



**University of
Zurich^{UZH}**

Department of Informatics

Principle-Driven Continuous Integration: Simplifying Failure Discovery and Raising Anti-Pattern Awareness

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Carmine Vassallo
from Benevento, Italy

approved in October 2020

at the request of
Prof. Dr. Harald C. Gall
Prof. Dr. Laurie Williams
Dr.-Ing. Sebastian Proksch



**University of
Zurich^{UZH}**

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, October 21, 2020

The Chairman of the Doctoral Board: Prof. Dr. Thomas Fritz

Acknowledgements

“The road of life twists and turns and no two directions are ever the same. Yet our lessons come from the journey, not the destination.”

— Don Williams, Jr.

Getting a PhD means acquiring skills that go (way) beyond the scope of this dissertation. Looking back, I simply feel I am a better person now. This was thanks to several people that I have encountered during my *journey*.

I thank my advisor, Prof. Harald C. Gall, that made this *journey* possible. Harald, I have sincerely appreciated your unconditional support and the freedom that you granted me. You encouraged me to look beyond the scope of my work and think about its impact on society. I was humbled to be part of your group.

Meeting Sebastian Proksch was a crucial step in my *journey*. Seb, you have been not only a collaborator but also a mentor. I learned a lot from you. Besides being the toughest reviewer in our community, you are the person that taught me how challenges have to be faced rather than avoided and how every occasion can turn into a chance of promoting your profile and your good work. I will never forget the late afternoons spent together. One might think we were working until late. In reality, we were doing all but working, discussing the most diverse topics.

I had the pleasure to meet a few more people that influenced my *journey*. Fabio Palomba, getting in touch with you and your passion for research was fundamental to continue my *journey*. Prof. Alberto Bacchelli, you have been the senior person that was always available when I needed. I liked the way you challenged me during presentations. A special mention goes to Prof. Massimiliano Di Penta. Max, you introduced me to the research world and we collaborated on my most important

projects: thank you for your help. I also want to thank Prof. Laurie Williams for taking the time to examine my dissertation and her insightful comments. Many thanks to all my co-authors: I could not complete this *journey* without you!

Several were the colleagues that eased my *journey*. Christoph Laaber, Gerald Schermann, Giovanni Grano, you are the best mates one could desire to have. I will remember the great discussions about research and, more importantly, all the jokes and the funny moments we had! Anna Jancso, I was impressed to see how you grew as a researcher: thanks for working closely in the last mile of my *journey*. I am grateful for the time spent with Carol Alexandru, André Meyer, Alexander Lill, Jian Gu, Manuela Züger, Sebastiano Panichella, Jürgen Cito, Katja Kevic, Sebastian Müller, Gül Çalikli, Fernando Petruccio, Pavlína Wurzel Gonçalves, Prof. Philipp Leitner, and Prof. Thomas Fritz, and particularly with Enrico Fregnan, Larissa Barbosa, Adelina Ciurumelea, and Pasquale Salza. I had the privilege of co-advising several bachelor and master students during their theses or projects. Thank you all for contributing to my *journey* with your individual work.

Many thanks to my mother Maria, my father Giovanni, my sisters Antonella and Lucia, and my friends for spurring me to chase my dreams. We learned how to master Skype over the last few years!

I reserve the final words of this *journey* for Chiara: *Grazie per aver resistito alla lontananza e per farmi diventare ogni giorno una persona migliore.*

Carmine Vassallo
Zurich, 18 September 2020

Abstract

Continuous Integration (CI) is a software development practice that enables developers to build software more reliably and quickly. Most organizations have started adopting CI, however, only a few of them achieve the expected benefits. The reason is that living up to the recommended practices (called principles) of CI is not easy and developers tend to follow anti-patterns, which are ineffective solutions to recurrent problems. Anti-patterns violate principles and lower the effectiveness of CI.

In this dissertation, we characterize the problem of anti-patterns to implement solutions that help developers follow principles. We start with classifying the anti-patterns encountered by developers in practice and identifying their four root causes, which are (i) the poor knowledge of the prerequisites for adopting CI, (ii) the difficulty of inspecting build failure logs, (iii) the presence of bad configurations, and (iv) the wrong usage of a CI process. While only better coaching in CI can efficiently remove the former, we implement several approaches to address the other causes. To improve the understandability of build failure logs, we develop Bart, a tool that produces summaries for the most common build failure types. To identify anti-patterns caused by configuration smells that developer should remove, we propose CD-Linter, a semantic linter for CI/CD configuration files. We implement CI-Odor, an automated reporting tool that leverages information from repository and build history, to monitor the wrong adoption of CI over time.

The results of multiple empirical studies conducted with professional developers show that the proposed approaches are effective at identifying and removing the aforementioned causes of anti-patterns and, consequently, at enforcing a *principle-driven continuous integration* practice.

Contents

1	Synopsis	3
1.1	The Problem of Anti-patterns	5
1.2	Thesis Statement and Research Questions	7
1.3	RQ ₁ : On the Nature of Anti-patterns in CI	10
1.3.1	RQ _{1.1} : Investigating the Existence of Anti-patterns	11
1.3.2	RQ _{1.2} : Characterizing Anti-patterns	11
1.4	RQ ₂ : Simplifying the Inspection of Build Failure Logs	14
1.4.1	RQ _{2.1} : Categorizing Build Failures	14
1.4.2	RQ _{2.2} : Build Failure Summarization	15
1.5	RQ ₃ : Detecting Configuration Smells that Affect Build Pipelines . .	17
1.6	RQ ₄ : Reporting the Wrong Usage of a Build Pipeline	20
1.7	Scope and Limitations	22
1.7.1	Scope	22
1.7.2	Limitations	24
1.8	Scientific Implications	26
1.9	Opportunities and Future Work	28
1.10	Related Work	30
1.11	Summary and Contributions	32
1.12	Outline	33
2	Continuous Code Quality: Are We (Really) Doing That?	39
2.1	Introduction	40
2.2	Background and Related Work	41

2.2.1	Continuous Code Quality	41
2.2.2	Related Work	42
2.3	Overview of the Research Methodology	43
2.4	Continuous Code Quality Data Collection	43
2.4.1	Collecting CCQ Data	44
2.4.2	Collecting CI Data	44
2.4.3	Overlaying CCQ and CI Information	45
2.5	Continuous Code Quality in Practice	46
2.5.1	Definition of CCQ Metrics	46
2.5.2	On the Current Application of CCQ	47
2.6	Discussion and Future Work	50
2.7	Threats to Validity	52
2.8	Conclusion	53
3	Bad Practices in Continuous Integration	55
3.1	Introduction	56
3.2	Related Work	58
3.2.1	Studies on Continuous Integration and Delivery Practice	59
3.2.2	Continuous Integration Bad Practices and Barriers	60
3.3	Empirical Study Definition and Planning	62
3.3.1	Research Questions	62
3.3.2	Study Methodology	63
3.4	Empirical Study Results	73
3.4.1	Overview of the CI Bad Smell Catalog	73
3.4.2	Perceived Importance of CI Bad Smells	81
3.4.3	Mapping and Comparison with Duvall's Anti-Patterns	88
3.5	Threats to Validity	94
3.6	Implications	96
3.6.1	Implications for Practitioners	96
3.6.2	Implications for Researchers	98
3.6.3	Implications for Educators	99
3.7	Conclusions and Future Work	100
4	A Tale of CI Build Failures	103

4.1	Introduction	104
4.2	Study Design and Planning	106
4.2.1	Study Context	106
4.2.2	Data Extraction Approach	107
4.2.3	Definition of the Build Failure Catalog	108
4.2.4	Analysis of Build Failure Proportions	109
4.3	Study Results	110
4.3.1	What Types of Build Failures Affect Builds of OSS and ING Projects?	110
4.3.2	How Frequent Are the Different Types of Build Failures in the Observed OSS and ING Projects?	118
4.4	Discussion	124
4.5	Threats to Validity	126
4.6	Related Work	127
4.7	Conclusion	129
5	Developer-Oriented Assistance for Build Failure Resolution	131
5.1	Introduction	132
5.2	Creating a Build Abstraction and Recovery Tool (BART)	136
5.2.1	Detecting Failure Information in the Log	138
5.2.2	Summarizing Build Log Information	140
5.2.3	Hints from External Sources	144
5.3	Investigating the Effect of Bart on the Build Fixing Practice	145
5.3.1	Context	146
5.3.2	Experimental Procedure	148
5.3.3	Understandability of Build Breaks	150
5.3.4	Resolution Time of Build Failures	153
5.3.5	General Feedback on Bart	156
5.4	The Developer's Perception of the Build Fixing Practice	157
5.4.1	Survey on the Perception of Build Failures	157
5.4.2	How Do Developers Approach Different Types of Build Failures?	159
5.4.3	What Types of Build Failures Are Hard to Fix?	173
5.5	Discussion	178

5.6	Threats to Validity	181
5.7	Related Work	182
5.8	Summary	185
6	Configuration Smells in Continuous Delivery Pipelines	187
6.1	Introduction	188
6.2	Methodology Overview	190
6.3	CD-Linter Description	191
6.3.1	Background	192
6.3.2	Selection of Relevant CD Smells	193
6.3.3	Parsing CD Configuration Files	194
6.3.4	Detection of the CD Smells	196
6.4	Empirical Study Design	199
6.4.1	Context Selection	199
6.4.2	Monitoring of the Opened Issues	200
6.4.3	Manual Validation of CD Smells	201
6.4.4	Measurement of CD Smell Occurrences	202
6.5	Empirical Study Results	202
6.5.1	Are the Four CD Smells Relevant to Developers?	202
6.5.2	How Accurate Is CD-Linter?	205
6.5.3	How Frequent Are the Investigated CD Smells in Practice?	208
6.6	Threats to Validity	211
6.7	Discussion	212
6.8	Related Work	214
6.8.1	Bad Practices in CI/CD	214
6.8.2	Detection of Smells in Development Workflows	215
6.9	Summary	216
7	Automated Reporting of CI Anti-Patterns and Decay	217
7.1	Introduction	218
7.2	Methodology Overview	219
7.3	Which Anti-Patterns to Detect, and how?	220
7.3.1	Pre-Selection	221
7.3.2	Survey on the Practical Relevance	222

7.4	Reporting CI Practices	227
7.4.1	Slow Build	228
7.4.2	Skip Failed Tests	230
7.4.3	Broken Release Branch	230
7.4.4	Late Merging	231
7.5	Empirical Assessment of the CI-Odor Summaries	232
7.5.1	Projects Selection	233
7.5.2	Quantification of the Phenomenon	234
7.5.3	Survey on Generated Reports	236
7.6	Discussion	239
7.6.1	Threats to Validity	240
7.6.2	Future Work	241
7.7	Related Work	242
7.8	Summary	243

List of Figures

1.1	A CI-based development workflow	5
1.2	Overview of the Research Questions	8
1.3	The enhanced CI-based development workflow	23
1.4	Structure of the next chapters of this dissertations and their relations with the formulated research questions	34
1.5	The other research works that are not included in this dissertation .	37
3.1	Process for CI bad smell catalog's creation and validation	64
4.1	Percentage of build failures for each category	119
4.2	Clusters of projects based on the distribution of build failures . . .	122
5.1	Overview of the Build Summarization approach	135
5.2	Meta-Model that is available to the hint generators in Bart	137
5.3	An example of hints derived from StackOverflow discussions	144
5.4	Understandability of build summaries	151
5.5	Relevance of solutions proposed in the hints	152
5.6	Applicability of solutions proposed in the hints	152
5.7	The process of fixing <i>Compilation</i> build failures	166
5.8	The process of fixing <i>Dependencies</i> build failures	167
5.9	The process of fixing <i>Testing</i> build failures	169
5.10	The process of fixing <i>Code-Analysis</i> build failures	170
5.11	Build failure types our participants struggle with	174
5.12	Root causes of infrastructure-related build failures	176
6.1	Overview of the methodology for evaluating the usefulness of CD-Linter	189
6.2	Example excerpt of a GitLab configuration	193
6.3	Example of a pom.xml file	195
6.4	Reactions to the issues opened to evaluate CD-Linter	203
7.1	Overview of the methodology used to build and evaluate CI-Odor .	220
7.2	Main questions of the survey on the relevance of CI anti-patterns .	222
7.3	Likert-scale answers to the survey on the relevance of CI anti-patterns	225

7.4	CI-reporting process in CI-Odor	227
7.5	Example summaries of the four anti-pattern detectors	229
7.6	Example of different late-merging scenarios	233
7.7	Questions of the survey on reports generated by CI-Odor	236
7.8	Likert-scale answers to the survey on reports generated by CI-Odor	238

List of Tables

2.1	CCQ usage indicators applied to our projects	47
3.1	Summary of the study data used to create and validate the CI bad smell catalog	65
3.2	Companies involved in the interviews on CI bad smells	66
3.3	Experts interviewed on CI bad smells	66
3.4	Results of CI bad smells' perception - Repository	74
3.5	Results of CI bad smells' perception - Infrastructure Choices	75
3.6	Results of CI bad smells' perception - Build Process Organization	76
3.7	Results of CI bad smells' perception - Build Maintainability	78
3.8	Results of CI bad smells' perception - Quality Assurance	79
3.9	Results of CI bad smells' perception - Delivery Process	80
3.10	Results of CI bad smells' perception - Culture	81
3.11	CI bad smells considered relevant by the majority of respondents.	83
3.12	CI bad smells considered as not relevant by the majority of respondents.	84
3.13	Mapping between Duvall's patterns (and their corresponding anti- patterns) and CI bad smells	90
4.1	Build failure categories	113
4.2	Distribution of build failure types in each cluster	123
5.1	Projects used in the controlled experiment for evaluating Bart	146
5.2	Mutated components in the systems used to evaluate Bart	148
5.3	Resolution time per build failure type	156
5.4	Questions about the perception of build failures	158

6.1	Fuzzy Version syntax in Python and Maven	198
6.2	Detection precision for the four CD smells	206
6.3	Detection precision for the Fuzzy Version subtypes	206
6.4	CD smells in the analyzed projects and owners	208
6.5	CD smells across different .gitlab-ci.yml sizes	209
6.6	Break-down of Fuzzy Version smell	210
7.1	CI anti-patterns detected in the analyzed projects	235

Acronyms

α Krippendorff's reliability coefficient

API Application Programming Interface

BART Build Abstraction and Recovery Tool

CB Percentage of Checked Branches

CCQ Continuous Code Quality

CD Continuous Delivery

CI Continuous Integration

CQCR Code Quality Checking Rate

EFC Elapsed Frame between Checks

ETC Elapsed Time between Checks

FP False Positive

GDM Goal Question Metric

IaC Infrastructure as Code

IDE Integrated Development Environment

k Cohen's kappa inter-rater agreement

OSS Open Source System

REST REpresentational State Transfer

RQ Research Question

SCM Source Code Management

SO StackOverflow

TP True Positive

VCS Version Control System

1

Synopsis

The complexity of software systems and the continuous demand for new features require software to be produced in a collaborative manner [122]. Developers work in teams and manage their code contributions sharing a repository and an issue-tracking system. Every developer picks or gets assigned to one or more coding tasks. The implementation starts when the developer checks out the current state of a project, which typically corresponds to the latest version stored in the repository, and works independently on her tasks. As a developer completes her tasks, she is ready to copy her code to the repository. If the existing code is unchanged, a developer can safely add her new changes. Otherwise, several merge conflicts may arise, for example when different developers have worked on the same method's implementation or when a developer has relied on methods whose behavior has meanwhile been modified by others. Resolving merge conflicts can be difficult, especially when changes diverge significantly [73]. The likely severity of those conflicts increases as more people work on the same project components, which might lead to several changes to the same code in parallel, and as more time passes since the last checkout from the repository, which might decrease the validity of developers' assumptions about the current project's state [74]. Severe conflicts have negative consequences for the business [73]. Testing and fixing integration bugs can take long and the planned releases are running late.

Continuous Integration (CI) is the practice that enables developers to reduce the frequency and severity of merge conflicts, and to build reliable software that is immediately ready for being deployed and potentially released. In their seminal book on this topic, Duvall et al. [25] define CI as follows:

Continuous Integration is the practice that requires developers to integrate changes into a shared repository frequently, preferably several times a day. Each integration gets typically verified (i.e., changes get compiled, tested, and quality checked) through an automated build infrastructure.

Given its proven benefits, such as faster releases [46] and increased developer productivity [124], most of the organizations started adopting CI. Based on a recent study, 62% of the organizations have a build system configured in their projects.¹ However, the mere introduction of an automated build infrastructure in the development workflow is not sufficient to practice CI well and most organizations do not achieve the expected benefits. Compared to a few big companies such as Amazon, Facebook, Netflix, and Google that deploy code to production thousands of times a day,² the release frequency in other companies is much lower. The results of a recent study³ show that most companies release changes between once per week and once per month. One of the reasons for lower performances is the fact that many organizations invest in new tools without putting enough effort into establishing a new culture within teams [137]. Instead, CI implies a cultural shift that involves several challenges [45]: (i) debugging build failures is not straightforward as they happen on a remote server with limited access; (ii) waiting for the results of long builds threatens developer productivity; (iii) automating manual steps requires dedicated personnel; (iv) the size of each coding task has to be small enough to enable developers to integrate frequently their code contributions.

To correctly apply CI and fully benefit from it, developers need to overcome these challenges with the definition of a new development workflow that supports them in following CI principles [25]. For example, developers should establish policies to timely merge feature branches into the mainline, make a runnable version of the code always available, and automate all testing activities. Although the extensive list of CI principles has been thoroughly described in several books [25, 47] and has gained wide acceptance since its publication, following those principles is not easy and developers deviate from them generating anti-patterns. CI anti-patterns are temporary and ineffective solutions to recurrent problems.

¹<https://www.infoq.com/articles/infoq-reader-survey-2019-results/>

²<https://dzone.com/articles/release-frequency-a-need-for-speed>

³<https://services.google.com/fh/files/misc/state-of-devops-2019.pdf>

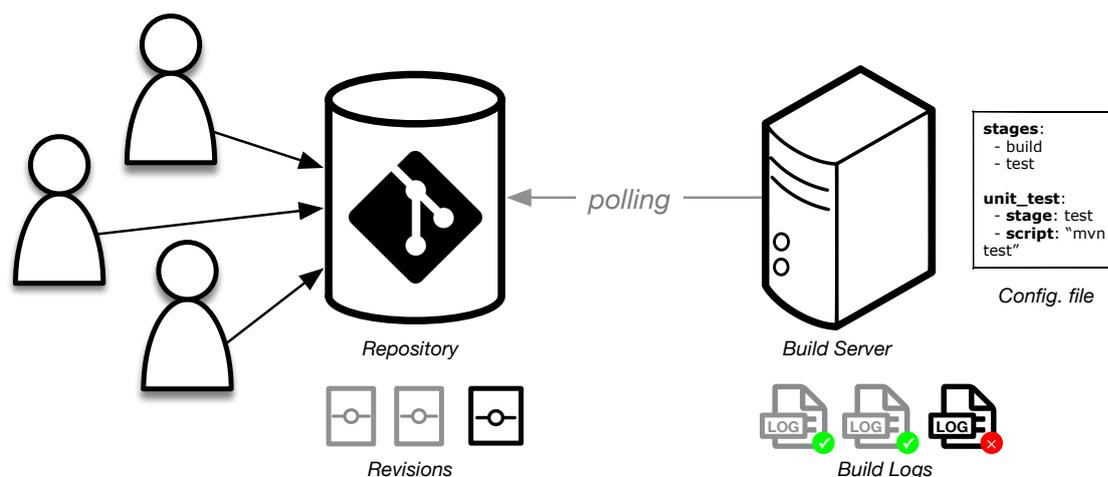


Figure 1.1: A CI-based development workflow

1.1 The Problem of Anti-patterns

The core of a CI-based development workflow (Figure 1.1) is the build server that constantly checks the shared repository for newly committed changes. Examples of build servers are TravisCI,⁴ Jenkins,⁵ and GitLab.⁶ When a new change introduced by developers is detected (i.e., a new *revision* is generated), the build server triggers a new build, which is a pipeline consisting of several *build steps* (e.g., compilation, testing) that verify the absence of issues (e.g., compilation errors, bugs) in the new code. Based on the type and severity of the identified issues, a build can succeed (*build success*) or fail (*build failure*). The outcome of a build success is a new software version that can later be deployed. A *build log* containing the output of all the actions performed during build steps is produced at the end of the build. Developers can configure steps and define their own *build pipeline* through a *configuration file* that is provided by the build server (e.g., `.gitlab-ci.yml` in case of GitLab, `travis.yml` in case of TravisCI). Build servers need to execute specialized *build tools* to perform build steps. Differently from build servers, the choice of build tools depends on the programming language used in the project. Examples of build tools are Maven⁷ and Gradle⁸ for Java projects. The execution

⁴<https://travis-ci.org/> ⁵<https://www.jenkins.io/> ⁶<https://about.gitlab.com/>
⁷<https://maven.apache.org/> ⁸<https://gradle.org/>

of some build steps can be offloaded on external services that do the work and communicate the results to the build server. This is the case of *Continuous Code Quality (CCQ) services* like SonarCloud⁹ that check source code for code quality issues. Finally, developers can implement a *monitoring layer* (built with tools like Splunk¹⁰) that analyze build outputs such as revisions and build logs with the goal of continuously assessing the performance of the pipeline, verifying its velocity and stability.

Developers can sometimes manipulate or misuse the pipeline with modalities that, consciously or not, violate principles and generate anti-patterns. Unprincipled choices are typically well-intended reactions to immediate, critical issues that would, for example, delay a release. However, these choices have negative mid/long-term effects and are therefore considered harmful for a project, because more effort is needed to fix issues afterward. In this way, the expected advantages of CI such as improving the reliability and quality of released changes can get compromised. In the following, we illustrate a few exemplar situations where developers generate anti-patterns and describe their consequences.

Skipping Tests or Code Quality Checks Delaying a release is typically not possible because it can cause the termination of a contract already agreed with the client. It might happen that one or more tests are failing or that the number of vulnerabilities exceeds the threshold shortly before the due date of the next release. In such cases, developers can decide to skip tests or ignore code quality warnings. This has the immediate advantage of letting developers release on time. Nevertheless, a future bug in production can cause a more serious loss of money.

Long-Running Branches Developers prefer to work independently on the assigned features and share code with the other teammates only when the implementation is completed [3]. Developers create feature branches and work on their features without disturbances from other teammates. However, some features require several days or even weeks to be completed. Thus, if they adopt feature branching, developers might end up committing a big amount of changes at once after some time. This choice will provoke severe merge conflicts that are hard to solve.

⁹<https://sonarcloud.io/> ¹⁰<https://www.splunk.com/>

Allowing Builds to Fail Build failures are usually very time consuming to solve [50]. Developers need to inspect long and verbose build logs to understand the root causes of build failures. They also happen on a remote build server that developers do not fully control and where debugging capabilities are limited. Given the difficulty of troubleshooting build failures, developers can decide to allow builds to fail so that their productivity is not hindered by possible errors. In this way, all the effort of defining a build infrastructure is wasted and defects might escape unnoticed.

These examples show that anti-patterns do not really solve problems but they just postpone their resolution. The longer anti-patterns “survive”, the more problems, such as bugs, accumulate and become more difficult to solve. Prompt identification and removal of anti-patterns are crucial to achieve the expected benefits of CI.

1.2 Thesis Statement and Research Questions

The presence of anti-patterns has a disruptive effect on the CI practice and its goals. We want to support developers in removing the causes of anti-patterns from their development workflows and we state our thesis as follows:

The empirical characterization of anti-patterns that exist in the practice of CI allows to implement solutions that reinforce a principle-driven software development through a better understanding of failures and the awareness of bad practices.

As depicted in Figure 1.2, we formulated several research questions to investigate our thesis. Each of them is described in the following sections and corresponds to a chapter of this dissertation (see Section 1.12 for more details about its structure).

We started our work investigating and characterizing the problem of anti-patterns by answering the following research questions:

RQ₁ *How do developers fail to get CI done right?*

RQ_{1.1} *Do developers deviate from CI principles?*

RQ_{1.2} *What are the anti-patterns encountered by developers in practice?*

Developers can deviate from CI principles generating anti-patterns [25]. However, while the phenomenon of anti-patterns was hypothesized in several

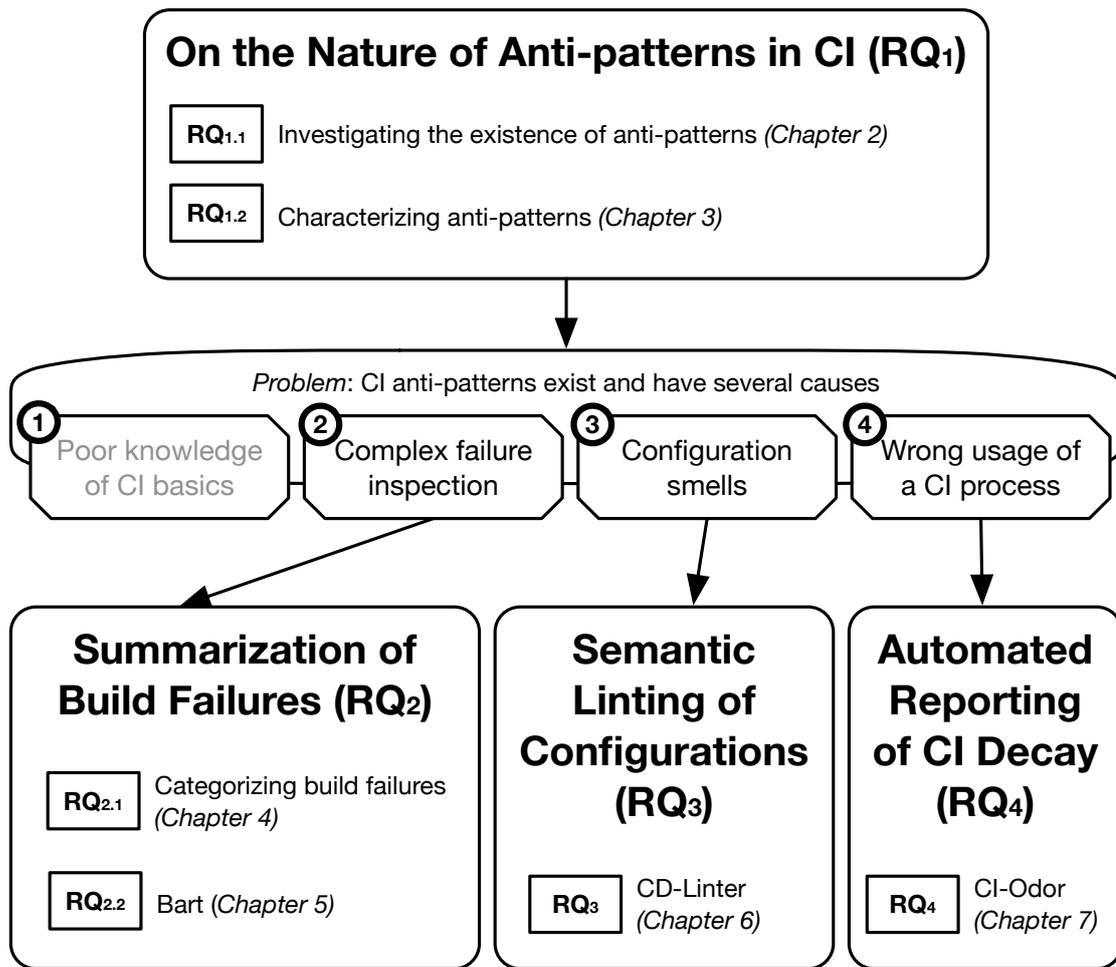


Figure 1.2: Overview of the Research Questions

books [25, 47], none had assessed their existence in practice. To gain initial empirical evidence of the presence of anti-patterns, we conducted a preliminary study on the violation of core CI principles (**RQ_{1.1}**). Specifically, we studied the extent to which the principle of continuous inspection is violated investigating how frequently developers that adopt CI automatically check for code quality in their projects. Because of the initial results revealing the existence of deviations from principles, we decided to perform a broader study on the phenomenon of CI anti-patterns (**RQ_{1.2}**). We derived a catalog

¹¹<https://stackoverflow.com/>

of existing CI anti-patterns conducting semi-structured interviews with practitioners and manually analyzing StackOverflow¹¹ posts that were discussing issues related to the adoption of CI.

From the interpretation of bad practices encountered by developers in practice, we identified four main causes for the presence of anti-patterns (as illustrated in Figure 1.2). The poor knowledge of prerequisites for implementing a CI workflow (1), such as the adoption of a dependency management system, can only be addressed with better coaching in the CI discipline and is out-of-scope of this dissertation. Novel approaches for eliminating the other causes, which are instead provoked by limitations of current tools and lack of support when configuring and using CI, are needed. **RQ₂** investigates the difficulty of understanding build logs during failures (2). **RQ₃** and **RQ₄** study the presence of bad configurations in build pipelines (3) and the wrong usage of a CI process (4), respectively. While answering research questions, we built approaches that empower developers to avoid anti-patterns.

RQ₂ *How can we reduce delays caused by the difficulty of inspecting build failure logs?*

RQ_{2.1} *What are the most common causes of build failures?*

RQ_{2.2} *How can we summarize build failure logs?*

CI principles advocate a prompt resolution of any stop in the pipeline [25]. A build failure stops the line because developers cannot successfully check in new changes until the build is fixed. Although build failures are important to detect issues early in the development process, troubleshooting them is often a difficult task because developers need to inspect logs that mostly contain information that is irrelevant to the failure. To improve the understanding of build failures and avoid long build resolution sessions, we built an approach for summarizing build failure logs. We conducted several iterations of card sorting applied to errors present in build logs of open-source and closed-source projects and were able to derive a taxonomy of build failures based on their causes in the logs (**RQ_{2.1}**). We then developed a tool that produces summaries for the most common build failure types. We evaluated its effectiveness through a controlled experiment and its limitations via a survey (**RQ_{2.2}**).

RQ₃ *How can we warn developers about improper configurations of a build pipeline?*

Some anti-patterns occur when developers define and maintain a build pipeline, which are not easy tasks [45]. Build servers typically offer online linters that can be used to validate configuration files. However, those tools only spot basic syntactic errors such as the use of reserved keywords when naming build steps. To identify violations of CI principles in the form of configuration smells that developer should remove, we developed a semantic linter for CI/CD configuration files. We evaluated our tool opening issues in open-source projects and monitoring the acceptance of our bug reports and the removal of reported smells.

RQ₄ *How can we detect the wrong usage of a build pipeline?*

Anti-patterns can also slowly creep into a project over time and provoke CI decay. For example, to deliver a new feature faster, one might consider skipping failing tests that currently stop the line. While it gives an immediate advantage, this choice might result in a more serious delay caused by a bug in production. To increase the awareness about CI decay caused by anti-patterns and help developers react to them, we built a monitoring tool that leverages information from the repository and the build history to identify the presence of bad practices over time. We surveyed developers to identify a good set of detectors and received feedback on our idea. We then evaluated our tool sending developers reports produced for open-source projects.

1.3 RQ₁: On the Nature of Anti-patterns in CI

In his landmark work about CI [25], Duvall outlines best practices but also warns developers against the danger of introducing anti-patterns. Duvall emphasizes that CI aims at “improving software quality and reducing risks”. A fundamental principle that developers should follow to accomplish this mission is continuous inspection, also known as Continuous Code Quality (CCQ), which includes running automated tests and performing code quality analyses at every committed change.

1.3.1 RQ_{1.1}: Investigating the Existence of Anti-patterns

We started investigating the existence of CI anti-patterns in development workflows by studying the adoption of CCQ.

Research Methodology We considered 148,734 builds of 119 Java projects that were produced over 5 years of development to study the current adoption of CCQ. We selected those projects that were both using TravisCI as the build server and SonarCloud as the code quality analyzer. For each build, we collected *build id*, *message* and *id* of the triggering commit, *branch's name*, *build status*, *starting* and *ending date*. If the build was inspected on SonarCloud, we also marked the build as checked. We devised several indicators to assess the application of CCQ based on the extracted features. We measured the *Code Quality Checking Rate* (CQCR), the *Elapsed Time and Frame between Checks* (ETC and EFC), and the *Percentage of Checked Branches* (CB).

Results When investigating the adoption of CCQ, we found that only 11% of the builds are qualitatively checked. Developers typically perform a code quality inspection after 18 builds, and, most likely at the end of a sprint. Furthermore, only 36% of the branches are checked for the presence of code quality issues. These results show that CCQ is not applied in practice. The existence of such a major deviation from CI principles prompted us to perform a second and broader study to derive a catalog of CI anti-patterns.

1.3.2 RQ_{1.2}: Characterizing Anti-patterns

Based on the previous findings revealing the existence of deviations from core principles, we decided to conduct a broader study aiming at characterizing the anti-patterns encountered by developers in practice.

Research Methodology Although CI anti-patterns are mentioned in several books [25, 47] and even catalogs [27] exist, previous works did not study which anti-patterns are relevant and occur in practice. We empirically derived a catalog of existent anti-patterns conducting semi-structured interviews with 13 practitioners

that use or manage the CI pipelines of 6 medium/large companies and manually analyzing over 2,300 StackOverflow posts where developers discuss issues related to the adoption of CI. After the inception phase that we used to increase our knowledge about CI anti-patterns by inspecting existing resources [25, 26, 27], we interviewed practitioners and extracted from the transcriptions those sentences that contain good and bad CI practices. We also extracted CI practices from the selected StackOverflow posts. Specifically, two authors independently read the posts and marked as relevant those posts where developers talk about CI practices. The authors agreed on the relevance of 553 posts and assigned sentences from both these posts and interviews to one or more categories of anti-patterns based on the discussed violations. From the 182 anti-patterns that we recognized in the sentences, we excluded, during several iterations, those that are not specific to CI and are more bugs than misuses of a CI pipeline. We also merged similar anti-patterns. To assess the completeness of our catalog, we classified a random sample of CI-related posts from StackOverflow that were not analyzed before and verified that the catalog was comprehensive enough to classify unseen discussions. We measured the importance of each anti-pattern surveying 26 CI experts working in 21 different companies and compared our catalog with the one previously defined by Duvall [27].

Results The final version of the catalog includes 79 anti-patterns grouped into 7 categories or aspects of a CI process management and extends the Duvall’s catalog [26] without including practices that are not specific to CI. Anti-patterns occur when developers keep working on many branches for a long period or they do not version the necessary configurations (*Repository* category). The improper choice of hardware and software components can make the build slow and unreproducible (*Infrastructure Choices* category). A wrong definition of build steps, the way steps are scheduled in the build process, and the complexity of build logs cause *Build Process Organization* anti-patterns. Duplications and environment-specific build steps are examples of bad practices that decrease *Build Maintainability*. Other anti-patterns concern the inadequacy of testing and code inspection strategies (*Quality Assurance* category), the inefficient preparation for release (*Delivery Process* category), and the lack of a DevOps culture within teams (*Culture* category).

To better interpret the results of the research question, the author of this dissertation performed an open card sorting of the derived anti-patterns and categorized them based on their causes. The produced classification was also reviewed by a different researcher (while he was working in a software company that adopts CI) to have an external opinion about the validity of the identified causes.¹² Four main causes of anti-patterns were found:

- *Poor knowledge of the fundamentals of CI.* Anti-patterns, such as missing use of a dependency management system and lack of automated testing, result from a superficial understanding of the prerequisites for doing CI.
- *Difficulty of understanding build failure logs.* Because of the complexity of build failure logs, developers are typically slow in repairing build failures. The build stays broken for long periods of time preventing developers from checking out functioning code.
- *Configuration smells in build pipelines.* Bad configurations of the build pipeline cause anti-patterns. For example, developers can configure build steps that are triggered manually or allow some of them to fail. These settings threaten the effectiveness of build pipelines.
- *Wrong usage of a build pipeline.* A well-configured build pipeline can be wrongly used generating anti-patterns. Developers do not commit often causing severe merge conflicts or remove failing tests instead of fixing their underlying root causes that lead to incidents in production.

While better coaching in CI is needed for the former, developers require better support to recognize and avoid the other causes of anti-patterns.

Answer to RQ₁

Developers deviate from core CI principles such as the continuous inspection of code. We built a catalog featuring 79 anti-patterns that regard several aspects of the CI process and interpreted their causes.

¹²<https://doi.org/10.5281/zenodo.4446239>

1.4 RQ₂: Simplifying the Inspection of Build Failure Logs

Troubleshooting a CI build failure is the most important barrier that developers face when approaching to CI [45]. They cannot easily debug build failures setting breakpoints to investigate a program's state right before a crash, as the failures happen on the CI server that developers do not fully control. To understand the root causes of build failures developers inspect build logs, which contain all the actions performed by tools executed during the build (see Figure 1.1). Because of their length and verbosity, developers are slow in extracting the information that is relevant to formulate a fix. Long build debugging sessions cause stops in the build pipeline, preventing developers from checking out functioning code.

1.4.1 RQ_{2.1}: Categorizing Build Failures

As the first step towards more understandable build failures, we built an approach that automatically classifies build failures according to their root causes.

Research Methodology To build a comprehensive taxonomy, we studied build failures occurring both in industrial and open-source contexts. Specifically, we investigated Maven build failures in 418 projects from ING,¹³ which is a large financial organization located in The Netherlands, and in 349 open-source projects from the TravisTorrent dataset [11]. We manually inspected the most popular error messages contained in 34,182 build logs (both from ING and TravisTorrent) and extracted those keywords (e.g., the failed build step, which is called `goal` in Maven) that better describe the type of failure (e.g., a testing failure). Two of the authors grouped keywords into categories and other two reviewed the produced categorization. The respective reviews were discussed among the authors that, after several iterations, reached a consensus. The resulting taxonomy was then used to assign all previously collected build failure logs to the corresponding categories based on matched keywords. A random and statistically significant sample of classified failures per category was inspected by two authors to determine the

¹³<https://www.ing.nl/>

validity of the produced classification. If changes to the taxonomy were needed, all build failure logs were re-classified and a new sample was validated. This process was iterated until the taxonomy became stable. We used the final version of the taxonomy to measure and compare the frequency of build failure types in industrial and open-source contexts.

Results The final version of the taxonomy contains 171 keywords grouped into 20 categories. We measured their frequency both in ING's and TravisTorrent's projects. If we exclude failure types corresponding to steps that are not scheduled in a typical build process (i.e., preparation for a new release and support), and those (i.e., Deployment) that can include other failure types, *Compilation* failures (due to errors while compiling production and test code), *Testing* failures (occurring while performing unit, integration, and non-functional testing activities), *Code Analysis* failures (caused by unmet code-quality criteria), and *Dependencies* failures (when required libraries are missing) are the most frequent types in both industrial and open-source context. The occurrences of other types are negligible.

1.4.2 RQ_{2.2}: Build Failure Summarization

Starting from the previously defined taxonomy, we devised a summarization technique that provides developers with summaries for common build failure types.

Research Methodology Bart (that stands for Build Abstraction and Recovery Tool) is an approach for summarizing Maven build failures and ease their comprehension. We decided to focus on the most common build failure types when implementing Bart, which has been released as a plugin for Jenkins. It parses build logs and constructs a meta-model that only contains information that is relevant to a failure (e.g., failed build steps and their execution lines). Such a meta-model is used to provide an overview of the modules that failed, determine the failure type (based on the previously defined taxonomy) and, finally, produce summaries for common failure types. Summaries are build-failure-type specific but they have the same structure including hints about the location of errors causing failures (e.g., name of the class and line number, name of the dependency) and its reasons (e.g., an abstract class is instantiated, a dependency is missing). When similar

failures are discussed on StackOverflow, Bart also provides solution hints, which are links to relevant StackOverflow posts that can guide the resolution process. We conducted a controlled experiment with 17 developers to evaluate Bart's capability to improve the understandability of build failures and accelerate their resolution process. We injected bugs in three well-known software systems written in Java and built with Maven to generate instances of build failure types supported by our tool. We then used Bart to generate summaries for these build failures and measured the performances of our participants in fixing build failures with and without summaries. We also asked our participants to rate the understandability of summaries compared to logs and the usefulness of solution hints. Given the divergent opinions on the applicability of solution hints, we conducted a survey among developers on how they react to build failures and what types they find difficult to solve. We obtained 101 valid responses. Two authors analyzed the received open answers through a multi-step open-coding approach [111] and derive resolution workflows for each build failure type.

Results The results of our controlled experiment show that Bart improves a lot the understandability of the most common build failure types and developers are faster in troubleshooting build failures. The resolution time is reduced by 23% when solving Testing failures, 20% when repairing Compilation failures, 43% when fixing missing Dependencies, and 62% when dealing with Code Analysis failures. However, the results also show that an optional part of our summaries that consists of solution hints (i.e., links to relevant StackOverflow discussions) is not always useful. While they seem to be helpful for some failure categories (e.g., compilation and code analysis), solution hints are not so valuable in case of dependency-related failures and simply do not work for testing. Comparing the resolution workflows obtained by analyzing the results of our survey, we found that build failures are rarely solved in a collaborative effort and developers typically do not search for similar solutions on the Internet. Apart from Testing failures that require also to be reproduced and debugged locally, developers mostly rely on the sole build log to define a fix strategy. For this reason solution hints, especially in case of testing failures that are project-specific, are not needed. Build logs contain enough information to fix a failure and, reducing their complexity, we are able to

speed up the resolution process. The results of our survey confirm the difficulty of solving common types such as testing and dependencies failures but also reveal the complexity of environmental failures (i.e., failures that are typically not expected within a standard application development lifecycle, such as infrastructure failures).

Answer to RQ₂

Build failures are mainly caused by compilation errors, failing tests, missing dependencies, and poor code quality. We built an approach for automatically detecting and summarizing those recurrent failure types with the indication of reasons and error locations. Our build summaries increases the understandability of build failure logs and let developers solve build failures faster avoiding severe stops of the build pipeline.

1.5 RQ₃: Detecting Configuration Smells that Affect Build Pipelines

Setting up a build pipeline is not trivial [45]. Because of the great flexibility offered by CI servers and the high competence required to configure them, developers can easily make mistakes even when defining a simple workflow. For this reason, developers validate their build configuration using the online linters that are directly provided by CI tool vendors. The biggest limitation of these linters is that they only spot syntactic errors, such as the use of reserved keywords when naming build steps, while previous researchers have shown that a semantic linting is also needed for build configurations. For example, linters should also detect misuses of configuration features [34], like executing commands in unsuitable build steps, and security weaknesses [94], such as the use of hard-coded passwords. Besides generating misuses and security smells, developers can also, consciously or not, follow anti-patterns when configuring a build pipeline. Therefore, we decided to implement CD-Linter, a novel semantic linter that warns developers against Continuous Delivery (CD) smells, which are process-related violations of CD

principles present in configuration files. CD is the practice that extends CI enabling the automated deployment of a new software version in production. While our tool is meant to scan build pipelines that also includes a deployment step, the current version of CD-Linter mainly detects violations of CI principles, which are also valid for CD. Despite being CI anti-patterns, we call those violations CD smells to avoid generating confusion with future extensions of the tool. CD-Linter inspects GitLab configuration files and detects four types of CD smells in Python projects and in Java projects built with Maven. Our tool parses `.gitlab-ci.yml` files and identifies cases where build steps are allowed to fail (*Fake Success*), manually executed (*Manual Execution*), or set to rerun multiple times after a failure (*Retry Failure*). Dependencies whose versions are not fully specified (*Fuzzy Version*) are also reported as CD smells.

Research Methodology To validate our tool we conducted an empirical study opening and monitoring issues in open-source projects hosted on GitLab.com. We selected 5,312 starred projects that are not forks and contain a `.gitlab-ci.yml` file in their repositories. We run CD-Linter against these projects and identified a random sample of detected CD smells in a way to achieve a balanced set of CD smells of each type and at most one CD smell per project owner. Out of the resulting 168 CD smells, we discarded 23 that contain false positives and opened 145 issues. We have monitored the issues over a period of 6 months and used the issue state (i.e., closed, open), the number of upvotes/downvotes, and the received comments to classify each reaction. We also inspected the source code to see how smells were removed. Based on the feedback to our tool received in the comments, we produced a new version of CD-Linter and generated a new random sample from the newest set of detected smells to be manually validated. We selected for each project owner, one CD smell of each type, if detected. Since for *Fuzzy Version* we have four sub-categories, we considered one of each sub-category, if present. Two authors independently validated the resulting sample of 868 issues. We measured the Cohen's kappa inter-rater agreement [18] ($k = 0.76$, high agreement) and the authors discussed and solved the disagreement cases. We computed the overall precision based on the validated sample and the recall using another randomly

selected sample of 100 projects. Finally, we measured the occurrences of CD smells in the latest snapshot of the analyzed projects.

Results During the 6-month observation period, 74% of the active projects reacted to our issues. Overall, 53% of the project maintainers reacted positively, with 9% that confirmed the validity of the reported problem and 44% that fixed it. 32% of the issues were rejected because the problem affected build steps that were not fully integrated into the pipeline yet or that used predefined templates containing smells. Other developers did not trust the output of an automated issue-reporting tool or they considered the smell being not applicable in some cases (i.e., differently from libraries, required tools should be always updated to the latest version). During the manual validation, CD-Linter achieved a precision of 87% and a recall of 94%. The identified false positives and negatives were caused by very specific cases, such as the use of unconventional names for steps executed during deployment, that can only be removed enabling developers to configure CD-Linter. From the analysis of the occurrence of CD smells in the wild, we detected 2,874 instances that affect 13% of the projects (and 14% of the owners). Fuzzy Version is the most common smell and is present in 37.1% of the analyzed projects. We found that longer configuration files (that potentially lead to more complex pipelines) are also more prone to contain CD smells. 69.7% of the overall detected smells affect long `.gitlab-ci.yml` files, with 50.9% of the Fuzzy Version problems, 88.2% of the Fake Success smells, 86.4% of the Manual Execution incidents, and even 98.7% of the Retry Failure instances.

Answer to RQ₃

A semantic linter can be used to warn developers against improper configurations that generate CI anti-patterns. Most of the developers confirmed the validity of configuration smells detected by our linter and fixed them. Our approach is accurate, scoring a precision of 87% and a recall of 94%. Smells tend to affect longer configuration files, where a linter can be more effective to spot hidden issues.

1.6 RQ₄: Reporting the Wrong Usage of a Build Pipeline

A well-defined build pipeline is only the prerequisite for a good CI practice. Developers still need to use the build pipeline correctly. They have to automate tasks, such as testing and code quality assurance that in some organizations are performed manually or only partially automated [45], and establish a branch-merging strategy to avoid the disproportionate effort of integrating changes from long-lived feature branches. Developers have also to be careful with adding dependencies and selecting tests not to raise the build time too much. Unfortunately, living up to those principles while using a build pipeline is hard and several are the anti-patterns that slowly creep into projects over time. To identify those anti-patterns, we implemented CI-Odor, an automated detection and reporting tool that monitors the usage of a build pipeline. Specifically, CI-Odor parses build logs of Java projects built with Maven on TravisCI and inspects their repositories to compute CI-related metrics, such as build durations, number of build failures on the release branch, and date of the last commit on a given branch. Those metrics are then used by detectors to spot the presence of anti-patterns.

Research Methodology We surveyed developers to identify a good set of detectors to implement in our tool. Specifically, we asked developers to rate the relevance of proposed detection strategies for several anti-patterns that were good candidates based on their feasibility to be identified from the extractable metrics. Our respondents had also the possibility to give us feedback on the detection strategies in the open answers, which we analyzed through several sessions of open card sorting [111]. We then refined strategies and developed the final set of detectors in CI-Odor. We conducted a study on open-source software projects to assess the accuracy and usefulness of our reporting tool, and to measure the occurrences of detected anti-patterns over the history of real projects. We selected active Maven projects built on TravisCI that were forked at least once and had more than 5 maintainers to ensure a certain community size. Among them, we only considered 36 projects whose development teams could be reached out through a public mailing list or an available group channel. We sent to those teams the reports produced by

CI-Odor for their projects. Finally, we also sent those reports to participants of the previous survey that were interested in seeing the output of our tool.

Results Based on the analysis of 124 valid responses to our survey, we decided to implement four detectors: Late Merging (i.e. branches are kept open for a long time without being merged into the master branch), Slow Build (i.e., the duration of some builds is significantly high), Broken Release Master (i.e., build breaks occur on the master branch and are not timely fixed), and Skip Failed Test (i.e., tests are skipped to repair build failures). Our respondents also confirmed that developers are typically unaware of anti-patterns occurring when using CI and that this causes CI decay (i.e., the goals of CI are not achieved). In the next study, we analyzed the reactions of developers to the reports generated by CI-Odor. According to 13 original developers, reports are generally useful and anti-patterns are properly identified. Many developers (67%) expect a positive effect of using our generated reports on their CI discipline. Even though they asked for an increased configurability and more suggestions on possible fixes for detected anti-patterns, 55% of the respondents were already willing to integrate CI-Odor in their CI processes. Late Merging is the most frequent problem affecting 97% of the projects and 67% of the analyzed branches. We detected at least one Broken Release Branch instance in all projects while Slow Build was identified in half of the projects and 27% of the overall builds. The least prominent anti-pattern is Skip Failed Test, which occurs in 42% of the projects and 0.65% of the builds.

Answer to RQ₄

Build logs and repositories contain valuable information to identify those anti-patterns that slowly creep into projects over time. Our tool leverages these resources to detect the presence of anti-patterns, which occur very frequently in the analyzed projects. Our reporting tool is perceived as useful and developers want to integrate it into their monitoring layer.

1.7 Scope and Limitations

In this section, we describe the scope of this dissertation and its main limitations.

1.7.1 Scope

The focus of this dissertation was on understanding what anti-patterns occur in the practice of CI and on effectively helping developers with the identification and removal of their causes. The catalog of anti-patterns that we derived from the analysis of StackOverflow posts and interviews with experts, was a step required to build the foundation of this thesis. The catalog is general and valid for every organization adopting CI, regardless of its domain and size. We identified common causes of anti-patterns to better guide the definition of approaches that help developers remove these causes. Our build summarization approach, which tackles the problem of having long stops caused by build failures, is designed for build servers that can be extended (for example, by means of plugins) to show our summaries in the build overview. Our semantic linter identifies configuration smells and can be used in organizations adopting build servers that can be configured through configuration files. This is possible in the majority of build servers, which are highly configurable and follow the infrastructure-as-code pattern [2]. Our monitoring tool for anti-patterns caused by the wrong usage of a build pipeline can be installed in those organizations that store build logs and repository information for a certain period of time (without rewriting the recent history at least).

Potential for Industrial Adoption Figure 1.3 illustrates the enhanced version of the basic development workflow shown in Figure 1.1. It contains the three approaches developed in this dissertation for the identification and removal of anti-pattern causes.

Developers have to avoid introducing configuration smells that provoke anti-patterns. When developers define the build pipeline for the first time or when they simply modify it, our linter (1) can be used to improve the effectiveness of the other tools that only spot syntactic errors. For example, our linter can remind developers to remove an anti-pattern that was temporarily added to avoid a build failure. Nevertheless, build failures play an important role in the build verification

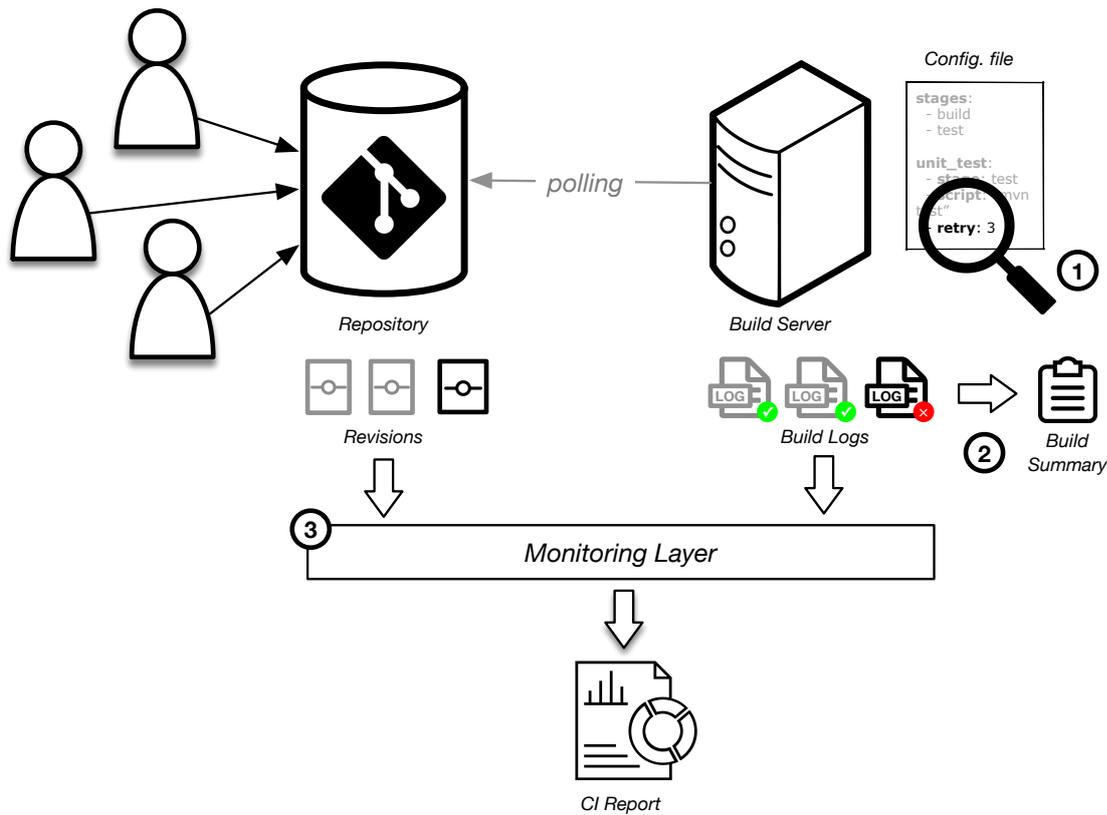


Figure 1.3: The enhanced CI-based development workflow

process because they spot the presence of defects that otherwise will be released in production. Build failures have to be solved in a short time to let other developers check out functioning code. Instead of inspecting huge build logs, developers can look at the build summaries produced by our tool (2) and understand the root causes of a failure more quickly. Several organizations adopt monitoring tools (e.g., Splunk) to measure throughput and stability of the released code over time. A monitoring layer (3) can be extended by including our automated reporting tool that assesses the quality of the development process in terms of anti-patterns accumulated over time, which have to be fixed on par with other smells that lower code quality. The generated report highlights the presence of worrisome trends and lets developers take action against CI decay. While they do not enhance directly the development workflow with new tools, the descriptions and examples

of anti-patterns contained in our catalog can be consulted by developers to learn about other important bad practices that are currently not detected by our tools.

1.7.2 Limitations

We present the main limitations of our dissertation. We motivate the reasons for some choices and assumptions that might be perceived as limiting factors. We also illustrate possible threats to validity, which will be discussed in the other chapters, together with our mitigation strategies.

Focus on Specific Technologies and Languages Although our approaches can be easily adapted and integrated into other build infrastructures, we chose a few technologies and programming languages to support. The reason of our choices was to maximize the number of projects that we could analyze and receive feedback from. We mainly supported projects written in Java because this has been the most popular language during recent years.¹⁴ If we consider the Java ecosystem, Maven is still the most used build tool.¹⁵ Thus, we summarized logs generated by Maven and detected anti-patterns in `pom.xml` files. As regards the build server, we supported Jenkins, TravisCI, and GitLab for several reasons. We decided to build a proof-of-concept implementation of our build summarization approach for Jenkins, not only because it is easy to extend through plugins but also because Jenkins is still the most popular build server.¹⁵ This enabled us to reach out to a bigger group of candidates for our controlled experiment on build summaries. We implemented our anti-pattern detectors for projects built on TravisCI and GitLab because those are very popular solutions among open-source developers, which made it easier to measure the presence of anti-patterns and communicate the results of our analysis. Finally, we chose SonarCloud for its seamless integration with other collaborative tools such as GitHub and the availability of past measurements, which made it possible to perform a historical analysis of code quality measurements.

Representativeness of Participants in Our Studies The number of participants in some of our studies is limited. The effort required to review a long catalog of

¹⁴<https://www.tiobe.com/tiobe-index/>

¹⁵<https://www.jetbrains.com/lp/devecosystem-2020>

anti-patterns or to participate in a 45-minute interview made it difficult to recruit a high number of participants. In the case of controlled experiments that need to be held in person, recruiting was even more problematic. Another source of difficulties was the availability of open-source developers to be contacted. Because we did not want to send unsolicited emails, we only targeted those projects that provide an official way (e.g., a public mailing list or group channel) to get in contact with their developers. Despite the impossibility of reaching a big number of participants based on the above-mentioned reasons, we always strove to build a diverse sample, including developers of different expertise levels and belonging to various organizational domains and sizes.

Assumptions about the Development Workflow We made several assumptions about the development workflow. When investigating the frequency of code quality checks, we assumed that projects use only one CCQ service to assess the overall code quality. Despite being unlikely based on the analyzed data, it might be that developers use multiple services with different settings and modalities of use. When assessing the usefulness of build summaries, we assumed that developers typically inspect raw build logs to understand a build failure. Our assumption was later confirmed by the analysis of build resolution workflows, where we actually saw that developers mostly rely on the sole build log to define a fix strategy.

Confirmation Bias in Surveys and Interviews We conducted interviews and surveys. Confirmation bias, which is the tendency of humans to confirm their preexisting beliefs, can affect both the construction of surveys or interview guidelines and the interpretation of the results. We tried to mitigate this threat by conducting pilot runs with other researchers to canvass them about the quality and neutrality of the questions. Instead of asking yes-or-no questions, we typically measured the perception of developers about our approaches through Likert scales [62] and employed as many open-ended questions as we could. When analyzing the results of open-ended questions, several people were involved in multiple sessions of card sorting.

Selection of Anti-patterns We did not implement detectors for all anti-patterns encountered by developers in practice. We rather focused on a subset of them based on two main criteria. One was the possibility to detect anti-patterns using information made available by open-source projects (e.g., configuration files, build logs, and repository). The other was the validity of our detection strategies for anti-patterns. While we cannot claim that we built the most relevant detectors, our survey among developers has shown the relevance of our selection. Furthermore, we used the feedback that we received from developers to further refine the detection strategies.

1.8 Scientific Implications

My thesis has both implications for practitioners interested in removing CI anti-patterns from their development workflows and researchers that want to improve the current practice of CI.

CI Is a Culture, not a Tool One of the main causes of anti-patterns is the missing adherence to fundamentals of CI. For example, developers do not automate their testing activities and they do not have a rollback strategy in place. Organizations seem to be mostly focused on introducing the newest build servers and integrate the most popular tools into their pipeline. However, tools are only means to assist developers in establishing a CI process. A build server cannot ensure that every developer commits changes at least once a day or that a build failure gets immediately fixed. Organizations need to invest more resources in coaching developers in CI principles to help them adopt CI correctly and adapt their development style to the practices that CI requires.

Summaries Enhance the Build Failure Overview In the presence of failures, most of the build servers display information about the commit that provoked the failure and the build duration. Developers have to inspect huge build logs to understand the root causes of a failure. Build summaries have the advantage of considering only those sections of the build logs that are relevant to the failure. In this way, they improve the understandability of build failures and reduce the time

needed to fix them. We argue that build servers should include our summaries as a supplement for build logs in the failure overview.

Semantic Linters as “Mentors” when Configuring Build Pipelines Semantic linters, which go beyond syntactic errors and highlight the effects of a configuration choice, can efficiently support developers when defining their build pipelines. For example, our linter warns developers against configurations that violate CI principles. Linters can mentor developers that approach the configuration of a build pipeline for the first time and have the advantage of being lightweight. They are fast and do not require special hardware (i.e., they can be run locally). Given that improper configurations mostly affect long configuration files, linters are even more needed to spot those issues that hide in the lines.

Monitor CI Processes to Prevent Decay We saw that developers typically lack awareness of anti-patterns that emerge over time and that lead to CI decay in the long term. Monitoring tools (e.g., Splunk), that are available on the market, assess only the performance of a build pipeline in terms of throughput and stability. We argue that those tools should be set to monitor also for the presence of anti-patterns such as aged branches. In addition to that, developers should adopt well-defined notification mechanisms (e.g., build failure notifications) that avoid following bad practices such as postponing the resolution of build failures.

Automated Reporting Tools Need to Earn Trust First Nowadays, there is an increasing use of software bots to assist developers during various phases of the development process [60]. For example, software bots close abandoned issues after a period of inactivity and alert to the presence of vulnerabilities in the code. In our evaluations, we opened *automatically* issues containing configurations smells and sent *automatically* generated reports to developers. Despite the presence of a real problem, several developers first closed issues or ignored our reports and they only showed interest when we started a conversation with them. Based on our experience, automated reporting tools should demonstrate the presence of humans behind them, otherwise, developers are less willing to accept their suggestions regardless of their relevance.

Do not Rely Blindly on Default Configurations As well-known static analysis tools, also our anti-pattern detectors generate false positives or are intrinsically imprecise. Despite performing very well both in terms of precision and recall, there are cases where our tools report anti-patterns that are not valid in a specific context (e.g., the manual execution of a deploy job does not violate principles). Those inaccuracies can be removed only by enabling developers to configure our tool and define exclusion criteria for our detectors. The default configuration and the use of predefined thresholds only capture the common case. It is important that developers configure tools to better adapt them to their contextual needs.

1.9 Opportunities and Future Work

This section describes possible extensions of our dissertation. Future works should address its limitations and broaden the scope.

Suggesting Fixes for Anti-patterns Fixing a detected anti-pattern is typically straightforward. Developers just have to unfollow the incorrect behavior. For example, they are required to better define dependencies with missing versions or merge a feature branch when it is kept open for a long period. However, in some cases, the solution is not immediate. When the build duration raises to dangerous levels, developers cannot simply make the build faster removing steps or part of the code. They need to deeply investigate bottlenecks and their causes. Future research should study how to help developers when removing an anti-pattern is not trivial.

Detecting more Anti-patterns Our approaches do not detect all anti-patterns that developers can encounter in practice. Future works should alert developers when they do not version all the necessary configurations and when a proper notification mechanism is missing. Duplicated build steps should be avoided because they make builds unnecessary slow and lower the build maintainability. We saw that Fuzzy Version is perceived as a bad configuration only when it involves libraries and dependencies. According to the received feedback, tools, that are executed during build steps, do not need a fully-specified version. They have to

be always updated to the latest version that can contain security improvements. However, having automatic updates threaten the build reproducibility. More research is needed to investigate what is a good trade-off between security and reproducibility.

Enhancing the Debugging of Build Failures Developers typically rely on the sole build logs to solve failures. However, we saw that, especially in the case of testing failures, developers also reproduce the failure and debug it. Unfortunately, this is always not possible because some builds require high-performing machines and are not meant to be executed locally. Researchers should study innovative approaches that bridge the gap between local (e.g., the IDE) and remote (e.g., build server) environments. Furthermore, the recent history of build fixes can help developers with identifying the root causes of build failures.

Tackling Infrastructure-related Build Failures Infrastructure-related build failures, such as the ones due to unreachable machines, are not expected in the standard development workflow. They are caused by weaknesses of the adopted build infrastructure rather than by defects introduced by developers. Future work should investigate how to extend our approach to effectively summarize also those failures. For example, new hint generators can leverage information from different nodes (e.g., servers) of the infrastructure to provide developers with the location of failures. Researchers should also verify the relation between infrastructure failures and bad configurations.

Using Contextual Information to Calibrate Thresholds We developed anti-pattern detectors that employ default thresholds. However, these thresholds are not applicable to all contexts and have to be adjusted based on the needs. Contextual information from different sources (e.g., bug trackers, task management systems, communication platforms) can be used to automatically calibrate thresholds.

1.10 Related Work

The works that are mainly related to this dissertation can be categorized into research papers that (i) investigate the adoption of CI and the difficulty of applying it, (ii) study the phenomenon of build failures and their solutions, (iii) identify defective infrastructure as code scripts. More details about the works that are closely related to the studies included in this dissertation will be described in the corresponding chapters.

Continuous Integration: Benefits, Anti-patterns, and Barriers Several researchers have studied the adoption of CI in open-source and industrial contexts. Hilton et al. [46] found that popular open-source projects are more likely to install build servers and CI is used because it helps developers to catch bugs early increasing the release frequency. Those expected benefits have been investigated by Vasilescu et al. [124]. CI not only helps developers to discover more bugs but also increases developer productivity making teams more effective at merging pull requests. The benefits originally formulated in books [25, 47] are then proven in practice. This strengthened our motivation for studying ways to better enforce a correct CI adoption. Despite its clear definition, Staahl and Bosch [112] found that industrial organizations have different interpretations of the CI practice, which might lead to very specific anti-patterns. Our goal was to build a catalog of anti-patterns that was general, so we made sure to involve companies varying in size and domain to improve the reliability of our results.

We defined anti-patterns as deviations from CI principles, which are instead those practices that developers should adopt to obtain expected benefits. CI principles have been defined in several books that describe not only CI [25] but also CD [47], and have been later collected in catalogs [26, 27]. For each principle, those books identified several risks that can derive from the incorrect application of CI. For example, Duvall et al. [25] advocated the need for a fully automated build process and a centralized dependency management system highlighting the possible risks of not being able to create deployable software or to solve transitive-dependencies-related issues. Humble and Farley [47] also provided suggestions on how to construct a build pipeline that enforces best practices. Differently from

these works, we empirically derived anti-patterns and validated how many bad practices, that were only hypothesized before, are encountered in practice.

One reason for introducing anti-patterns is the difficulty of living up to CI principles. Hilton et al. [45] investigated the barriers that developers encounter when adopting CI. Developers struggle with repairing build failures that require the inspection of complex logs and that happen on remote machines where developers have limited debugging capabilities. Furthermore, the build duration tends to increase as long as a project evolves. Long build durations as well as long build resolution activities reduce developer productivity. Introducing a build pipeline and automating an established development process are not trivial because organizations need to invest in dedicated personnel that not only set up a build server but also coach people in CI principles. The burden of adopting CI correctly is a strong motivation for supporting developers to remove the causes of anti-patterns that emerge because of the aforementioned reasons.

The Causes of Build Failures and Approaches for Their Automated Fix Previous works have investigated the causes of build failures. Miller [75] studied build failures occurring at Microsoft and found that are mainly due to poor code quality, testing failures, and compilation errors. Those results are also valid in open-source projects where build fails mostly because of failed test cases [10, 100]. Researchers have also produced taxonomies focusing on specific types of build failures, i.e., compilation [107] and code quality [142] related failures. In our dissertation, we propose a broad taxonomy considering all types of build failures that we could identify in the open-source and industrial projects that we studied. The frequency of build failure types that we measured in our dataset confirms the results obtained by the previous researchers. There is a cluster of build failures caused by the instability of build environments that is becoming more frequent [35]. We did focus only on build failures that are related to development activities, but we plan to extend our taxonomy to cover also those failures that were perceived as even more difficult to solve in our study on build resolution workflows.

Repairing build failures is a time-consuming task. Kerzazi et al. [50] observed a high percentage of build failures, whose resolution costs more than 2,000 man-hours over a 6-month period. On average, each failure resolution requires one

hour. Because of the effort required to fix them, several researchers have proposed approaches for the automated resolution of build failures. Macho et al. [66] developed an approach that automatically repairs build failures due to some types of dependency-related issues. Urli et al. [120] implemented a bot that repairs build failures caused by failing tests. Hassan and Wang [43] leveraged patterns extracted from previous build fixes to generate patches that repair those failures that are caused by changes to the build configuration file. Our approach for build failure summarization is complementary to automated fix approaches. Most of the failures cannot be solved automatically, so we try to empower the developer by improving build log understandability.

Bad Configurations in Infrastructure as Code Scripts Previous researchers have investigated the nature of bad configurations in infrastructure as code (IaC) scripts. Sharma et al. [108] derived a catalog of configuration smells that assess the quality of IaC scripts starting from best practices that are traditionally used to improve source code quality. Other researchers studied configuration smells [92], development anti-patterns [93], and source code properties [96] that are associated with defective IaC scripts, which might cause severe system outages.

Several approaches have been developed to automatically detect various kinds of bad configurations. Gallaba and McIntosh [34] developed a tool that detects and eliminates misuses such as the presence of unused properties and bypassed security checks. Rahman et al. [94] implemented a tool that automatically identifies seven types of security smells that can generate security vulnerabilities in the infrastructure. We share with these existing works the focus on detecting bad configurations. However, our goal is different. We want to detect those configuration smells that cause violations of CI principles.

1.11 Summary and Contributions

Developers can violate the principles of CI introducing anti-patterns that quickly solve urgent problems such as meeting the due date of a release. However, if not properly identified and removed, anti-patterns threaten the long-term goal of CI, which is delivering reliable and better software faster. We started this dissertation

investigating the existence of anti-patterns in practice. We built a catalog of anti-patterns encountered by developers and identified their main causes. Apart from the poor knowledge of CI fundamentals, such as the use of automated tests and a dependency management system, that can be addressed with better coaching in CI, developers require proper assistance to remove anti-patterns that are caused by the complexity of build failure logs and that emerge when defining or using a build pipeline. To assist developers, we built (i) an approach for summarizing build failure logs that increases their understandability and reduce the time spent on solving build failures, (ii) a semantic linter that detects the presence of anti-patterns in configuration files, and (iii) a monitoring tool that identifies anti-patterns that slowly creep into projects over time.

In summary, we made the following contributions:

- A catalog of CI anti-patterns encountered by developers in practice;
- A taxonomy of build failures that occur in open-source and industrial projects;
- An approach for summarizing build failure logs that increases their understandability;
- An approach for identifying anti-patterns in build configuration files;
- An approach for detecting anti-patterns that occur over time analyzing build logs and repository information.

1.12 Outline

The next chapters of this dissertation contain research works that were published at international peer-reviewed conferences and journals. As shown in Figure 1.4, those works answer the research questions previously illustrated in Section 1.2.

Chapter 2 investigates the existence of deviations from a core principle of CI, which is continuous inspection also known as continuous code quality (CCQ). This work was done in collaboration with my former colleague Fabio Palomba, Alberto Bacchelli, and my supervisor Harald C. Gall. I contributed to the study design

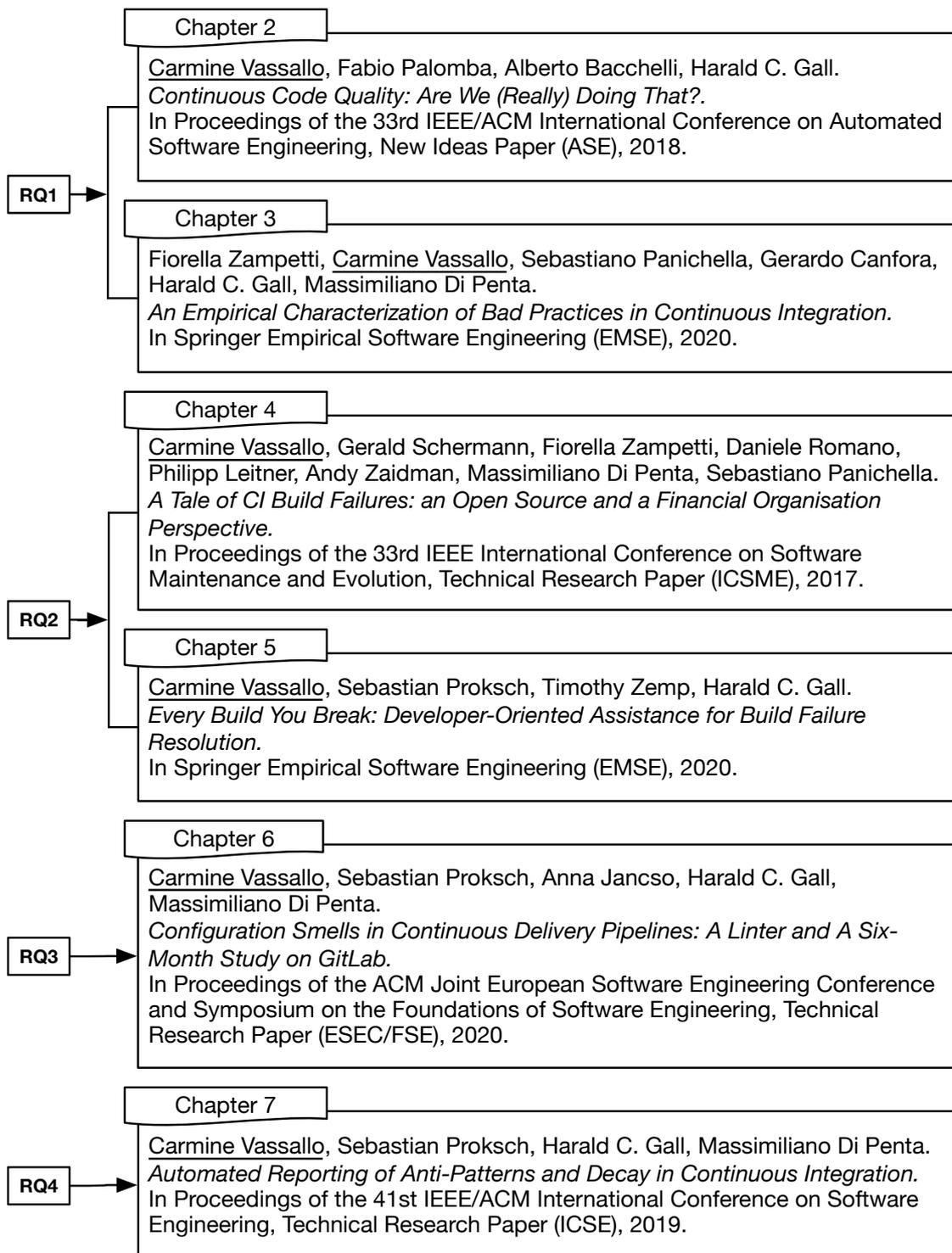


Figure 1.4: Structure of the next chapters of this dissertations and their relations with the formulated research questions

and paper writing. I collected build and code quality data, implemented the CCQ indicators, and analyzed the results.

Chapter 3 characterizes the anti-patterns encountered by developers when adopting CI. I collaborated with Fiorella Zampetti, Gerardo Canfora, and Massimiliano Di Penta from University of Sannio, Sebastiano Panichella from Zurich University of Applied Sciences, and my supervisor Harald C. Gall. I contributed to the study design, the paper writing, and I conducted the majority of the interviews (together with their transcription). The manual labeling of StackOverflow posts and interview transcripts, that we used to build the catalog of anti-patterns, was equally distributed between the first author and me.

Chapter 4 defines a taxonomy of build failures based on their root causes. My collaborators were Fiorella Zampetti and Massimiliano Di Penta from University of Sannio, Daniele Romano from ING Nederland, Andy Zaidman from Delft University of Technology, and my former colleagues Gerald Schermann, Philipp Leitner, and Sebastiano Panichella. I contributed to the study design, the collection of build logs, the manual inspection of error messages, the data analysis, and the paper writing. Note that part of the data collection (i.e., build logs from ING) was performed during my master thesis work.

Chapter 5 presents an approach for summarizing build failure logs. This work was done in collaboration with my former colleague Sebastian Proksch, a former bachelor student Timothy Zemp, and my supervisor Harald C. Gall. I contributed to data analysis and paper writing. I designed prototype, experiment, and survey.

Chapter 6 describes an approach for detecting configuration smells in build pipelines. I collaborated with Massimiliano Di Penta from University of Sannio, my former colleague Sebastian Proksch, Anna Jancso, and my supervisor Harald C. Gall. I contributed to the prototype and experimental design, the manual analysis of configuration smells, and the paper writing. I selected the input data of our study and analyzed the results.

Chapter 7 illustrates an approach for identifying CI decay. My collaborators were Massimiliano Di Penta from University of Sannio, my former colleague Sebastian Proksch, and my supervisor Harald C. Gall. I contributed to survey, experiment, and tool designs, and to the paper writing. I implemented the tool, collected data used as input of our study, and analyzed the results.

During my Ph.D., I also published other research works that are not included in this dissertation. As described in Figure 1.5, they are grouped into the following topics:

- *Configuration of Static Analysis Tools* We conducted a study where we investigated how people configure static analysis tools and the importance of considering contextual information to reduce false alarms. We also studied the relevance of static analysis tools in open-source projects.
- *Improvement of a CI process* In addition to helping developers with establishing CI, we analyzed possible directions for further improvement of the CI practice. This topic also includes a research work that was extended later (see Chapter 5).
- *Continuous Refactoring* We studied how and when developers perform refactoring in open-source projects. We then surveyed developers on the need for continuous refactoring (i.e., the practice of continuously looking for refactoring opportunities) in CI.

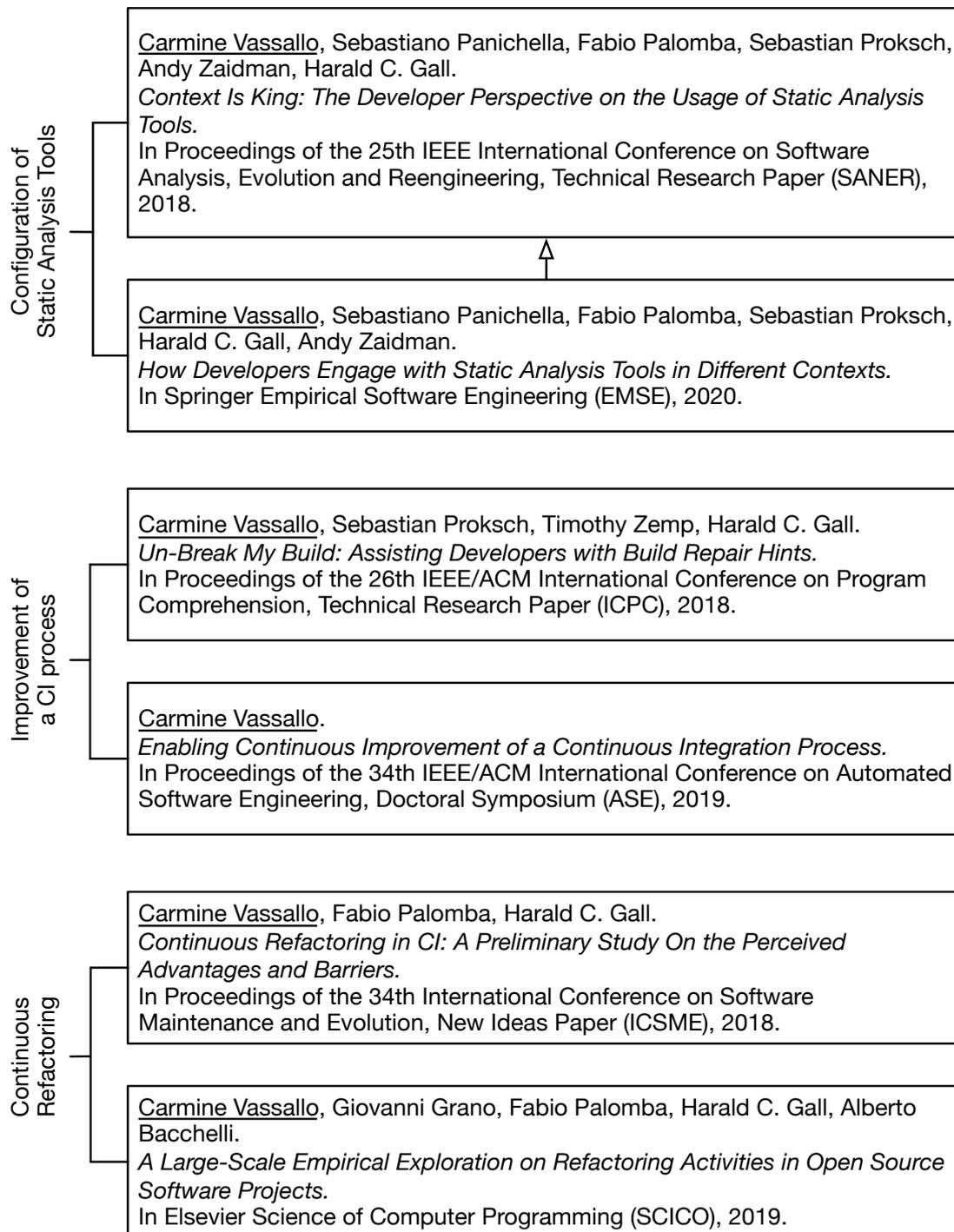


Figure 1.5: The other research works that are not included in this dissertation

2

Continuous Code Quality: Are We (Really) Doing That?

Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall
Published in Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, New Ideas Paper (ASE), 2018
Contribution: study design, data collection, implementation of CCQ indicators, data analysis, and paper writing

Abstract

Continuous Integration (CI) is a software engineering practice where developers constantly integrate their changes to a project through an automated build process. The goal of CI is to provide developers with prompt feedback on several quality dimensions after each change. Indeed, previous studies provided empirical evidence on a positive association between properly following CI principles and source code quality. A core principle behind CI is *Continuous Code Quality* (also known as CCQ, which includes automated testing and automated code inspection) may appear simple and effective, yet we know little about its practical adoption. In this paper, we propose a preliminary empirical investigation aimed at understanding how rigorously practitioners follow CCQ. Our study reveals a strong dichotomy between theory and practice: developers do not perform continuous inspection but rather control for quality only at the end of a sprint and most of the times only on the release branch.

2.1 Introduction

“Improving software quality and reducing risks” [25]. This is how Continuous Integration (CI) has been put forward by Duvall et al. [25] and is widely perceived by developers and students [59]. Concretely, CI is an agile software development process aimed at continuously integrating changes made by developers working on a shared repository; a build server that is used to build every commit, run all tests, and assess source code quality [47].

Duvall et al. [25] have proposed a set of principles that developers should methodically follow to adopt CI. For instance, CI users should build software as soon as a new change to the codebase is performed, instead of building software at certain scheduled times (e.g., nightly builds). A key principle of CI, as advocated by Duvall et al. [25], is *continuous inspection*, which includes running automated tests and performing static/dynamic analysis of the code at every build, as a way to ensure code quality. This aspect of CI is also known as *Continuous Code Quality* (CCQ).

Previous work provided evidence on the potential of CI in achieving its stated goals. Vasilescu et al. [124] quantitatively explored the effect of introducing CI on the quality of the pull request process, finding that it improves the number of processed pull requests. Khohm et al. [51] studied whether and how shifting toward a shorter release workflow (i.e., monthly releases) had an effect on the software quality of Firefox, reporting significant benefits. Others found evidence of reduced time-to-market associated with CI [135] and the possibility to catch software defects earlier [46].

However, empirical knowledge is still lacking on the actual practice of CCQ: How strictly do practitioners adopt CCQ? What are the effects resulting from practitioners’ approach to CCQ? To scientifically evaluate CCQ and its effects, as well as to help practitioners in their software quality efforts, one has to first understand and quantify current developers’ practices. In fact, an updated empirical knowledge on CCQ is paramount both to focus future research on the most relevant aspects of CCQ and on current problems in CI adoption, as well as to effectively guide the design of tools and processes. To this aim, we conduct a large-scale analysis that involves a total of 148,734 builds and 5 years of the development change

history of 119 Java projects mined by SonarCloud and TravisCI, two well-known providers of continuous code quality and continuous integration data, respectively. We study the adoption of continuous code quality by measuring metrics like the number of builds subject to quality checks and frequency of the measurements.

Our findings reveal that only 11% of the builds are subject to a code quality check and that practitioners do not apply CCQ, rather run monitoring tools just at the end of a sprint. Moreover, only 36% of branches are checked.

2.2 Background and Related Work

This section provides an overview of the principles behind continuous code quality as well as the related literature.

2.2.1 Continuous Code Quality

There are a few basic principles at the basis of continuous integration [25]. Besides maintaining a single source code repository, the idea behind CI is to automate the correct integration of code changes applied by developers as much as possible. This is normally obtained by having a dedicated build server responsible for taking all the new commits as input and automatically build, test, and deploy them. In addition, code quality assessment tools are used in order to control how much the performed change respects the qualitative standards of the organization. Thus, the principle of *continuous code quality* translates into having a development pipeline composed of a repository, a CI build server, and a CCQ Service. The developer commits a change to a repository (e.g., hosted on GitHub), triggering a new build on the CI build server (e.g., TravisCI). The server transfers the change to a different server (called *CCQ Service*) that is in charge of performing the quality analyses and reporting back the outcome to the CI build server. Based on its configuration, the CI build server decides on whether the build fails depending on the results of the CCQ service.

CI build server users can configure the build in a customized way, e.g., sending only specific builds or builds on specific branches to the CCQ service for inspection. This configuration allows users to depart from the continuous quality practice as

prescribed [25] and to follow a different strategy. The decision to depart from the prescribed CCQ practice is at the basis of our work, which is focused on a deeper understanding of the actual CCQ practices.

2.2.2 Related Work

In the last years, researchers have proposed a growing number of studies targeting CI practices [10, 134, 135], also thanks to the increasing availability of publicly available CI data [11].

Hilton et al. [46] employed a mixed-method approach to study the use of CI in open-source projects. They first mined the change history of 34,544 systems, finding that CI is already adopted by the most popular projects and that the overall percentage of projects using CI is growing fast. In the second place, the researchers surveyed 442 developers on the perceived benefits of CI. The main perceived advantage is that CI helps projects release more often.

Hilton et al. [45] proposed a qualitative study targeting the barriers developers face when using CI. The study comprised two surveys with 574 industrial developers, with the main findings presenting the trade-offs between (i) speed and certainty, (ii) information access and security, and (iii) configuration options and usability. The authors motivated the need for new methods and tools able to find a compromise between those perspectives. The results discussed so far were also confirmed by Laukkanen et al. [59] and Kim et al. [53], who reported on industrial experiences when using CI.

Complementing the studies mentioned above, our investigation aims at understanding how rigorously developers adopt CCQ.

Other researchers investigated the use of automated static analysis tools (know as ASATs) in CI. Specifically, Zampetti et al. [142] observed that a low number of builds fail because of warnings raised by ASATs, while Vassallo et al. [126] reported that developers configured static analysis tools only at the beginning of a project. Our study further elaborates on how developers use ASATs in CI, by exploring how they use them in order to perform CCQ.

2.3 Overview of the Research Methodology

As Duvall et al. stated in previous work [25], the time between discovery and fix of code quality issues can be significantly reduced by continuously inspecting the code. Thus, the application of the *continuous inspection* principle is stated to be crucial for fulfilling the main advantage of CI, i.e., “improving software quality and reducing risks” [25].

The *goal* of the study is to quantify the gap (if any) between the *continuous inspection* principle (also known as *continuous code quality*) and the actual practices applied by developers with the *purpose* of providing initial guidelines and tools for future research in the field of continuous integration. Thus, our investigation is structured around one research question: *how is CCQ applied to projects in CI?*

The *perspective* is of researchers and practitioners interested in understanding whether code quality assessment is performed continuously in CI.

In order to answer our research question and guide future research on CCQ practice, we first need to construct a dataset containing projects developed through a CCQ pipeline. The *context* of our study consists of such a dataset, which includes 119 projects selected as reported in Section 2.4.

Then, we devise a set of CCQ metrics for assessing the actual CCQ adoption (described in Section 2.5.1) and measure them over the history of the projects in our dataset (Section 2.5.2).

2.4 Continuous Code Quality Data Collection

To conduct our investigation, we need to study projects that not only use CI, but also: (i) adopt a CCQ pipeline, (ii) adopt a static analysis tool that stores the quality measurements performed over their history, and (iii) have CI-related events available, so that we can contextualize CCQ measurements in their evolution.

Since an already built dataset that fulfills our criteria is not available, we build our own. The definition of an ad-hoc data collection strategy is necessary because CCQ and CI events are stored on different servers and the alignment of the CCQ change history over the change history recording all the events occurred on the CI build server required the definition of heuristics to properly match the two sources.

In the next sections, we describe the procedure we follow to build the dataset, which is composed of three main steps such as (i) collecting data from the CCQ server, (ii) collecting data from the CI build server, and (iii) aligning the change history coming from the two sources.

2.4.1 Collecting CCQ Data

SonarCloud is a cloud service based on SonarQube¹ that continuously inspects code quality and detects bugs, vulnerabilities, and code smells. SonarQube is one of the most widely adopted code analysis tools in the context of CI. SonarQube is a SonarSource product that is adopted by more than 85,000 organizations and that support more than 20 languages—including the most popular ones according to the TIOBE index. SonarQube provides developers with its own rules and incorporates rules of other popular static and dynamic code analysis tools. As an example, SonarQube runs all the most popular code analysis tools (i.e., CheckStyle, PMD, Findbugs, Cobertura) by default on Java projects. Thus, the relevance of SonarQube in the context of CI motivates the decision to focus on systems using SonarCloud as CCQ service.

Overall, 14,152 projects are actively using SonarCloud, even though some of them are private and, thus, not accessible. We query SonarCloud using the available web APIs² and extract the list of all the open source projects that use the free analysis service, reaching 1,772 candidate systems.³

2.4.2 Collecting CI Data

Starting from the initial population of 1,772 candidate systems, we keep projects that use TravisCI as build server, as this ensures that the project actually adopts a CCQ practice. We select TravisCI as it provides the entire build history, as opposed to other build servers (e.g., Jenkins) where only the recent builds are typically stored [124].

Selecting projects using TravisCI as CI server and SonarCloud as CCQ service is not trivial. While TravisCI provides a direct and easy integration with SonarCloud,

¹<https://www.sonarqube.org> ²https://sonarcloud.io/web_api

³The complete list is available in our online appendix [125].

there is no explicit link between those two services, meaning that one cannot directly infer which projects use both services at the same time. Thus, we need to create such a link. Among the information available on SonarCloud, the projects report the URL referring to the source code repository; this URL provides us with an exploitable solution to identify the desired systems. In particular, TravisCI is used to build projects hosted on GitHub: therefore, we first consider all the projects available on SonarCloud that expose a GitHub URL. This step reduces the number of candidate projects to 439 (i.e., 25% of all SonarCloud systems). Subsequently, using the GitHub URL we query the TravisCI APIs⁴ and check if a certain URL is present on the platform: 390 projects match the selection criteria, i.e., SonarCloud systems that are on TravisCI. As a final step, we remove projects having less than 20 CCQ checks over their history.⁵ This filter is needed to avoid the analysis of projects that do not really integrate a CCQ service in their pipeline; in other words, we only consider projects that *actively* apply CCQ. At the end of this process, our dataset comprises 119 projects.

2.4.3 Overlaying CCQ and CI Information

Once the explicit link between SonarCloud and TravisCI is available, the final step of the data collection process is to overlay the separate change history information available in two sources. Also, in this case, there is no explicit way to link a data point available SonarCloud to one on TravisCI. We solve this as in the following. For each of the 148734 builds available on TravisCI we first collect (i) `build id`, (ii) `triggering commit` (i.e., commit message and id), (iii) `build status` (i.e., failed, errored, passed), (iv) `starting date`, and (v) `ending date`. Then, we use the `starting date` parameter of the build to identify the corresponding data point on SonarCloud.

Specifically, let $\mathbf{b}_i \in T_i$ be a build done on the branch br in the CI history T_i of the project i available on TravisCI, and let $\mathbf{m}_{ik} \in S_{ik}$ be a measurement of a certain metric k for project i on the branch br in the CCQ history H_{ik} available on

⁴<https://developer.travis-ci.com>

⁵The threshold of 20 is fixed in a similar way as done in previous work [20, 49, 70].

SonarCloud, we considered m_{ik} to be the measurement corresponding to b_i if the following relation held:

$$date(m_{ik}) \geq startingDate(b_i) \wedge date(m_{ik}) \leq startingDate(b_{i+1})$$

In other words, for each of the 119 considered projects, we compute the time interval in which two subsequent builds (i.e., b_i and b_{i+1}) are performed on TravisCI and assign a quality measurement to the build b_i if it was started within that time window. For each considered project, the final result is an *overlaid change history*, which contains information about the measured metric(s) and value(s), for each measured build (i.e., a build subject to a measurement on SonarCloud).

2.5 Continuous Code Quality in Practice

In this section, we discuss how continuous code quality is applied in the selected projects. Specifically, we first present the CCQ metrics that we conceive to automatically assess the CCQ practice. Then, we show how our projects perform against the CCQ metrics over their development's history.

2.5.1 Definition of CCQ Metrics

Our study aims at assessing the practical use of CCQ. Based on the constructed *overlaid change history* of the 119 subject projects, we devise four indicators for measuring the actual CCQ usage:

CQCR – Code Quality Checking Rate: Number of builds subject to a code quality check divided by the total number of builds.

EFC – Elapsed Frame between Checks: Average number of builds between two builds subject to a code quality check.

ETC – Elapsed Time between Checks: Average number of days between two builds subject to a code quality check.

Table 2.1: CCQ usage indicators applied to our projects

Project Set			CCQ Usage Indicators			
Feature	Level	# projects	CQCR	EFC	ETC	CB
Age	Low	30	0.14	7.72	14.49	0.62
	Medium	58	0.17	10.86	18.33	0.25
	High	30	0.06	39.08	16.49	0.33
Contribution	Low	27	0.14	8.99	16.67	0.39
	Medium	61	0.12	9.74	16.69	0.41
	High	30	0.05	36.76	17.33	0.22
Popularity	Low	30	0.14	9.42	15.63	0.49
	Medium	58	0.12	10.61	15.47	0.33
	High	30	0.06	37.34	20.20	0.27
Overall			0.11	18.30	16.91	0.36

CB – Percentage of Checked Branches: Number of branches containing at least one build subject to a code quality check divided by the number of total branches scheduled for build.

We design these CCQ usage indicators (based on the guidelines by Duvall et al. [26]) to understand how well CCQ is performed from different perspectives. CQCR is the basic metric that reveals the fraction of builds that are qualitatively measured during the history of a project, thus giving a view on the extent to which developers use to check builds in their projects. EFC and ETC measure the frequency of the quality checks in the considered projects, in terms of the average number of builds and days, respectively, that are waited before performing a new quality check. CB indicates if there are branches that are not checked at all: in this case, we want to measure whether there are branches that are more prone to be subject of qualitative checks.

2.5.2 On the Current Application of CCQ

Table 2.1 reports the results of our study aimed at investigating how CCQ is applied in practice. The table reports the overall values (row “Overall”) of each considered metric, i.e., *Code Quality Checking Rate (CQCR)*, *Elapsed Frame between Checks (EFC)*, *Elapsed Time between Checks (ETC)*, and *Percentage of Checked Branches*

(CB). Moreover, with the aim of deeper understanding whether the characteristics of the projects influence our observations, we also report the overall metric values when splitting the systems by age, contribution, and popularity.

We exploit the GitHub APIs⁶ to identify (i) the number of performed commits, (ii) the number of contributors, and (iii) the numbers of stars of a certain repository, respectively. For each considered perspective (i.e., age, contribution, and popularity), we split projects into three different subsets, i.e., *low*, *medium*, and *high*. Specifically, we calculate the first (Q_1) and the third (Q_3) quartile of the distribution representing the number of commits, contributors, and stars of the subject systems. Then, we classify them into the following categories: (i) *low* if they have a number of commits/contributors/stars n lower than Q_1 ; (ii) *medium* if $Q_1 \leq n < Q_3$, and (iii) *high* if n is higher than Q_3 . As shown in Table 2.1 (column “# projects”), we inadvertently achieved a good balance among the different subsets in terms of the number of contained projects.

Looking at the results, we can first observe that, overall, only 11% of the builds are qualitatively checked (CQCR value). This is a quite surprising result, because it clearly indicates that projects are **not** continuously inspected. In the lights of this finding, we can claim that the *continuous inspection* principle is generally not respected in practice.

When considering projects split by categories, i.e., *low*, *medium*, and *high* for age, contribution, and popularity, we can perceive a trend in the results. Young and medium-age projects exhibit higher values for CQCR with respect to the more mature projects, yet still have a pretty low percentage of monitored builds (14% and 17%, respectively). This finding seems to suggest that the application of CCQ becomes even harder when increasing the number of commits, and consequently the number of builds of a software project. We find that only 6% of the builds pass for a quality check in long-lived systems, while the percentage is 5% in case of an high number of contributors. This result triangulates the findings by Hilton et al. [46], revealing that developers are still not very familiar with all the CI principles and tend to not apply them properly. At the same time, it seems that community-related factors play a role in the application of CCQ. Indeed, our findings suggest that communities with a large number of contributors are less

⁶<https://docs.github.com/en/rest>

prone to apply CCQ: this is in line with previous work that showed how large communities generally have more coordination/communication issues, possibly resulting in technical pitfalls [14, 37, 114].

The most popular projects are generally more likely to use CI [46], however—according to our results—they do not apply CCQ properly. This is visible in Table 2.1, where we observe that only 6% of the builds of popular projects are qualitatively monitored. Conversely, low and medium-popular systems exhibit a higher number of measured builds.

The projects using CI do not continuously inspect the source code. Moreover, the percentage of qualitatively monitored builds is lower for systems with large numbers of commits and contributors.

Elapsed Frame between Checks (EFC) measures the average number of builds between two builds subject to a code quality check on the same branch. The overall result for EFC strengthens our initial findings on the lack of CCQ. On the average, developers perform a code quality check every 18 builds. This number still increases where taking into account the size of the projects. Indeed, systems with a high number of commits and contributors have an EFC score of 39 and 37, respectively. It is important to highlight that such projects have a higher number of builds with respect to small projects, and therefore might benefit more of a continuous check of code quality.

Looking at and *Elapsed Time between Checks* (ETC), we can confirm what we observe for the elapsed time between quality checks: developers do not perform a continuous code quality assessment, but rather they monitor the quality at time intervals of 17 days. This number is very close to the usual duration of a SCRUM Sprint [7], which is often used in the CI context [57]: thus, our findings suggest that likely the current practice merely consists of checking code quality at the end of a sprint. This observation holds when splitting projects based on their characteristics, as we confirm that quality checks are performed at fixed intervals.

Developers perform a code quality inspection after several builds (on average every 18 builds) and, most likely, at the end of a sprint.

As the last indicator, we compute the percentage of *Checked Branches* (CB). Table 2.1 shows a similar trend as for the other CCQ usage indicators. Also in this case, the higher the number of commits and contributors, the lower the percentage of branches that are subject to a quality check. This result confirms the possible role of community-related factors, as large communities tend to be more reluctant to apply CCQ.

Overall, only 36% of branches are checked, meaning that most of them are developed without a formal quality control.

A low percentage of branches follow CCQ.

2.6 Discussion and Future Work

Our results highlight a number of points to be further discussed, and in particular:

- **CCQ Is not Applied in Practice** A clear result of our study demonstrates a poor usage of continuous code quality, and that indeed only a very low number of builds (11%) are qualitatively monitored. This finding opens up a number of observations. In the first place, the low use of CCQ may be due to a general biased perception that developers have with respect to source code quality [12, 85]: code quality is not the top-priority for developers [28], who prefer not to improve the existing code for different reasons, including time pressure or laziness [117]. Most of the time developers and product managers do not consider a quality decrement enough to fail the build process, or they do not know how to properly set up quality gates [104]. Besides this, our study somehow confirms the findings reported by Hilton et al. [45], highlighting once again that developers face several barriers when adopting CI principles.

- **The Relevance of a Development Community** A key finding in our study reports that the size of a project plays a role in the adoption of continuous code quality. While projects having few developers perform a (slightly) higher percentage of code quality checks, systems with a larger community face more difficulties. This can be explained by the presence of community-related factors that might preclude an effective management of the development activities. Indeed, wrong communication and coordination within software communities have been not only largely associated to the emergence of socio-technical issues [24, 37, 63], but also related to continuous integration aspects. In particular, Kwan et al. [56] reported a strong negative impact of socio-technical congruence, i.e., a measure indicating the alignment between technical dependencies work relations among software developers, on build success. Our findings confirm the importance of studying such factors and how they influence technical aspects of software systems more deeply.
- **On the Size of Change History** According to our results, projects having a longer change history are less likely to apply CCQ. This may suggest that a possible co-factor influencing the lack of continuous code quality control falls in the difficulty of developers to switch toward such new continuous monitoring in case the project is already mature.

Our initial findings pave the way to further study that we plan to conduct in future work:

1. **On the Value of Continuous Code Quality** Despite previous work in the area of agile processes [51], there is still a lack of study empirically assessing the benefits deriving from the actual practice of code quality assessment in CI. We build a dataset of projects using both CI Server and CCQ Service (as explained in Section 2.2). Thus, compared to previous work [124, 142] we are able to analyze the decisions of developers (i.e., whether perform code quality or not) and the obtained measurements without rerunning the analysis on projects' snapshots that might cause several threats, such as the unavailability of the configuration file or the impossibility to build a snapshot [118]. As future work, we plan to measure the effectiveness of the actual CCQ practice in maintaining software quality.

2. **Key Scenarios in Continuous Code Quality** Given the fact that code quality is not continuously assessed in CI, we are interested in determining the circumstances (e.g., development tasks) where the use of CCQ should be particularly encouraged, as they can lead to significantly decrease the quality of source code. It might be that CCQ is particularly effective in certain scenarios compared to others.

3. **Code Quality Recommendation in CI** Slow builds are serious barriers faced by developers using CI [45]. Automated testing and code quality assurance tasks are possible causes in slowing down builds. Code quality tasks are usually postponed and scheduled in *nightly builds*, thus preventing CCQ to be performed. We aim at finding a good trade-off between scheduling code quality tasks at every new change and slowing down the build. Our vision is to predict which quality measurements perform before triggering a new build. Given the actual build context described in terms of several features (e.g., checked-out branch, type of development task, etc.), a recommender will automatically schedule a new code quality task enabling the proper warnings.

2.7 Threats to Validity

This section discusses possible threats that might have affected the validity of our observations.

We mined information from different sources and combined them using heuristics that were needed because of the lack of an explicit link between them. To infer projects using both SonarCloud and TravisCI we used their Github URL—exposed on the first platform—as a means for understanding whether they also use TravisCI as build server. This linking process can be considered safe, as the Github URL of a project is unique and, thus, there cannot be cases where the history of a project on SonarCloud was overlaid with the one of another project on TravisCI. As for the overlay of the change history information of the two platforms, we exploited the build and measurement dates to understand to which build a certain measurement referred to. Also, in this case, the linking procedure cannot produce false positives

because there are not cases in which different builds might have been performed between the dates considered.

As for the generalizability of the results, we conducted this study on a large dataset composed of 119 projects. We also made some precautions to take into account only projects that actively adopt CI and CCQ. We limited our study to Java projects since some of the exploited platforms (e.g., SonarCloud) mainly contained information on this type of systems. Replications aimed at targeting projects written in different programming languages as well as industrial ones would be desirable.

2.8 Conclusion

In this paper, we analyzed the current practice of Continuous Code Quality (CCQ). Our findings showed that the theoretical principles reported by Duvall et al. [25] are not followed in practice. We found that only 11% of the builds are subject to a quality control. More importantly, the current CCQ practice merely consists of checking code quality at the end of a sprint, thus basically ignoring the CCQ principle.

Based on the dataset that we built overlaying change history information coming from SonarCloud and TravisCI, we plan to investigate the impact of the current CCQ practice on the software quality and the circumstances where developers are particularly encouraged to check code quality more frequently. Our future research agenda includes also the definition of techniques for assisting developers during continuous monitoring of code quality.

Acknowledgements

Vassallo and Gall acknowledge the support of the Swiss National Science Foundation for the project “SURFMobileAppsData” (SNF Project No. 200021-166275). Bachelli and Palomba also gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

An Empirical Characterization of Bad Practices in Continuous Integration

Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, Massimiliano Di Penta

Published in Springer Empirical Software Engineering (EMSE), Volume 25, Pages 1095–1135, 2020

Contribution: study design, interviews (and their transcription), manual analysis of StackOverflow posts and interview transcripts, survey design, and paper writing

Abstract

Continuous Integration (CI) has been claimed to introduce several benefits in software development, including high software quality and reliability. However, recent work pointed out challenges, barriers and bad practices characterizing its adoption. This paper empirically investigates what are the bad practices experienced by developers applying CI. The investigation has been conducted by leveraging semi-structured interviews of 13 experts and mining more than 2,300 Stack Overflow posts. As a result, we compiled a catalog of 79 CI bad smells belonging to 7 categories related to different dimensions of a CI pipeline management and process. We have also investigated the perceived importance of the identified bad smells through a survey involving 26 professional developers, and discussed how the results of our study relate to existing knowledge about CI bad practices. Whilst some results, such as the poor usage of branches, confirm existing literature, the study

also highlights uncovered bad practices, e.g., related to static analysis tools or the abuse of shell scripts, and contradict knowledge from existing literature, e.g., about avoiding nightly builds. We discuss the implications of our catalog of CI bad smells for (i) practitioners, e.g., favor specific, portable tools over hacking, and do not ignore nor hide build failures, (ii) educators, e.g., teach CI culture, not just technology, and teach CI by providing examples of what not to do, and (iii) researchers, e.g., developing support for failure analysis, as well as automated CI bad smell detectors.

3.1 Introduction

Continuous Integration (CI) [6, 15] entails an automated build process on dedicated server machines, with the main purpose of detecting integration errors as early as possible [10, 25, 112]. In certain circumstances, the next step is Continuous Delivery (CD), in which code changes are also released to production in short cycles, i.e., daily or even hourly [47].

Industrial organizations that moved to CI reported huge benefits, such as significant improvements in productivity, customer satisfaction, and the ability to release high-quality products through rapid iterations [17]. The undisputed advantages of the CI process have motivated also many Open Source Software (OSS) contributors [46, 112] to adopt it, promoting CI as one of the most widely used software engineering practices [17, 46].

Despite its increasing adoption, the heavy use of automation in CI makes its introduction in established development contexts very challenging [17, 46]. For this reason, in recent work researchers investigated the barriers [45] and challenges [17] characterizing the migration to CI. They found that developers struggle with automating the build process as well as debugging build failures [45].

Also, once CI is in place, it might be wrongly applied, thus, limiting its effectiveness. Previous work [25, 26, 47, 103] highlights some bad practices in its exercise concerning commits frequency, management of built artifacts and overall build duration. In this context, Duvall [27], by surveying related work in literature, created a catalog featuring 50 patterns (and their corresponding anti-patterns) regarding several phases or relevant topics of the CI process.

In summary, previous work discussed the advantages of CI, outlined possible bad practices and identified barriers and challenges in moving from a traditional development process to CI. However, to the best of our knowledge, there is no prior investigation aimed at empirically analyzing what bad practices developers usually incur when setting and maintaining a CI pipeline.

This paper aims at empirically investigating what bad practices developers incur when using CI in their daily development activities. This is done through (i) semi-structured interviews with 13 practitioners directly involved in the management and use of CI pipelines of 6 medium/large companies, and (ii) manual analysis of over 2,300 Stack Overflow (SO) posts, all related to CI topics. By relying on such sources of information and using a card sorting approach [111], we derived a catalog of 79 CI bad smells organized into 7 categories spanning across the different dimensions of a CI pipeline management (e.g., Repository, Build Process Organization, Quality Assurance, Delivery). Afterward, we have assessed the perceived importance of the 79 bad smells through a survey, which involved 26 developers working in 21 different companies.

To document the identified CI bad smells, we provide traceability (in our dataset) between the catalog and the SO posts and interviews' sentences. For example, the CI bad smell "Pipeline related resources (e.g., configuration files, build script, test data) are not versioned" is originating from a SO post¹ where a user asked:

... Do you keep all your CI related files checked in with your project source? How would you usually structure your CI and build files?

whereas the CI smell "Quality gates are defined without developers considering only what dictated by the customer" was inferred from an interview where developers reported they were not satisfied by their static analysis, and mentioned that:

... in general when we use SonarQube we configure only those checks that the customers feel important.

While a catalog of CI patterns and corresponding anti-patterns exist [27], this paper aims at empirically inferring and validating bad practices through a

¹<https://stackoverflow.com/questions/1351755>

reproducible process, which has manifold advantages: (i) it allows us to investigate what is the current perception of CI bad practices (based on SO posts and interviews with experts), (ii) it allows others to replicate our process, and, possibly, to improve our catalog or even confute our results, and (iii) it allows us to perform an unbiased comparison between CI bad smells emerged from our study and what stated in Duvall’s catalog. To investigate (i) the correspondence or contradiction between what is known in literature and what we observed in the reality, and (ii) cases in which developers follow a bad practice because they aim at pursuing trade-offs between conflicting goals, we have analyzed and discussed the relationships between the CI bad smells we identified, and the pattern/anti-patterns of Duvall’s catalog [27]. Finally, the study shows that different bad smells (including those matching Duvall’s anti-patterns) have a different degree of perceived importance.

The paper is further organized as follows. Section 3.2 provides the context of our study by discussing the related literature. Section 3.3 defines the study, its research questions, and details the study methodology. Section 3.4 reports and discusses the study results, while Section 3.5 discusses the threats to the study validity. The study implications are outlined in Section 3.6, while Section 3.7 concludes the paper.

3.2 Related Work

This section discusses related work on CI and CD, going more in-depth on bad practices and barriers in their usage. Before discussing related work, we clarify the terminology used hereinafter:

- We use the term “bad smells” similarly to previous work to denote “symptoms of poor design or implementation choices” [30]. In our case, “CI bad smells” are symptoms of poor choices in the application and enactment of CI principles.
- We use the term “bad practice” when we generically discuss the bad application of CI principles, without referring to a specific problem.
- Finally, when referring to the catalog by Duvall [27], we use the terms “pattern”/“anti-pattern” to be consistent with the terminology used there.

However, in this paper, Duvall's anti-patterns and our CI bad smells have the same meaning.

3.2.1 Studies on Continuous Integration and Delivery Practice

Many researchers have studied the CI/CD practices adopted in industry and open source projects [22, 46, 112, 113, 124]. Hilton et al. [46] conducted an extensive study on the usage of CI infrastructure and found that CI is currently very popular in OSS. Ståhl and Bosch [113] pointed out that in industry there is not a uniform adoption of CI. More specifically, they have identified the presence of different variation points in the CI term usage. Similarly, we involved different companies, varying in size and domain, to guarantee diversity of respondents and reliability of our results. Other researchers have focused the attention on the impact of CI adoption on both code quality and developers' productivity. Vasilescu et al. [124] showed how CI practices improve developers productivity without negatively impacting the overall code quality. From a different perspective, Vassallo et al. [135] investigated, by surveying developers of a large financial organization, the adoption of the CI/CD pipeline during development activities, confirming what known from existing literature (e.g., the execution of automated tests to improve the quality of their product), or confuting them (e.g., the usage of refactoring activities during normal development).

While the studies mentioned above investigated CI practice and served as inception for our work (Section 3.3.2), our perspective is different i.e., identifying and categorizing specific CI problems into a catalog of CI bad smells.

Concerning CD practices, Chen [17] analyzed four years' CD adoption in a multi-billion-euro company and identified a list of challenges related to the CD adoption. Also, Chen identified six strategies to overcome those challenges such as (i) selling CD as a painkiller, (ii) starting with easy but important applications and (iii) visual CD pipeline skeleton. Savor et al. [103], by analyzing the adoption of Continuous Deployment in two industrial (Internet) companies, found that the CD adoption does not negatively impact developer productivity, even when the project increases in terms of size and complexity.

3.2.2 Continuous Integration Bad Practices and Barriers

Duvall et al. [25] identified the risks that can be encountered when using CI, e.g., lack of project visibility or the inability to create deployable software. Such risks highlighted the need for (i) a fully automated build process, (ii) a centralized dependencies management to reduce class-path and transitive dependencies' problems, (iii) running private builds and (iv) the existence of different target environments on which deploy candidate releases.

Hilton et al. [45] investigated what barriers developers face when moving to CI, involving different and orthogonal dimensions namely: assurance, security, and flexibility. For instance, they found that developers do not have the same access to the environment as when they debug locally and their productivity decreases when dealing with blocking build failures.

Humble and Farley [47], instead, set out the principles of software delivery and provided suggestions on how to construct a delivery pipeline (including the integration pipeline), by using proper tools, automating deployment and testing activities. Based on such principles, Olsson et al. [82] explored the barriers companies face when moving towards CD. More specifically, they identified as barriers: (i) the complexity of environments (in particular of network environments) where software is deployed; (ii) the need for shortening the internal verification to ensure a timely delivery; and (iii) the need for addressing the lack of transparency caused by an incomplete overview of the current status of development projects.

Our work shares with previous work the challenges encountered when applying CI, but also CD. However, the observation perspective and the stage in which the problems are observed are different. Indeed, we do not look at problems encountered in the transition to CI or CD, while we infer CI bad smells when CI is already being applied. However, in some cases, the consequences of the bad smells we inferred share commonalities with the barriers affecting the transition [82]. As an example, we foresee some CI bad smells concerning (i) deployment of artifacts that have been generated in a local environment, and (ii) deployment without a previous verification in a representative, production-like environment. Also, our catalog features a bad smell named “Authentication data is hardcoded (in clear) under VCS”, which considers the lack of an appropriate and secure authentication when

deploying from a CI server. In summary, our findings indicate that, even when organizations have performed a transition towards CI/CD, some issues previously identified as barriers still arise, and some more can occur.

Duvall defined a comprehensive set of 50 patterns and related anti-patterns regarding several phases or relevant topics in the CI/CD process [27]. In the catalog, it is possible to find bad habits concerning the versioned resources, the way developers use to trigger a new build, test scheduling policies, and the use of feature branches. Duvall also highlighted the need for a fully automated pipeline having a proper dependency management, a roll-back release strategy and pre-production environments where the release candidate should be tested. As we explained in the introduction, while it is not our intent to show whether our catalog is better or worse than the one by Duvall, we (i) define and apply an empirical methodology to derive CI bad smells, (ii) study the developers' perception of such bad smells, and (iii) finally, we discuss the differences between the two catalogs, as well as cases in which a bad smell present in both catalogs was considered as not particularly relevant by the study respondents.

Vassallo et al. [129] proposed CI-Odor, an approach that detects the presence of four CI anti-patterns (slow build, broken master, skip failed tests, and late merging) inspired by Duvall [27]. While they focused on the automated detection of on some Duvall's anti-patterns, our aim is to investigate what bad practices (beyond those by Duvall) are relevant for practitioners, also to develop further detection strategies.

The phenomenon of slow builds was also investigated by Ghaleb et al. [36], finding that long builds (e.g., exceeding the 10 minutes rule-of-thumb [25]) do not necessarily depend on the project size/complexity, but may also be related to build configuration issues, such as multiple (and failed) build attempts. To this extent, our catalog includes various kinds of build configuration issues that can result in such a side effect. Also, Abdalkareem et al. [1] propose to optimize the build process by determining, through a tool named CI-Skipper, which commits can be skipped. While this is not necessarily related to the presence of CI bad smells, it may be a mechanism useful to solve problems related to slow builds.

The work by Gallaba and McIntosh [34] investigates configuration smells for CI infrastructure scripts in Travis-CI (i.e., `.travis.yml` files), and proposes Hansel

and Gretel, two tools to identify and remove four types of anti-patterns in Travis-CI configuration scripts. Such anti-patterns are related to (i) redirecting scripts into interpreters, (ii) bypassing security checks, (iii) having unused properties in `.travis.yml` files, and (iv) having unrelated commands in build phases. Also in this case, our work is complementary to such specific detectors as it provides a comprehensive, empirically-derived catalog of CI bad smells along with developers' perception of such bad smells.

3.3 Empirical Study Definition and Planning

In the following, we define our study according to the Goal Question Metric (GQM) paradigm [4].

The goal of this study is to identify the bad smells developers incur when adopting CI and assess the perceived importance of such bad smells. The quality focus is the overall improvement of the CI process and its associated outcomes, e.g., improving developers' productivity and software reliability. The perspective is of researchers interested, in the short term, to compile a catalog of CI bad smells and use them for education and technology transfer purposes, and in the long term to develop monitoring systems aimed at automatically identifying CI bad smells, whenever this is possible. The context from which we have inferred the catalog of CI bad smells consists of six companies (where we interviewed 13 experts), and 2,322 discussions sampled from SO. To assess the perceived importance of the identified CI bad smells, we have surveyed 26 CI practitioners belonging to 21 companies, that are not involved in the previous phase of the study.

3.3.1 Research Questions

The study aims at addressing the following research questions:

RQ₁: *What are the bad practices encountered by practitioners when adopting CI?*

This research question addresses the main goal of the study, which is the empirical identification and categorization of CI bad practices. The output of this categorization is a catalog of CI bad smells, grouped into categories.

As explained in the introduction, while a catalog of CI patterns and related anti-patterns already exists [27], our goal is to infer bad practices from pieces of evidence, i.e., SO posts or semi-structured interviews.

RQ₂: *How relevant are the identified CI bad smells for developers working in CI?*

While in the previous research question we derive a catalog of CI bad smells by interviewing experts and analyzing SO discussions, it could be possible that different bad smells might have a different degree of perceived importance. By surveying developers, this research question aims at assessing the importance of the bad smells identified in **RQ₁**, and, therefore, at performing an external validation of the compiled catalog.

RQ₃: *How our pieces of evidence confirm/contradict/complement the existing CI pattern/anti-pattern catalog by Duvall [27]?*

This research question aims at comparing our catalog of CI bad smells with those from literature. This is done by performing a mapping between our catalog and the one by Duvall [27]. It is important to remark that it is not our goal to determine which catalog is better. Instead, we want to determine the extent to which the anti-patterns defined by Duvall [27] are reflected by problems occurring in real practice, and whether there are problems not considered by Duvall [27].

3.3.2 Study Methodology

Figure 3.1 shows the overall methodology we followed to create and validate our catalog of CI bad smells. The methodology comprises a set of steps to create the catalog and further steps to validate it. Also, the figure reports which steps produce results for our study research questions. Table 3.1, instead, provides essential information about data processed/collected during our interviews and SO mining.

Inception

As first step, we performed an inception phase to enrich our knowledge on the possible CI misuses and therefore be able to effectively conduct interviews and

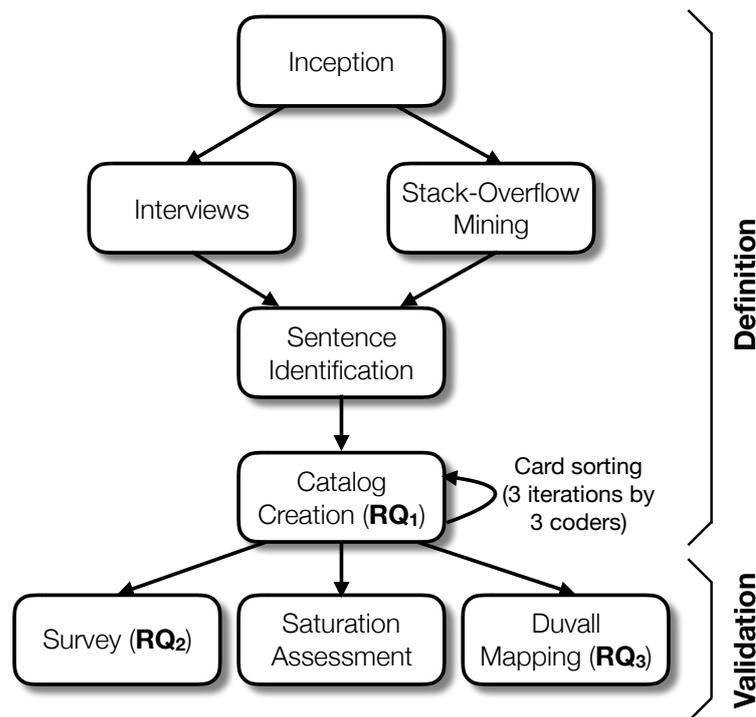


Figure 3.1: Process for CI bad smell catalog's creation and validation

mine online discussions. To this aim, we relied on well-known books, together with the related online resources: the Duvall et al. book on CI [25] and the book by Humble and Farley [47]. We intentionally did not use Duvall's catalog [27] in this phase, because we wanted to proceed bottom-up from the problems experienced by developers adopting CI. This allows us to perform an unbiased comparison of our catalog with the one by Duvall in a subsequent phase of the study.

In addition, we looked at previous research articles in the area: surveys about CI conducted in industrial context [112, 113], as well as studies focusing on specific CI phases [10, 68, 142].

Semi-structured Interviews

The goal of the semi-structured interviews was to understand and discuss with practitioners the problems they encountered when maintaining and using the whole CI pipeline in practice. We conducted interviews instead of a survey because (i)

Table 3.1: Summary of the study data used to create and validate the CI bad smell catalog

Interviews	
Interviewed Experts	13
Companies	6
Sentences	102
SO mining	
Queries	192
Retrieved Posts	4,645
Manual analysis	2,322
Posts found to be relevant	533

we expected that problems varied a lot depending on the context, and therefore it might have been worthwhile to deepen the discussion on specific CI areas; and (ii) interviews allowed us to discuss and capture specific situations the practitioners experienced doing CI activities. To conduct the interviews, we created an interview guide composed of the following sections:

- *Logistics*: consent and privacy/anonymization notices.
- *Demographics*: we asked the participant the: study degree, years of experience, programming languages used, role in the CI process, and company size/domain.
- *Characteristics of the CI pipeline adopted*: when CI was introduced, the number of projects adopting it, the typical pipeline structure and adoption of practices such as the use of branches and nightly builds.
- *CI bad practices encountered*: to guide the interview, we asked the main reasons inducing a restructuring of the CI process and also, when and how such maintenance activities have been performed. Finally, we asked the participants to tell us what are the typical issues they have experienced using the CI pipeline, in terms of symptoms and consequences.

The interviewed companies were identified based on the personal list of contacts of the authors. Thus, the interviewees were either our direct contacts, or we were forwarded to people specifically working on the CI pipeline. As shown at the top of Table 3.1, we identified in total 13 people from six different companies (in four cases we interviewed multiple people from the same company, working on rather

Table 3.2: Companies involved in the interviews on CI bad smells

Company	Domain	Size
#1	Software for Public Admin.	30
#2	IT consultant	100
#3	IT consultant for Financial Services	800
#4	Software for PA	9,000
#5	Telco	50,000
#6	IT consultant	100,000

Table 3.3: Experts interviewed on CI bad smells

Comp.	ID	Role	CI Intro (years)
#1	#1	Pipeline Configuration	2
	#2	Pipeline Configuration	4
#2	#3	Pipeline Configuration	—
#3	#4	Pipeline Configuration	—
	#5	DevOps	2
	#6	Design solution for Jenkins	2
	#7	Pipeline Configuration	—
	#8	DevOps	—
#4	#9	Pipeline Configuration	5
	#10	Pipeline Configuration	5
#5	#11	Pipeline Configuration & DevOps	5
#6	#12	Pipeline Configuration & DevOps	3
	#13	Pipeline Configuration & DevOps	> 5

different projects). The interviews were conducted either in person or using a video-conferencing system (Skype). In both cases, the audio of the interview was recorded. Each interview lasted between 30-45 minutes on average, depending on the participants' availability.

All the participants have more than eight years of experience in software development; all of them use Java; eight are also very familiar with JavaScript, and five are knowledgeable of Python. Table 3.2 summarizes the domain/size (number of employees) of each company, while Table 3.3 reports, for each interviewee, her role in the CI pipeline and also from when her company started to adopt the CI process. As shown in Table 3.2, the involved companies vary in terms of size and domain. More importantly, as shown in Table 3.3, the majority of our respondents

declared to be actually in charge of configuring/maintaining the whole CI pipeline, while only two respondents are simply using the pipeline as is. Finally, each company started to use CI two or more years ago (“–” indicates that the respondent did not precisely know when). As an outcome, we obtained a set of transcribed sentences from the taped interviews, referring to possible CI bad practices to use in the subsequent phase to create the catalog. The sentences report either the current practice, (e.g., “Our client dictates the quality gates. We have only to meet these gates”), a perceived bad practice (e.g., “Hardcoded configurations are a CI smell”) or, in some cases, both (e.g., “Monitor the global technical debt ratio, without paying attention to a single developer activity. It’s up to developers to decide when performing refactoring”). The complete set of collected sentences is available in our replication package [144].

Analysis of Stack Overflow Posts

As a first step, we needed to identify SO tags to use for retrieving candidate SO posts that could be relevant to our study. By scrutinizing the whole set of SO tags available,² we identified four CI-related tags, namely *continuous-integration*, *jenkins*, *hudson*, and *travis-ci*, used to better contextualize the discussions (i.e., issues/problems identification) to CI, since in some cases you could face the same problem in a different development practice. Then, we identified a total of 48 tags (a complete list is available in our online appendix [144]) that could relate to the specific activities and properties we were interested to investigate, such as Version Control Systems - VCS (e.g., *branching-strategy*, *version-control*), build (e.g., *batch-file*, *build-process*), testing (e.g., *acceptance-testing*, *automated-tests*), performance (e.g., *build-time*) and other related tags. In the end, we performed SO queries expecting at least one of the four CI-related tags, and one of the 48 specific tags (192 queries).

As a result, we downloaded 4,645 candidate, non-duplicated posts. We randomly divided such posts into two sets (ensuring proportions for each query), the first half used for inferring the catalog of CI bad smells, and the second half used for verifying the catalog saturation, as detailed in Section 3.3.2.

²We have queried the SO database in May 2017

Each selected post, belonging to the first half (in total 2,322 posts), was fully read by a tagger (one of the authors) to establish its relevance. The relevance was expressed tagging each post as “Yes”, “Maybe”, or “No”. In the presence of a potentially relevant post, the tagger had to report the relevant text extracted from it in a spreadsheet. Otherwise, the tagger wrote, if needed, a short justification for the lack of relevance (i.e., “No” tag). Tagging was performed based on the full content of the post, not just the title.

While it was unpractical (due to the size of the dataset) to afford multiple taggers per post, to minimize false positives, all the “Yes” or “Maybe” were independently re-analyzed by a different tagger. Posts with a “No” were discarded after an evaluator different from the tagger skimmed the annotation related to the lack of relevance. While a full analysis of posts tagged with one “No” could have reduced the number of false negatives, we preferred to reduce false positives instead, while keeping the number of posts to analyze reasonable enough. Out of the 2,322 manually-analyzed posts, 635 were labeled as “Yes” or “Maybe” by at least one tagger, with an agreement of 86.3%. While we promoted to “Yes” all posts on which the two taggers were in agreement (“Yes/Yes” or “Yes/Maybe”), we had to resolve, by means of a discussion (involving a further author that did not participate in the initial tagging), all the “Maybe/No” cases (128). Of such cases, only 26 were promoted to “Yes”. This resulted in a total of 533 posts to be used in the subsequent phases to create the catalog.

Identification of Sentences Reflecting Symptoms of CI Bad Practices

Two authors (hereinafter referred to as “coders”) analyzed the 533 SO posts identified in the previous phase, as well as the sentences transcribed from the interviews referring to possible symptoms of CI bad practices. The two coders used a shared spreadsheet to group sentences and to encode the specific problem/symptom instance using a short sentence. Upon adding the short sentence, each coder could either select — through a drop-down menu — one of the previously defined short descriptions, or add a new one. In other words, when performing the encoding, each coder could browse the already created short descriptions. If no description suited the specific case, the coder added a new short description in the list of possible ones, making it available for the upcoming annotations. As an example,

the coders used the sentence “Test too long in the commit stage” to highlight the usage of a build process that does not adhere to the fast feedback practice Duvall [27] or “Manual steps while triggering different stages in the pipeline” to point out the lack of full automation of a CI process.

As a result, the coders defined a total of 162 initial symptoms reflecting possible CI misuses, from the SO posts. The interviews’ transcripts contained a total of 102 sentences related to CI bad practices. The encoding of such sentences resulted in the identification of a total of 62 symptoms of CI misuses. Of such cases, 20 were not previously found in the SO posts. This ended up in a set of 182 initial symptoms reflecting CI bad practices.

Elicitation of the Catalog of CI Bad Smells

To address **RQ₁**, and therefore provide a systematic classification of bad smells faced by developers when applying CI, we performed card-sorting [111] starting from the symptoms identified in the previous phase. We relied on an online shared spreadsheet to perform the task.

The task was iteratively performed by three of the authors following three steps:

1. We discarded bad smells related to problems in the development process that do not have a direct impact on CI. For instance, we excluded generic test smells [121] or code smells [30].
2. We discarded symptoms reflecting possible CI misuses that we recognized to be “bugs”. As an example, we found some SO discussions in which developers discussed build failures due to the presence of bugs in third-party libraries included in the project. While the inclusion has a negative impact on the build process, this does not represent a CI misuse.
3. We merged related bad smells. For example, in some circumstances, a very similar bad practice was mentioned in two different CI activities, e.g., arbitrarily skipping a failing static analysis check or a failing test case.

The process was iterated three times, until no further changes were applied to the catalog. In the end, the final version of the catalog, discussed in details in Section 3.4.1 (RQ₁), features a total of 79 bad smells grouped into 7 categories.

Catalog Validation on Unseen Stack Overflow Posts

To verify the catalog saturation, i.e., its capability to cover bad smells not encountered in our manual analysis, we took the remaining set of $4,645 - 2,322 = 2,323$ SO posts and extracted a statistically significant sample of 330 posts. Such a sample size was chosen to ensure a significance level of 95% and a margin of error of $\pm 5\%$. Also, the 330 posts were sampled in proportion across the query tags. In other words, a stratified-random sampling was performed, where strata are represented by the set of posts returned by the different queries (i.e., tags). After that, two independent evaluators (two of the authors) selected the relevant posts and mapped them onto the 79 CI bad smells. Where the evaluators could not find a bad smell in the catalog matching the SO post, they added an annotation in the spreadsheet to be able to discuss these cases (8 cases in total). After the first round of independent classification, we found that the evaluators agreed in 76% of the cases on whether a post was related to CI bad smells or not. While this percentage seems high, it is still possible that they could have agreed by chance. Therefore, we computed the Cohen's k [18], which resulted to be 0.46 (moderate). After that, a third author identified the inconsistent classifications and discussed with the evaluators the reasons why this occurred, e.g., somebody classified as relevant SO posts that were seeking technical information (howto) about given pieces of technology. Such posts were not questioning an appropriate adoption practice, but, instead, seeking technical details, e.g., installation or usage instructions. After that, the two evaluators reworked again on the inconsistent cases. After the re-coding, the Cohen's k increased to 0.79 (substantial agreement), and the agreement rate to 91%. Also, we computed the agreement rate in terms of the kind of smell each evaluator associated to the post. In this case, we used the Krippendorff's α reliability coefficient [55] as the labeling was incomplete (i.e., for a post an evaluator could have indicated a bad smell, and another none). We obtained a reliability coefficient $\alpha = 0.67$, which is considered an acceptable agreement.

In the end, 131 out of 330 posts were classified as related to CI bad smells by at least one of the evaluators, out of which only 1 was not included in the previous version of the catalog. More specifically, one SO post pointed out cases where the build fails as soon as the first test case fails. This implies having a great

number of build failures without having a clear vision about the whole changes being implemented and pushed.

In summary, the results of the validation indicate that, with some exceptions, the identified catalog is general enough. However, this does not exclude that, in the future, further bad smells could emerge and be therefore included in the catalog.

Evaluating the Catalog through a Survey

To address **RQ₂**, we conducted a survey involving practitioners adopting CI in their organization. To ensure a good generalizability of the validation and to encourage participation, we adopted the snowball [38] sampling. That is, we shared the survey link to some contact points, and encouraged them to indicate us further participants, or people in the company better suited to participate in the survey. We followed this strategy because, while we had personal knowledge with a relatively limited set of contacts, snowballing helped to reach the relevant people (i.e., those involved in CI) and, in general, to favor participation. The online survey presented to the participants had:

1. An introduction explaining the meaning of our CI bad smells, as well as some basic terminology definitions for avoiding misunderstandings;
2. A demographic section similar to the one described in Section 3.3.2.
3. A set of 7 sections in which bad smells belonging to each category are evaluated.

We asked each respondent to evaluate the relevance of each bad smell over a 5-level Likert scale [83] (strongly agree, weakly agree, borderline, weakly disagree, strongly disagree), and we also gave the option to answer “Don’t know”. At the end of each section, we had an optional free comment field where the respondent could provide additional insights. The questionnaire has been administered through Survey Hero.³ The link has been sent to the people using an invitation email, in which we encouraged to spread the links to other CI experts. We kept the questionnaire open for four weeks. Nobody reported to have particular issues (e.g., privacy issues) with the used survey administration tool.

³<https://www.surveyhero.com>

After closing the survey, we obtained 26 responses from developers working in 21 different companies. Among all respondents, 15 were generic developers, while others covered different roles including project managers (3), solution/software architects (3), and researchers (5). All of them were involved in the CI process as maintainers and/or used the CI pipeline.

In the company/units where the respondents work, CI was introduced less than 5 years ago (7 cases), between 5-10 years (8 cases), and more than 10 years ago (12 cases). Note that this varied even within the same company for different units/projects. Most of them used Jenkins as CI automation infrastructure (17 cases), or GitLab (9 cases), while others used various kinds of infrastructures including Concourse, Bitbucket, or even in-house solutions. The build automation was performed mostly with Gradle (14 responses), Maven (10) and Ant (4),⁴ but also with various other tools such as cmake or npm.

Mapping onto Duvall's Anti-Patterns

To address **RQ₃**, we analyzed the overlap of our set of bad smells with those proposed by Duvall [27] to (i) investigate the exhaustiveness of our catalog; (ii) determine whether the empirically derived set of bad smells are not covered in Duvall's catalog, pointing out also more specialized cases of CI bad smells and/or outlining trade-offs. Note that we consider the last version of Duvall's catalog [27] and, in Section 3.4.3, we explain why some anti-patterns are out of the scope of this study.

The analysis was conducted by two authors independently (each author tried to create a mapping between Duvall's catalog and ours). When performing the mapping, we considered the possibility of assigning one Duvall's anti-pattern to multiple CI bad smells in our catalog or vice versa. In other words, the mapping is not one-to-one.

After the first round of annotation, the two authors agreed in the mapping with a Krippendorff $\alpha=0.65$, which is just below the minimum acceptability of $\alpha=0.66$ defined in the literature [55]. Similarly to what was done in the previous case (validation), a third author analyzed the disagreement cases and discussed

⁴The sum is > 26 as multiple build automation tools may be used.

them with the two annotators. As an example of inconsistent annotations, for the “Repository” pattern in Duvall’s catalog [27] the first annotator used the “Pipeline related resources are not versioned” while the second one mapped it onto the “Missing Artifact’s Repository” bad smell. With the help of a different author, and looking at the whole description of the pattern (and related anti-pattern) in Duvall’s catalog, the annotators converged on the fact that the anti-pattern is clearly highlighting the needs for having all the resources required to execute the build process under version control to avoid unnecessarily build failures. After that, the two annotators performed the mapping of the inconsistent cases again. The new mapping yielded an agreement rate of 80% and a Krippendorff $\alpha=0.84$.

3.4 Empirical Study Results

Table 3.4-Table 3.10 provide an overview of the 79 CI bad smells emerged from our empirical investigation. As explained in Section 3.3.2, we grouped them into 7 categories related to different dimensions of a CI pipeline management.

The third column (D) of the tables reports whether or not the bad smell maps onto at least one of Duvall’s anti-patterns [27] (RQ₃, Section 3.4.3). The fourth and fifth column report results related to the perception of CI bad smells (RQ₂, Section 3.4.2) and, specifically, the number of respondents that evaluated that specific bad smell (Resp.), and the evaluation results in form of asymmetric stacked bar charts. Note that each bar chart also reports three percentages: (i) respondents providing a disagree/strongly disagree answer, (ii) respondents providing a neutral answer, and (iii) respondents providing an agree/strongly agree answer.

3.4.1 Overview of the CI Bad Smell Catalog

In the following, we provide a general overview of the 7 categories of CI bad smells, without necessarily enumerating and describing all bad smells belonging to each category. For more detailed information, the complete catalog is available in our online appendix [144].

Repository groups bad smells concerning a poor repository organization, and misuse of version control system (VCS) in the context of CI (see Table 3.4). Some

Table 3.4: Results of CI bad smells' perception - Repository
 (*D* is the mapping with Duvall's anti-patterns, *RESP* is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results				
R1	Project decomposition in the repository does not follow modularization principles	✗	26	30%		39%		
R2	Test cases are not organized in folders based on their purposes	✗	26	48%		22%		
R3	Local and remote workspace are not aligned	✗	25	30%		57%		
R4	Number of branches do not fit the project needs/characteristics	✓	26	41%		27%		
R5	A stable release branch is missing	✗	25	45%		45%		
R6	Feature branches are used instead of feature toggles	✓	24	30%		45%		
R7	Divergent Branches	✓	26	38%		48%		
R8	Generated artifacts are versioned, while they should not	✓	25	50%		18%		
R9	Blobs are unnecessarily checked-in at every build instead of being cached	✗	25	41%		23%		
R10	Pipeline related resources are not versioned	✓	25	18%		64%		

Strongly disagree
 Weakly disagree
 Borderline
 Weakly agree
 Strongly agree

smells (R1–R3) deal with problems related to the repository structure which may affect the modularity of CI solutions (e.g., to build different modules of a software project separately). Moreover, a poor project decomposition into sub-modules (R1) might make parallel work on branches more difficult, but also prevent from having (some) faster builds limited to certain modules only. Then, there are bad smells about branch misuses (R4–R7), e.g., wrong choice between the use of feature branches and feature toggles (R6) or the use of an unbalanced number of branches that do not fit the project's characteristics (R4). Finally, some smells concern the poor choice of configuration items (R8–R10).

Infrastructure Choices groups bad smells related to a sub-optimal choice of hardware or software components while setting a CI pipeline (see Table 3.5). Hardware issues are mainly related to a poor allocation of the CI process across hardware nodes that could overload development machines or lose scalability (I1, I2). Software-related bad smells (I4–I7) concern poor tool choices and configuration,

Table 3.5: Results of CI bad smells’ perception - Infrastructure Choices
 (*D* is the mapping with Duvall’s anti-patterns, *RESP* is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results		
I1	Resources related to the same pipeline stage are distributed over several servers	✗	22	64%		14%
I2	The CI server hardware is used for different purposes other than running the CI framework	✗	25	48%		24%
I3	External tools are used with their default configurations	✗	25	28%		28%
I4	Different releases of tools/plugins versions are installed on the same server	✗	25	40%		48%
I5	Different plugins are used to perform the same task in the same build process	✗	25	48%		28%
I6	A task is implemented using an unsuitable tool/plugin	✓	25	40%		32%
I7	Use shell scripts for a task for which there is a suitable plugin available	✗	25	44%		20%

e.g., use of inadequate plugins for certain tasks (I6), abuse of ad-hoc shell scripts (I7), as well as the use of external plugins with default configurations not suitable to the specific development scenario (I3). Indeed, each tool has to be configured according to the (i) developers’ needs, (ii) final product requirements, and (iii) policies adopted by the organization.

Build Process Organization. This category, the one with the largest number of bad smells (29), features CI bad smells related to a poor configuration of the whole CI pipeline, as detailed in Table 3.6. Some of such bad smells are related to the CI environment’s initialization. Specifically, inappropriate clean-up strategies (BP1) could have been chosen. This, on the one hand (aggressive clean-up), may unnecessarily slow-down the build, and, on the other hand (lack of clean-up where needed), would make the build less effective to reveal problems.

There are two CI bad smells dealing with cases in which either monolithic builds are used where they should not (BP4), and where there is a poor decomposition of build jobs (BP3), e.g., including several activities in one single job or duplicating the same activities in multiple different jobs.

Table 3.6: Results of CI bad smells' perception - Build Process Organization
(D is the mapping with Duvall's anti-patterns, RESP is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results	
BP1	Inappropriate build environment clean-up strategy	✓	25	28%	40%
BP2	Missing Package Management	✗	24	25%	58%
BP3	Wide and inchoesive build jobs are used	✓	25	44%	40%
BP4	Monolithic builds are used in the pipeline	✗	25	28%	56%
BP5	Independent build jobs are not executed in parallel	✓	25	48%	36%
BP6	Only the last commit is built, aborting obsolete and queued builds	✓	25	44%	24%
BP7	Build steps are not properly ordered	✗	25	40%	48%
BP8	Pipeline steps/stages are skipped arbitrarily	✗	24	46%	46%
BP9	Tasks are not properly distributed among different build stages	✓	23	48%	35%
BP10	Incremental builds are used while never building the whole project from scratch	✗	25	36%	36%
BP11	Poor build triggering strategy	✓	24	33%	50%
BP12	Private builds are not used	✗	22	32%	36%
BP13	Some pipeline's tasks are started manually	✓	25	44%	44%
BP14	Use of nightly builds	✓	25	36%	48%
BP15	Inactive projects are being polled	✗	23	52%	17%
BP16	A build is succeeded when a task is failed or an error is thrown	✓	25	28%	64%
BP17	A build fails because of some flakiness in the execution, whereas it should not	✗	24	12%	67%
BP18	Dependency management is not used	✓	25	40%	44%
BP19	Including unneeded dependencies	✗	25	36%	44%
BP20	Some tasks are executed without clearly reporting their results in the build output	✗	25	24%	56%
BP21	The output of different build tasks are mixed in the build output	✗	23	17%	48%
BP22	Failures notifications are only sent to teams/developers that explicitly subscribed	✓	25	44%	40%
BP23	Missing notification mechanism	✓	25	28%	56%
BP24	Build reports contain verbose, irrelevant information	✗	25	24%	56%
BP25	Time-out is not properly configured	✗	25	36%	52%
BP26	Unneeded tasks are scheduled in the build process	✗	25	28%	44%
BP27	Build time for the commit stage overcomes the 10-minutes rule	✓	23	22%	52%
BP28	Unnecessary re-build steps are performed	✗	25	20%	40%
BP29	Authentication data is hardcoded (in clear) under VCS	✓	25	32%	52%

From a different perspective, looking at the build execution, our catalog includes CI bad smells related to the lack of parallelization while executing independent build jobs/tasks (BP5), the skip of certain phases/tasks just to make a previously failing build passing (BP8), or the use of an unsuitable ordering of build phases/tasks (BP7), e.g., a phase failing often such as static analysis checks follows a long and expensive test phase, making the latter worthless when the build fails.

Another key issue in configuring builds is related to the triggering strategy that may lead to some bad smells (BP10–BP15), e.g., related to builds started manually, or to the abuse of nightly builds, useful in certain circumstances, e.g., to run computationally-expensive tasks, but otherwise defeating the purposes of CI.

An inappropriate setting of the build outcome, e.g., succeeding a build when a task is failed (BP16), is also considered a bad smell. The same applies when the outcome of a build depends on some flakiness in the execution (BP17). Although flakiness has been extensively studied in some specific context, e.g., for testing [8, 65, 86, 115], we focus on the extent to which different kinds of flakiness (related to testing, but also to the temporary unavailability of some online resources) affect the build outcome on a CI server. Furthermore, the build output needs to be properly configured (BP20–BP24). Here the bad smells are related to inadequate observability and low readability of logs/notifications.

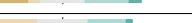
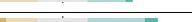
Inadequate/wrong dependency management is also felt like a very important problem (BP18, BP19). Previous studies reported how the majority of build failures is due to these kinds of problems [50].

There are some bad smells dealing with those cases in which the builds result to be particularly long such as the presence of unnecessary tasks (BP26), or unnecessary rebuilds (BP28).

Finally, although we did not focus on security-related issues, as there is a specific work by Rahman et al. [94], we got interview responses as well as SO discussions related to the presence of security-sensitive data (e.g., authentication information) hard-coded in the VCS (BP29).

Build Maintainability. Since build configuration files often change over time and their changes induce more relative churn than source code changes [72], their maintainability is also an important concern. As reported in Table 3.7, problems can arise when a build configuration is coupled with a specific workspace (see BM1–

Table 3.7: Results of CI bad smells' perception - Build Maintainability
 (*D* is the mapping with Duvall's anti-patterns, RESP is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results			
BM1	Absolute/machine-dependent paths are used	✗	26	31%		65%	
BM2	Build scripts are highly dependent upon the IDE	✓	26	27%		58%	
BM3	Environment variables are not used at all	✓	26	35%		31%	
BM4	Build configurations are cloned in different environments	✓	24	50%		29%	
BM5	Build jobs are not parametrized	✗	25	36%		36%	
BM6	Lengthy build scripts	✗	25	40%		28%	
BM7	Missing smoke test, set of tests to verify the testability of the build	✓	25	24%		44%	
BM8	Missing/Poor strict naming convention for build jobs	✗	26	15%		46%	

BM3, e.g., environment variables are not used when they should), or when the build script is poorly commented, uses meaningless variable names, and modularity is not used when it should be. While a recent work by Gallaba and McIntosh [34] deals with specific problems related to the maintainability of Travis-CI `.travis.yml` files, our bad smells are technology-independent and cover problems that can occur in all scripts involved in a build pipeline.

Quality Assurance. This category relates to CI bad smells that are linkable to testing and static analysis phases (see Table 3.8). Bad smells related to testing are due to the lack of optimization for testing tasks within a CI pipeline:⁵ for example a branch is not tested before merging it (Q5), or all permutations of features toggles are tested (Q6) and, as a consequence, the build gets slow or fails without a reason.

Moreover, this category features smells related to how test coverage thresholds (resulting in build failures, when not reached) are set (Q3, Q4), or the lack of a clear separation between test suites related to different testing activities (Q10) [68, 134]. Our catalog also includes problems related to (i) the use of production resources

⁵This is beyond test suite optimization, which is an important problem in testing, but out of scope for this investigation.

Table 3.8: Results of CI bad smells’ perception - Quality Assurance
 (D is the mapping with Duvall’s anti-patterns, RESP is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results			
Q1	Lack of testing in a production-like environment	✓	25	8%	17%		75%
Q2	Code coverage tools are run only while performing testing different from unit and integration	✗	23	30%	35%		35%
Q3	Coverage thresholds are fixed on what reached in previous builds	✓	23	35%	35%		30%
Q4	Coverage thresholds are too high	✓	24	46%	29%		25%
Q5	Missing tests on feature branches	✗	26	31%	19%		50%
Q6	All permutations of feature toggles are tested	✗	19	32%	32%		37%
Q7	Production resources are used for testing purposes	✓	25	36%	12%		52%
Q8	Testing is not fully automated leading to a non-reproducible build	✓	26	27%	8%		65%
Q9	Test suite contains flaky tests	✗	26	19%	12%		69%
Q10	Bad choice on the subset of test cases to run on the CI server	✗	25	28%	32%		40%
Q11	Failed tests are re-executed in the same build	✗	25	52%	28%		20%
Q12	Quality gates are defined without developers considering only what dictated by the customer	✗	24	29%	29%		42%
Q13	Use quality gates in order to monitor the activity of specific developers	✗	25	36%	12%		52%
Q14	Unnecessary static analysis checks are included in the build process	✗	26	38%	19%		42%

during testing operations (Q7), (ii) the test suite not being fully automated provoking a non-reproducible build (Q8), and (iii) the presence of flaky tests that may unnecessarily fail the build (Q9). Note that we have excluded from our analysis the presence of test smells [121], since they can negatively impact the understandability and maintainability of the product under development independently from the use of CI.

Bad smells related to static analysis are mostly due to how tools are configured and used within a CI pipeline. More in detail, they include failing to select

appropriate checks given the project characteristics (Q14), or setting quality gates not representative of what is relevant for developers and/or customers (Q12).

Table 3.9: Results of CI bad smells' perception - Delivery Process
(D is the mapping with Duvall's anti-patterns, RESP is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results
D1	Artifacts locally generated are deployed	✗	26	35%  58%
D2	Missing Artifacts' repository	✓	26	33%  46%
D3	Missing rollback strategy	✓	26	15%  62%
D4	Release tag strategy is missing	✗	26	27%  50%
D5	Missing check for deliverables	✗	24	25%  62%

Delivery Process. This category of bad smells concerns the storage of artifacts related to a project release. As reported in Table 3.9, they are related to poor/lack of usage of artifact repositories giving the possibility of rollback of deployed artifacts (D2, D3), or to the adoption of bad deployment strategies (D1), e.g., deployment of locally-generated artifacts.

Furthermore, this category includes bad smells related to software release in the production environment. These are related to not using a strategy aimed to validate the produced deliverables/artifacts taking part in a release (D5), or missing a clear and well-defined tagging convention for artifacts related to specific releases (D4).

Culture. We found bad smells whose symptoms might not be inferred by observing the CI pipeline, but are more human-related (see Table 3.10). They deal with the lack of a shared culture on how developers should behave when adopting CI. These include (i) bad push/pull practices (C1, C2), e.g., pushing changes before a previous build failure is being fixed; (ii) poor prioritization of CI-related activities (C5, C6), including the fixing of build failures, and (iii) Dev/Ops separation (C3, C4), i.e., developers and operators roles are kept separate, which is against the DevOps practice and, among other negative effects, increases the burden when fixing build failures affecting different stages of a CI pipeline.

Table 3.10: Results of CI bad smells' perception - Culture
(D is the mapping with Duvall's anti-patterns, RESP is the number of respondents that evaluated the smell, and the asymmetric stacked bar chart is presented with the percentages of strongly disagree/disagree answers, neutral answers, and agree/strongly agree answers.)

ID	CI Bad Smell	D	Resp.	Survey Results			
C1	Changes are pulled before fixing a previous build failure	✗	23	30%		43%	
C2	Team meeting/discussion is performed just before pushing on the master branch	✗	22	32%		36%	
C3	Developers and Operators are kept as separate roles	✓	25	24%		44%	
C4	Developers do not have a complete control of the environment	✓	26	19%		46%	
C5	Build failures are not fixed immediately giving priority to other changes	✓	26	12%		58%	
C6	Issue notifications are ignored	✓	25	16%		64%	

3.4.2 Perceived Importance of CI Bad Smells

In the following, we describe examples of CI bad smells perceived as very relevant, or, on the contrary, not particularly relevant by the respondents to our survey. As previously explained, a summary of the perceived relevance is depicted as asymmetric stacked bar charts on the right-side of Table 3.4-Table 3.10.

Overall, our results indicate that:

1. 26 bad smells received a strongly agree/agree assessment by over 50% of the respondents (i.e., the third percentage in the asymmetric stacked bar charts is greater than 50%). Such bad smells are listed in Table 3.11.
2. 26 bad smells had more strongly agree/agree (but $\leq 50\%$) than strongly disagree/disagree assessments.
3. 14 bad smells had more strongly disagree/disagree (but $\leq 50\%$) than strongly agree/agree assessments.
4. 7 bad smells received a strongly disagree/disagree assessment by over 50% of the respondents (i.e., the first percentage in the asymmetric stacked bar charts is greater than 50%). Such bad smells are listed in Table 3.12.

5. 5 bad smells received an equal number of agreement and disagreement assessments.

Note that the above grouping merely serves to facilitate the discussion of relevant/less relevant bad smells, and that, depending on the use one wants to make of the catalog, it could be possible to group (or even rank) bad smells differently.

As regards the **Repository** organization (see Table 3.4), out of 10 CI bad smells, 3 were considered relevant by the majority of respondents, and other 3 received more positive than negative assessments. Respondents found particularly relevant the lack of alignment between the local (developers') workspace and the CI server workspace (R3), but also the lack of a stable release branch (R5). This was also remarked by one of our interviewees: *“Production builds have to be “shiny”, the development ones can be “cloudy”. The last build in production must not be a failed build.”*. If there is a stable release branch, the development team always has a software product ready to be released.

Moreover, the survey respondents felt the need for versioning all pipeline-related resources (e.g., configuration files, build scripts, test data) as highly relevant (R10). At the same time, our respondents gave relatively negative importance to CI bad smells going into the opposite direction, i.e., those discouraging the versioning of binary large objects (R9) for performance reasons), or of previously generated artifacts (R8). While the lack of versioning for all needed resources makes impossible the execution of the build process, the presence of previously-generated artifacts could make the build unreproducible, and it could lead to a release of a software product that does not reflect the last changes being applied.

Most of the bad smells belonging to the **Infrastructure Choices**, see Table 3.5, received more negative than positive assessments. While the original book on CI by Duvall et al. [25] stressed this aspect, nowadays the availability of adequate hardware to support a CI server is generally not an issue, unless one has to deal with very specific contexts, such as cyber-physical systems for which the pipeline requires to be connected with hardware-in-the-loop or simulators. We would have expected positive feedback about bad smells related to software infrastructure choices, but this was not usually the case. In particular, the distribution of resources related to the same pipeline stage over multiple servers (I1) was not considered a problem, likely because, at least for some pipeline stages (i.e., the ones

Table 3.11: CI bad smells considered relevant by the majority of respondents.

Category	Smell
Repository	Local and remote workspace are misaligned (R3)
	A stable release branch is missing (R5)
	Pipeline related resources are not versioned (R10)
Build Process Organization	Missing package management (BP2)
	Monolithic builds are used in the pipeline (BP4)
	Build steps are not properly ordered (BP7)
	A build is succeeded when a task is failed or an error is thrown (BP16)
	A build fails because of some flakiness in the execution, whereas it should not (BP17)
	Some tasks are executed without clearly reporting their results in the build output (BP20)
	Missing a build notification mechanism (BP23)
	Build reports contain verbose, irrelevant information (BP24)
	Time-out is not properly configured (BP25)
Build time for the “commit stage” overcomes the 10-minutes rule (BP27)	
Build Maintainability	Authentication data is hardcoded (in clear) under VCS (BP29)
	Absolute/machine-dependent paths are used (BM1)
Quality Assurance	Build scripts are highly dependent upon the IDE (BM2)
	Lack of testing in a production-like environment (Q1)
	Production resources are used for testing purposes (Q7)
	Testing is not fully automated leading to a non-reproducible build (Q8)
	Test suite contains flaky tests (Q9)
Delivery Process	Use quality gates in order to monitor the activity of specific developers (Q13)
	Artifacts locally generated are deployed (D1)
Culture	Missing rollback strategy (D3)
	Build failures are not fixed immediately giving priority to other changes (C5)
	Issue notifications are ignored (C6)

Table 3.12: CI bad smells considered as not relevant by the majority of respondents.

Category	Smell
Infrastructure Choices	Resources related to the same pipeline stage are distributed over several servers (I1) Different plugins are used to perform the same task in the same build process (I5)
Build Process Organization	Independent build jobs are not executed in parallel (BP5) Only the last commit is built, aborting obsolete and queued builds (BP6) Tasks are not properly distributed among different build stages (BP9) Inactive projects are being polled (BP15)
Quality Assurance	Failed tests are re-executed in the same build (Q11)

different from the commit stage), the overhead due to the download of required resources from different servers may still be acceptable. Only one bad smell received more positive than negative assessments. Such a bad smell is related to the use of multiple plugin versions (I4), possibly causing conflicts or inconsistencies. One SO post, while identifying the root cause of a build failure, remarked that “*When there are multiple version of [a tool] installed side by side in Build Server, ensure right version of [the tool] is used by Build Server to execute the unit test*”. Truly, an inconsistency may not necessarily result in a build failure, but may produce slightly different outputs that might confuse developers or even fail external tools consuming such outputs. The usage of default configurations in build scripts (I3) received an equal proportion of positive and negative assessments. While previous research has highly motivated the need for properly configuring tools used in the build (e.g., static analysis tools, see the work of Zampetti et al. [142]), it was also found that developers rarely pay attention to that [9].

As expected, we received positive feedback by the majority of respondents for several (11 out of 29) bad smells belonging to the **Build Process Organization**, i.e., on how all steps of a build are configured through build automation scripts, as detailed in Table 3.6. In this context, a relevant bad smell is the lack of a proper package management mechanism (BP2) in the build automation. Specifically, if the CI pipeline does not include a package manager, developers might wrongly assume the presence of resources no longer available or use an outdated version of

them. Furthermore, it becomes difficult to manage tools included in the pipeline, in terms of installing, upgrading, configuring, and removing them. By using a package manager it is possible to specify when checking for updates, thus avoiding to download all dependencies at each build. The CI bad smell discussed above originates from an interviewee who clearly stated that: *“Sometimes we don’t clean the dependencies (that are not useful anymore) in our project. As a result, we have huge packages that are time-consuming to deploy”*.

Another relevant bad smell in this category is the lack of decomposition of builds into cohesive jobs, resulting in a monolithic build (BP4). This bad smell produces several side effects, including the difficulty in (i) identifying the cause for a build failure, (ii) parallelizing multiple jobs, and (iii) maintaining the overall build process. A slightly different problem that is considered relevant by our respondents is the one dealing with having a sub-optimal ordering of tasks (BP7) in a build process. In other words, some build steps should be always performed before others, e.g., integration testing should be scheduled before deployment in the production environment in order to discover faults earlier.

The way the build output is reported is also particularly important. First of all, respondents believe that ignoring the outcome of a task when determining the build status (BP16) defeats the primary purpose of CI. These kinds of smells may occur when, for example, static analysis tools produce high-severity warnings without failing a build. While a previous study found that this practice is indeed adopted for tools that may produce a high number of false positives [138], one SO post remarked that *“... if the build fails when a potential bug is introduced, the amount of time required to fix it is reduced.”*, and a different user in the same discussion highlighted that *“If you want to use static analysis do it right, fix the problem when it occurs, don’t let an error propagate further into the system.”*. A related bad smell judged as relevant is the lack of an explicit notification of the build outcome (BP23) to developers through emails or other channels. In other words, having the build status only reported in the CI dashboard is not particularly effective, because developers might not realize that a build has failed.

Furthermore, it can be troublesome if tools do not (clearly) report results in the build logs (BP20, BP21) since their output might be difficult or impossible to access when the tool is executed on a CI server. At the same time, a very verbose

build log (BP24) containing unnecessary details (this depends on how different tools/plugins are configured) would make the build result difficult to browse and understand. Quite surprisingly, even if an interviewee remarked that “*To skip tests is always an anti-pattern*”, our survey results report the same proportion of agreement and disagreement on this bad smell (BP8). This is a surprising result since it is difficult to imagine some circumstances in which hiding the presence of issues/problems in the build is a good practice.

Concerning **Build Maintainability**, out of eight bad smells, two received a positive assessment by the majority of respondents and other two received more positive than negative responses (see Table 3.7). The two most positively-assessed bad smells were related to the usage of absolute paths in the build (BM1), and the coupling between the build and the IDE (BM2). The high perceived relevance of such smells is justified considering that their presence will unavoidably limit the portability of the build resulting in statements such as “*but it works on my machine*”. The need for performing smoke-testing (BM7) was also considered relatively important, as well as the usage of suitable naming conventions inside the build scripts (BM8).

While there were no bad smells for which the majority of respondents provided a negative assessment, quite surprisingly, the lack of usage of environment variables (BM3), the cloning of build scripts (BM4), and the presence of lengthy build scripts (BM6) received more negative than positive assessments. As regards the build configurations being cloned in different environments, our interviews and the mined SO posts highlighted that developers should make use of parametrized build jobs and environment variables to enable reuse of build jobs. However, this bad smell (BM4) may be very project-dependent. Indeed, it may be crucial in presence of critical projects where developers need to deploy and test in different environments in which they need to slightly change the build configuration based on the target environment, and define a different chain of build jobs for each environment.

About **Quality Assurance**, out of 14 bad smells, five received a positive assessment by the majority of respondents, and six more positive than negative assessments (see Table 3.8). Respondents considered as relevant the lack of testing in a production-like environment (Q1), while at the same time they considered dangerous the use of production resources for testing purposes (Q7). The latter has

been also highly discussed in SO. Indeed, in a SO post, a user searching “...for the CI server to be useful, my thoughts are that it needs to be run in production mode with as close-as-possible a mirror of the actual production environment (without touching the production DB, obviously)”. By analyzing the provided answers on SO, we found that “Testing environment should be (configured) as close as it gets to the Production” concluding the discussion by highlighting that “The best solution is to mimic the production environment as much as possible but not on the same physical hardware”.

Concerning testing automation, the need for a fully-automated testing process (Q8) was considered important, because manual tests would be excluded from automated build within CI. Furthermore, flaky tests (Q9) were considered particularly problematic in the context of CI, indeed we found many SO discussions in which developers struggled improving the overall build reliability while having randomly failing tests. As an example, a SO question in which a user asked “I’d like to know if there is a way to ... warn of failing tests only if the same tests fail in the previous builds. This, albeit not a perfect solution, would help mitigate the randomness of tests and the effort to analyse only those that are real bugs.”. The above findings help us in explaining why the majority of respondents considered irrelevant the re-execution of failed tests within the same build (Q11).

Concerning the usage of static analysis, the majority of respondents considered a problem the use of CI (and static analysis tools) to monitor specific developers (Q13) since, as also highlighted in our interviews, this kind of monitoring only negatively impacts the overall developers’ productivity and their ability to work in a team. Moreover, one interviewee also highlighted that “Monitor the global technical debt ratio, without paying attention to single developer activity. It’s up to developers to decide when to perform refactoring.”. As regards the definition of quality gates only based on customers’ guidelines (Q12), we got more positive than negative judgments. Indeed, customers could not have enough knowledge about the software development process and the implemented source code. This might result in an excessive number of build failures, due to some quality checks suggested by developers [48] (e.g., on a too low cyclomatic-complexity threshold or too high test-coverage threshold) or, on the contrary, might result in the omission

of some relevant checks that only developers might consider, e.g., because they know details about the source code.

In general, we received particularly positive feedback for the bad smells related to the categories **Delivery Process** and **Culture**. All bad smells belonging to the **Delivery Process** category obtained a positive assessment either by the majority of respondents or, at least, they received more positive than negative assessments (see Table 3.9). The most relevant bad smells were related to the deployment of artifacts generated locally (D1) (i.e., on the developers' machine rather than on the CI infrastructure), to the absence of an artifact repository where to deploy released products (D2), and to the lack of a rollback mechanism (D3).

About the **Culture** category, the majority of respondents considered relevant bad smells related to not trying to fix a build failure immediately (C5), and ignoring CI notifications (C6), see Table 3.10. Other bad smells related to how work is organized received more positive than negative feedback. They concern (i) pulling from the repository when a build has failed (C1), (ii) organizing a meeting just before pushing to the master (C2), (iii) keeping separate the developer and operator roles (C3), and (iv) developers that do not have complete control of the CI environment (C4). This indicates how the lack of a shared view on how developers (and teams) should behave when adopting CI would diminish the advantages introduced by putting in place a suitable (and in some cases complex) CI infrastructure.

3.4.3 Mapping and Comparison with Duvall's Anti-Patterns

Table 3.13 reports the mapping between Duvall's patterns and our CI bad smells. Note that we use the pattern name, because Duvall, under each pattern, reports a brief description of the good practice (pattern) and the corresponding bad practice (anti-pattern). For brevity, in Table 3.13 and in the remainder of this section, we refer pattern/anti-pattern using the pattern's name, however, we discuss the comparison considering Duvall's anti-patterns and our CI bad smells. Where necessary, we also report the corresponding anti-pattern in our text. The last column summarizes the perception of the CI bad smell (detailed values are in Tables 3.4-3.10) corresponding to each Duvall's pattern. In particular, we use

(i) $\uparrow\uparrow$ to indicate bad smells that received a positive assessment by the majority of respondents; (ii) \uparrow for bad smells that received more positive than negative feedback (but $\leq 50\%$ of positive answers); (iii) $-$ for bad smells that received an equal proportion of positive and negative feedback; (iv) \downarrow for bad smells that received more negative than positive feedback (but $\leq 50\%$ of negative answers); and (v) $\downarrow\downarrow$ to indicate bad smells that received a negative assessment by the majority of respondents.

The mapping is not one-to-one. For example, on the one side, the “Pipeline related resources (e.g., configuration files, build script, test data) are not versioned” (R10) in our catalog covers 3 different patterns from Duvall’s catalog, namely “Repository“ (where the anti-pattern mentions “some files are checked in others, such as environment configuration or data changes, are not. Binaries — that can be recreated through the build and deployment process — are checked in”), “Single Path to Production” (the anti-pattern mentions “parts of system are not versioned”) and “Scripted Database” (the anti-pattern mentions “manually applying schema and data changes to the database.”). All these anti-patterns highlight the presence of resources needed for the CI process that are not versioned. On the other side, the “Build Threshold” pattern in Duvall’s catalog (the anti-pattern mentions “learning of code quality issues later in the development cycle.”) matches 3 different bad smells in our catalog, namely “Coverage thresholds are too high” (Q4), “Coverage thresholds are fixed on what reached in previous builds” (Q3), and “A build is succeeded when a task is failed or an error is thrown” (BP16).

Our catalog, composed of 79 bad smells, covers 39 out of 50 Duvall’s anti-patterns, while 11 of them are left uncovered. Going more in-depth into anti-patterns uncovered by our CI catalog, we found two cases, namely “Configuration Catalog” (the anti-pattern says “configuration options are not documented”) and “Commit Often” (the anti-pattern says: “source files are committed less frequently than daily ...”). Such anti-patterns are mainly related to versioning system’s usage not directly specific to the CI context. Moreover, there are other 4 Duvall’s anti-patterns mainly related to planning activities, which are out of the scope of CI, and therefore were not covered in our interviews nor in the analyzed SO posts. These are:

Table 3.13: Mapping between Duvall’s patterns (and their corresponding anti-patterns) and CI bad smells

Duvall Pattern	CI Bad Smell	Rel.
Configurable Third-Party Sw.	A task is implemented using an unsuitable tool/plugin (I6)	↓
Configuration Catalog	✘	
Mainline	Number of branches do not fit the project needs/characteristics (R4)	↓
Merge Daily	Divergent branches (R7)	↑
Protected Configuration	Authentication data is hardcoded under VCS (BP29)	↑↑
Repository	Pipeline related resources are not versioned (R10)	↑↑
Repository	Generated artifacts are versioned, while they should not (R8)	↓
Short-Lived Branches	Divergent branches (R7)	↑
Single Command Environment	Some pipeline’s tasks are started manually (BP13)	–
Single Path to Production	Pipeline related resources are not versioned (R10)	↑↑
Build Threshold	A build is succeeded when a task is failed or an error is thrown (BP16)	↑↑
Build Threshold	Coverage thresholds are too high (Q4)	↓
Build Threshold	Coverage thresholds are fixed on what reached in previous builds (Q3)	↓
Commit Often	✘	
Continuous Feedback	Missing notification mechanism (BP23)	↑↑
Continuous Feedback	Failures notif. only sent to teams that explicitly subscribed (BP22)	↓
Continuous Feedback	Issue notifications are ignored (C6)	↑↑
Continuous Integration	Use of nightly builds (BP14)	↑
Continuous Integration	Poor build triggering strategy (BP11)	↑↑
Continuous Integration	Only the last commit is built, aborting obsolete and queued builds (BP6)	↓↓
Stop The Line	Build failures are not fixed immediately giving priority to other changes (C5)	↑↑
Independent Build	Build scripts are highly dependent upon the IDE (BM2)	↑↑
Visible Dashboards	Failures notif. only sent to teams that explicitly subscribed (BP22)	↓
Automate Tests	Testing is not fully automated (Q8)	↑↑
Isolate Test Data	Production resources are used for testing purposes (Q7)	↑↑
Parallel Tests	Independent build jobs are not executed in parallel (BP5)	↓↓
Stub Systems	Production resources are used for testing purposes (Q7)	↑↑
Deployment Pipeline	Some pipelines’ tasks are started manually (BP13)	–
Value-Stream Map	✘	
Dependency Management	Dependency management is not used (BP18)	↑
Common Language	✘	
Externalize Configuration	Environment variables are not used at all (BM3)	↓
Externalize Configuration	Build configurations are cloned in different environments (BM4)	↓
Externalize Configuration	Authentication data is hardcoded (in clear) under VCS (BP29)	↑↑
Fail Fast	Wide and incohesive jobs are used (BP3)	↓
Fast Builds	Build time for the “commit stage” overcomes the 10-minutes rule (BP27)	↑↑
Fast Builds	Tasks are not properly distributed among different build stages (BP9)	↓↓
Scripted Deployment	Some pipelines’ tasks are started manually (BP13)	–
Unified Deployment	Build configurations are cloned in different environments (BM4)	↓
Binary Integrity	Missing artifacts’ repository (D2)	↑
Canary Release	✘	
Blue-Green Deployments	✘	
Dark Launching	✘	
Rollback Release	Missing rollback strategy (D3)	↑↑
Self-Service Deployment	Developers and operators are kept as separate roles (C3)	↑
Automate Provisioning	Developers do not have a complete control of the environment (C4)	↑
Behavior-Driven Monitoring	Missing smoke test, set of tests to verify the testability of the build (BM7)	↑
Immune Systems	✘	
Lockdown Environments	Developers do not have a complete control of the environment (C4)	↑
Production-Like Environments	Lack of testing in a production-like environment (Q1)	↑↑
Transient Environments	✘	
Database Sandbox	Lack of testing in a production-like environment (Q1)	↑↑
Database Sandbox	Inappropriate build environment clean-up strategy (BP1)	↑
Decouple Database	✘	
Database Upgrade	Some pipelines’ tasks are started manually (BP13)	–
Scripted Database	Pipeline related resources are not versioned (R10)	↑↑
Branch by Abstraction	Number of branches do not fit the project needs/characteristics (R4)	↓
Toogle Features	Feature branches are used instead of feature toggles (R6)	↑
Delivery Retrospective	Developers and operators are kept as separate roles (C3)	↑
Cross-Functional Teams	Developers and operators are kept as separate roles (C3)	↑
Root-Cause Analysis	✘	

1. “Canary Release”, where the anti-pattern occurs when “software is released to all users at once”;
2. “Dark Launching”, where the anti-pattern occurs when “software is deployed regardless of the number of active users”;
3. “Value-Stream Map”, where the anti-pattern relates to “separately defined processes and views of the check in to release process”;
4. “Common Language”, where the anti-pattern occurs when “each team uses a different language making it difficult for anyone to modify the delivery system”.

Also, our CI catalog does not cover specific Duvall patterns mainly related to deployment and delivery activities, namely:

1. “Blue-Green Deployments”, highlighting the need for deploying software to a non-production environment while production continues to run;
2. “Immune System”, that emphasizes the need for deploying software one instance at a time while conducting Behavior-Driven Monitoring;
3. “Decouple Database”, aimed at ensuring that the application is backward and forward compatible with the database giving the possibility of independent deployment activities.

Finally, our catalog does not cover the (i) “Transient Environments”, where the anti-pattern occurs when environments are fixed or pre-determined, and (ii) “Root-Cause Analysis”, where the anti-pattern occurs when developers “accept the symptom as the root cause of the problem”.

From a different perspective, looking at the second column in Table 3.4-Table 3.10, only 35 of our CI bad smells are covered by Duvall, while 44 are completely uncovered. More specifically, our catalog includes specific CI bad smells related to (i) the way the CI infrastructure is chosen and organized, as well as (ii) how a build process is organized and configured, and (iii) testing and quality checks.

Focusing on the testing phase (and therefore on bad smells we categorized under Quality Assurance), it is possible to state that, even if Duvall's catalog has a category aimed to cover problems occurred in doing testing activities consisting of four patterns/anti-patterns fully covered by our catalog, we provide other six further bad smells not contemplated by Duvall's catalog. Specifically, two of them — “Missing tests on feature branches” (Q5) and “All permutations of feature toggles are tested” (Q6) — focus on the way a testing strategy is adopted in the CI pipeline. Their consequence is a negative impact on the build duration, also preventing the availability of fast feedback.

Other two CI bad smells not covered by Duvall, namely “Bad choice on the subset of test cases to run on the CI server” (Q10) and “Failed tests are re-executed in the same build” (Q11), deal with the way in which test cases are executed in the CI pipeline.

Looking at the second column in Table 3.8, we can notice that three CI bad smells related to the definition/usage of quality gates are not mapped onto Duvall's catalog, and, at the same time, are highly relevant for our survey participants. More in detail, the “Quality gates are defined without developers considering only what dictated by the customer” (Q12) highlights that developers need to be involved in the definition of the quality gates, since customers do not have enough knowledge of software development and implementation details. The “Use quality gates in order to monitor the activity of specific developers without using them for measuring the overall software quality” (Q13), instead, emphasizes the fact that the overall team spirit will be negatively impacted in presence of monitoring activities. Finally, the “Unnecessary static analysis checks are included in the build process” (Q14) may unnecessarily slow down the build duration as well as may decrease the developers' productivity, since developers will waste their time trying to fix violated checks that do not completely fit the need of their organization.

In the following, we discuss some examples of CI bad smells (partially) contradicting common wisdom and/or outlining trade-off situations for developers.

Nightly builds are a particular type of scheduled builds that are usually performed overnight (i.e., out of working hours). Duvall reports their usage as an anti-pattern corresponding to the “Continuous Integration” pattern (“Scheduled builds, nightly builds, building periodically, building exclusively on developer's

machines, not building at all”). Indeed, Duvall highlights as good practice the need for building and testing software with every change committed to a project VCS [27].

In our catalog, the CI bad smell “The project only uses nightly builds when having multiple builds per day is feasible” (BP14) is relevant for 48% of our respondents. Differently from Duvall, we do not consider nightly builds as something to avoid; instead, we foresee caution in using them. On the one side, to follow the “Fast Builds” rule it is important to execute time-consuming tasks over the night and not during the regular builds. On the other side, if developers schedule the whole set of tasks during regular builds, the usage of nightly builds could be redundant. The above bad smell is discussed in many SO posts related to the build duration. More in detail, a SO discussion stated: “... *on each commit, I would like to run a smaller test suite, and then nightly it should run a full regression test suite ... which is much more involved and can run for hours ...*”

Having an adequate branching strategy is another key point in the CI process. Our catalog provides four different CI bad smells dealing with the adoption of a poor branch management strategy in the CI process. Among them, the “Feature branches are used instead of feature toggles” (R6) and “Number of branches do not fit the project needs/characteristics” (R4) identify situations in which the right decision depends upon the organization needs. More specifically, the bad smell (R6) discourages the use of branches when developing features, and it is in agreement with what stated by Duvall. However, a recent study by Rahman et al. [97] found that feature toggles, despite their advantages, could introduce technical debt involving maintenance effort for developers. To this regard, we found quite controversial opinions in SO posts. For example, in one SO post, the preferred choice is the feature toggles even if “... requires very strict discipline as broke/dark code is making it to production”. The discussion ends with the following suggestion: “*I do not believe in a better choice in all the cases*”.

Furthermore, the “Number of branches do not fit the project needs/characteristics” (R4) bad smell, mapped onto Duvall’s anti-pattern “Multiple branches per project” (related to the “Mainline” pattern), occurs when a project has several branches that might not be needed. While Duvall indicates as an anti-pattern the “Feature Branching”, our CI bad smell suggests the usage of a proper number of

branches according to a well-defined strategy. An interviewee of our study agrees with the usage of different branches evolving in parallel (“*We use a branch for each service and usually 3/4 developers work on it. There is a straight separation between each branch.*”), while some SO discussions discourage to exceed in the number of branches: “*You must have your changes included in the main trunk so you can reduce the number of conflicts related to merge operations*”.

Since 39 patterns/anti-patterns defined by Duvall are covered by 35 CI bad smells of our catalog, it is interesting to look at their perceived relevance according to our survey respondents. More specifically, we discuss some cases considered highly relevant as well as some cases considered as less relevant. On the one side, unsurprisingly, the “Protected Configuration” pattern (the anti-pattern mentions “open text passwords and/or single machine or share”) mapped onto our “Authentication data is hardcoded (in clear) under VCS” bad smell (BP29), is considered still relevant by our survey participants since it is mainly related to security issues. Also for “Single Path to Production” pattern defined by Duvall, mapped onto our “Pipeline related resources are not versioned” bad smell (R10), the relevance is high since that this bad smell negatively impacts the reproducibility of the overall build process. On the other side, the “Parallel Tests” pattern is felt as less relevant by our survey respondents. Developers, at least for builds that are not executed in the commit stage, do not account for the build duration. Indeed, the main goal of parallelizing independent tasks is to reduce the overall build time.

Finally, Duvall considers as an anti-pattern the execution of dependent build jobs/tests in parallel. This bad smell has been found in many SO discussions, however, as already detailed in Section 3.3.2, we discarded it because we consider this to be a bug rather than a bad smell.

3.5 Threats to Validity

Threats to *construct validity* relate to the relationship between theory and experimentation. These are mainly due to imprecision in our measurements. We relied on SO tags to filter SO discussions. While it is possible that we could have missed some relevant posts because their tags were not directly related to CI, we at least made sure to pick all tags of interest by performing a manual, exhaustive analysis

of all SO tags. Concerning the protocol used for the semi-structured interviews, it is possible that the incompleteness of our interview structure could have led to some CI bad smells. However, to mitigate this threat, the questions being asked in the last part of our interview took into account what we learned from the existing literature [25, 27, 47]. Concerning the way we collected bad practices relevance through the survey, we used a 5-level Likert scale [83] to collect the perceived relevance of each CI practice. To limit random answers, we added a “Don’t know” option and the opportunity to explain the answers with a free comment field.

Threats to *internal validity* are related to confounding factors, internal to the study, that can affect our results. Internal validity threats can, in particular, affect the extent to which the protocol followed to build the catalog could have influenced our results. As detailed in Section 3.3.2, we used different countermeasures to limit the influence of our subjectiveness. As for the analysis of SO posts, it was not possible to afford two independent taggers per post. However, the independent re-check on the “Yes”/“Maybe” limited the false positives, though could not mitigate possible false negatives. The mapping onto Duvall’s anti-patterns and the validation of the catalog were performed by two independent evaluators, inter-rater agreement was computed, assignment conflicts were resolved through a discussion, and inconsistent cases were re-coded again to improve the inter-rater agreement. For what concerns the creation of the catalog itself, we interleaved multiple iterations done by discussions of comments each author raised during each iteration.

Threats to *conclusion validity* concern the relationship between theory and outcome, and are mainly related to the extent to which the produced catalog can be considered exhaustive enough to capture CI bad practices. While we are aware that there might be CI bad smells we did not consider, to mitigate the threat and verify saturation, we validated a statistically significant sample of SO posts not used when building the catalog.

Threats to *external validity* concern the generalizability of our findings. These are mainly due to (i) the choice of SO as a source for mining CI-related discussions, and (ii) the set of interviewed people. Concerning SO, it is the most widely used development discussion forum to date. Although specific forums for CI exist (e.g., DZone⁶), such forums are more portals where white papers (e.g., the one with

⁶<https://dzone.com>

Duvall's anti-patterns [27]) are posted rather than a Question & Answer (Q&A) forum. The number of participants (and companies) to the interviews and survey is admittedly, limited. At the same time, the companies are pretty diverse in terms of domain and size, and the interviewed/surveyed people always had several years of experience. Nevertheless, we could still have missed some relevant problems, e.g., we have shown (Section 3.4.3) that eight of Duvall's anti-patterns did not emerge from our study.

3.6 Implications

This section discusses the implications of the identified CI bad smells for practitioners, researchers, and educators.

3.6.1 Implications for Practitioners

In the following, we discuss implications for practitioners, which could either be developers just using the CI pipeline, but also developers having the responsibility and rights to configure it.

In this context, the aim of the catalog is providing developers with concrete misuses of the CI pipeline. These misuses can occur at different stages, including: (i) setting up and configuring the Software Configuration Management (SCM) infrastructure, as well as the CI infrastructure and tooling, and (ii) performing daily activities in the context of a CI pipeline, which translates into creating and synchronizing/merging branches, pushing changes, interpreting and leveraging results of the CI builds to improve software quality.

In the following, we report some scenarios where the catalog of CI bad smells could help developers.

Favor Specific, Portable Tools over Hacking One of the CI bad smells we identified is related to a sub-optimal selection of tools composing the CI pipeline.

Specifically, the adoption of a tool that does not provide the features needed by developers may cause delays or, even worse, force the adoption of “hacking solutions”, e.g., custom shell scripts. While such scripts can represent a quick-and-dirty solution for the immediate, in the long term they can exhibit maintainability

issues, or even introduce portability problems on different machines. This CI bad smell was pointed out by one of our interviewees, saying that they “*replaced the old scripts with Jenkins*”. Our catalog includes the bad smell “Use shell scripts for a task for which there is a suitable plugin available” (I7) that covers these aspects. Moreover, developers should pay attention to avoid different versions of tools conflicting with each other, as indicated by the bad smell “Different releases of tools/plugins versions are installed on the same server” (I4).

Do not Use *Out-of-the-Box* Tools, nor Listen Customers Only Even when suitable tools are chosen, such tools have to be properly configured. One of our bad smells is “External tools are used with their default configurations” (I3). Moreover, in this context, a relevant and frequent bad habit is to not involve developers in the definition of the quality gates, but just listen to customers’ requirements. Indeed, as reported by an interviewee, the “*client dictates the quality gates . . . [they] have only to meet these gates*” even though they do not have enough expertise or knowledge about the software development skills. For those reasons, our catalog features the bad smell “Quality gates are defined without developers, considering only what dictated by the customers” (Q12) that reminds how quality gates that have been established without developers could increase the number of irrelevant warnings and slow down the entire CI process.

Do not Ignore nor Hide Build Failures When working with the CI pipeline, it is possible that developers do not give adequate priority and importance to warnings outputted in build logs, or even worse, to broken builds. As a consequence, they tend to hide actual problems by just disabling failing tests or quality checks instead of solving them. In other cases, developers might improperly use the features provided by the whole environment or by specific toolkits. A concrete example is reported in a SO post⁷ in which a user highlighted that there are cases in which developers focus more on “*. . . keep the build passing as opposed to ensuring we have high quality software*”. The latter translates into “*. . . comment[ing] out the tests to make the build pass*” resulting in having a passed build while the overall software quality is going down.

Our catalog features the bad smell “Pipeline steps/stages are skipped arbitrarily” (BP8), pointing out that it is important to not skip pipeline tasks only to have a

⁷<https://stackoverflow.com/questions/214695/>

passed build. Also, the Culture category features bad smells occurring when “Build failures are not fixed immediately giving priority to other changes” (C5), or “Issue notifications are ignored” (C6).

3.6.2 Implications for Researchers

Development of Automated CI Bad Smell Detectors Researchers can use the catalog as a starting point to develop (semi) automated recommender systems able to identify CI bad smells by mining or monitoring VCS data, CI build logs, configuration files, and other artifacts. A CI bad-smell detector could, for example, identify poor choices in build or configuration files by analyzing the build scripts’ complexity, length, and readability, or the presence of hard-coded configuration parameters. Then, the detector can highlight the bad smells, and possibly, provide suggestions for refactoring them/making them easier to read.

Also, a CI bad-smell recommender could go beyond that, and “observe” the activity carried out by developers through the pipeline, including branching strategies, push/pull frequency, distribution of build failures, fix time etc.. As an example, the CI bad smell “Pipeline steps/stages are skipped arbitrarily” (BP8) could be detected by analyzing the build history of the project, and detecting cases where a build that failed because of test cases becomes successful again without changes to the production code but, instead, because test cases were commented out or disabled.

Support for Failure Analysis A key element for quick fixing a build is the capability of developers to properly analyze the output of a build. Our CI bad smells provide indications of what should be avoided when configuring the build log (e.g., “Build reports contain verbose, irrelevant information” (BP24) or “The output of different build tasks are mixed in the build output” (BP21)) or other notification mechanisms (e.g., “Failures notifications are only sent to teams/developers that explicitly subscribed” (BP22)). Also, techniques similar to those used to summarize release notes [78] or bug reports [99] could be used to extract and summarize relevant information from verbose and complex build logs.

Keep the Context into Account When building tools to identify CI bad smells, researchers should take into account the developers’ context and, if possible,

their feedback to past recommendations to properly tune the recommender and avoid overloading developers with irrelevant suggestions. This could be achieved by collecting a set of preferences about the CI practices adopted in one organization (e.g., when to open a branch, push/pull practices, testing, and static analysis needs), by also observing the past history of developers' activity. For example, self-admitted technical debt [89] can be used to learn bad smells relevant to developers, and consequently help to configure static analysis tools properly.

Expand the Catalog Finally, having provided a methodology to infer bad smells and a full study replication package, the catalog can be extended through further bad smells other researchers might discover, and replication studies on bad-smell relevance can confirm or contradict the results of our survey.

3.6.3 Implications for Educators

Teach CI by Providing Examples of What not to Do The proposed catalog can be valuable for educators introducing CI/CD principles in software engineering curricula. A typical course about CI would introduce the topic, illustrate the main claimed advantages of CI, explain the technologies that can be used, and, importantly, provide principles to properly set up and use the CI pipeline. So far, such principles are typically being taught mainly based on the content of known books [25, 47] or by using catalogs/white-papers reflecting existing knowledge in this area coming from the available literature.

With the proposed catalog, it would be possible to explain CI misuses, which are based on specific experiences occurred to developers and discussed in SO or our interviews. In other words, this could enable the introduction of CI using a “learn by example” methodology, illustrating practices that should be avoided. Also, when applying CI in their homeworks, students can be thought to monitor the occurrence of CI bad smells, open issues to this regard, and solve them whenever possible.

Teach CI Culture, not Just Technology Given the importance provided by our survey respondents to the Culture-related bad smells, it should be avoided to introduce technological content about CI without fostering the right understanding of the metaphor and its related processes.

3.7 Conclusions and Future Work

The adoption of Continuous Integration and Delivery (CI/CD) as development practice is increasing. Studies conducted in industrial organizations and open source projects highlight benefits of CI/CD, including increased developer productivity, software quality, and reliability [17, 46, 124]. Despite those advantages, the introduction of CI/CD in well-defined development contexts is still challenging, as also highlighted in previous work [27, 45, 82].

This paper empirically investigates, by analyzing over 2,300 SO discussions and by interviewing 13 industry experts, on bad practices developers encounter when adopting CI. As a result of the study, we compiled a catalog of 79 CI bad smells organized into 7 categories. The catalog has been validated through a survey involving 26 professional developers, indicating their perceived relevance of the 79 bad smells.

While a catalog of CI patterns and anti-patterns exists [27], this is, to the best of our knowledge, the first catalog empirically derived from concrete problems discussed in SO or highlighted by interviewed experts, and following a replicable methodology.

As we have shown in Section 3.4.3, while in some cases our results confirm what known from the existing literature, there are quite a few controversial cases, such as the appropriate balancing in the usage of nightly builds or feature branches. Moreover, while our catalog does not cover 11 of Duvall's anti-patterns (though covering all categories), there are 44 of our CI bad smells not covered by Duvall, and they especially pertain to the CI infrastructure, the build configuration, as well as testing and quality checks. We also discuss cases in which the two catalogs agree and disagree, also highlighting examples of Duvall's anti-patterns not perceived as important by our survey participants.

The catalog resulting from our study has implications for different stakeholders. Practitioners can use it to avoid/recognize the application of bad practices degrading the overall build process. Also, the catalog could be of benefit for educators introducing CI in software engineering curricula, and researchers interested in conceiving CI bad smell detectors. Indeed, besides enlarging the study to different

contexts, our future work is in the direction of building automated recommenders to detect and possibly remove CI bad smells.

Interestingly, while some researchers previously investigated barriers towards CI [45] and CD [82] adoption, the inferred bad practices indicate that some of those issues still arise when the CI process has been adopted. They include, for example, long in-the-loop feedback during development, or network configuration issues.

Future work should further analyze the observed bad smells through other kinds of empirical studies, e.g., in-field studies conducted within companies.

Acknowledgements

We would like to thank experts/developers involved in our interviews and those who participated in our online survey. Vassallo, Panichella, and Gall also acknowledge the Swiss National Science Foundation's support for the project SURF-MobileAppsData (SNF Project No. 200021-166275).

4

A Tale of CI Build Failures: an Open Source and a Financial Organisation Perspective

Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, Sebastiano Panichella
Published in Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, Technical Research Paper (ICSME), 2017
Contribution: study design, data collection, manual inspection of error messages in build logs, data analysis, and paper writing

Abstract

Continuous Integration (CI) and Continuous Delivery (CD) are widespread in both industrial and open-source software (OSS) projects. Recent research characterized build failures in CI and identified factors potentially correlated to them. However, most observations and findings of previous work are exclusively based on OSS projects or data from a single industrial organization. This paper provides a first attempt to compare the CI processes and occurrences of build failures in 349 Java OSS projects and 418 projects from a financial organization, ING Nederland. Through the analysis of 34,182 failing builds (26% of the total number of observed builds), we derived a taxonomy of failures that affect the observed CI processes. Using cluster analysis, we observed that in some cases OSS and ING projects share similar build failure patterns (e.g., few compilation failures as compared to frequent testing failures), while in other cases completely different patterns emerge. In

short, we explain how OSS and ING CI processes exhibit commonalities, yet are substantially different in their design and in the failures they report.

4.1 Introduction

Continuous Delivery (CD) is a software engineering practice in which development teams build and deliver new (incremental) versions of a software system in a very short period of time, e.g., a week, a few days, and in extreme cases a few hours [47]. CD advocates claim that the practice reduces the release cycle time (i.e., time required for conceiving a change, implementing it, and getting feedback) and improves overall developer and customer satisfaction [16]. An essential part of a CD process is Continuous Integration (CI), where an automated build process is enacted on dedicated server machines, leading to multiple integrations and releases per day [10, 25, 31, 112]. One major purpose of CI is to help developers detect integration errors as early as possible. This can be achieved by running testing and analysis tools, reducing the cost and risk of delivering defective changes [25]. Other collateral positive effects introduced by CI in industrial environments are the improvement in developer communication [23] and the increase of their productivity [75]. Consequently, CI has become increasingly popular in software development of both, industrial and OSS projects [46, 112].

At the same time, the study of CI builds has also become a frequent research topic in academic literature. Miller [75] studied a Web Service project in Microsoft and, by observing 69 failed builds, mainly found failures related to code analysis tools (about 40%), but also to unit tests, server, and compilation errors. The latter have been investigated in depth by Seo et al. [107] in a study at Google. Recently, Rausch et al. [100] studied the Travis CI builds of selected OSS projects, and identified correlations between project and process metrics and broken builds. Other works focused on test failures [10] and on the use of static analysis tools [142]. However, no previous research provides a broad categorization of build failures and compared their occurrences between industry and OSS.

This paper studies build failures in 418 projects (mostly Java-based) from a large financial organization, ING Nederland (*referred to as ING*), as well as in 349 Java-based open source software (OSS) hosted on GitHub and using Travis

CI. The purpose of the study is to compare the outcome of CI in OSS and in an industrial organization in the financial domain, and to understand commonalities and differences. As previous work by Ståhl et al. [113] suggested that the build process in industry varies substantially, we aim to understand the differences (also in terms of build failure distributions) between OSS and one industrial case. In total we analyzed 3,390 failed builds from ING and 30,792 failed builds from OSS projects. Based on this sample, we address the following research questions:

RQ₁: *What types of failures affect builds of OSS and ING projects?*

This research question aims to understand the nature of errors occurring during the build stage in ING and the analyzed OSS projects. We use an open coding procedure to define a comprehensive taxonomy of CI build errors. The resulting taxonomy is made up of 20 categories, and deals not only with the usual activities related to compilation, testing, and static analysis, but also with failures related to packaging, release preparation, deployment, or documentation. Overall, the taxonomy covers the entire CI lifecycle. We then study build failures along the taxonomy, addressing our second research question:

RQ₂: *How frequent are the different types of build failures in the observed OSS and ING projects?*

Given the catalog of build failures produced as output of **RQ₁**, we then analyze and compare the percentages of build failures of different types for both, ING and OSS projects. Furthermore, based on these percentages, we cluster projects and discuss characteristics of each cluster, observing in particular whether different clusters contain ING projects only, OSS projects only, or a mix. Finally, we investigate the presence of build failure patterns shared by ING and OSS projects in the various clusters.

Our study shows that ING and OSS projects share important commonalities. For example, both exhibit a relatively low percentage of compilation failures, and, instead, a high percentage of testing failures. However, while unit testing failures are common in OSS projects (as also discussed in [84]), ING projects have a much higher frequency of integration test failures. ING projects also exhibit a high

percentage of build failures related to release preparation, as well as packaging and deployment errors. We also found that projects cluster together based on the predominance of build failure types, and some clusters, e.g., related to release preparation or unit testing, only contain or are predominated by ING or OSS projects respectively. In summary, while the behavior of CI in OSS and closed source exhibits some commonalities, closed source has some peculiarities related to how certain activities, such as testing and analysis, are performed, and how software is released and deployed.

4.2 Study Design and Planning

The *goal* of the study is to investigate the types of build failures that occur in the analyzed OSS and ING projects, how frequently they occur, and to understand the extent to which these failure frequencies differ in the analyzed industrial and OSS projects. This analysis has the *purpose* of understanding the commonalities and differences in the CI process of OSS and of an industrial environment (ING), at least from what one can observe from build failures and from the knowledge of the adopted CI infrastructure.

4.2.1 Study Context

The study *context* consists of build failure data from 418 projects (mostly Java-based, as declared by the organization) in ING and 349 Java-based OSS projects hosted on GitHub.

In both, ING and OSS, the CI process is triggered when a developer pushes a change to the Git repository. The code change is detected by the CI server (Jenkins¹ in ING and Travis CI² in OSS) and the build stage is started. The outcome of the build process is either a *build failure* or a *build success*.

In ING, if a triggered build succeeds, the generated artifacts of the new version are deployed to a remote server that simulates different environments (i.e., testing, production) to perform further activities (e.g., load, security testing). Indeed ING adopts a well-defined CD pipeline (illustrated in a survey conducted in ING [135])

¹<https://jenkins.io> ²<https://travis-ci.org>

where the build is only one node of the entire process of an application's release. Some external tools are usually plugged into the build process to augment the actions performed at some steps. For example, SonarQube³ is used as source code quality inspector, sometimes complemented by PMD⁴ and Checkstyle.⁵

In our study, we analyzed 12,871 builds belonging to 418 different Maven projects in ING, and mined data from 4 different build servers. Specifically, we extracted, for each project, the Maven logs related to the failed builds that occurred between March 21st, 2014 and October 1st, 2015. The resulting total number of Maven build failures is 3,390 ($\approx 26\%$ of builds). Due to restrictive security policies in the financial domain, the Maven logs were the only resources we could access during the study. Hence, we did not have access to any other resource that would be valuable in order to investigate the nature of build failures in more depth, e.g., data from versioning systems, testing or detailed outputs of static analysis tools (except the data printed to build logs). Therefore, our observations are limited to the information available in build logs.

As for OSS, we selected 349 projects from the *TravisTorrent* dataset [11], which contains build failure data from GitHub projects using Travis CI.⁶ We restricted our analysis to all projects of the TravisTorrent dataset using Maven (in order to be consistent with ING projects) and mainly written in Java (according to the dominant repository language reported by GitHub). In addition, we only considered projects having at least one failing build. In total, the 349 projects underwent 116,741 builds, of which 30,792 ($\approx 26\%$) failed. It is interesting to notice how the percentage of build failures is approximately the same in OSS and ING.

4.2.2 Data Extraction Approach

Similarly to the work by Désarmieux *et al.* [21] we focused our analysis on Maven projects and their related build logs. Differently from the work by Désarmieux *et al.*, we did not analyze build scripts as they were not provided by the involved organization and thus we did not perform a more fine-grained analysis of the kinds of failures.

³<https://www.sonarqube.org> ⁴<https://pmd.github.io>

⁵<https://checkstyle.sourceforge.io> ⁶Data accessed on 27/10/2016

To extract build failure data for ING, we used the Jenkins REST APIs, which retrieved the log associated to each build and allowed us to reconstruct the history of a job in terms of build failures and build successes. To extract build failure data for OSS, we started from the information contained in the TravisTorrent database. Then, we downloaded build logs for each job IDs related to a build ID labelled as failed, by means of the Travis APIs.

Subsequently, we extracted the error messages contained in each log using a regular expression, e.g., the lines that contain the word “ERROR”, and used them to define our Build Failure Catalog.

4.2.3 Definition of the Build Failure Catalog

Our process for defining the Build Failure Catalog consists of five steps:

Keyword Identification. In this step, two authors mined the most relevant keywords associated with an error section. In Maven, each build fails because of the failure of a specific step (i.e., a goal). As outcome of this failure, an error section is generated in the build log. From each error section, we extracted as keywords only the phrase related to the failed goals (generated by Maven), as well as sentences generated by Maven plugins, making sure of their stability (i.e., no changes in the structure of error messages) during the period of observation. For example, if an error during the execution of the “compilation” goal (e.g., “*org.apache.maven.plugins:maven-compiler-plugin:compile*”) causes a build failure, the goal name is assumed as keyword. In other cases, when there is no evidence of failed goals in the build logs, we mined other sentences (e.g., “*Could not resolve dependencies*” in case of missing Maven artifacts).

Keyword Grouping. Two of the authors (hereby referred to as A and B) produced a first grouping of keywords into categories. For this step, we used the Maven documentation⁷ and the documentation of the respective plugins in order to understand the keywords. Then, two other authors (C and D) reviewed the first categorization and made change suggestions. This process will be explained in more detail in Section 4.3.1. Finally, A and B checked the resulting catalog again, which led to convergence.

⁷<https://maven.apache.org/guides/index.html>

Build Log Classification. The build logs were automatically classified according to the previously defined catalog. Based on the matched keywords, each build log was assigned to zero, one, or more categories.

Result Validation. After the automatic classification, a second, manual classification of a statistically significant sample has been performed to determine the error margin introduced by the automatic classification. Each sample has been validated by two authors and the reliability of the validation was computed using the Cohen k inter-rater agreement statistic [18]. After that, we discussed cases in which the raters disagreed in the classification and in which there was a mismatching with the automatic classification.

Catalog Refinement. If, as a result of the manual validation, it turned out that the first automatic classification was erroneous, the manual validation helped to further refine the categories. When categories were refined, a new validation (limited to these categories only) was performed, if necessary.

This five-step procedure was first applied to ING build failures and subsequently to the OSS data set. This means that the two inspectors, starting from the taxonomy derived from the analysis of ING build failures, extended, if necessary, the taxonomy with categories specific to OSS. Regarding the validation of the automatic classification, for ING data we randomly extracted for each identified category a sample size of 764 build failures considering a confidence level of 95% and a confidence interval of $\pm 10\%$. For OSS projects we did not perform a new stratified sampling. Instead, we conducted a complementary validation. We extracted a sample of 377 build failures (which is significant with 95% confidence level and $\pm 5\%$ of confidence interval), making sure the sample contains a number of failures, for each category, proportional to the distribution of failures across categories.

4.2.4 Analysis of Build Failure Proportions

In order to investigate the extent to which proportions of build failures of different types vary among ING and OSS, we perform two types of analyses. First, we statistically compared and discussed the frequency of build failures for different types among ING and OSS projects. Then, we used the relative frequencies of different types of failures to cluster projects exhibiting similar distributions of build

failures. We used k-means clustering [42] (the *kmeans* function available in the R [90] *stats* package). The k-means function requires to upfront specify the number of clusters k in which items (in our case projects) should be grouped. To determine a suitable value of k , we plotted the *silhouette* statistics [54] for different values of k . Given a document d_i belonging to a cluster C and, $a(d_i)$ the maximum distance of d_i from the cluster's centroid, and $b(d_i)$ from the centroids of other clusters, the silhouette for d_i is given by

$$S(d_i) = \frac{b(d_i) - a(d_i)}{\max(a(d_i), b(d_i))}$$

and the overall silhouette of a clustering is given by the mean silhouette across all documents (all projects in our case). Typically, the optimal number of clusters corresponds to the maximum value of the silhouette.

After this clustering step, we analyzed the clusters' content, looking in particular at the extent to which clustering separates ING projects from OSS projects and the occurrences of particular failures patterns.

4.3 Study Results

This section reports the results of the study we conducted for addressing our research questions.

4.3.1 What Types of Build Failures Affect Builds of OSS and ING Projects?

In the following, we first report how we obtained the Build Failure Catalog, and then discuss the catalog categories in detail.

From Maven Phases to the Build Failure Catalog

With the aim of defining a complete and possibly generalizable Build Failure Catalog, we firstly attempted to group build failures into categories closely following the Maven lifecycle phases, i.e., validate, process-resources, compile, test-compile, test,

package, integration-test, verify, install, and deploy. When a build failure could not be mapped onto specific Maven phases (e.g., those related to non-functional testing), we created a new category.

A first result was made up of 16 categories, specifically (i) a validation category; (ii) a pre-processing category; (iii) three compilation categories related to compiling production code, compiling test code, and missing dependencies; (iv) three separate categories for unit testing, integration testing, and non-functional testing; (v) a static analysis category; (vi) a packaging category; (vii) an installation category; (viii) a deployment category; (ix) a documentation category; and (x) three cross-cutting (i.e., placeable in more than one phase) categories related to testing (where mapping to specific testing activities was not possible), release preparation, and other plugins.

After the first categorization, subsequent iterations produced the following changes:

- *Dependencies* were initially mapped to *Compilation*. Then, we realized that it is not possible to generally map dependencies to the compilation phase (i.e., dependencies related issues impact on several phases). Therefore, we created a crosscutting category for *Dependencies*.
- Some tools, such as *Grunt*, *Cargo*, and *MojoHaus*, were initially grouped as *Other Plugins*. Then, we decided that, due to the nature of the tasks they perform (e.g., allowing the execution of external commands), it was more appropriate to group them together with *Ant* as *External Tasks*.
- The goal “prepare” of the *maven-release-plugin* was initially put into *Validation*. However, we realized that it is not bound exclusively to the “validate” phase, but it covers all phases prior “deploy”. Therefore, we placed this goal into the *Release Preparation* category.
- The “installation” phase was considered as a sort of “local” deploy, therefore we created a category named *Deployment* with two sub-categories related to *Local Deployment* and *Remote Deployment*.
- We decided to keep *Clean* separate from *Validation*, as Maven separates the cleaning of all resources generated by the previous build from the default

lifecycle. Instead, as Maven provides a dedicated clean lifecycle, we created a specific *Clean* category.

- For some goals (e.g., “migrate” of *org.flywaydb:flyway-maven-plugin*) we decided to create a category *Support*, as they have a supporting nature for the build process. Then we grouped those goals supporting a specific phase of the build into a subcategory of the related category.
- The *Static Analysis* category was renamed into *Code Analysis* and split in two sub-categories, namely *Static Code Analysis* and *Dynamic Code Analysis*.
- We created a category *Testing* hosting test execution involving *Unit testing*, *Integration testing*, and *Non-functional testing*. Due to missing goals we were not able to classify some failed tests into one of these three sub-categories, thus we assigned them to the crosscutting sub-category *Crosscutting*.

Once we had adapted our initial categorization as described, we automatically classified the logs, sampled the logs (764 ING and 377 OSS) to be manually validated, and proceeded with the validation. The validation of ING build failures was completed with an inter-rater agreement of $k = 0.8$ (strong agreement) and of OSS build failures with $k = 0.62$ (again, strong agreement). It is particularly important to discuss the 51 cases (6.7%) for which there was a consistent disagreement with the automatic categorization of ING builds. This was related to failures raised by the *org.apache.maven.plugins:maven-surefire-plugin*, initially assigned to the *Unit Testing* category. However, we found that the plugin name was usually followed by a label “(default-test)” or “(integration-testing)”, the former suggesting it was indeed unit testing, while the latter pertaining to integration testing. Therefore, we decided to rematch the latter. Since this was just a matter of refining the regular expression, a new validation of the *Unit Testing* and *Integration Testing* was not necessary, but this information was used in the classification of the OSS build failures.

⁸http://www.ifi.uzh.ch/seal/people/vassallo/build_failures_catalog.pdf

Table 4.1: Build failure categories

Category	Subcategory	Description
CLEAN		Cleaning build artifacts
VALIDATION		Check on the project's resources
PRE-PROCESSING (RESOURCES)		Generation and processing of the project's resources
COMPILATION	PRODUCTION	Compilation of production code
	TEST	Compilation of test code
	SUPPORT	Code manipulation & processing
TESTING	UNIT TESTING	Running unit tests
	INTEGRATION TESTING	Running integration tests
	NON-FUNCTIONAL TESTING	Running load Tests
	CROSSCUTTING*	Crosscutting test failures
PACKAGING		Packaging project artifacts
CODE ANALYSIS	STATIC	Code analysis (without executing it)
	DYNAMIC	Code analysis (by executing it)
DEPLOYMENT	LOCAL	Project's artifacts installation
	REMOTE	Project's artifacts deployment to a remote repository
EXTERNAL TASKS		Capability for calling other environments
DOCUMENTATION		Documentation generation and packaging
RELEASE PREPARATION*		Preparation for a release in SCM
SUPPORT*		Database migration and other activities
DEPENDENCIES*		Project's dependencies resolution and management

The Build Failure Categories

Table 4.1 reports the final version of the catalog we devised. It is made up of 20 categories (including sub-categories) and based on 171 keywords.⁸ Crosscutting categories are tagged with an asterisk, e.g., *Dependencies**. In the following, we will briefly describe each category, in terms of included goals/keywords and corresponding standard Maven or new phase, with qualitative insights about the logged errors.

Clean. This category includes builds failed while executing the Maven lifecycle goal “org.apache.maven.plugins:maven-clean-plugin:clean”; it tries to remove all files generated during the previous build.

Validation. Our *validation* category is mapped to the Maven *validation* phase that aims to validate a project by verifying that all necessary information to build it is both available and correct. Indeed, the goals included in this category check Maven classpaths (the main purpose of “org.apache.maven.plugins:maven-

enforcer-plugin:enforce”) or environment constraints, such as Maven and JDK versions.

Pre-Processing. This category contains all failed goals related to the generation of additional resources typically included in the final package (corresponding to Maven phases *process-resources* and *generate-resources*). For example, the goal “org.apache.maven.plugins:maven-plugin-plugin:helpmojo” generates a “HelpMojo” class, corresponding to an executable Maven goal used in the subsequent steps of the build process. Other goals, such as “com.simpligility.maven.plugins:android-maven-plugin:generate-sources”, generate source code (in this case R.java) based on the resource configuration parameters.

Compilation. Build failures in this category are mainly caused by errors during the compilation of the production and test code (respectively *compile* and *test-compile* in the Maven lifecycle). This leads to two sub-categories: (i) failures related to the compilation of production code, and (ii) failures related to the compilation of test code.

Production. The compilation of production code can fail due to typical programming errors that are detected by the compiler while running goals such as “org.apache.maven.plugins:maven-compiler-plugin:compile”, but also because of language constructs unsupported by the build environment.

Test. Test code compilation failures are similar to production code ones, although we noticed many failures due to wrong exception handling in test code (e.g., “org.apache.maven.plugins:maven-compiler-plugin:testCompile” failed because “unreported exception must be caught or declared to be thrown”).

Support. In addition to the previous compilation sub-categories, we added another one related to failures occurring during activities complementary to standard compilation. Examples of these activities are represented by the goal “org.bsc.maven:maven-processor-plugin:process” that processes annotations for jdk6, or the goal “net.alchim31.maven:yuicompressor-maven-plugin:compress” that performs a compression of static files.

Testing. This category includes the execution of unit, integration, and non-functional system tests. Moreover, we identified multiple failed builds for which we were not able to classify them into one of these three sub-categories, hence the crosscutting category *Crosscutting*.

Unit Testing. The Maven phase *test* directly corresponds to this category. Builds fail while executing goals such as “org.apache.maven.plugins:maven-surefire-plugin:test”. Those failures are related to the presence of failing test cases. Also other issues raised, e.g., the goal execution fails with an error message that specifies the presence of unit tests which invoke `System.exit()`, or a crashing virtual machine.

Integration Testing. Integration test results are typically verified by means of the *Failsafe* Maven plug-in, which has a goal “integration-test” producing the error message “There are test failures” in case that tests fail. This category corresponds to the Maven phases *pre-integration-test* and *integration-test*. Moreover, we found that integration testing is often performed using the goal “test” of the *Surefire* plugin, even if the latter is mainly intended to be used to execute unit tests.

Non-Functional System Testing. While there is no corresponding Maven phase, the single failing goal in this category is “io.gatling:gatling-maven-plugin:execute”. This goal launches *Gatling*, a load testing tool, that keeps track of load testing results across builds. As Gatling accepts *Scala* code, some build failures have occurred because of incompatibilities between Gatling and specific Scala versions, e.g., Gatling 2.1 and Scala versions prior 2.11.

Crosscutting Tests. There are testing failures that we could not assign to a specific category because builds ended without reporting the failed goal. The reported message (“There are test failures”) highlights the presence of test failures and is very similar to the one that occurred for unit and integration testing. Moreover, such test failures could occur within various testing related phases of the build process.

Code Analysis. Similar to *Non-Functional System Testing*, we could not identify a Maven phase related to the failed goals of the *Code Analysis* category. Failed goals in this category can be subdivided into *Static* and *Dynamic*.

Static. Static code analysis [9] within the build process is conducted by running goals such as “org.codehaus.mojo:sonar-maven-plugin:sonar”, which launches the analysis of source code metrics via SonarQube, and “org.codehaus.mojo:findbugs-maven-plugin:findbugs”, which is used to inspect Java bytecode for occurrences of bug patterns via FindBugs.

Dynamic. Dynamic code analysis is performed by executing goals such as “org.codehaus.mojo:cobertura-maven-plugin:instrument”, which instruments the classes for the measurement of test coverage (with Cobertura).

Many goals included in the *Code Analysis* category failed because of (failed) quality checks, or in case of SonarQube, because of connection timeouts, e.g., “server can not be reached”.

Packaging. This category concerns all the builds failed while bundling the compiled code into a distributable format, such as a JAR, WAR, or EAR (Maven *prepare-package* and *package* phases). This category includes goals such as “org.apache.maven.plugins:maven-war-plugin:war”. There are several errors underlying these failed goals, such as the presence of a wrong path pointing to a descriptor file or non-existing files (e.g., “The specified web.xml file does not exist”).

Deployment. We observed two types of deployment: local and remote.

Local. This sub-category is mainly related to the *install* phase of the standard Maven lifecycle, in which the build adds artifacts to the local repository (e.g., by “org.apache.maven.plugins:maven-install-plugin:install”). To this end, the build process, using the information stored in the POM file, tries to determine the location for the artifact within the local repository. Failures concern the impossibility to find and parse the needed configuration data.

Remote. The sub-category *Remote* corresponds to the Maven phase *deploy* and includes goals, e.g., “org.apache.maven.plugins:maven-deploy-plugin:deploy”, which try to install artifacts in the remote repository. Failures are often due to wrong server URLs and authentication credentials. Other cases include the unsuitability of the specified repository for deployment of the artifact or a not-allowed redeployment of the same artifact.

External Tasks. This category includes failures caused by the usage of external tools scheduled to execute within the build process. Some failures are related to the execution of Ant tasks (e.g., “org.apache.maven.plugins:maven-antrun-plugin:run”) or SQL statements (e.g., “org.codehaus.mojo:sql-maven-plugin:execute”). We also added goals to this category that have the task of manipulating application containers and allowing the execution of external Java programs (e.g., “org.codehaus.mojo:exec-maven-plugin:exec”) from a POM file. Many errors re-

ported by these goals are related to timeout problems (e.g., “Execution start-container:start failed: Server did not start after 120000 milliseconds”), but there are also failures related to erroneous environment specifications, e.g., wrong port numbers.

Documentation. Documentation is mapped onto the phase *site* of the Maven Site lifecycle. It concerns build failures occurring during the generation of documentation, e.g., using the Javadoc tool. Reasons for those failures include the specification of wrong target directories, incompatibilities between the JDK and the Javadoc generation tool, or syntax errors in the Javadoc comments. Moreover, failures are caused by the inability to create an archive file from the previously generated Javadocs (“Error while creating archive”).

Crosscutting Categories. We will now discuss failure categories that can not be associated with one specific phase, but can rather be mapped onto several phases.

Release Preparation. This category concerns failures occurring during the preparation for the deployment of a packaged release. We included in this category the goal “org.apache.maven.plugins:maven-release-plugin:prepare”, that is used to (i) check the information regarding the current location of the project’s Source Configuration Management (SCM) and whether (ii) there are no uncommitted changes in the current workspace. There are several reasons for failures in these tasks, including failed executions of SCM-related commands (e.g., Git commands), or the presence of uncommitted changes. In many cases, failed builds simply report the name of the failed goal without specifying additional information.

Support. Builds fail while executing tasks that are not scheduled to execute within an usual build process. This category includes goals such as “org.flywaydb:flyway-maven-plugin:migrate”, which is used for database schema migration. As they are not used in a regular build process, their categorization into a specific phase (e.g., *Support* of *Compilation*) is difficult.

Dependencies. This category contains goals such as “list” and “copy” of the “org.apache.maven.plugins:maven-dependency-plugin” plugin. We also put there the keywords “Could not resolve dependencies” and “Failed to resolve classpath resource”, which are used to catch dependency-related failures when there is no further information about the failed goal. Typical errors occurring in this category

are invalid resource configurations in the POM file, or failed downloads due to unavailable artifacts. As is the case for all crosscutting categories, this category of failures can occur in each phase of the build process. It is possible that, in order to execute the test phase or a static analysis check, the build process needs to use (and possibly download) the proper plugins via dependency resolution.

Answer to RQ₁

Our build failure catalog comprises 20 categories. 3 are related to Compilation, and 6 to specific Testing and Code Analysis activities. Release preparation and Deployment represent 2 separate categories, and the remaining are related to other build activities (e.g., Packaging).

4.3.2 How Frequent Are the Different Types of Build Failures in the Observed OSS and ING Projects?

Percentage of Failing Builds. For both the OSS and ING analyzed projects, the overall percentage of failing builds during the period of observation is 26%. In contrast, Kerzazi et al. [50] and Miller [75] observed lower percentages of 17.9% and 13% respectively; Seo et al.'s study at Google revealed a rate of 35% failing builds.

Fig. 4.1 provides, for both ING (blue bar) and OSS projects (yellow bar), a break-down of the build failures across the (sub-)categories identified in RQ₁. Note that we performed this classification on $3,390 - 779 = 2,611$ ING and $30,792 - 9,774 = 21,018$ OSS build failure logs for which we were able to mine sentences to support the identification of the causes of failures (e.g., failed goal or any of the identified regular expressions). In 779 cases for ING and in 9,774 cases for the OSS projects, we were not able to classify the failure based on the information contained in the log. However, the percentages in Fig. 4.1 are related to the original set of failures (including non-classified build failures).

By observing Fig. 4.1, we can notice that the percentage of *compilation*-related failures is fairly limited, for both *production* code (4.2% ING and 7.1% OSS)

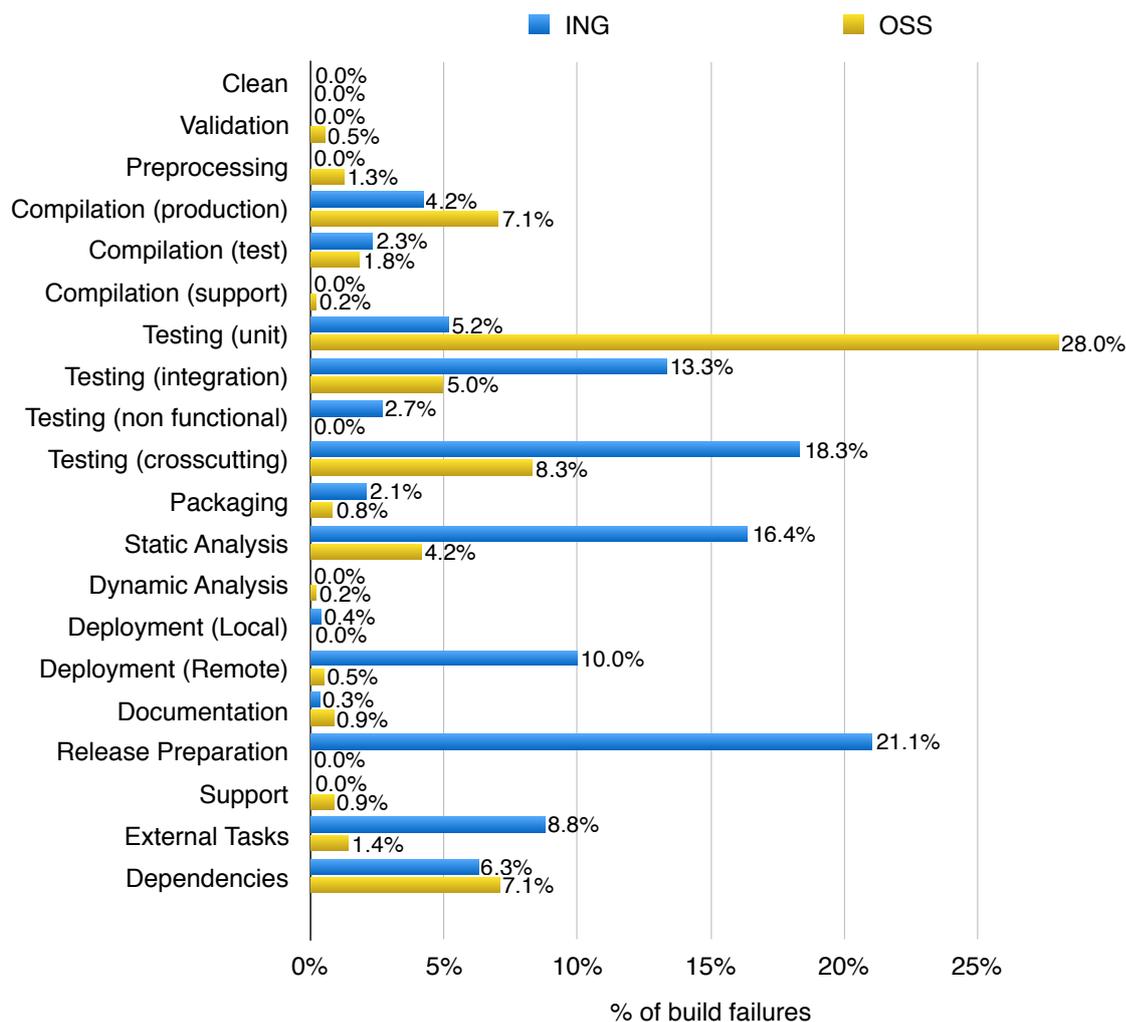


Figure 4.1: Percentage of build failures for each category

and *test* code (2.3% ING and 1.8% OSS). This is below the 26% observed by Miller [75]. In the case of ING, one possible reason is that — as also confirmed by a survey conducted within ING [135] — private builds (i.e., builds executed on the developer’s machine) are used to limit the number of build failures due to compilation errors.

Interesting considerations can be drawn for build failure categories pertaining to testing. The *unit testing* category is the one for which we can notice, on the one hand, the highest percentage of build failures for OSS (28%, perfectly in line with

the percentage observed by Miller [75] at Microsoft), and, at the other hand, a large difference with respect to the relatively low percentage (5.2%) observed for ING projects. Conversely, we observe a relatively high percentage of *integration testing* failures in ING (13.3%) and only 5.0% for OSS. When we compare this insight with the one reported above about unit testing, we can confirm the findings of Orellana et al. [68] indicating that in OSS projects unit testing failures dominate integration testing failures. We can also see how ING projects are affected more by integration testing than unit testing failures. Previous research [135] indicates that unit tests are often executed within private builds in ING, therefore the number of remaining failures discovered on the CI server turns out to be fairly limited. We found no evidence of *non-functional testing* failures in OSS projects, while there is a relatively low percentage of them (2.7%) in ING. This low percentage of failures was surprising, as the CD pipeline used in ING explicitly perform non-functional testing (including load testing) at a dedicated stage (i.e., not during the build, as reported in a survey conducted in ING [135]). We deduced that in ING a preliminary load testing is performed at the build stage (by means of Gatling⁹), with the aim of an early and incremental discovery of bottlenecks. Finally, we found a relatively high number of *crosscutting* test failures (18.3% for ING and 8.3% for OSS) not attributable to specific testing activities.

Static analysis tools are also responsible for a relatively high percentage (16.4%) of build failures in ING; Miller [75] observed a substantially higher percentage of 40%, without discussing specific reasons. In contrast, for our OSS projects, the percentage is 4.2%, similar to the observation of Zampetti et al. [142] who found percentages almost always below 6%. One may wonder whether the higher percentage of static analysis related failures in ING are caused by stricter quality checks. We have no evidence of this, as we had no access to the SonarQube entries related to the build failures. Instead, OSS projects used static analysis tools on the CI server (without running them on a dedicate server as in ING) and the results are usually visible in the build logs. For this reason, a manual scrutiny of some build failure logs confirmed that, in such cases, build failures were indeed due to specific warnings raised by static analysis tools.

⁹<https://gatling.io/>

For ING projects we perceive a high percentage of *release preparation* problems (21.1% of the total). This can be attributed to the way ING handles the deployment of a new application: it relies on the standard Maven process instead of using a combination of different goals, as it is common in OSS projects. Also, for *deployment*, the percentage of build failures that occurred in ING (10.0%, higher than the 6% that Miller reported [75]) is much higher than for OSS (0.5%). As discussed in Section 4.3.1, such failures are mostly due to mis-configurations for accessing servers (i.e., wrong server's IP address) hosting the application artifacts. These mis-configurations could potentially be very costly. Indeed, earlier research in the area of storage systems [140] has shown that 16.1%–47.3% of mis-configurations lead to systems becoming either fully unavailable or suffering from severe performance degradations. The *packaging* category exhibits a relatively low percentage of build failures, but still much more frequent in ING (2.1%) than for OSS (0.8%). In summary, release preparation, packaging and deployment have quite different trends in ING and OSS. In ING, the CD machinery is massively used to produce project releases and deploy them on servers, and in particular on servers where further quality assurance activities (i.e., load and security testing) occur before the release goes in production. Instead, we assume that in most of the studied OSS projects this rarely happens (but it is not excluded), as the main goal of CI is to make a new release available for download on GitHub.

External tasks are responsible for respectively 8.8% (ING) and 1.4% (OSS) of the build failures. The use of external tasks (e.g., the execution of Ant tasks in a Maven build) might make the build process more complex and possibly difficult to maintain, e.g., when one has to maintain both Maven and Ant scripts. While there is no evidence that build maintenance is related to the increase of build failures, it is a phenomenon to keep into account, e.g., by planning, whenever possible, build restructuring activities.

Dependency-related failures exhibit similar percentages for ING and OSS (6.3% and 7.1% respectively). Finally, we found a very small failure percentage ($< 1.5\%$) for other categories, such as clean, validation, pre-processing, compilation support, documentation, and support (crosscutting) both in the case of ING and OSS projects.

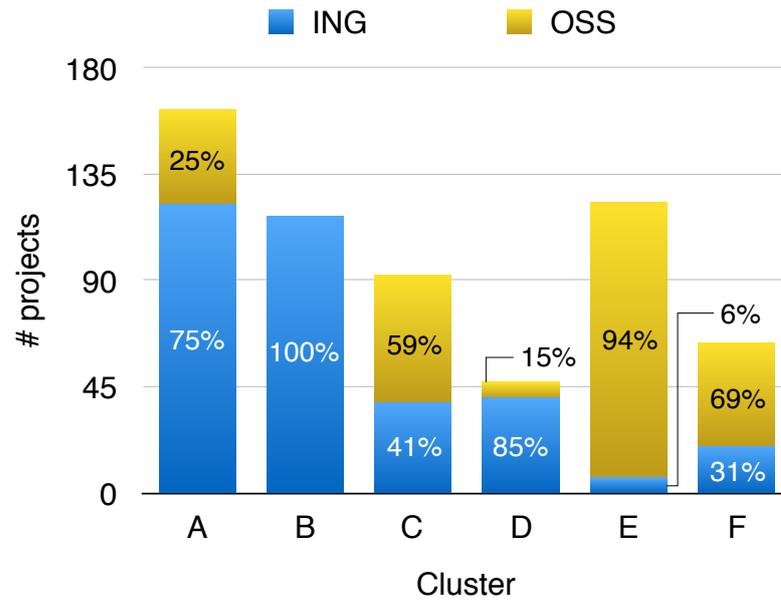


Figure 4.2: Clusters of projects based on the distribution of build failures

Clustering Based on Build Failure Percentages. As explained in Section 4.2, to better investigate the differences in the distribution of build failures in ING and OSS projects, we clustered the whole set of projects (both OSS and ING) using the *K-means* algorithm [67]. While the first analysis of Fig. 4.1 provides a broad overview of build failure distribution in the entire context (ING or OSS), the clusters help to understand the extent to which there are projects (within ING, within OSS, or in both environments) exhibiting certain distributions of build failures and, in general, to investigate the build failure distribution on a single project level.

We modelled the projects as vectors where each dimension is a category of our taxonomy (Table 4.1). Note that only for Testing we considered the sub-categories as dimensions, because they lead to a significantly different distribution of build failures. Then we computed for each project the percentage of its build failures belonging to each category, and assigned values to the related vector dimension. Finally, we applied the *K-means* algorithm several times in order to find the optimal value of Silhouette statistics.

Fig. 4.2 depicts the clusters' composition for $k = 6$ (where k is the number of clusters), which corresponds to the optimal value of Silhouette. The clusters can

Table 4.2: Distribution of build failure types in each cluster

Class	A	B	C	D	E	F
Clean	0%	0%	0%	0.02%	0.01%	0%
Validation	1.46%	0%	0.78%	0%	0.23%	0.38%
Pre-Processing	0.73%	0%	8.62%	0%	0.49%	0.22%
Compilation	6.06%	2.79%	9.57%	1.42%	12.07%	74%
Unit Testing	4.05%	3.02%	7.16%	0.67%	72.81%	15.12%
Integration Testing	7.52%	0.54%	2.03%	0.06%	0.70%	0.28%
Non Functional Testing	0%	0%	9%	0%	0%	0%
Crosscutting Testing	2.57%	0.19%	2.5%	93.54%	0.42%	1.53%
Packaging	1.16%	2.93%	5.88%	1.10%	1.71%	0.87%
Code Analysis	32.12%	0.24%	4.59%	0.64%	1.64%	1.03%
Deployment	22.20%	5.01%	1.93%	0.79%	1.96%	0.83%
External Tasks	14.29%	3.77%	0.88%	0.33%	1.02%	0.14%
Documentation	1.75%	0%	1.23%	0%	0.98%	1.21%
Release Preparation	3.66%	80.11%	3.15%	1.18%	0.26%	0.23%
Support	0.57%	0.17%	3.80%	0%	0.63%	0.08%
Dependencies	1.67%	1.23%	38.85%	0.22%	5.07%	4.09%

be interpreted by looking at Table 4.2, showing for each cluster the percentage of build failures of its mean point (as computed by the k-means algorithm).

Three main clusters can be observed: A, B and E (containing 27%, 19% and 20% of the projects respectively). All projects in Cluster B come from ING, while Cluster E includes almost exclusively OSS projects. Cluster A contains again a high percentage of ING projects (75%).

Cluster A includes projects that fail mostly because of Code Analysis (32%), Deployment (22%) and External Tasks (14%), while B exhibits mainly build failures belonging to Release Preparation (80%). The last result is not surprising since, as shown in Fig. 4.1, only ING builds are affected by this type of failures.

Cluster E contains projects that typically fail because of Unit Testing (on average the 73% of the build failures belongs to this category) and this justifies that within the cluster we mainly found OSS projects.

Cluster C is balanced (59% OSS projects vs. 41% ING projects) and contains projects mainly exhibiting dependency failures, while the projects in Cluster D are mainly ING and exhibits mostly Crosscutting Testing build failures for which we cannot specify the type of testing involved.

Cluster F is relatively small (64 projects) and catches only projects that usually fail for compilation errors (74%).

In summary, the clustering analysis highlights groups of projects exhibiting similar characteristics in terms of build failure distributions. Specifically, some clusters are mainly constituted by ING (only) or OSS projects, as certain failures dominate in one case or the other, while there are also some clusters that are almost equally distributed. Also, we noticed how some clusters group together projects mainly exhibiting one specific kind of build failure.

Answer to RQ₂

The OSS and ING projects exhibit a different distribution of build failure types. Overall, in OSS projects build failures happen mainly due to unit testing failures, while ING projects fail, above all, because of release preparation failures. Finally, the clustering analysis of the projects (based on the distribution of build failure types) points out how projects from the two different contexts in some cases spread out into different clusters.

4.4 Discussion

In this section we discuss the main findings and their implications for future research. Specifically, given the differences and commonalities that we observed in RQ₂, some CI practices deserve additional investigation.

Compilation Errors Are Typically Fixed in Private Builds Compilation failures are considered to be particularly relevant during the build process, such that the study of Seo et al. [107] conducted at Google focused entirely on that. Our study reports a small (6.5% in ING and 9% in OSS), yet non-negligible percentage of compilation-related build failures. On the one hand, our study confirms that compilation success is a key prerequisite for code promotion (as stated in [135]), and therefore mostly achieved in private builds. On the other hand, even such low percentage highlight how CI can still be beneficial to spot compilation errors due to the adoption of anti-patterns – e.g., when a change is pushed without compiling

it – or due to the usage of different development environments, e.g., incompatibility between the JDK versions installed on the CI server and the local machine.

OSS Projects Run Every Test on CI Servers, ING Mostly Integration Testing Integration testing failures are more frequent in ING than in OSS. Instead, and consistently with a previous study by Orellana Cordero et al. [84], OSS projects exhibit more unit testing related failures. This indicates a different distribution of testing activities across the development process in the open source and industrial context. In OSS projects, developers often rely directly on the CI server to perform testing (as the very high number of unit testing failures may suggest). In ING, a conservative strategy is preferred, revealing unit testing failures before pushing changes on the server (by running unit tests on local machine), and referring to the build server mostly to verify the correct integration of the changes made by different developers. This is also confirmed by results of a previous survey with ING DevOps [135].

Early Discovery of Non-functional Failures in ING For large and business-critical projects like the one used by ING for their online banking, appropriate non-functional system testing is crucial. As explained in Section 4.2, this is mostly done offline, on a separate node of the CD pipeline. Nevertheless, as our study shows, developers in ING rely on the build process to spot, whenever possible, non-functional issues, and specifically load test failures. While this happens in a relatively small percentage of the observed build failures (2.1%), it suggests that the early discovery of some problems during the CD process (i.e., as soon as a change is pushed) could save time to solve some performance bottlenecks that would otherwise be discovered at a later stage only. We have no evidence of this activity in the OSS builds.

Release Preparation and Deployment Failures Are Very Common in Industry, Less so in OSS We noticed how release preparation errors are very frequent in ING (21.2% of our build failures are associated to this category). At the same time, we did not find such evidence in OSS projects. In ING, the CI process is built using fewer steps (i.e., Maven goals). Developers prefer to use (when possible) predefined bundled steps (as in “prepare” goal, that “covers” several default Maven lifecycle stages), instead of adopting a combination of different goals (each covering a single stage) at least for the most critical stages of the build process (i.e., all

stages before the deploy of a new release). Deployment errors are also conspicuous (10%), and mostly due to mis-configurations. For OSS projects, in most of the cases, there is no real deployment of a new release, it is just a matter of making the new release available on GitHub, and using CI for assessing and improving its quality.

Static Analysis (SA) Tools: on CI Server in OSS, Remotely in ING

Our results indicate an intensive percentage of failures related to static analysis tools in ING (16.4%) compared to OSS (4.2%). Looking at the failed goals in the build process of projects in both contexts, we noticed how OSS developers prefer to run SA tools on the build server while in ING SA tools are run on a different server (via SonarQube). This choice did not necessarily lead to less build failures (we noticed more static analysis related failures in ING), but it is an indication about the willingness of ING to have i) well collected data, easy to query and monitor, and ii) a separate analysis of code smells without overloading the CI server.

4.5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. The most important threat of this type in our study is due to our limited observability of the ING build failures (i.e., we had to rely on build failure logs only). For example, we could not perform a fine-grained classification of some testing or static analysis failures whose nature was only visible from separate reports.

Another threat is due to possible mis-classification of build failures, given that the classification is only based on failed build goals and, in some cases, on keyword matching. As explained in Section 4.2, we mitigated this threat by performing a manual validation on a statistically significant sample of ING build failures, mainly aimed at verifying that such a matching was correctly performed, and reported results of such a validation. We checked whether the agreement of such a validation was due to chance by using the Cohen k inter-rater agreement. As for OSS failures, we repeated the process, however on only 377 build failures in total, as it was not a complete new validation of the taxonomy, but rather a check of its validity when applied to OSS projects.

Threats to *internal validity* concern internal factors of our study that could influence our results. The most important threat of this class is related to the subjectiveness likely introduced when devising the build failure catalog. We limited such a subjectiveness in different ways, i.e., by performing multiple iterations conducted by different authors independently, and by using the Maven standard lifecycle as a roadmap for creating such a catalog. For ING projects, we could not observe and control the projects' programming languages (although an ING team leader reported us that are mostly written in Java). Another threat is related to the choice of the number of clusters in \mathbf{RQ}_2 . We mitigated this threat by using the silhouette statistic [54] to support our choice. Furthermore, while the number of OSS and closed-source projects is comparable, the number of build failures for closed-source is one order of magnitude greater than for OSS, and this sample imbalance could have affected our results.

Threats to *external validity* concern the generalization of our findings. On the one hand, results related to our closed-source sample are necessarily specific to ING. It is possible that these results partially generalize to other organizations in the financial domain, but this has not been specifically validated in our work. In Section 4.4, we discussed in how many cases our findings are consistent with those of previous studies, and also in which cases we obtained different results. OSS results, on the other hand, are representatives of OSS Java-based projects using Maven and relying on the Travis-CI infrastructure. In some cases (e.g., for testing), we found confirmations of results from previous studies. Our study does not necessarily generalize to applications built using other languages, build scripts, and different CI infrastructures. Finally, it is possible that the different domain of closed-source projects and open source projects could have impacted our results.

4.6 Related Work

Ståhl et al. report that CI is becoming increasingly popular in software development [112]. With this continued uptake in industry, researchers have also focused more on build systems, and more specifically on build failures.

Build Failures. Miller's seminal work [75] on build failures in Microsoft projects describes how 66 build failures were categorized into compilation, unit

testing, static analysis, and server failures. Our observation is larger (18 months vs 100 days, 13k builds vs 515, and 3,390 build failures vs 66), and describes a more detailed categorization of build failures, but covers a different domain. Rausch et al. [100] studied build failures in 14 popular open source Java projects, finding that most of the failures (> 80%) are related to failed test cases, and that there is a non-negligible portion of errors due to Git interaction errors. Seo et al. [107] conducted a study focusing on the compiler errors that occur in the build process. They devised a taxonomy of compilation errors that lead to build failures in *Java* and *C++* environments. Kerzazi et al. [50] analyzed 3,214 builds in a large software company over a period of 6 months to investigate the impact of build failures. They observed a high percentage of build failures (17.9%) that brings a potential cost of about 2,035 man-hours considering that each failure needs one hour of work to succeed.

Beller et al. [10] focused on testing with an in-depth analysis of 1,359 projects using both *Java* and *Ruby* programming languages. Testing is the main activity responsible for failing builds (59% of build failures during test phase for Java projects). While our results confirm their finding, we also highlight the importance of failures due to other tasks. Orellana Cordero et al. [84] studied test-related build failures in OSS projects. They identified that unit test failures dominate integration test failures. Our results for OSS projects are similar, yet our findings for ING projects are the opposite. This may be due to more intensive usage of private builds to deal with unit tests in an industrial environment, which is not the case for the OSS projects studied by Orellana Cordero et al. [84]. Zampetti et al. [142] looked at build failures (mostly related to adherence to coding guidelines) produced by static analysis tools in Java-based OSS projects. Our work found a high percentage of build failures due to static analysis (for ING projects), although in most cases due to infrastructure (Jenkins and SonarQube) mis-configuration.

Build Activities. McIntosh et al. [71] studied the relationship between changes to production code, test code, and the build system. They noticed that a strong relation between changes made at all three levels. McIntosh et al. [72] studied version histories of 10 systems to measure the overhead that build system maintenance imposes on developers. Finally, Desarmeaux et al. [21] investigated how build maintenance effort is distributed across the build lifecycle phases of

systems built through Maven. They observed that the compile phase requires most maintenance activity.

Continuous Integration (CI) and Continuous Delivery (CD). Hilton et al. [46] investigated why developers use or do not use CI, concluding that this concept has become increasingly popular in OSS projects and then [45] presented a qualitative study of the barriers and needs developers face when using CI. Ståhl et al. [113] noticed that there is no homogeneous CI practice in industry. They identified that there are many variation points in the usage of the CI term. Not only CI practices vary between different industries, we also identified noticeable differences between OSS and industry projects. Vassallo [135] investigated CD practices in ING focusing attention on how they impact the development process and the management of technical debt. Conversely, Savor et al. [103] reported on an empirical study conducted in two high-profile Internet companies. They noticed that the adoption of CD does not limit the scalability in terms of productivity of one organization even if the system grows in size and complexity. Schermann et al. [105] derived a model based on the trade-off between release confidence and the velocity of releases. Finally, Schermann et al. [106] investigated the principles and practices that govern CD adoption in industry and concluded, amongst others, that architectural issues are one of the main barriers for CD adoption.

4.7 Conclusion

This paper investigates the nature and distribution of Continuous Integration (CI) build failures occurring in 418 Java-based projects from ING (an organization in the financial domain), and in 349 Java-based OSS projects hosted on GitHub and using Travis CI as CI infrastructure. The results of our study highlight how OSS and ING projects exhibit substantially different distributions of build failure types, confirming but also contradicting some of the findings of previous research which are based on OSS projects' data or only data from a single industrial organization. Our findings are important for both researchers and practitioners since they shed some more light on the differences and commonalities of CI processes adopted in the analyzed OSS projects and the observed industrial organization highlighting interesting build failure patterns.

Work-in-progress aims at replicating the study in other industrial environments and further open source projects, and at performing a deeper analysis on the build failures observed, e.g., studying the difficulty in fixing different kinds of problems. Additionally, we plan to use our results to aid developers to properly maintain build process pipelines to make it more efficient, e.g., by deciding what to do in private builds on the developer's local machine and what to delegate to CI servers, or how to mitigate problems by conceiving approaches able to automate their resolution.

Acknowledgements

The authors would like to thank developers from ING that provided precious inputs during this study.

Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution

Carmine Vassallo, Sebastian Proksch, Timothy Zemp, Harald C. Gall
Published in Springer Empirical Software Engineering (EMSE), Volume 25, Pages
2218–2257, 2020

Contribution: prototype design, experiment design, survey design, data analysis, and paper writing

Abstract

Continuous integration is an agile software development practice. Instead of integrating features right before a release, they are constantly being integrated into an automated build process. This shortens the release cycle, improves software quality, and reduces time to market. However, the whole process will come to a halt when a commit breaks the build, which can happen for several reasons, e.g., compilation errors or test failures, and fixing the build suddenly becomes a top priority. Developers not only have to find the cause of the build break and fix it, but they have to be quick in all of it to avoid a delay for others. Unfortunately, these steps require deep knowledge and are often time-consuming. To support developers in fixing a build break, we propose Bart, a tool that summarizes the reasons for Maven build failures and suggests possible solutions found on the internet. We will show in a case study with 17 participants that developers find Bart useful to understand build breaks and that using Bart substantially reduces the time to fix

a build break, on average by 37%. We have also conducted a qualitative study to better understand the workflows and information needs when fixing builds. We found that typical workflows differ substantially between various error categories, and that several uncommon build errors are both very hard to investigate and to fix. These findings will be useful to inform future research in this area.

5.1 Introduction

Continuous integration (CI) is an agile software development practice that advocates frequently integrating code changes introduced by different developers into a shared repository branch [47]. An automated system builds every commit, runs all tests, and verifies the quality of the software, e.g., through automated static analysis tools [25]. This helps to detect issues earlier and locate them more easily [50]. CI is widely adopted in industry and open source environments [136] and has already proven its positive effects on release frequency, software reliability, and overall team productivity [46].

Despite its undisputed advantages, the introduction of CI in established development contexts is anything but trivial. Hilton et al. [45] found that *build breaks* are a major barrier that hinders CI adoption and various reasons exist for a build to break [134], e.g., compilation errors, testing failures, poor code quality, or missing dependencies. Developers need to learn how to efficiently identify the reasons for a build break and, unfortunately, the required skill set is still different to traditional debugging. Established techniques that are widely used in the development environment [81], like setting breakpoints to investigate a program right before a crash, are not applicable, which makes it difficult and time consuming to remove a build break [45]. As a result, developers spend a significant amount of their working time comprehending and solving build breaks. It takes on average one hour to fix build breaks [50].

Those results motivate the need for new ways to support developers in *understanding* build breaks and in *deriving a fix*. Existing works have already proposed automatic build-fixing techniques, e.g., [66]. However, such approaches are typically limited to a specific type of build break (i.e., fixing unresolved dependencies). In this paper, we propose a *developer-oriented* assistance system that supports

build break fixes by summarizing available information and by linking to external information. We do not focus on a specific build problem, but empower the developer by providing relevant information in a wide range of build failures. To the best of our knowledge, we are the first to propose such an *information-centric* developer support during build breaks.

In the first part of this paper, we will investigate whether generated summaries can help developers with comprehending build logs. We will also empirically analyze the effect of a build summarization tool on the time needed for understanding and fixing a build break. More specifically, we will answer two research questions:

RQ₁: *Are summarized build logs more understandable?*

RQ₂: *Does a semi-automated support system influence the time that is required to fix a broken build?*

We have implemented the *Build Abstraction and Recovery Tool* (Bart) to study these questions. Bart is a Jenkins plugin that summarizes logs of failed Maven builds and that links related StackOverflow discussions to help solve the build failure. To answer both research questions, we deployed Bart in an empirical study with 17 developers. Our results show that developers consider the generated summaries helpful for fixing build breaks; as a further result, the resolution time for fixing the build can be significantly reduced.

However, the results also show that developers only find some solution hints valuable, other cases are perceived as less helpful. To understand this observation, we asked the following research questions and discuss the corresponding results in the second part of the paper.

RQ₃: *How do developers approach different types of build failures?*

RQ₄: *What types of build failures are hard to fix?*

To answer these questions, we have conducted a large-scale qualitative study that involved 101 developers. Our results show that fundamental differences exist in how developers approach the different failure categories that have been investigated. For example, while code-analysis build failures are easy to fix with information contained in the build log, other scenarios like testing failures require a much deeper investigation of information that needs to be collected elsewhere. Our survey has also revealed that problems related to the build infrastructure are not only hard to

investigate, but also hard to address, once the problem is understood. While these categories are known, they have not yet been the focus of an investigation, likely because they occur too infrequently. Our results suggest that they are still worth investigating, because they represent a major pain-point for developers when they occur.

In summary, this paper makes the following contributions:

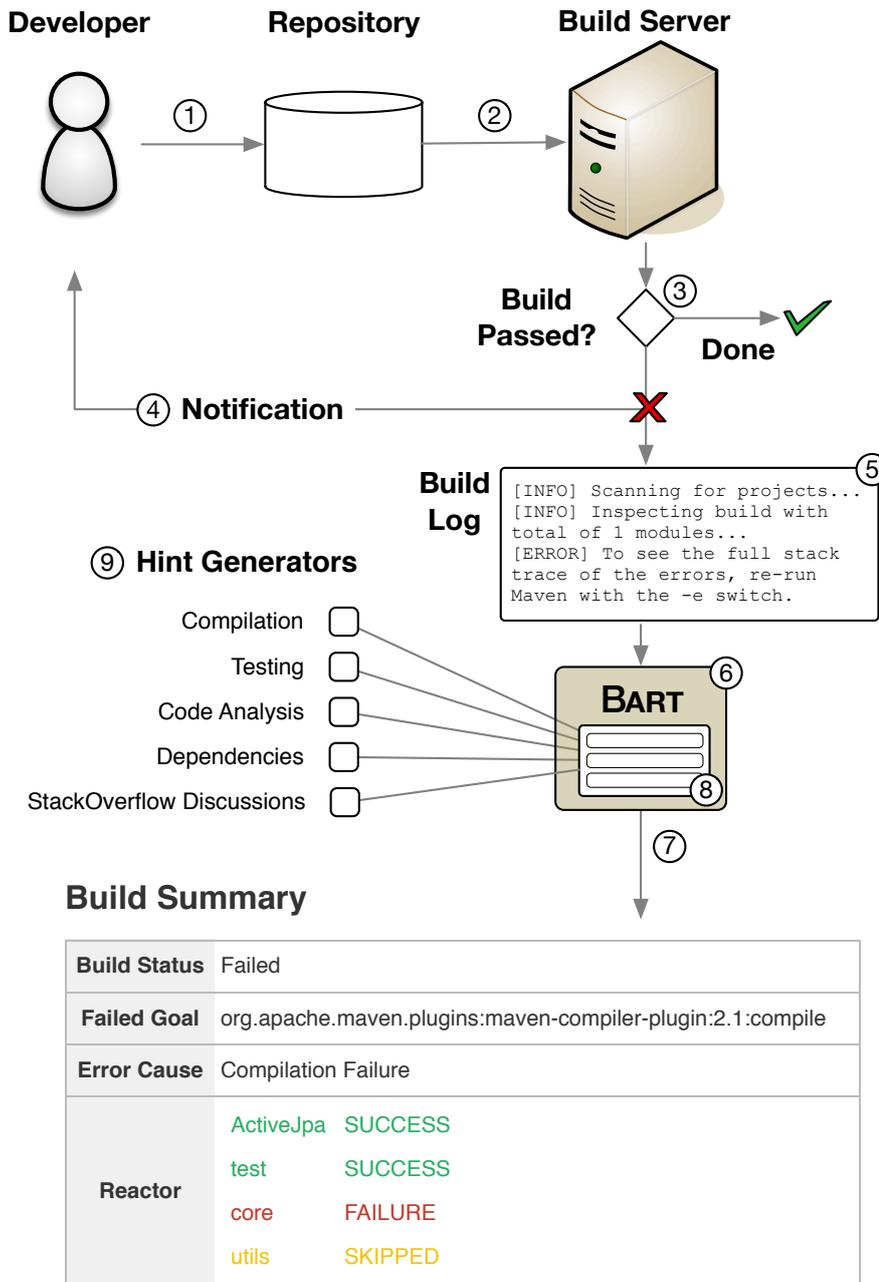
- Presentation of a novel idea to support build fixing through build log summarization and linking to StackOverflow resources.
- Proof-of-concept implementation for Bart.
- Investigation of the effect of Bart on the understandability of build failures and the time for fixing a build failure
- A qualitative analysis of the workflows and information needs of developers that resolve build failures.

This paper is an extension of previous work [131]. Compared to the original paper, it contains the following new contents.

- We have added two research questions that investigate qualitative aspects.
- We have doubled the number of the participants in the experiment to improve the evaluation of Bart and to make our findings more convincing.
- We have conducted 9 semi-structured interviews with several study participants to better understand information needs and the limitations of Bart.
- We have sent out a survey and analyzed 101 answers from a diverse set of developers to validate our interview results with a more general crowd.
- We have derived a methodology to open-code build fixing workflows.
- We have published Bart and open-sourced the implementation.¹

Scripts, data, and additional material are available in the online appendix [132].

¹<https://plugins.jenkins.io/bart>



Reason for Build Failure: *Compilation Failure*

Hint: Compilation

Your build contains a compilation error. Please check the following file:

File: Model.java
Line: 224
Reason: Model is abstract; cannot be instantiated

Figure 5.1: Overview of the Build Summarization approach

5.2 Creating a Build Abstraction and Recovery Tool (Bart)

To understand our vision of *developer-oriented* assistance, it is important to reflect on the typical CI pipeline that is illustrated in Figure 5.1. Developers working in such a pipeline synchronize their *working copy* frequently with the central repository that is shared by all team members (1). They pull changes from others and push their own contributions. The repository is being monitored by the build server. Every time a new commit is pushed to the repository, the build server will update its working copy and build the project (2). This typically includes multiple build stages, for example, compiling the sources, running the tests, generating documentation, or validating the software quality through static analysis. If all these stages have passed (3), the build is considered to be successful, which typically results in the release of the software. If the build fails, on the other hand, developers are being notified by the build server about the error. This typically happens through sending an email or by visiting the web frontend of the build server (4). The developers have to consult the build log (5) to understand the problem and provide a fix, a difficult and time-consuming task that typically consists of three steps.

Log Inspection The developer investigates the build log to get further information about the build failure. While it is often simple to spot the part in which the build failed, it is very often difficult to read the log and to understand the failure reason.

Hypothesis Once the developer has an intuition about the root cause for the break, the problem should be replicated, if possible on the developer machine and ideally by providing a test. This makes it possible to use a debugger to inspect the failure.

Fix If the root cause of the build failure is identified, fixing it is usually the easy part. The developer implements the fix, pushes it to the repository, and waits for the result of the new build. All the steps are re-executed if the build fails again.

Executing these three steps is difficult and deriving a hypothesis about the root cause of the failure requires experience. If the developer gets stuck, a common

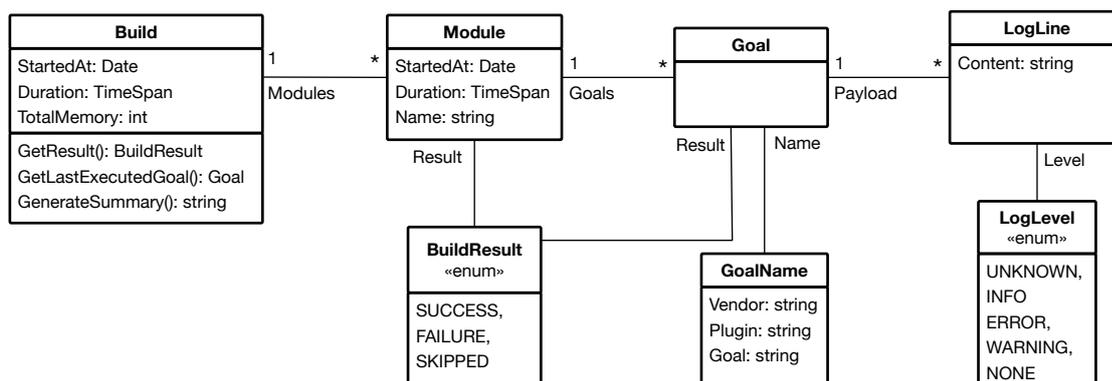


Figure 5.2: Meta-Model that is available to the hint generators in Bart

strategy is to ask more experienced team members [58] or to search on the Internet for solutions [116].

In this paper, we present Bart (6), the *Build Abstraction and Recovery Tool* that supports developers by enriching the build log through summarization and linking of external resources. We have designed Bart as a support tool for Maven builds and we have created a proof of concept implementation for the build server Jenkins. Our solution is complementary to the existing pipeline that we have discussed before. Bart does not replace the inspection of the build log, instead, the build log is embellished with further information to facilitate a faster and better decision making of the developer. For example, our screenshot (7) shows the *Build Summary* (a general summary of the build result) as well as a list of *hints*, in this case details about a compilation error. These hints are included in the Jenkins build overview page.

Bart facilitates the generation of these hints with two reusable parts. First, it parses the build log, extracts all relevant sections (e.g., keywords, commands, built modules), and stores this preprocessed information in a meta-model (8). Second, Bart is extensible through additional *Hint Generators* (9). We have built five different hint generators that summarize the information found in the build log, as well as hint generators that use information from the build log to search for solutions in the Internet. For example, our proof-of-concept implementation can link to related discussions on StackOverflow.

We will introduce these individual parts in the remainder of this section. Section 5.2.1 introduces our build-log meta model and describes our parsing. Section 5.2.2 contains the extension point mechanism for hint generators and a description of the four different hint generators that summarize build log information. Section 5.2.3 discusses the hint generator that links build failures to external information, such as discussions on StackOverflow.

5.2.1 Detecting Failure Information in the Log

Build tools typically log all their actions in a detailed log that allows developers to reconstruct their actions after the fact. Such a build log is typically stored as plain text. All details about the build are contained, but such logs are very large, e.g., even the build log of the relatively small Maven build tool itself ($\sim 130\text{K}$ LoC) results in a build log of more than 1,500 lines. To make the creation of new hint generators in Bart straight-forward, we preprocess these logs. We provide an abstraction over a Maven build log that makes it easy to find exactly the information that the hint generators need. While we are going to focus on the Maven build system,² the most used build tool among Java developers [69], the underlying idea is general and can also be applied to other build systems. In this section, we will first briefly describe Maven's building concept, the structure of its build log, and our parsing. We will then introduce our meta model that we use to store the relevant build information.

Maven follows the concept of convention over configuration. It provides a standard build configuration that defines several *phases* that are run one by one in the default build lifecycle (e.g., `compile`, `test`, `verify`). The set of phases is fixed and most of them are empty by default. A concrete build job can now add specific *goals* to the different phases, if needed. So, for example, a project could add the invocation of a *static analysis tool* to the `verify` phase. In practice, build files contain the configuration for many of such *goals* that range from dependency resolution in the very beginning of builds to packaging or deployment that typically take place at the end.

²<https://maven.apache.org>

Maven builds are organized hierarchically. In addition to the goals that are configured in the build file for the current *module*, parent configurations can be referenced to inherit configuration options. In addition, it is possible to refer to *submodules* that are then build together with the current *module*.

At each build and starting from the module for which the build was triggered, Maven creates the dependency tree between all (sub-) modules and schedules the individual builds in an order that does not violate their dependencies. In Maven terminology, this *build plan* that contains the names of all *modules* is called the *reactor*. During the build, one *section* is dedicated to each *module*. This section contains entries for each executed *goal*, which might also prints additional output to the log. At the end of the build, Maven generates a *reactor summary*, which lists the individual build results. In addition, the reactor summary will also show consumed memory, and -for build failures- further information about the *module* and *goal*, in which the build broke.

As a first step in the failure resolution, a developer that has to read such a build file, has to navigate through a big log to find the relevant information. This is also a hard task for an automated processor, because the individual parts need to be parsed or otherwise processed with string utilities.

To simplify the access to the contained information, we implemented a parser that, taking a build log as input, fills the meta-model that we have created, as depicted in Figure 5.2. The model follows the structure that we have introduced before. The root entity of a build log is a `Build`, which has basic properties like the required memory for the build. A build refers to several `Module` definitions that are part of the build. In addition to timing information, each module has a unique name and a result. It also contains information about the different `Goals` that were executed while building it. Each `Goal` combines the `GoalName` (i.e., a reference to the tool that was run), the `BuildResult` or the invocation, and a `Payload`, which contains all output that was generated for this `Goal`. Each line is annotated with a `LogLevel` and contains content as `string`.

The original build log contains a *reactor summary* at the very end that can be requested by calling `Build.getSummary()`. We do not explicitly store the contained information in the meta-model, because it can be fully inferred from the stored data. In general, this meta-model does not lose any information of the original log.

It splits information into individual sections and provides easy access, but it could be transformed back into the original log file.

Please note that the parser used for this paper does not support the complete meta-model yet. Our implementation supports all parts that were used for the summary generation, leaving out details like the build duration or the amount of memory that were irrelevant in this context. This does not represent a conceptual limitation of the meta model though and can be solved by spending more implementation effort on the parser.

5.2.2 Summarizing Build Log Information

Understanding the extensive build logs generated through a Maven build is tedious. The *reactor summary* that is automatically generated at the end of a build contains basic information about a build failure and represents a first step in the right direction. However, the information is presented as plain text without any highlighting and it is hard to read by developers. In addition, only the lines of the failing goal that are marked as *error* are included in the summary, surrounding information, which could further explain the error, is omitted. We think that this current build summarization is not sufficient. Thus, we propose an improved summarization and a highlighting of the important pieces to ease the life of developers and to make them more efficient in understanding the build log.

Inspecting the failing section of the build log should provide all required information to understand the cause of the failure, so it represents the starting point of any investigation of a build failure. However, the actual information that is included in the corresponding part of the build log can be very extensive (e.g., long lists of executed tests) and it also heavily depends on the failing goal. Fortunately, it has already been shown that build failures can be assigned to different categories, based on the goal or build step that failed [134]. We propose to provide a better guidance in the fixing process by tailoring the summary to the failure category.

We have created a conceptual framework and use it to present build summaries. The implementation is modular and general, but we provide an integration into the build server Jenkins, which adds the polished output to a build report that is shown in Figure 5.1. To create this output, we first parse the build log into our

meta-model. Build status and individual build goals are directly available in the model. We do not add hints to successful builds, so we can directly skip them in the processing. For a failing build, we determine the failure category based on the last executed goal. Using this category, it is possible to find applicable hint generators to save execution time. These specialized hint generators can provide hints that might point the developer towards a build fix. A `Hint` is just a key-value dictionary that can contain arbitrary contents. Hint generators can return an empty list, a single hint, or also multiple hints. The presentation to the user is achieved by iterating over all hints and by putting all their key/value pairs into a table. We do not imply any restriction on the type of key that can be generated, because the actual hint generator should select the most meaningful information for the developer.

In this paper, we focus on a proof-of-concept implementation that supports the most frequent build break categories. Based on the dataset of industrial and open-source projects analyzed in previous work [134], the most common build failure types are *Compilation* (9% of open-source and 6.5% of industrial projects), *Dependencies* (7.1% of open-source and 6.3% of industrial projects), *Testing* (41.3% of open-source and 39.5% of industrial projects), and *Code Analysis* (4.2% of open-source and 16.4% of industrial projects). Please note that we did not consider cross-cutting categories such as *Release Preparation* as suitable for our implementation, because (i) they are not associated with a specific step of the build process and (ii) the structure of the log is highly variable. In addition, we want to provide an additional *Build Summary* that provides an overview over the whole build. In the following, after providing an overview, we will briefly introduce these different hint generators, which information we want to show to the developer in each case, and how we can get access to the required build log data.

□ **Build Summary** The Build Summary provides a high-level overview over the result of the build. It mimics Maven's reactor summary, but reduces the amount of information to a minimum. Rich formatting options are applied to highlight the different information. You will find an example of the build summary in Figure 5.1. Each summary is composed by the following sections.

Reactor Summary The list of modules can be requested from the `Build` object, their individual results can be directly used.

Build Result Can directly be requested from a `Build` object.

Failed Goal The last executed goal of a failing `Build`.

Error Cause The error cause consists of the error message that is printed by the failing goal. These can be extracted by selecting all lines of the goal that have the log level “error”.

□ **Compilation Failures** The hint generator should provide detailed information about the location of the compilation error. All this information can be found in the `Payload` of the failing goal.

Type Name of the type (e.g., class) that could not be compiled.

Line Line number, in which the error has occurred.

Reason Textual description of the compilation error, e.g, instantiation of an abstract class, when provided by the build log.

□ **Dependency Failures** Declared dependencies can lead to various build failures. Our summarizer helps understanding the dependency error by showing the following information.

Dependency The name of the dependency that causes the failure. The Maven coordinates of the dependency are mentioned in the error message and we use a regular expression to parse them.

Reason Textual description of the dependency error. Typical reasons are invalid versions numbers or missing internet access.

□ **Testing** Testing failures are particularly tricky to fix, because they can occur after introducing a change in a completely different part of the system. For this reason, it is important that a hint does not only contain the location of the test, which is required to replicate the failure locally, it should also contain the reason that explain the failure. As a result, the hint generator reports the following:

Location The location, in which the testing failure occurs. This contains both the *test class* and the failing *test case*.

Reason A textual description of the test failure. This is taken from the *failed* assertion statement, so the quality of these descriptions depends on the concrete test case. In case of an *error*, also the stack trace of the failure will be included.

□ **Code Analysis** Many builds use static analysis tools to validate properties (e.g., a consistent programming style) of the system. Like for all other types, also for code-analysis failures developers need to find the relevant information that is spread across different sections of the log. Let us consider a simple example, a Checkstyle error. The Maven’s reactor summary only reports “You have 1 Checkstyle violation”, but to understand the actual problem, the developer first has to find the right section of the log, which then includes the detailed description (e.g., “Line is longer than 120 characters”). Each static analysis tool produces a different output and individual hint generators are needed to cover them. We selected Checkstyle as a representative for such static analysis tools because is the most frequently used in open-source Java projects [9, 142]. We include the following information that help to understand related build failures:

Location The path to the *file*, in which the style violation was detected. The location also includes the *line number*.

Reason Name of the style rule that caused the failure. These names are typically very expressive, e.g., “method name too long”, so the proposed hints are potentially very meaningful.

□ **Future Extensions** Future hint generators might use other build-log information in their hints. They can either reuse our meta model or provide their own extraction strategy to find the interesting information in the build log. It is possible, for example, to use custom regular expressions to parse specific information from the `Payload`. As a fallback, it is always possible to recover a full build log from our meta-model, which ensures support for all hint generators that work on the build log. Extensions that require *external files* in addition to the build log, like test coverage reports, represent a special case. These files are not contained in the build log. Hint generators that require access can still parse the respective path from the build log and open these files separately.

Stack Exchange Analysis

Related Post 1:

This is normal behaviour. Abstract classes are not supposed to be instantiated. You should test the classes which inherit from the abstract class, not the abstract class itself.

Full Discussion: <https://stackoverflow.com/questions/5028082/rails-3-activerecord-abstract-objects>

Figure 5.3: An example of hints derived from StackOverflow discussions

5.2.3 Hints from External Sources

Summarizing local information improves the ability to understand the contents of the build log. However, developers may encounter situations, in which the error message is easy to understand, but requires a complex fix. For example, if the *source level* is not configured in Maven, it will use Java version 5 by default. If the developer now writes Java code in a newer version, e.g., version 8, and uses one of the newly introduced constructs, e.g., lambda expressions, the Maven compiler plugin will fail with a syntax error, even though no problem will be reported in the development environment. While the summary will point out a syntax error very clearly, in this case, an inexperienced developer will struggle to solve this on their own and will either ask a more experience colleague for help or simply search for a solution on the internet. For this reason, we also need to provide an infrastructure in Bart that allows the creation of hint generators, which can go beyond a *local summarization*. These *external* hint generators should be able to add additional hints and link to external resource in their suggestions for possible solutions.

It is very likely that, in case of a build failure, a similar build break has already been discussed online. Previous work has already shown that question and answer sites, like StackOverflow, can provide a great source of information to support developers [88]. The site contains almost 60K discussions that are related to Maven development,³ which makes us very confident that it can also be a good source for tips on how to fix a broken build. An example of a hint that refers to a StackOverflow discussion is shown in Figure 5.3. The example hint explains a specific compilation failure and also links the full discussion to provide additional context.

³<https://stackoverflow.com/questions/tagged/maven>

To obtain relevant discussions from StackOverflow, we use a twofold approach. First, we query StackOverflow for discussions that are related to the build log. Second, we rank the returned posts and present the most relevant discussions to the developer. The hint engine starts with requesting the build log and the hints that have already been generated in the *local summarization* step. Given that the *local* hint generators have already identified the key parts of the build log, we make use of this information to create a query that is as specific as possible. The hints are being *cleaned* by removing local information (e.g., paths or file names) and common overhead that is added in every Maven build (e.g., formatting characters or *goal* names). The resulting query mainly contains the error message that describes the failing build and it is used to search on StackOverflow.

In a second step, we rank the returned posts to identify the ones that are most related to the actual build log. To achieve this, the build log is first *cleaned* in the same way as the query and then *tokenized* to create a set of keywords that describe the build. Common english stop words (e.g., “the”, “or”, “and”) are removed to improve this set of keywords. For each post, we calculate a post score by counting how many different keywords are used in the body of the discussion. After ordering the posts by their score, the top four proposals are selected and shown to the developer. We decided to list the top four posts because because this number typically avoid that developers have to further scroll the page. We did not want to come up with a long list that adds an additional burden on the developers.

5.3 Investigating the Effect of Bart on the Build Fixing Practice

To understand the effects that a tool like Bart has on the practice of fixing a build, we conduct an empirical study, in which we will quantitatively measure Bart’s capability to improve the *understandability* of build failures and to improve the *performance* of developers when fixing broken builds. This first study consists of two parts, a *controlled experiment* and a follow-up *questionnaire*, that covers the different aspects that we want to investigate.

Table 5.1: Projects used in the controlled experiment for evaluating Bart

Project Name	Version	Size (LoC)	#GitHub Stars	#Builds
ActiveJPA	0.3.5	39,335	143	123 (master)
Sentry-java	5.0.0	113,332	312	509 (master)
Fongo	2.1.1	31,088	374	404 (master)

Understandability In the first part, we assess whether or not the summaries generated by Bart make it easier to understand the cause of a build break and to formulate a solution strategy.

Performance In the second part, we measure Bart’s effect on the required time to fix certain types of build breaks.

In the following, we will first introduce the context and our methodology that we have applied to investigate both aspects and we will then answer the first two research questions.

5.3.1 Context

The *context* of our study includes (i) as *objects*, build breaks that we have generated from selected Java projects, and (ii) as *subjects*, developers that participated in our controlled experiment.

The three software systems that we considered in our study are illustrated in Table 5.1. We followed the methodology of Bavota et al. [5] to select systems that developers can easily get familiar with and that are, at the same time, representative for real software systems. ACTIVEJPA⁴ is a Java library that implements the active record pattern on top of Java Persistence APIs (JPA). SENTRY-JAVA⁵ is an error tracking system that helps developers to monitor and fix crashes in real time. FONGO⁶ is an in-memory Java implementation of MongoDB. The considered systems are hosted on GitHub and built on the cloud-based build infrastructure of TravisCI. While our selected systems have less than 500K lines of code, they

⁴<https://github.com/ActiveJpa/activejpa/>

⁵<https://github.com/getsentry/sentry-java>

⁶<https://github.com/fakemongo/fongo>

are very popular (more than 100 stars on GitHub) and frequently built (more than 100 builds on the master branch). For our study, we have injected bugs into these systems to generate broken builds. The introduced bugs belong to the four most recurrent categories of build failures [134], i.e., *compilation*, *dependencies*, *testing*, and *code analysis*. We created different mutations of the extracted systems for every category of broken builds and ended up with five broken ACTIVEJPA versions, two broken SENTRY-JAVA versions, and one broken FONGO version.

More details about these broken versions are depicted in Table 5.2. We always created two mutated versions to avoid learning effects in both tasks of the study. To generate the *testing* build breaks we changed an assertion in the test class `FongoAggregateProjectTest` from `assertNotNull` to `assertNull`. We have also altered the `count` method of the class `org.activejpa.entity.Model` by adding an incorrect offset to the returned value. We chose an obvious mistake to fail the build, i.e., we added 100, which is easy to spot. To provoke *dependency* build breaks, we have first inserted an obvious non-existing dependency and, second, we included a typo in another (existing) dependency. We have introduced two *code analysis* build breaks in Sentry-java, by adding a new method with a very long name to the class `SentryAppender` and by deleting the Javadoc comment of the method `doClose` in the class `AsyncConnection`. Both are picked up by CheckStyle, which will raise the errors *Very long function name* and *Javadoc has empty description section*. Finally, to create *compilation* errors we added a return statement in the void method `close()` of the class `org.activejpa.JPA` and inserted an illegal combination of `static` and `abstract` in the signature of the method `deleteAll` of class `org.activejpa.entity.Model`.

We contacted participants by sending out invitations to students from the University of Zurich (UZH) and Swiss Federal Institute of Technology in Zurich (ETHZ) and to professional developers that we reached through our personal contacts. In total, 17 participants completed our controlled experiment. We made sure that all participants have used Maven before. The majority of our participants (9, 53%) report three to five years of programming experience, while six participants (35%) declare that their experience exceeds five years. Only two participants reported less than three years of programming experience. 11 (65%) participants work as professional developers. We asked the participants to self-

Table 5.2: Mutated components in the systems used to evaluate Bart

Build Break Type	Project	Mutated Component
<i>Task 1</i>		
Test	Fongo	com.github.fakemongo.FongoAggregateProjectTest
Compilation	ActiveJPA	org.activejpa.jpa.JPA
Code Analysis	Sentry-java	net.kenochrane.raven.connection.AsyncConnection
Dependencies	ActiveJPA	pom.xml
<i>Task 2</i>		
Test	ActiveJPA	org.activejpa.entity.Model
Compilation	ActiveJPA	org.activejpa.entity.Model
Code Analysis	Sentry-java	net.kenochrane.raven.log4j.SentryAppender
Dependencies	ActiveJPA	pom.xml

estimate their programming experience in a five-point *Likert scale* [62] from *very low* to *very high*. Out of all participants, 10 report an experience level of *above average* or higher (*very high*: 2). Only 2 participants rate their experience as *below average* and no one rated their experience as *very low*. Our participants represent a diverse group with different backgrounds.

5.3.2 Experimental Procedure

The empirical study we conducted with our participants consists of two different tasks and was supervised by one of the authors. We provided summaries and solution hints generated by Bart to our participants to study the *understandability* of build breaks. In the second task, we investigate whether Bart can speed up the fix.

First Task: Understandability In the first task, our participants answered a questionnaire about the *understandability* of the build break summaries provided by Bart. We used Bart to generate summaries and solution hints for the broken builds of the mutated software components in Table 5.2 (*Task 1*) and asked our participants to evaluate them. We provided our participants with the following three questions and we asked them to answer on a five-point Likert scale from *very high* to *very low*:

- How much did your understanding of the build failure improve through the summary of the build log?
- To what extent do the suggested solutions help you in conceiving a strategy to solve the build failure?
- To what extent are the suggested solutions applicable to the specific build failure?

Second Task: Resolution Performance In the second task, we measured the time it takes developers to fix a broken build to analyze the effect of Bart. Every participant was asked to fix two of the four manually injected bugs for Task 2. Specifically, each participant had to fix one bug with treatment (i.e., support through Bart) and another one from a different category without. We have avoided learning effects between the two different fix attempts of each participant by changing the type of build failure and by changing the software component, in which the bug was introduced. In total, we tested eight scenarios and each of the four different build failures was fixed twice, once with and once without treatment. The participants always started with fixing the build failure without treatment. All participants managed to fix both assigned builds without external help.

One of the authors supervised the task and started the experiment with an introduction to Bart. Participants were then asked to import the assigned projects into their development environment. We were using a Jenkins build server, which, in case of a build failure, produces a build overview that indicates the build result (i.e., Failed), the last Git commit that was pushed to the remote repository, and the name of the committer. In addition, Jenkins provides access to the generated build log. The supervisor gave our participants time to get familiar with the projects and with the Jenkins instance that was used to build the projects. To start the fix attempt of the build failure, our participants were asked to trigger a new build of the assigned project and to repair the resulting build failure. The task supervisor measured the *resolution time*, i.e., the time between the build break and the next build success. The same methodology was applied for the second build fix attempt.

General Feedback After finishing the experiment, our participants filled a survey and participated in semi-structured interviews, in which we asked them for their opinions on the build-failure summarization. The survey was focused on three

main questions. The first question was about Bart’s ability to provide assistance for build failure resolution. We discussed with our participants what they like or dislike about our approach. In the second question, we asked for the perceived benefits of using Bart. The last question asked whether our participants would integrate a tool like Bart in their regular CI pipeline.

All 17 participants filled out the survey, but only nine participants agreed to participate in a follow-up interview. We recorded and transcribed these interviews and then performed card sorting to analyze the answers to our open interview questions [111]. We started by splitting the answers into individual statements, grouped common arguments, and finally organized these arguments hierarchically. We will use statements from the participants later, when we discuss the results of our experiment, and indicate which participant made them (e.g., I3 is interviewee with id 3, S5 is survey participant with id 5).

5.3.3 Understandability of Build Breaks

Our first research question was how build summarization can improve understandability. To answer this question, we evaluate the ratings of our participants for the generated summaries of Bart. We visualize the answers in three *diverging stacked bar charts* [101] that illustrate their rating regarding the *understandability* of the summaries (Figure 5.4), their *relevance* (Figure 5.5), and their *applicability* to the build break (Figure 5.6). We use the *Likert* values *very high*, *above average*, *average*, *below average*, and *very low*.

Understandability Figure 5.4 shows how participants rate the understandability of Bart’s summaries compared to the raw build logs that are provided by Jenkins. All participants agree across the board that the understandability of the build break summaries is at least *above average*, with the majority saying that it is *very high*. Only in two cases, one for compilation and one for dependencies a participant found the build summary *below average* and *average* respectively. Specifically the participant found that Bart’s summary for the dependency error was comparable to the default build-log presentation in Jenkins and that it did not improve.

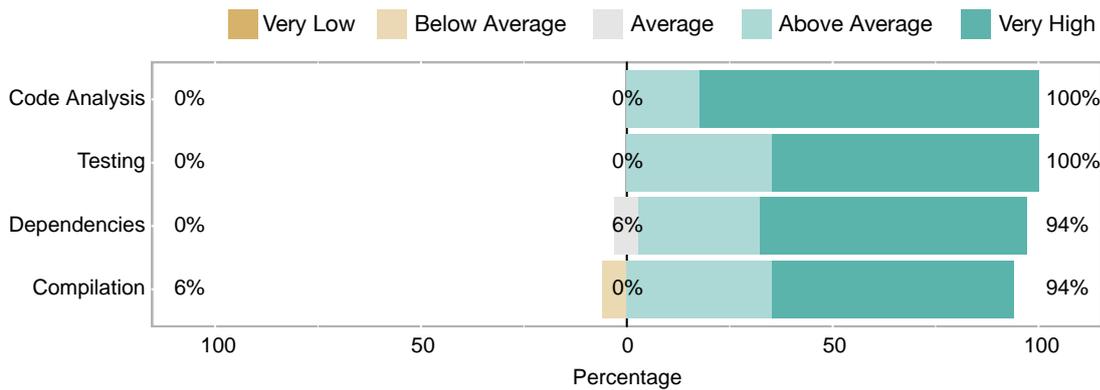


Figure 5.4: Understandability of build summaries

The developers seem to agree that Bart’s summaries helps them to better understand the build log. One of the participant describes the actual build logs as “*cryptic*” (I8), which could be caused by a lack of experience in reading it. However, the overloading amount of information that is contained in a build log is a recurring theme in the answers of our participants, even from experienced developers. Another participant said that “*Maven logs tend to be verbose, having a quick summary [...] greatly reduces the time needed to find and correct a build failure*” (I5) and another one that “[Bart] helps to find the programming errors quickly” (I4) and “*a structured summary is way easier to grasp than many unstructured lines of text*” (I4).

Our participants almost unanimously agree that Bart’s build summaries improve the understandability of build logs.

Relevance & Applicability Figures 5.5 and 5.6 illustrate relevance and applicability of the proposed solution hints from StackOverflow. The solutions hints for *compilation* and *code analysis* breaks were mostly positively rated. Most our participants found their relevance and applicability *above average*, more than half of them rated them *very high*. However, two participants found the applicability of the solution for the *code analysis* break *below average* and one of them, according to the background information a very skilled developer, has also considered the relevance of the solution *below average*. The one participant that has considered

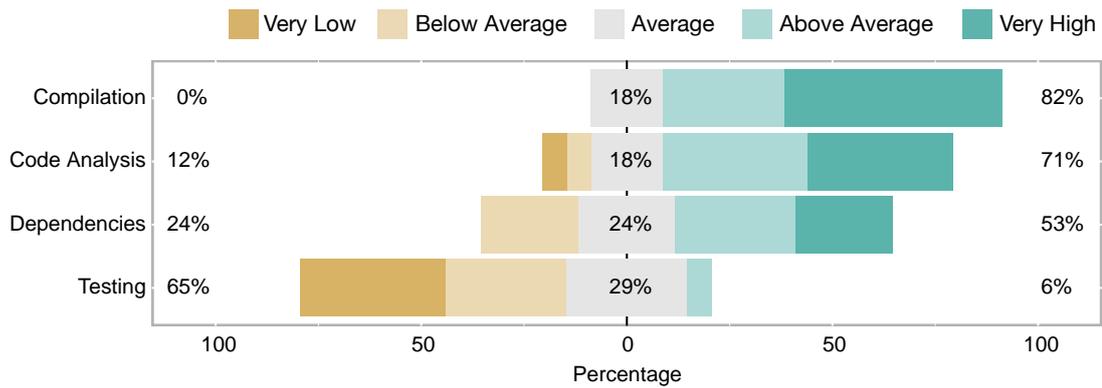


Figure 5.5: Relevance of solutions proposed in the hints

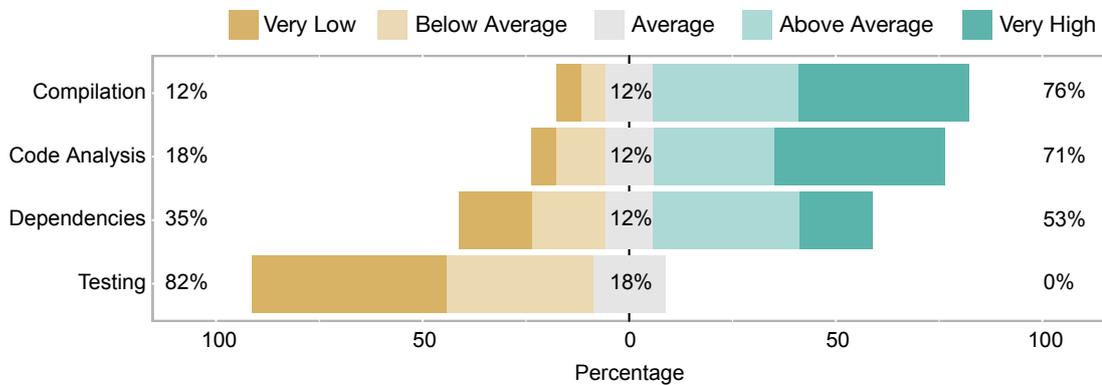


Figure 5.6: Applicability of solutions proposed in the hints

the solution hint for the *compilation* build break as *very low* has little programming experience and uses Java only occasionally. We assume that he simply did not understand the suggested hint.

Most study participants find the solution hints for build breaks caused by compilation and code analysis errors relevant and applicable.

In case of the *dependency* build break, the participants do not agree on a rating for the relevance and applicability of the solution hints. The ratings are centered around *average*, some of the participants find the suggestions relevant and applicable (one participant considered it even *very high*), while others rate it *below*

average. Two participants even think that the applicability of the solution hints is *very low*. One of them is no frequent Java user, but the other one has a very strong background in Java programming, so a lack of expertise alone does not explain the different ratings.

Suggested solutions for dependency errors are often not considered as valuable hints by our participants.

Our respondents were also not convinced about the relevance and applicability of solution hints for *testing* build breaks. Most of our participants consider them *below average* or even *very low* when compared to the original build log. One possible explanation is that the available information in the build log of a failing *testing* build, i.e., the name of the failed test, is project specific. This makes it unlikely to find related solutions for such local errors in external resources without taking other information into account.

Testing-related build failures are project specific. The build log alone is not sufficient to identify related external resources.

5.3.4 Resolution Time of Build Failures

Our second research question was whether Bart can reduce the time that is required to fix a build. To answer this questions, we have asked our participants to fix the failing builds of the second task while measuring the required time. Table 5.3 illustrates the results of this experiment. It shows the average time to repair the different build failure types with and without the support of Bart and the corresponding standard deviations. We computed the coefficients of variation (CV) [29] for all the combination of build failure types and fixing modalities. All our calculated CV values are smaller than the traditional threshold 1 (in fact, all values are < 0.5), which shows that the measured times are centered around the mean. For this reason we refer to the average values in the following. While the previous research question has revealed that the ratings for relevance and

applicability of Bart’s solution hints differ between the build break types, the results of the second task shows that using Bart leads to a substantial reduction from 20% to up to 62% in the required time to fix a build across all scenarios. We will discuss the different break types individually to explain what seems to be a contradiction at a first glance.

Code Analysis and Compilation The study participants found that Bart’s summaries improve understandability and that the solution hints are both relevant and applicable. The positive ratings can also be confirmed in the practical task. The time to fix a build break could be reduced by 62% for build breaks related to *code analysis* and by 20% for build breaks related to *compilation* errors.

The error messages of both *compilation* and *code analysis* are self-explanatory, but a certain degree of expertise is needed to understand them. Fortunately, the exact same error messages and warnings appear in other projects as well, so it is easy to find information online that provides context to understand the error message. Our solutions hints are able to enhance the description of a warning or even replace it and the developers get an explanation of an error or of a violated rule without having to look it up on external resources. This aspect is particularly useful when the developer is not used to a specific code analysis tool.

Our participants found it very helpful that Bart integrates all required information to understand the meaning of a rule violation *“Less searching for the relevant part in the error message, hence faster bug resolution”* (I2). Moreover, the links to StackOverflow are highly appreciated when the meaning of a warning is non-obvious. *“In the less obvious error causes, the stack overflow extracts prove to be very useful”* (I5). In these cases, the StackOverflow discussion about the proper solution can provide additional context information to understand the problem. The StackOverflow solution hints speed up the development process, because *“You can often get the information from bart without having to search the internet”* (I6).

In addition to the summary, solution hints can provide the required context that helps with understanding the cause of a build break.

Dependency and Testing The relevance and applicability of the suggested solution hints were not considered useful for *dependency* breaks and *testing* failures. These low ratings can easily be explained though. A search for the corresponding error message would either return many unrelated resources (e.g., cases in which other developers had trouble with some other dependency) or none (because the error message of a test failure is project specific). However, we could still see a substantially reduced fixing time for both categories. The improvement for *dependency* related build breaks (43%) has even been the second most significant reduction among all considered cases.

When considering error messages in both categories, it becomes apparent that both are typically very self-explanatory. The errors immediately point to the missing dependency or name the failing test. The required action to fix such issues is straightforward: search for the missing library in the *Maven Repository* and add it to the build file or fix the failing test, respectively. The participants that fixed such kind of breaks have reported that the reorganization of the information contained in the build logs significantly reduced the amount of time needed to understand the cause of a build break. One of our participants stated that “[*Bart is*] mostly a timesaver, not really a skill enhancer. Carefully reading the log usually allows the extraction of the same information” (I5). Another participant found that “directly serving the relevant solution, the debugging process is drastically sped up” (I1).

Another important aspect that affects the time to fix a build is the debugging environment. Previous work has shown [45] that CI server like Jenkins do not provide sufficient support to debug a build break. According to our participants, however, Bart summaries “add more capabilities to the environment” (I6) compared to the raw build logs and “might make debugging unnecessary when the bug becomes evident” (I2). They report that “if some tests fail, the Bart output can be helpful in finding out why” (I5).

Dependency breaks and testing failures seem to be easy to understand. Providing a good summary that highlights the locality of the issue seems to be the most crucial factor on fixing time.

Table 5.3: Resolution time per build failure type
 (AVG. stays for Average and STD. DEV. stays for Standard Deviation)

Failure Type	excl. Bart (s)		incl. Bart (s)		Reduction (%)
	Avg.	Std. Dev.	Avg.	Std. Dev.	
Testing	436	154	334	33	23%
Compilation	187	19	150	32	20%
Dependencies	223	74	127	16	43%
Code Analysis	280	109	106	39	62%
Overall Average					37%

Overall, it seems that the different build break categories require different support strategies. Some categories benefit from links to external resources that provide additional context about an error (e.g., *compilation* errors), others benefit more from an improved summarization (e.g., *testing* failures). Bart combines both in one tool and substantially reduces the time to fix a build break across all scenarios in our study, on average by 37%.

5.3.5 General Feedback on Bart

Most of our participants (76%) benefited from using our build summaries during failure resolution. 53% reported that the big advantage provided by Bart is the speed up in solving those failures. One participant stated that *“especially with very large build logs and more complex errors bart really helps to reduce the time needed to figure out where the error happened”* (I7). For 23% of our participants, the main advantage of using build failure summaries is an easier analysis of the root cause of the failure because *“a structured summary is way easier to grasp than many unstructured lines of text”* (I4). 12% were not sure about Bart’s benefits and another 12% did not think that Bart provides particular advantages. 63% like Bart because they have to read less information in order to fix a build failure. This is strictly connected to the most cited benefit provided by Bart, i.e., a faster build resolution. 16% also appreciated having links to relevant StackOverflow discussions that can point them to the right solution. They found those links useful especially when the problem is quite complex to solve as stated by a participant *“Additional links to related StackOverflow posts can be helpful in fixing more complex*

problems” (I9). 21% really like the overview of the failure provided by Bart that highlights the root cause of the failure. The last feedback that we received was about the applicability of Bart in the actual development context of the participants. 35% would immediately install our tool. 23% would like to install Bart, but they are not using Jenkins or Maven in their current projects. Furthermore, one participant (I4) believes that he would install Bart for big projects and another would use it only if there is “*there is no installation overhead*” (I2). Finally, 12% would maybe install Bart, 6% do not know, and 24% do not want to install our tool mainly because it does not apply to their context.

5.4 The Developer's Perception of the Build Fixing Practice

The results presented in the previous section show that Bart can provide assistance that improves the developer's performance when fixing build failures. Besides a summarization of the error cause, which seemed to be universally helpful across all failure scenarios, Bart also provides solution hints that help developers to derive fixing strategies. However, the results of our follow-up questionnaire have shown that the applicability of these solution hints differs between the failure categories. While they seem to be helpful for some failure categories (e.g., *compilation* and *code analysis*), the results are no longer unanimous for *dependency*-related failures, and our solution hints do not work for *testing*. To further investigate and explain this observation, we analyze the developers' perception of build failures. We first identify the characteristics of typical workflows for fixing build failures in various categories and answer RQ₃. After this, we answer RQ₄ by exploring what types of build failures developers consider more difficult to fix. We will use these findings to discuss the current limitations of our StackOverflow-based approach and we will envision future solution-hint generators.

5.4.1 Survey on the Perception of Build Failures

The *goal* of this study was to understand how people react to build failures and what types they consider hard to fix. In order to fulfill our goal, we sent out a

Table 5.4: Questions about the perception of build failures
(MC stays for Multiple Choice and O stays for Open answer)

Section	Summarized Question	Type	#
<i>What makes it hard to fix a build?</i>			
Q1.1	What typically takes longer, finding relevant information or understanding the problem to derive a fix?	MC	101
Q1.2	For which kind of build failures does it take long to find the relevant information?	O	101
Q1.3	For which kind of build failures does it take long to understand the problem and to derive a fix?	O	101
<i>How you would react to a build failure notification in the following scenarios?</i>			
Q2.1	Scenario 1: A compilation error broke the build.	O	101
Q2.2	Scenario 2: The build broke due to a dependency error.	O	101
Q2.3	Scenario 3: The execution of a test has failed.	O	101
Q2.4	Scenario 4: A build failed, because a quality concern was detected.	O	101

survey. In the following, we describe our survey’s questions and the demographics of our participants.

Structure Our survey consisted of 15 questions and was divided into 4 sections. In the first section, *Background*, we collected the demographics of our participants. In the next two sections illustrated in Table 5.4, we surveyed developers on the difficulty of solving build failures and on their reaction to the most common build failure types. We wanted to identify whether it is harder to inspect the root cause of the build failures instead of understanding the problem and plan the fix (Q1.1) and for which types (Q1.2 and Q1.3). In Q2.1-Q2.4 questions, we asked our participants to reflect on their workflows for solving compilation, dependencies, testing, and code-analysis build failures. In the last section, we let our participants add opinions on the survey and report other build failure types that researchers should investigate more.

Recruitment Our survey was implemented using Google Forms. To recruit participants we posted our survey on the Question and Answer Site Reddit,⁷ on

⁷<https://www.reddit.com>

which we targeted 15 specific subforums (so-called “subreddits”), namely `r/DevOps`, `r/LearnProgramming`, `r/WebDev`, `r/iOSProgramming`, `r/Docker`, `r/learnJava`, `r/Python`, `r/cpp_questions`, `r/dotnet`, `r/javascript`, `r/ruby`, `r/swift`, `r/coding`, `r/androiddev`, and `r/csharp`. We selected these communities because they (i) allow users to post surveys, (ii) have a large number of active subscribers, thus increasing our potential audience (e.g., the `r/DevOps` subreddit has approximately 300 members that are online daily), and (iii) include members that frequently encounter build failures. We also sent an email to the mailing lists of Maven,^{8,9} Ant,¹⁰ and Jenkins.¹¹ The survey was available for 2 months in two different periods (from mid-October to mid-November 2018 and March 2019).

Demographics A total of 101 respondents completed the entire questionnaire. Among all survey respondents, 88.1% work as professional developers. 3% are spare-time developers and 4% are students. The remaining part of our participants (4.9%) consists of build engineers, DevOps engineers, site-reliability engineers, and industrial researchers. 86.2% of our respondents rate their programming experience *High* or *Very High*. In most of the cases (58.4%), the highest qualification is a Bachelor Degree, while 16.8% hold a Master Degree, and 4% have a Ph.D. degree. Self-studied developers represent 14% of our respondents. Our participants report that 26.7% encounter build failures frequently and 6.9% very frequently. The majority (48.5%) reports encountering build failures occasionally.

5.4.2 How Do Developers Approach Different Types of Build Failures?

The fact that solution hints work for some categories of build failures, but not for others, suggests that developer approach these kinds of build failures differently. To better understand these differences, we have asked our participants to describe the typical workflow in which they would address several kinds of build failures, under the assumption that they do not know from the start what caused the

⁸users@maven.apache.org ⁹dev@maven.apache.org ¹⁰user@ant.apache.org

¹¹jenkinsci-users@googlegroups.com

problem. Participants could give an open answer and were supposed to illustrate the individual steps.

Methodology

To identify the typical workflows for the four main build failure categories that are covered by Bart, i.e., *compilation*, *dependencies*, *testing*, and *code analysis*, we have conducted an open-coding approach [111] to encode the described workflows from the open answers of our participants. Given that the output of the coding is not simple labeling, but a complex graph describing the workflow, we had to adapt the traditional methodology. Two authors of this paper have performed the following steps to extract the workflows.

Common Vocabulary The first step in our methodology was to establish a shared vocabulary, i.e., a set of activities that we accept in the extracted workflows. Both authors individually inspected 80 answers that span over 80 participants and all four scenarios to create two individual sets of activities. In a joined discussion, the authors merged both lists, eliminated duplicates, agreed on activity labels, and clarified overlapping cases. We ended up with the following list of activities. The high-level concepts are only used for grouping and are not used as labels.

Start & End All workflows have the label **Error notification** as the start node and the label **End** to indicate a fixed build.

Locating Activities in this category are related to understanding the error message and to finding further information about the error.

- Most of the time, developers open the build log, either via a web browser or through clicking a link in an email. The first step is then to **Locate the Error** in the log to understand the type of error.
- We extract an **Understand details** activity, every time the developer state that they look for information in the build log, like line numbers or test names.
- Many developers state that they **Reproduce the problem**, either locally on their development machine or on a build host, to collect further information.

Inspect Changes Once the error message is understood, developers often continue to inspect the changes of the last commit or differences in the environments. We differentiate the following activities.

- The most common activity to understand the root cause of an error is to **Inspect code changes** that have recently been introduced.
- Another frequent activity is to **Inspect the Config** to understand recent changes or differences in, for example, tools and environment.
- Developers check the version control system to **Identify committers**, either to ask them for details or to delegate the fix of the build failure.
- Several developers state that they **Inspect process documentation** to understand the reason for a change, for example, through release notes, a changelog, or an issue tracker.
- Some developers mention that they **Inspect the build history** to extract historic information about past builds, like previously failing builds.

Investigation Developers perform various steps to further investigate the failing build and understand more details about the error cause.

- Many developers **Read/Debug** the affected source code. We combine both activities in one node because it is often not clear which one is meant.
- Automated Static-Analysis Tools (ASAT) can provide diagnostic information about a build, e.g., the dependency graph. We extract a **Use ASAT** activity when developers apply them to find hints towards a solution.
- Sometimes, the build log is not enough and developers need to **Consult external logs**, e.g., other log files or screenshot of the failing application.
- Often as a measure of “last resort”, developers perform a **Websearch** to find more information or an illustration of the problem online.
- In some cases, developers state that they **Lookup the documentation** to understand the meaning of a particular error message.
- To get further information about a change, it is sometimes necessary to **Contact an expert**. This person is typically either familiar with the affected part of the codebase or is the committer of the latest changes.

Solution Attempt In the final phase of a workflow, developers state how they approach the fix. We differentiate between five alternative activities.

- The most obvious activity is when a developer states that, after understanding the problem, it is clear how to change the project to **Fix** the build.
- Various answers mention that they would use **Automated Fixes**, for example, an environment's clean-up, dependencies' update, or an automated refactoring.
- A re-occurring strategy is to **Delegate the Fix** to someone else, usually to the original developer who committed the change that caused the build failure.
- The answers contain descriptions of very strict workflows, in which developers **Revert commits** to fix the build failure as quickly as possible.
- Developers, that come to the conclusion that the build failure is either irrelevant or an incorrect state, **Ignore** the problem, e.g., by removing a failing test case or by ignoring a quality check.

Verify Several authors describe steps to verify the solution, both locally and remotely. We use the label **Verify** to represent these cases.

Exclude Several of the answers had to be excluded from our analysis. Instead of extracting a complete workflow, we then only extracted a single label.

- We extract the label **Not a problem in practice**, whenever participants state that this particular problem does not occur in their typical workflows.
- If we can not extract meaningful information (e.g., in case of an empty answer), we treat the entry as an **Invalid answer**.
- In several cases, participants are not able to explain how they would approach the problem in the stated scenario (**I don't know**).

This set of activities was stable during the open coding session. The authors did not find any description that could not be encoded with these labels.

Coding Rules In a first training iteration, the authors have encoded the 80 answers (20 answers per scenario) that have been used in the vocabulary step. This step has first been performed individually. Afterward, the results have then been discussed and merged into a joined encoding. As part of this merge discussions, the authors have agreed on a set of coding rules that decide corner cases, in which the

authors disagreed in their separated coding sessions. The authors then performed a second training iteration separately (120 answers, 30 per scenario), using these rules to test their applicability. The resulting workflows were discussed again, agreed on, and the rules have slightly been refined. The authors ended up with the following set of rules.

1. The initial state of a workflow is always an **Error Notification**. The end state is always the **End** node, which indicates a fixed build. The participants do not know what has caused the error, so unless developers explicitly mention that they do not check, we include the **Locate Error** activity, through which they understand the type of error that has occurred.
2. **Locate Error** informs about the error type (e.g., the build failed due to a compilation problem), but it does not provide enough information for a **Fix** (i.e., **Locate error** → **Fix** is an invalid transition).
3. Just stating to read the log does not define any solution strategy and is considered an **Invalid Answer**. Also, ambiguous statements are considered invalid. If missing steps are implicit and clear from a mentioned strategy, we fill the gaps. However, incomplete cases (e.g., when a “branch” in a case distinction is missing) are also considered **Invalid Answer**.
4. We consider the answers of all four scenarios together, since some participants do not repeat details that are mentioned before.
5. Running the debugger requires to **Reproduce the Problem** first. Please note that **Read/Debug** can also just mean reading the code, which does not require to **Reproduce the Problem**.
6. The activity **Identifying committers** needs a reason. Typically, this is either done to **Delegate a Fix** or to **Contact the Expert** to understand further details.

These rules remained stable during the following open-coding session and both authors used them to encode all remaining answers separately.

Validation A workflow consists of *nodes* and *directed edges*. The nodes have labels that correspond to the established vocabulary, edges connect nodes to indicate a sequence of activities. To make the individual workflows comparable, we wrote them down in the Dot syntax for directed graphs. For example, a “diamond” workflow

that does a and then either b or c, but ultimately always d, can be expressed in the simple graph $G \{ a \rightarrow b \rightarrow d; a \rightarrow c \rightarrow d; \}$.

After encoding all remaining 204 open answers separately, we validated the reliability of our process. We could not calculate the traditional *Inter-Rater Reliability* [18], because the resulting workflows are more complex than simple labels that are either correct or not. Instead, we calculated the similarity of the resulting workflows. We use a normalized hamming distance to calculate the similarity of two encoded workflows [102]. The contained nodes and edges of the graph build a set of graph elements, the more of these elements a workflow has, the larger it is. When comparing two workflows, the set-size of the larger workflow defines the maximum distance n_{max} between both workflows. We count the number of elements n_{shared} that both workflows have in common. The *normalized* distance is then defined as $d = n_{shared}/n_{max}$, which is bounded by 0 (completely different workflows) and 1 (identical workflows).

After the coding phase of the remaining 204 open answers, we achieved a perfect agreement in 105 cases (51.5%). The workflows, in which we disagreed had an average similarity of 50.0%. Without looking at the other solution, both authors re-coded these cases and many cases turned out to be trivial mistakes like forgetting the **End** node. After fixing these obvious mistakes, the agreement increased to an exact match in 145 cases (71.1%), with an average similarity of 53.3% for the workflows with disagreement. All remaining disagreements were solved in a joined coding session.

Aggregation After agreeing on the sorting results for all 101 participants, we merged the individual workflow descriptions into joined representations. The general approach was to simply overlay all workflows and to count how often each of the nodes and edges occurs in the dataset. The result was a graph with high complexity that was caused by various infrequently named transitions. To remove rare activity sequences that blur the overall picture, we have filtered activity sequences (i.e., edges) that were described less than three times. A straightforward implementation of this filtering would just leave out these edges from the graph, which would not only result in inconsistent counts, but it can also result in invalid workflows with “gaps”. To preserve this consistency and, at the same time, to

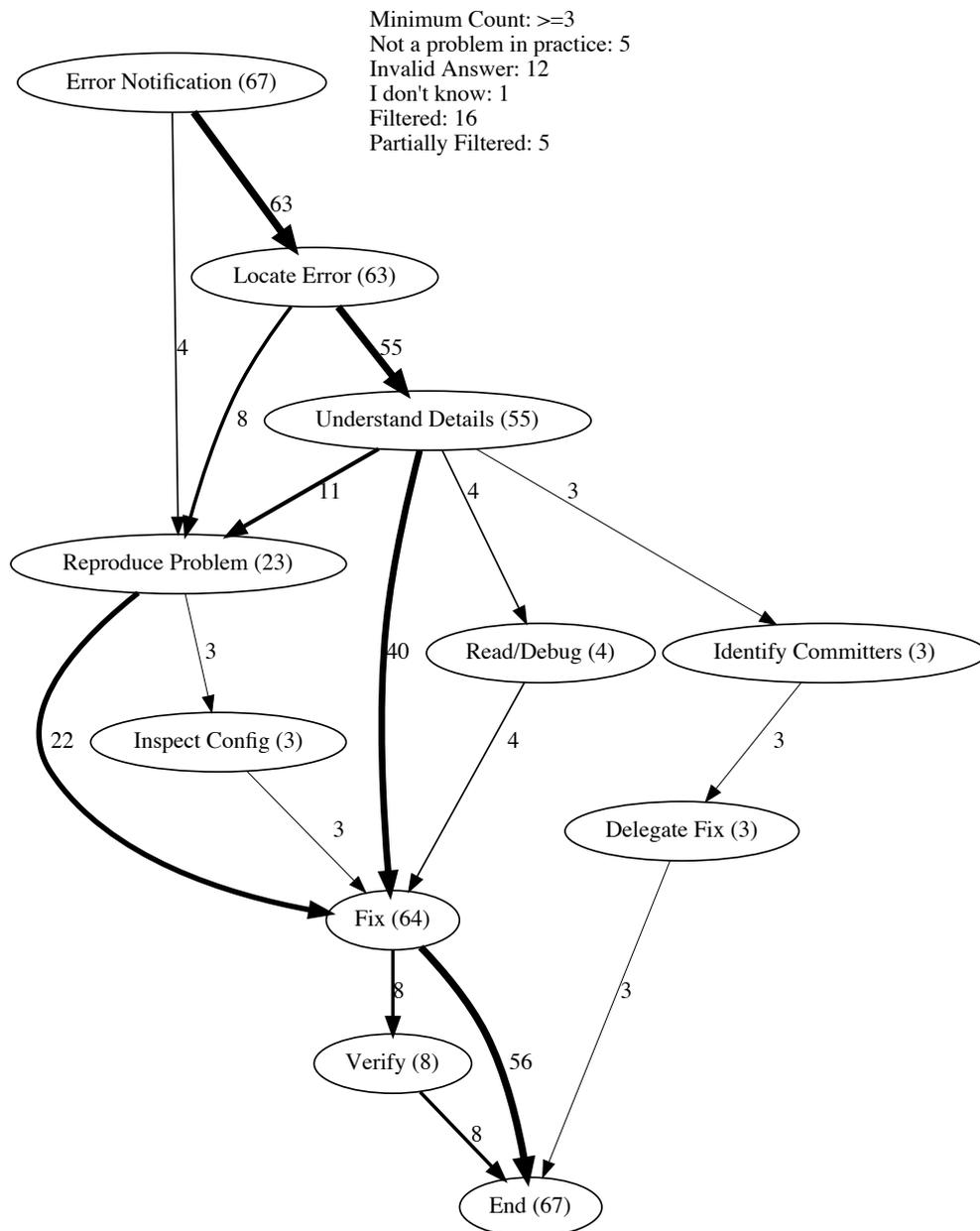
preserve as much information from the workflows as possible, we performed a *branch elimination*: We checked for each workflow whether it contained any of the filtered edges. If found, the edge was removed and so were all the previous and following edges that were not reachable otherwise. As an illustration, consider the previous example graph G . Assuming that the edge $c \rightarrow d$ is rare and should be filtered. In this case, we would not only remove $c \rightarrow d$, but also $a \rightarrow c$, which would otherwise create a “loose end”. We preserve the sequence $a \rightarrow b \rightarrow c$ in the graph, but count the workflow as a *partially filtered* case. In case the *branch elimination* removes all steps, we filter the whole workflow and count it as a *filtered* case.

The results of this aggregation are shown for the four different scenarios in Figures 5.7-5.10. Both the nodes and the edges contain counts that indicate how often the respondents have mentioned it. For the edges, we adapt the line width depending on the count to highlight the most common flows.

Extracted Workflows

In this section, we describe the workflows that we have identified for the main build failure categories. We illustrate the main steps that lead our participants to the resolution of the failures and we discuss the differences among the resulting workflows.

Compilation Failures (Figure 5.7) After being notified developers typically locate the error that informs about the compilation failure. In many cases, they need to understand further details from log such as the line number where the error occurs. This is often sufficient to immediately fix the problem. However, a few more steps are sometimes needed. Developers reproduce the failure locally (even immediately after the error notification or locating the error) and inspect changes made to environments such as a recently-introduced new version of the compiler. Others read the source code to increase their knowledge about the part of the system affected by the error. A few developers identify the authors of the last commits included in the failed build and ask them to fix the failure. Finally, only 8 developers verify the fix before committing it. Overall, compilation failures are considered relevant by the majority of our participants. Only 5 of our participants do not encounter these types of failures.

Figure 5.7: The process of fixing *Compilation* build failures

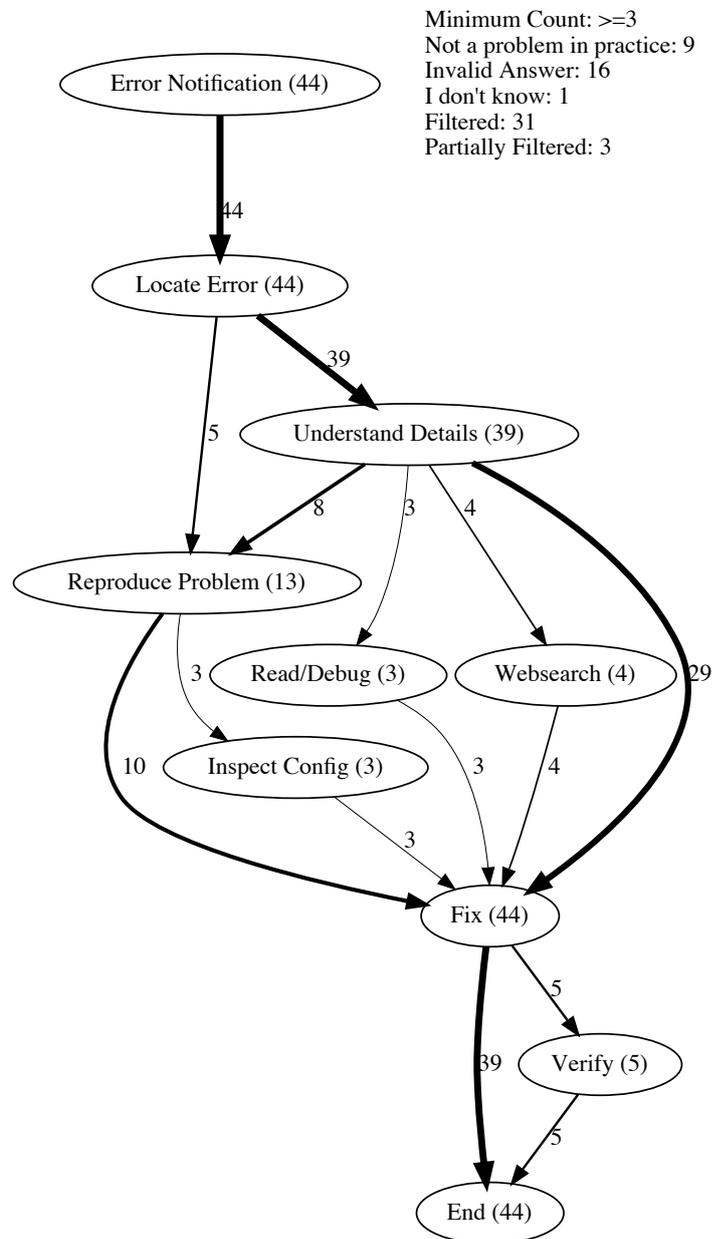


Figure 5.8: The process of fixing *Dependencies* build failures

Dependency Failures (Figure 5.8) Understanding the type of failure is the first step performed by developers when they run across a dependencies failure. Before directly applying the fix, the majority of our developers investigate additional information in the build log such as which dependency is missing or broken. In some cases, the information contained in the log is not enough to fully understand the problem and fix it. Thus, developers reproduce the failure locally (some of them avoid to fully inspect the log before) and depending on the severity of the problem they look at the differences between the local and remote environments (e.g., the dependency is available locally but not on the remote server). Others read source code requesting the dependency reported in the log or they search the Internet for similar failures. Finally, in a few cases, they verify the applied fix before committing it. Also in this case, only 5 participants consider dependencies failures as a problem not occurring in practice.

Testing Failures (Figure 5.9) Developers start fixing testing failures identifying the build failure type in the log. While a minority immediately reproduce the failure locally or start identifying people committing the last changes, many developers inspect the build log searching for the tests that did not succeed. Then 4 possible directions are taken before the problem is fixed. The majority of developers decide to investigate the test failure locally. They reproduce the last build and, if needed, debug the failed test or inspect the configuration files. Other developers inspect the history of code changes to identify (i) which was the previous version of the test or the code under test and (ii) which were the previous developers working on them. The goal of identifying committers is two-fold. On the one hand, developers can contact them to ask questions about the failure and generally to receive support while fixing the test. On the other hand, developers can delegate the fix to them. Surprisingly another direction is to ignore the failure. This happens when the code under test is not used or the test is obsolete or flaky (see Section 5.4.3). The last option is to immediately fix the problem based on the information retrieved from the build log. In a few cases, the fix is also verified. Testing failures are experienced almost by all participants. Only 5 developers do not consider them a real problem in practice.

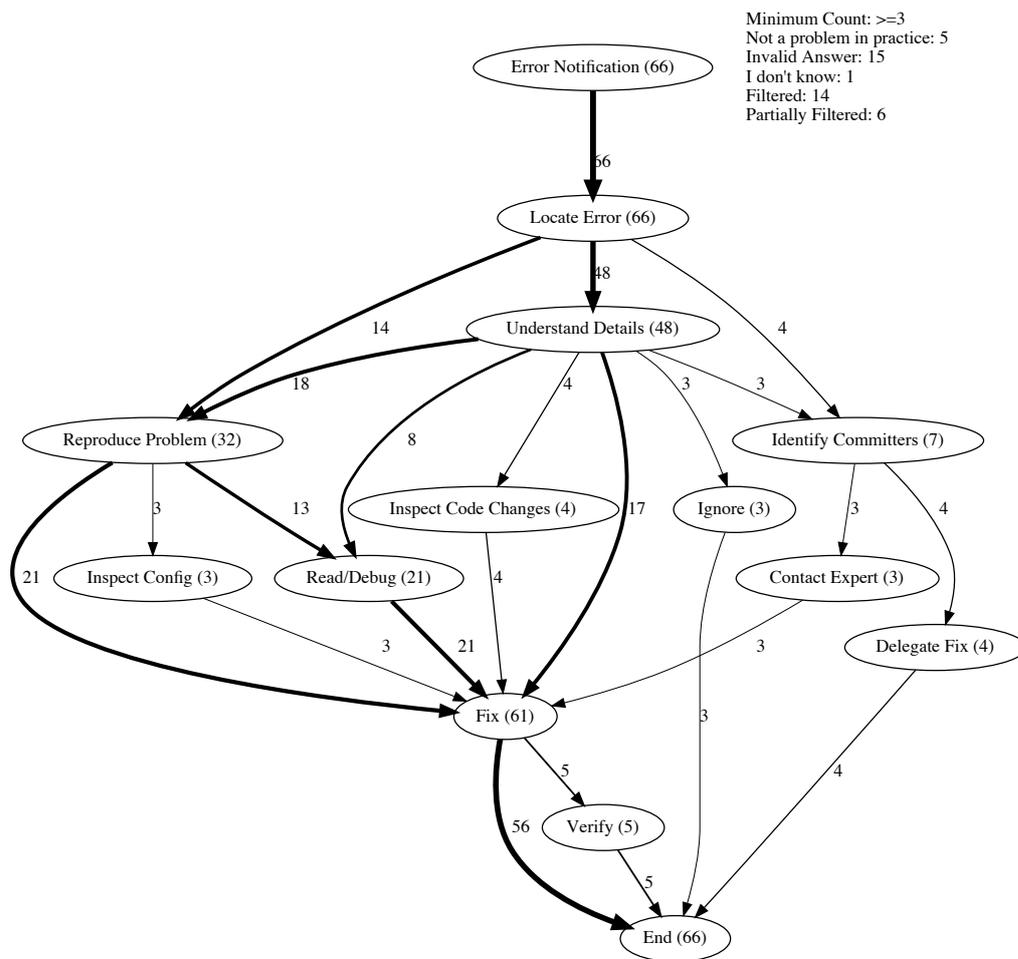


Figure 5.9: The process of fixing *Testing* build failures

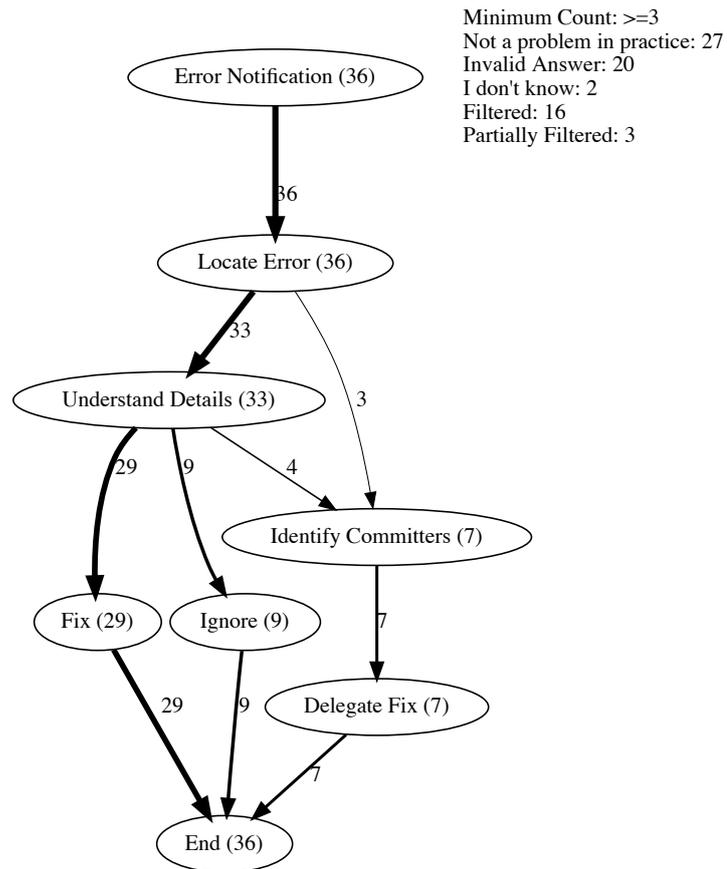


Figure 5.10: The process of fixing *Code-Analysis* build failures

Code-Analysis Failures (Figure 5.10) Code analysis are the failures with the highest percentage of “Not a problem in practice” answer (from 27 participants). These participants do not check for code quality during the build or, when it is checked, poor code quality is not set as a failing condition for the build. As regards the resolution workflow developers start investigating the type of the occurred failure. Then they either inspect the log to have more details about the warnings raised by the static analysis tools that provoked the failure or identify the committers of the last changes included in the failed build. The former step is used to judge the failure. If the violations are considered relevant developers apply the fix using the description available in the log. Otherwise, developers decide to ignore the violations and disable them in the configuration files of the invoked static analysis tools. The goal of the latter (that is sometimes performed without understanding more details from the log) is to delegate the fix to the authors of the last changes.

General Insights and Actionable Results

After discussing the four merged workflows separately for each scenario, we now reflect on observations that affect multiple of these workflows. Some interesting insights only appear when the various workflows are being compared.

Build failures are very rarely solved in a collaborative effort. Few developers mention that they get in contact with the authors of the “implicated” commits while fixing compilation, testing, and code-analysis failures. In the case of compilation and code-quality issues, the only purpose is to delegate the fix to them. For some testing failures that they feel competent to fix, developers get in contact with the committers to have more information about the change causing the failure and to receive feedback about the resolution strategy. Developers always try to solve dependencies failures themselves.

Developers often reproduce the failure locally to ease the task of inspecting the root cause in the code. If a build failure cannot be reproduced some developers also start inspecting differences between local and remote environments, suspecting that they caused the failure. While for other build failure types understanding details from the log is often sufficient to fix the error, most of the developers reproduce a testing failure locally before fixing. This means that the typical textual description

of the errors contained in the log is only a starting point for the investigation. To devise a resolution strategy developers also need to understand the semantic of the changes causing a testing failure and possibly debug the failure to identify bugs captured by the test. In such a workflow a better summarization of the build log would not help to plan a fixing strategy and solution hints are needed. However, websearch (which is the basis of our solution hints as described in Section 5.2.3) is rarely used and only to search for solutions to dependencies failures.

Code-analysis failures are considered “second-class” failures by many developers. They do not encounter build failures due to code-quality issues and such mistakes are typically unimportant/omittable and not worth scheduling during the build. Even when code-analysis is a proper step of the build, developers typically inspect the log not only to understand the issue causing the failure (that is typically easy to fix based on the description) but also to judge the violations and decide whether fix or ignore them.

Ignoring a failure is a much more common reaction for code-analysis failures. From the answers, the main reason seems to be that the high false-positive rate makes it necessary to ignore false warnings. Also, test failures are sometimes ignored, but here changing requirements are often named as the cause. The test is then ignored to give time to clarify requirements or fix implementation.

Developers often trust their devised solution. Only a few developers verify their solution, before integrating it into the master branch. This contradicts one of the key principles of CI, that is “run private builds” [25]. According to the principle, developers should emulate an integration build on their local workstation after committing their changes in order to prevent new build failures on the mainline.

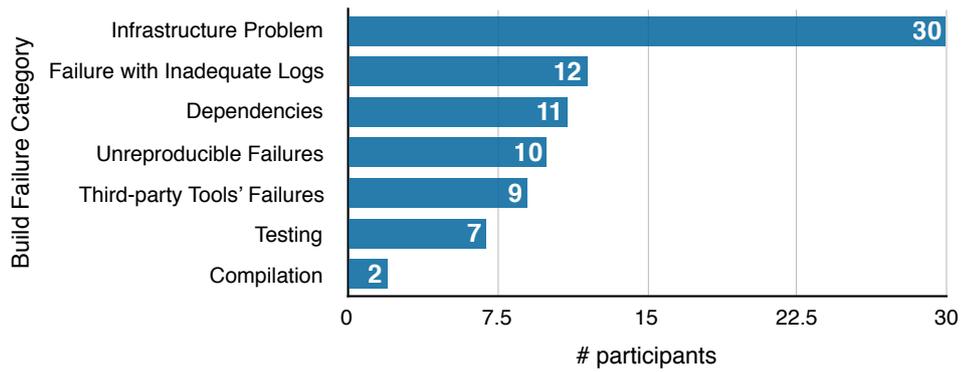
Coming back to our initial observation on how StackOverflow-based solution hints do not work for some categories, the most interesting insight from our study is that searching the internet for solutions is not a common practice. In the case of compilation, dependencies, and code-analysis failures developers often rely *only* on the information contained in the log. This confirms that a summarization approach pointing to the relevant information in the log can avoid the burden of inspecting thousands of lines of code and be sufficient to plan the fix. In the case of testing, developers need instead additional information to devise a fix strategy. However, our solution hints were considered irrelevant. Based on the workflow in

Figure 5.9, the reason is that developers typically reproduce the failure locally and debug it to guide the resolution process. Testing failures are too project-specific to search the internet for general solutions. Future extensions of Bart should ease the reproducibility of build failures and its solutions hints provide developers with the semantic content of the change causing the test failure.

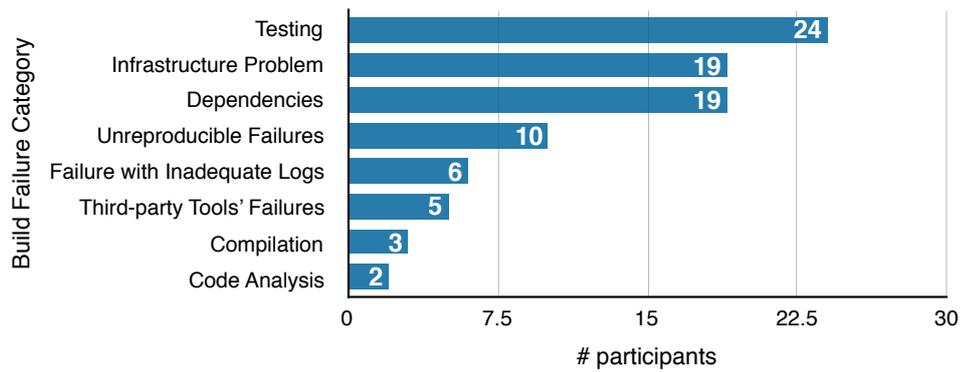
5.4.3 What Types of Build Failures Are Hard to Fix?

In the previous section, we have analyzed the most common workflows while solving build failures. Our participants typically start collecting relevant information about the problem and then try to understand it. In most of the cases build logs are enough to locate the error while especially in case of testing failures developers also need to reproduce the failure locally. As the answer to Q1.1, 62.4% of our respondents believe that understanding the problem is the hardest part. Instead, 19.8% think that finding relevant information is the most time-consuming activity, while 17.8% spend the same time both on collecting information relevant to the failure and on understanding the problem. We further investigate those two phases of the build resolution process and ask our participants in Q1.2 and Q1.3 which build failures are more difficult to inspect (i.e., require more time to collect information about their root causes) or which ones are instead harder to understand. We performed card-sorting to identify the recurring concepts in the answers to these questions and obtained the results presented in Figures 5.11a and 5.11b.

Failures Hard to Inspect Most of our participants consider *infrastructure problems* as the hardest to inspect (Figure 5.11a). They struggle to locate misconfigurations of the CI workflow that causes failures (e.g., the CI server is not able to fetch new changes from the remote repository) or locating errors that are caused by a poor definition of the build scripts. The second most cited failures (*failures with inadequate logs*) in Figure 5.11a do not relate to a particular category of failures. Failures sometimes happen on servers that are external to an organization, producing logs that cannot be accessed by developers. And build logs are often simply not enough to fully locate the cause of the failure. In the presence of *Dependency* breaks, looking for conflicting dependencies takes a long time for 11



(a) ... while finding relevant information



(b) ... while understanding the problem

Figure 5.11: Build failure types our participants struggle with

participants. Also the fourth most-mentioned failure (*Unreproducible failures*) is not specific to a particular failure category. These failures cannot be reproduced on the local machine of the developers and their build log is often not sufficient to fully locate the failure's cause. Developers need to replicate the error locally to generate additional log statements (e.g., by making Maven logging more verbose with the `-v` option) or reports (e.g., JUnit reports). Reproducing failures is sometimes impossible because of a significant difference between developer's and the remote build's environments. Other failures that developers find hard to investigate are caused by defects of the *third-party tools* that are used during the build. These failures are usually caused by unmet requirements (e.g., a specific version of the compiler is missing). Several participants reported *test* and *compilation* failures without providing a specific reason. Finally, 8 participants answered that looking for the error cause is straightforward (independently from the build failure type) while other 2 spend a lot of time in understanding all types of build failures.

Failures Hard to Understand Figure 5.11b illustrates the build failures for which our participants state that understanding is more difficult when deriving a fix. Despite finding the error that causes a *testing* failure is hard only for 7 participants (see Figure 5.11a), those failures are mentioned as the hardest to figure out. In this category, many developers mention integration and flaky tests. The second hardest type of failure is *infrastructure problems*. Developers report misconfigurations of the CI workflow and poor-defined build scripts as reasons. Many developers spend a long time not only on locating the outdated dependency but also on finding the right fix strategy. This is the reason why *Dependency* failures are also considered hard to fix. As in Figure 5.11a, also in Figure 5.11b 10 developers report *unreproducible failures*. The impossibility to reproduce the failure not only makes it hard to locate the error, but slows the understanding process down. Developers cannot verify (and refine) a hypothesized fix strategy without running a new build on the remote server. Third-party tools' failures sometimes require to replace a particular tool due to incompatibility with others used during the build or, in case of defects, as a temporary fix before the next update. This can take a long time and involve multiple developers. Finally, a few participants mention also *compilation* and *code-analysis* failures as the most time-consuming to understand. Only four

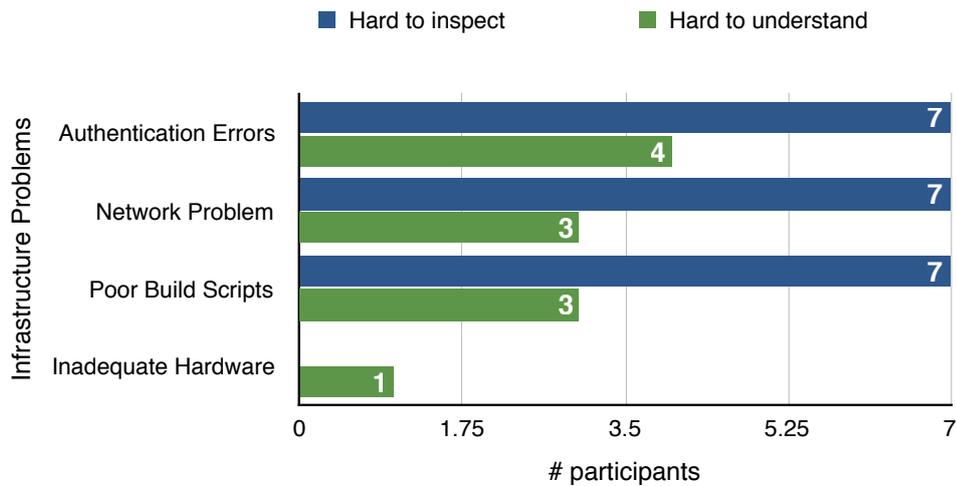


Figure 5.12: Root causes of infrastructure-related build failures

participants consider fixing all build failures a trivial task while one considers all types of build failures difficult to fix.

Cross-cutting Failures While the others can be directly mapped to known build failure types [134], *infrastructure problems*, *failures with inadequate logs*, and *unreproducible failures* are cross-cutting categories that are not associated with a particular stage of the build process. A *failure with inadequate logs* indicates the problem of having build failure logs that do not contain enough information to start inspecting the failure’s cause (sometimes the build log is not accessible). *Unreproducible failures* are build failures that cannot be reproduced in the developer’s machine. *Infrastructure problems* are heterogeneous and are also perceived by many developers as both hard to inspect and understand. Figure 5.12 illustrates which are the *infrastructure problems* reported by our participants as answers to Q1.2 and Q1.3. Note that the figure does not include answers that refer generally to infrastructure problems without specifying the problem.

Different machines are involved in a CI workflow. Some of them are also not directly controlled by an organization. *Authentication errors* provoked by wrong credentials or *network problems* bringing about the unavailability of servers are mentioned by several participants as causes of build failures. Poor build scripts (e.g. scripts which contain hard-coded credentials or other security smells, can

be susceptible to security breaches [94]) contribute to generating build failures. One participant mentions *inadequate hardware*. It can slow down the development process and generate server timeouts.

Discussion of Painful Build Failures Previous work [134] has shown that the most recurrent types of build failures are compilation, dependencies, testing, and code-analysis failures. While only a few participants mention compilation and code-analysis, our study on the perception of build failures reveals that testing and dependencies failures are also considered particularly hard to understand. These results confirm the importance of assisting developers during these types of failures and especially in case of testing failures that are considered as the most difficult among all the mentioned failures. Our build summarization approach increases the understandability of both categories. Together with a few “known” categories, new ones emerged from our analysis as the most painful categories. Those failures do not relate to a specific build phase and are not the direct consequence of committed code changes. They are caused by choices made while configuring the build process.

Developers use configuration files to define which tools are needed during the build and which are the requirements of the environments where the build is performed [34]. Depending on how the build process has been configured developers can encounter failures that are related to the infrastructure, i.e., the way those tools are connected and invoked during the build process. As shown in Figure 5.12, those failures can be caused by authentication or network errors between connected tools that are often hosted on different servers. Poorly defined build scripts or configuration files that define the build process lead to the introduction of smells, that are symptoms of deeper problems in the build process definition [34]. For example, build smells are indicative of security weakness [94] or misuse of CI features [34]. Based on our results, the presence of these smells also decreases the understandability of build failures. In the presence of less comprehensive build scripts, developers have difficulties understanding what to change in the configuration to resolve the failure.

Developers can control the amount of information that is generated by tools invoked during the build process through log statements. Having short logs reduces the amount of information developers have to read, but increases the time needed

to find relevant information and makes the failures even harder to be understood. Our summarization approach tackles this problem pointing developers directly to relevant information in verbose logs.

Developers can also configure the environment, in which the build is performed. Those environments are usually very different from the local environment, in which developers implement code changes. For example, using several operating systems or different versions of a compiler. Reproducing an error is a crucial step towards understanding, but when a failure occurs on the build server, developers find it very difficult to reproduce the error in cases, in which local and remote environments are significantly different.

Among the most recurrent types of build failures, testing and dependencies are considered hard to fix. Furthermore, other failures that are not build-phase specific and are instead caused by the misconfiguration of the build process are considered painful.

5.5 Discussion

In this section, we discuss the main findings of our study and their implications for researchers and practitioners.

For the RQ₁, we found that summaries improve the understandability of the build logs as it happens for source code [80]. Developers unanimously agree that build summaries improve the understandability of build logs across the most recurrent categories of failure.

In RQ₂, we discovered that improving the understandability of the build failures has a direct impact on the time needed to solve them. Across all the categories covered in our study, build summaries speed up the build fixing process: the time to fix is reduced by 23% in case of testing failures, by 20% in presence of compilation errors, by 43% in case of dependency-related issues, and even 62% in case of code violations. On average, developers using our tool spend 37% less time on solving build failures.

Among the hints included in our summaries, there is a category of hints that are generated by mining build failure solutions from StackOverflow discussions. The applicability and relevance of the proposed solutions are quite high in case of compilation and code analysis, less in case of dependencies failures. Furthermore, most of the times the solutions proposed for testing failures are not considered valuable. In RQ₃, we found that this is mainly caused by the lack of semantic information. Testing failures are project-specific and search for the root cause on the internet is not effective.

Although solution hints from the internet are ineffective, in RQ₄ we observed that developers struggle mainly while fixing testing and infrastructure-related failures. They find it difficult to derive a solution strategy for those failures and more effective solution hints that leverage other sources than the internet might be beneficial. Infrastructure-related failures do not refer to a particular build step and they are not directly caused by committed changes. They are issues provoked by the *smelly* configuration of a CI process.

Implications and Future Work Our findings have important implications for both researchers and vendor of CI servers. Existing tools provide a build overview, but refer developers to the build logs for details, for example, to understand the reason for a build failure. Our results suggest though that build logs are difficult to understand and that integrating summaries of the build failure into the build overview can support the comprehension process.

Despite not automating manual activity, we show that providing solution hints that link to external resources (i.e., different from the build logs) can be useful to developers. They can provide additional context that can help to derive a solution strategy, especially when the root cause of a build failure is unclear or when the solution is non-trivial. So far, our infrastructure only considers StackOverflow discussions as external resource. Future hint generators could consider other resources, like generated reports or information about deployed libraries, to create a more holistic picture of the failure.

A better context awareness of the summarization tool might help to overcome existing limitations, e.g., solution hints for testing failures. Based on the build fix workflows analysis, developers frequently inspect previous changes to understand

the cause of the failure or simply for learning from solutions applied to the same problems in the past. We argue that historical information and the semantics of a change can be an important source of hints for suggesting fixes to build failures.

This work introduces a technique to support developers when fixing a build break by providing them with summarization and solution hints. However, some build breaks cannot be reproduced locally and need to be solved on the server. Future work should investigate new ways of bridging this gap by considering differences between the remote CI environment and the local IDE environment when searching for solution hints. Novel debuggers, tailored to CI workflows, might help to improve the effectiveness when fixing build breaks.

Code-analysis failures are typically perceived as easy to fix, yet, they are the ones that benefit most from the adoption of our tool. Our participants also consider solution hints for code-analysis failures to be very relevant in the build-fix process. A comparison to other failure types indicates that developers tend to first judge the code-analysis failure, to decide whether the failure is worth fixing or whether it should be ignored. Solutions hints can support this decision by providing more details about the warning. For example, in case of the violation ‘line is too long’, a link to a related discussion of appropriate line length might help the developer to decide whether the warning is indeed a violation. Tool vendors can further support this step by ensuring that violation descriptions contain relevant contextual information.

For specific types of build failures, i.e., infrastructure-related failures, that are caused by weaknesses of the adopted development pipeline rather than by the change committed by a developer future hint generators should be conceived. In particular, by extracting information from different nodes (e.g., servers) of the pipeline those generators must be able to provide developers with the location of the error.

Assistance tools like Bart do not only have a positive effect on the developer that fixes the build. Supporting the individual developer has the potential to increase team productivity, because it reduce the team downtime that would normally be caused by the build break. Future work should investigate novel build log summarization techniques to reduce the required time even further.

5.6 Threats to Validity

The work presented in this paper was carefully planned and executed, but several threats to validity exist for our results. In the following, we will discuss them and our mitigation strategies.

Threats to *internal validity* concern the confounding factors that might have affected our results. We did not deploy our tool in a real industrial environment and only created versions of the software with artificial errors to evaluate our tool, which might not be representative of real errors. We tried to mitigate this by injecting bugs that resemble the most common causes of build breaks [107, 134]. Also, injecting realistic faults is a common trade-off in the design of many other studies (e.g., [87], [32]), so we strongly believe that our setting is valid. Another aspect that might affect the reliability of our results is the complexity of systems considered during the analysis. We tried to reduce this threat by considering build breaks in our study, which belong to projects that are not too big, but at the same time representative of real systems. It is also possible that our participants didn't fully understand the questions in our questionnaire. We have reserved time before starting, to allow participants to ask questions about the experimental procedure. Another threat is the manual time measurement that could introduce a bias. However, the substantial differences that we have measured far exceed the imprecision of the manual measurement. Other build summarizers might exist and requiring our participants to read a plain-text build log could introduce a bias in our experiment. However, we are not aware of any frequently used summarization tools and we think that using the information that is available in a standard Jenkins installation represents a valid baseline for our comparison. The subject understands the treatment. However, we could not really hide the treatment. Given that all participants have experience with Maven, every solution different from inspecting the raw build log would have been considered the treatment. In CI typically breaks are caused by recent changes, but our subjects do not have access to the history. However, breaking changes are hidden in big commits that tangle several changes, so a developer must first understand the error in order to know what to look for in the commit. Learning effects might blur the results because Bart is always used in the second task. We took strategies to mitigate this potential bias. Participants

have been coached in the beginning and we made sure that all participants have the required knowledge to fix build problems (in particular, all participants have used Maven before). Developers have worked on tasks that affect different areas in the systems and the workflows to address these problems differ, so we expect low learning effects.

Threats to *external validity* concern the generalizability of our findings. We considered only four types of build breaks in our study. However, those represent the most relevant and recurrent categories of build breaks that have been observed [10, 75, 107]. Furthermore, the participants in our study could be unrepresentative of all kind of developers. We mitigated such threat trying to reach for both survey and controlled experiment people with different programming skills, to make general consideration about beginner and expert developers. Our tool, Bart, is the first implementation of an approach for build logs summarization. The current design of our study only looks at errors introduced by users. Future work should expand the scope and investigate build errors that are caused by the environment of the build server (e.g., different locale settings). The results presented in this work might not generalize beyond the considered build failure types.

5.7 Related Work

This paper is related to three lines of research: works on build failures, source-code summarization, and mining Q&A sites. In the following, we will discuss the most related previous works from these areas and relate them to the work presented in this paper.

Build Failures Prior studies have investigated the nature of build breaks. Miller [75] found that the most recurrent causes of build failures in Microsoft projects are poor code quality, testing failures, and compilation errors. Other researchers [10, 100] studied the frequency of different build failure types in open-source projects, finding that builds generally fail because of failed test cases. In our study, we focused on the most common build failure types according to those studies. While several works focused on one particular type of build failure, e.g., code analysis [142] or compilation [107], Vassallo et al. [134] proposed a broader taxonomy of build

failures. They have analyzed 418 Java-based projects from a financial organization and 349 Java-based OSS projects and have identified differences and commonalities of failures between industrial and open-source projects. Because we summarized Maven build logs of Java projects, we decided to reuse this taxonomy to categorize our build failures. Kerzazi et al. [50] have analyzed 3,214 builds in a large software company over a period of 6 months to investigate the impact of build failures on the development process. They observed a high percentage of build failures (17.9%), which aggregate to a cost of more than 2,000 man-hours when each failure needs one hour of work to be fixed. Vassallo et al. [129] found that the problem of fixing build failures is even becoming more relevant in open-source projects. Developers often cause build failures on the release branch and it takes (on median) 17 hours to them. Some of these failures are noisy and complex [33]. Given that some builds have a misleading or incorrect status, developers are required to carefully inspect the build outcome to verify the presence of passing jobs. Thus, build failures slow down the release pipeline and decrease team productivity because they interrupt implementation activities [135]. This was one of the motivations for our study: providing developers with a tool able to support them while fixing build failures making the recovery process faster.

Existing plugins try to achieve the same goal. For example, Log Parser¹² is a Jenkins plugin that allows developers to add custom parse rules in the form of regular expressions. Matching parts of the build log are then highlighted for the developer. Bart pursues a different goal. It automatically selects the relevant information with no effort required from developers and organizes this information in summaries and by linking external information. Researchers proposed approaches to automatically repair builds that break due to dependency [66], testing-related issues [120], and that can be fixed by generating patches that are applied to buggy build scripts [43, 64]. The focus of Bart is *developer-oriented* and complementary to automated approaches. We assume that very often developer interaction is required to fix a build. Therefore, we try to empower the developer by improving build log understandability through summarization and linking to external resources.

¹²<https://wiki.jenkins.io/display/JENKINS/Log+Parser+Plugin>

Source-Code Summarization During their regular work, developers have to cope with a large amount of external data [41], e.g., bug reports or source code, which is produced during software development. They need support while trying to comprehend such data and summarization techniques can facilitate this process. Several techniques have been proposed to summarize source code [80]. Haiduc et al. [41] proposed automatic source code summarization leveraging the lexical and structural information in the code. Moreno et al. [77] conceived a technique to automatically generate human-readable summaries for Java classes relying on class and method stereotypes in conjunction with ad-hoc heuristics. Other approaches generate summaries from source code artifacts, such as code fragments [141] or code usage examples [79]. Moreover, Panichella et al. [87] studied the impact of test case summaries on the number of fixed bugs, proposing an approach that automatically generates summaries of the portion of code exercised by each individual test. Other researchers focused on the summarization of build reports [98] or user reviews [110]. Our approach complements these approaches and presents a novel summarization approach for another important software development artifact, i.e., the build log.

Mining Q&A Sites Question and answer websites like StackOverflow have been analyzed by several researchers to provide developers with helpful data during software development. Ponzanelli et al. [88] enhance the IDE with Prompter, a tool that automatically captures the code context in the IDE to propose related StackOverflow discussions. Bart is very similar to this work, it is integrated into the build server and acquires contextual information about failing builds to assist developers with deriving a fix. Other researchers, investigated the impact of using StackOverflow on development workflows. Vasilescu et al. [123] analyzed the interplay between StackOverflow activities and code changes on GitHub. While a switch to StackOverflow interrupts the coding, they were able to show a correlation between visits of StackOverflow and code changes. Developers seem to frequently switch between their IDE and StackOverflow when they get stuck, which supports our assumptions of Section 5.2. Finally, other researchers generated summaries for Java classes [139] and methods [127] by mining source code descriptions on StackOverflow. We also extract information from StackOverflow but follow a different goal, i.e., providing hints for build fixes.

5.8 Summary

This paper presented Bart, a system that supports developers in understanding build failures and effectively fixing them. Bart works on the build log, summarizes build failures, and provides solution hints using data from StackOverflow. We conducted an empirical study with 17 developers to assess the effect of Bart on repairing build breaks. Our results show that developers find Bart useful to understand build breaks and that using Bart substantially reduces the time to fix a build break, on average by 37%. To explain observations in our experiment, we have also conducted a qualitative study to better understand the workflows and information needs of developers fixing builds. We found that typical workflows differ substantially between various error categories and that several uncommon build errors, for example, errors related to the infrastructure, are very hard to investigate and to fix. These findings will be useful to inform future research in this area.

Acknowledgements

We would like to thank all the study participants. C. Vassallo and H. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275).

6

Configuration Smells in Continuous Delivery Pipelines: A Linter and A Six-Month Study on GitLab

Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, Massimiliano Di Penta
Published in Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Technical Research Paper (ESEC/FSE), 2020

Contribution: prototype design, experiment design, data selection, manual analysis of configuration smells, data analysis, and paper writing.

Abstract

An effective and efficient application of Continuous Integration (CI) and Delivery (CD) requires software projects to follow certain principles and good practices. Configuring such a CI/CD pipeline is challenging and error-prone. Therefore, automated *linters* have been proposed to detect errors in the pipeline. While existing linters identify syntactic errors, detect security vulnerabilities or misuse of the features provided by build servers, they do not support developers that want to prevent common misconfigurations of a CD pipeline that potentially violate CD principles (“CD smells”). To this end, we propose CD-Linter, a semantic linter that can automatically identify four different smells in pipeline configuration files. We have evaluated our approach through a large-scale and long-term study that consists of (i) monitoring 145 issues (opened in as many open-source projects) over

a period of 6 months, (ii) manually validating the detection precision and recall on a representative sample of issues, and (iii) assessing the magnitude of the observed smells on 5,312 open-source projects on GitLab. Our results show that CD smells are accepted and fixed by most of the developers and our linter achieves a precision of 87% and a recall of 94%. Those smells can be frequently observed in the wild, as 31% of projects with long configurations are affected by at least one smell.

6.1 Introduction

Continuous Integration (CI) and Delivery (CD) are widely adopted practices in software development. A CI/CD pipeline automates the process of building, testing, and deploying new software versions. There is plenty of empirical evidence for the positive effects of using CI/CD, including early defect discovery [46], increased developer productivity [124], and fast release cycles [25]. To achieve these benefits, it is recommended to follow various principles and best practices. For example, developers should build and test the software on every change that is committed to a project's version control system [27]. Several catalogs of CI/CD best practices exist [25, 26, 47, 103], but while their adoption has been advocated in research papers, white papers, and books, developers have difficulties to apply them in practice [45], deviating from principles and generating anti-patterns [129].

Some of these anti-patterns are related to the way developers use a CI/CD pipeline. For example, developers do not integrate their changes frequently or they remove failed tests to repair a build failure, and previous researchers [129] implemented tools that help developers avoid those bad practices by analyzing logs and past changes. Other anti-patterns emerge when the CI/CD pipeline is configured. To support developers when configuring CI/CD pipelines, DevOps build servers such as GitLab¹ can validate their configuration files using online linters.² However, those tools only spot basic syntactic errors such as the use of reserved keywords when naming build steps.

Previous works have proposed approaches for detecting misuses of specific configuration options. Gallaba et al., [34] achieved a high user acceptance when

¹<https://about.gitlab.com>

²<https://docs.gitlab.com/ce/ci/yaml/README.html#validate-the-gitlab-ciyaml>

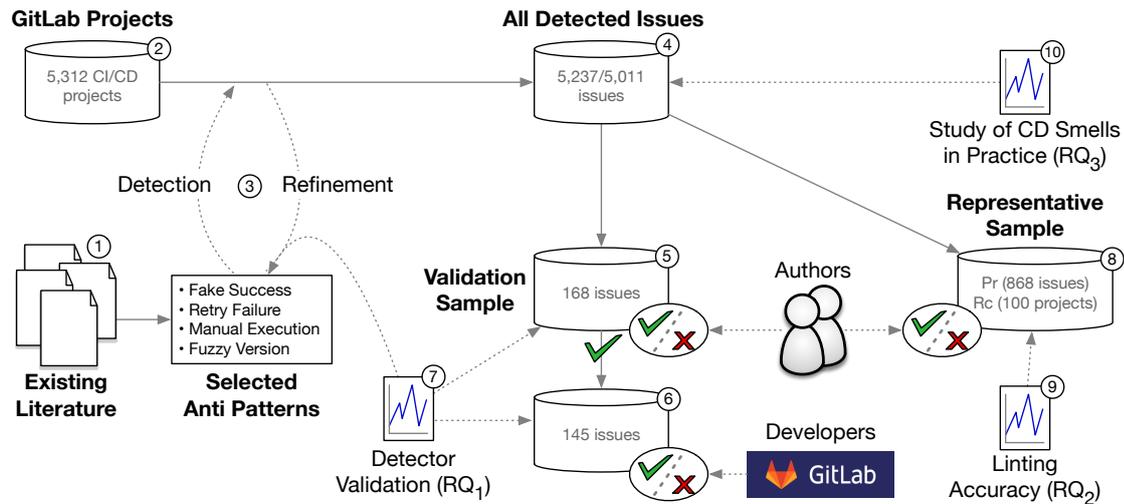


Figure 6.1: Overview of the methodology for evaluating the usefulness of CD-Linter

pointing out the misuse of four different configuration options, like executing commands in the wrong build step. Rahman et al. [94] focused on security-related issues and Sharma et al. [108] on Infrastructure-as-Code (IaC) smells. While these works show that semantic linting of CD pipelines is useful, they do not solve the problem of avoiding CD anti-patterns in configurations files. For example, a systematic manual job execution is not a misuse, but it violates a CD principle.

In this work, we want to help developers with configuring their CD pipelines by helping them to avoid violations of accepted CD principles in their configuration files. We propose a novel semantic linter named CD-Linter, to detect process-related violations of CD principles, in the following referred to as “CD smells”. CD-Linter is currently capable of detecting four types of CD smells that are related to violations of principles and best practices described in the literature [26, 47]. We evaluated CD-Linter through a large-scale and long-term study consisting of 145 issues opened in as many projects. We monitored the reactions to those issues over a period of 6 months and found that 53% of the project maintainers agreed with the reported CD smells either accepting the issues (9%) or directly fixing them (44%). We also analyzed the reasons for rejecting issues and use them to further improve CD-Linter. Finally, we measured the accuracy of the latest version of CD-Linter and investigated the occurrence of the four CD smells in the wild.

The contributions of this paper can be summarized as follows:

1. The operationalization of four violations of CD principles in pipeline configurations (CD smells), and the empirical validation of their relevance.
2. CD-Linter, an open-source semantic linter that can detect these CD smells in configuration files of GitLab pipelines. We show that, overall, CD-Linter has a precision of 87% and a recall of 94%.
3. A large-scale empirical investigation of the extent to which the considered CD smells occur in a large set of 5,312 open-source projects.

All datasets and scripts used in our studies (together with CD-Linter implementation) are available in our replication package [130].

6.2 Methodology Overview

This paper investigates the problem of violating CD principles (i.e., CD smells) in configuration files of CD pipelines. We propose CD-Linter, a semantic linter that detects CD smells and evaluate its usefulness by answering the following research questions:

RQ₁: *Are the four CD smells relevant to developers?*

RQ₂: *How accurate is CD-Linter?*

RQ₃: *How frequent are the investigated CD smells in practice?*

Figure 6.1 provides a high-level overview of the different parts of this paper, the details of the empirical study design will be covered in Section 6.4. Inspired by existing literature in this area, we started by selecting four CD smells that affect the definition of CD pipelines (1) (Section 6.3.2 provides more details about the selection). To study these CD smells, we selected a dataset of 5,312 open-source projects that are publicly available on the GitLab platform (2). We built detectors, ran them against the dataset, and incrementally improved the corresponding detection strategies (3). The four CD smells types and their detection strategies will be introduced in Section 6.3.

Initially, we detected 5,237 smells in our dataset (4). To validate the relevance of the selected CD smells and the correctness of our detectors, we started to open issues in the issue trackers of the affected open-source projects. We used feedback

from the early iterations to further improve the detection strategies, once we were confident that the detectors work properly, we created a balanced sample of 168 issues (5). After validating the reports manually, we rejected 23 issues and posted the approved 145 issues to the issue trackers of the corresponding open-source projects. We then monitored how professional developers reacted to the opened issues for 6 months (6). In Section 6.5.1, we will answer **RQ₁** by analyzing the internal rating of the authors and the reactions of the original developers to our reports (7).

The feedback that we received through rejected issues also enabled us to further improve our detection strategies, which reduced the total number of identified issues to 5,011 (4). We created a stratified sample of 868 issues to validate the precision of our detectors on a large scale, while we validate the recall by manually inspecting 100 projects (8). The sample size made it infeasible to open further issues on GitLab, because we could not have followed-up on all of them, so we only rated the validity of these issues internally. Our rating provided the required data to answer **RQ₂** (9), which will be discussed in Section 6.5.2.

Finally, we analyzed the results for the complete dataset of 5,312 projects to investigate how frequently CD smells occur in practice (10). These results will be discussed in Section 6.5.3.

6.3 CD-Linter Description

Organizations implement *CD pipelines* using pieces of technology such as Jenkins, TravisCI, or GitLab. While this paper copes with build-server agnostic smells, we implement CD-Linter on GitLab. GitLab is an integrated platform that hosts both the repository and the issue tracker, which is particularly interesting for our evaluation. In February 2020, a search via the GitLab API revealed that the site hosted more than 1.57M projects. As GitLab can also be used in private installations, this makes it a very popular solution for enterprises.³ By supporting GitLab, CD-Linter targets industrial and open-source projects alike.

³<https://about.gitlab.com/analysts/forrester-cloudci19>

6.3.1 Background

In the following, we provide some background information about the relevant configuration parts for this work.

Build Server A build server is a reusable infrastructure, which enables developers to define custom CD pipelines and is configured through configuration files. In GitLab the configuration file is *.gitlab-ci.yml*; other build servers have similar configuration files.

An example of such a configuration file is shown in Figure 6.2. In the top part of the file, the stages of the build, that every change committed to a version control system as `Git` has to pass during the build, are defined. If no stages are defined, the default stages in GitLab are `build`, `test`, and `deploy`. The automation tasks are defined as *jobs*, the basic unit of the CD pipeline. The example defines two jobs, `code_quality` and `unit_test`, which invoke specific shell scripts that are defined in the `script` line. For example, a Java project could include the script line `script: mvn test` to start all unit tests through the Maven build tool. Developers can also configure when to run a job (e.g., `when: manual`), how many times a job can be auto-retried in case of failures (e.g., `retry: 3`), and whether a job is allowed to fail (i.e., `allow_failure: true`).

Specialized Build Tools In addition to the configuration of the high-level orchestration of the build pipeline, most CD pipelines use specialized build tools to perform the actual automation tasks, which require separate configuration parameters. In contrast to the build server, these build tools depend on the programming language that is used in the project. For this paper, we chose to support the typical build tools of Java and Python to have two representatives for strictly-typed and dynamically-typed languages. The typical configuration differs between these languages and is too complex to be covered here. We will introduce the relevant bits, once we have described the CD smells that we are going to support.

```
stages:
  - build
  - test
  ...

code_quality:
  stage: build
  script: "mvn sonar:sonar"
  when: manual # Manual Execution

unit_test:
  stage: test
  script: "mvn test"
  retry: 3 # Retry Failure
  allow_failure: true # Fake Success
```

Figure 6.2: Example excerpt of a GitLab configuration

6.3.2 Selection of Relevant CD Smells

CD-Linter features the implementation of an initial set of CD smells to be evaluated. Clearly, there may be many smells in CD pipelines (e.g., Duvall [26] defined 50 anti-patterns). Practically speaking, a CD-Linter can detect a limited subset of smells, and, being a linter, only those that can be statically identified. Therefore, we aimed to find a set of suitable CD smells, not all the most relevant ones. We collected all the good and bad practices that are illustrated in the *Foundations* part of Humble and Farley [47], a well-known book about CD practices. Some CD smells require historical information for the detection (using artifacts like logs or repositories), which is only available after the CD pipeline is being used and not when it is configured [129]. This is out-of-scope for a static linter, so we judged the feasibility of detecting the anti-patterns from configuration files alone, without relying on other artifacts. The complete list is available in our replication package [130] and we selected four CD smells.

Fake Success Each stage of the CD pipeline checks for several categories of defects. For example, jobs executed in the code quality stage can reveal the presence of poorly-written code snippets, while jobs in the test stage typically

spot bugs at unit and integration levels. Every executed job should be able to fail the build. If not, developers can miss or ignore the underlying issue, which adds technical debt and might result in problems later. A *Fake Success* arises when a failure in a job does not affect the overall build result.

Retry Failure The build process has to be deterministic. Flaky behavior, e.g., tests that sometimes fail [65], should be avoided at any cost, because they hinder development experience, slow down progress, and hide real bugs. Some pipelines address this issue by rerunning a job multiple times after failures. However, this might not only hide an underlying problem but makes issues also harder to debug when they only occur sometimes.

Manual Execution CD means to keep the code base in a deployable stage at any given time. Thus, a fully automated build process up until the deploy stage is required. Manual jobs might introduce errors and delay the delivery of code changes to the customers. This CD smell occurs when a job (that is executed before the deploy stage) needs to be explicitly started by a user.

Fuzzy Version Developers should always specify the exact version of the external libraries that are used. If not, a build could not be reproduced. Failing to be specific on versions also leads to an occasional long debugging session tracking down errors due to the use of different library versions. Using the terminology of *semantic versioning*, we differentiate between the following sub-types of the smell: (i) *Missing Version*: No version number is defined; (ii) *Only Major Version*: Only a major release number is defined; (iii) *Any Minor Version*: Any equal or higher minor release with the same major version is allowed; or (iv) *Any Upper Version*: every equal or higher version can be used.

6.3.3 Parsing CD Configuration Files

To detect the CD smells, we parse the configuration files and map their content onto meta-models that we have created for each type of configuration files CD-Linter supports. These meta-models cover the parts of the configuration files that matter for the detection of the CD smells. CD-Linter considers three types of configuration

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>a</groupId>
  <artifactId>b</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties><x.version>1.0.0</x.version></properties>
  <dependencies>
    <dependency> <!-- correct -->
      <groupId>foo</groupId>
      <artifactId>x</artifactId>
      <version>${x.version}</version>
    </dependency>
    <dependency> <!-- Missing version -->
      <groupId>bla</groupId>
      <artifactId>blubb</artifactId>
    </dependency>
  </dependencies>
  <modules><module>module-example</module></modules>
</project>
```

Figure 6.3: Example of a pom.xml file

files used for GitLab (*.gitlab-ci.yml*), Maven (*pom.xml*), and pip (*requirements.txt*). In the following, we describe how we parse these configurations for our purposes.

GitLab Configuration From the *.gitlab-ci.yml* file, we capture the list of jobs, the stages, and the variables. For each job, we record the name, the stage, and the script lines (*script*, *before_script*, *after_script*) as well as the *retry*, *allow_failure*, *when*, and *environment* parameters. For the *retry* parameter, we keep track of the maximum number of retries (*max*) and for which kinds of failures the job is allowed to be retried (*when*). We filter out dot-prefixed jobs as GitLab does not process them.

Maven and pip Configurations The Maven build tool is very popular among Java projects. Maven can automate various tasks, for example, the dependency resolution and automated download from a centralized repository. Figure 6.3 shows a configuration excerpt (“*pom.xml*”), in which two dependencies, *foo.x* and *bla.blubb* are being defined.

From the *pom.xml*, we capture the unique *coordinates* of the artifact (artifact ID, group ID, version), all defined properties, and the *coordinates* of all dependencies. All properties are automatically replaced with their actual value. We also include all referenced *modules* recursively and link them together. Values such as versions are then inherited from ancestor POMs where available.

As regards pip, the most used package manager for Python code, two things are relevant. First, the file *requirements.txt* is often used to define all dependencies that are required in the Python environment to run a particular piece of software. These requirement files can be hierarchical and include other requirement files that are inherited. Second, the `script` line in the GitLab configuration file often contains manual calls to pip to download external dependencies. To find these, we search for the keyword `pip install`, strip other pip options, and remove quotes from the arguments. It is also possible to specify dependencies by pointing to files, folders, and URLs to version control systems. We use simple heuristics to detect these cases and exclude them from the linting.

6.3.4 Detection of the CD Smells

Having access to the parsed information in the meta-models, we could proceed to implement the various analyses that detect instances of the four CD smells.

Fake Success The `allow_failure` parameter set to `true` allows a job to fail without impacting the rest of the build. Figure 6.2 shows an example in which the build execution can succeed despite potential errors in the `unit_test` job. We detect a Fake Success every time a job's definition contains `allow_failure: true`. Note that we do not report Fake Success for the stages `sast` (static application security testing) and `dast` (dynamic application security testing). GitLab defines templates^{4,5} that contain `allow_failure` set to `true` for the default job used in the these stages.

Retry Failure The `retry` parameter allows developers to configure how many times a job is going to be retried in case of a failure (see the example in Figure 6.2).

⁴https://docs.gitlab.com/ee/user/application_security/dast

⁵https://docs.gitlab.com/ee/user/application_security/sast

We detect all cases in which `retry` is set to a positive value. The proposed solution for such a case is to control `retry` by matching a specific failure cause (e.g., `when:runner_system_failure`). We only found very few cases in which projects used `when`, so we decided to simplify the detection in CD-Linter and report all such usages of `retry` for now. Handling these cases properly is a simple matter of implementation.

Manual Execution The `when` parameter can also be used to specify when a job shall be executed. To detect manual triggers of steps, we selected all jobs that contain `when:manual` in their definitions. For example, job `code_quality` in stage `build` (Figure 6.2) needs to be manually started by a user.

Not all manual triggers are a problem though. CI/CD advocates the automated execution of all stages to ensure a releasable project state at every point in time, however, it is acceptable to manually decide when this release should happen. Therefore, we do not report cases in which the manual execution only affects deploy stages. Apart from using the default `deploy` stage, GitLab users can also define custom deploy stages.⁶ To build a comprehensive list of deploy stage names, we extracted the stage names from a random project sample of our dataset (see Section 6.4). We identified all keywords that hint at a deploy stage such as ‘`deploy`’, ‘`release`’ or ‘`publish`’ and exclude jobs and stages that contain these keywords in their name. Also, we did not report Manual Execution for jobs in the `triage` and `review` stages, because GitLab suggests that these stages should be started manually.^{7,8} Furthermore, we exclude cases where the `action` parameter of `environment` is set to `stop`, which used to define a manual way to shutdown an environment that is used in the build.

Fuzzy Version The way dependencies are declared is specific to the programming language and the corresponding dependency management tool. For what concerns versioning, CD-Linter supports Python and Java projects (the latter using Maven). Table 6.1 shows a comparison of the version syntax.

⁶<https://docs.gitlab.com/ee/ci/yaml>

⁷https://docs.gitlab.com/ee/ci/review_apps

⁸<https://gitlab.com/gitlab-org/gitlab-triage/blob/master/.gitlab-ci.yml>

Table 6.1: Fuzzy Version syntax in Python and Maven

Fuzzy Version type	Python	Maven
Correct	<code>==1.1.8</code>	<code>1.1.8</code>
Missing Version	<code>empty</code>	<code>empty</code>
Only Major Version	<code>1</code>	<code>1</code>
Any Minor Version	<code>1.*</code>	<code>n/a</code>
Any Upper Version	<code>>=1.1.8</code>	<code>[1.1.8,)</code>

Python projects typically use pip to manage their dependencies and our meta-model contains information about all dependencies that are either defined in the `requirements.txt` file or through direct invocations of `pip install`. CD-Linter distinguishes between several Fuzzy Version subcategories. (i) If no version is defined, we report a *Missing Version*, (ii) if a version specifier only consists of a single number, we report an *Only Major Version* violation, (iii) a *Any Minor Version* when the minor release number is an asterisk (*), and (iv) *Any Upper Version* if the version number only defines a lower bound, but omits the upper bound (e.g., `numpy>=10.4`).

In Java projects, dependency resolution is typically handled by the build tool. In the case of Maven, dependencies are defined in `pom.xml`. To detect *Missing Version*, we identify dependencies that do not specify a `<version>` tag. In dependencies that define the tag, we detect *Only Major Version* as we do in Python projects and *Any Upper Version* checking whether the upper version in a range is missing (e.g., `[1.2.3,)`). *Any Minor Version* is impossible by design, because at least a range will be always declared for minor releases. When analyzing dependencies, CD-Linter handles transitive dependencies by traversing the POM hierarchy recursively.

When reviewing the detection strategies (step 3 of Figure 6.1), we realized how some libraries self-manage dependency versioning, e.g., as Spring Boot, a popular framework for web apps, does. We have compiled a list of affected dependencies for which we do not report a Fuzzy Version CD smell, because omitting the version is acceptable in these cases.

As a reaction to developer feedback (RQ₁), we differentiate between libraries used in production code and tools used in the pipeline. Not specifying a version for a tool is less critical, because no source code relies on an API that might break in

newer versions. On the contrary, having a new version with fixed bugs and updated features might even be desired. To this end, we compiled a list of tools used for Python and Java projects. These include, for example, pipenv,⁹ pytest,¹⁰ pylint,¹¹ and pip¹² for Python, and JUnit,¹³ FindBugs,¹⁴ Checkstyle,¹⁵ and PMD¹⁶ for Java (the complete list is in our replication package [130]).

6.4 Empirical Study Design

The *goal* of this study is to evaluate CD-Linter, to determine whether it could be useful for developers to avoid CD smells in their CD pipeline. The *quality focus* is two-fold: the perceived usefulness from original developers of projects where CD smells are detected and the accuracy of CD-Linter. The *perspective* is of researchers that have developed CD-Linter and want to transfer it to practice. The *context* consists of 5,312 open-source projects hosted on GitLab and using CD. More specifically, the study answers the three research questions formulated in Section 6.2.

6.4.1 Context Selection

To answer our research questions, we selected open-source projects hosted on GitLab. Using the GitLab API, we filtered projects that do not have at least one star or that are forked from other projects to avoid duplicates. From the resulting 26,984 projects, we removed all the projects that do not contain a `.gitlab-ci.yml` file in their repositories (i.e., do not use GitLab as CD server). The last filter left us with 5,312 projects that we could analyze for the presence of CD smells. These projects have a diverse team size (from 1 to 633 members with a median of 2) and age (from 1 to 133 thousand commits with a median of 75). Regarding the languages, our dataset mainly includes JavaScript (16%), Python (14%), C (10%), Java (7%), Go (4%), and Ruby (4.4%) repositories. Also, there are projects with varying CD adoption history.

⁹<https://docs.pipenv.org> ¹⁰<https://pytest.org> ¹¹<https://www.pylint.org>

¹²<https://pypi.org/project/pip> ¹³<https://junit.org/junit5>

¹⁴<http://findbugs.sourceforge.net> ¹⁵<https://checkstyle.sourceforge.io>

¹⁶<https://pmd.github.io/>

6.4.2 Monitoring of the Opened Issues

We first run CD-Linter on the dataset of 5,312 projects that has been described in Section 6.4.1. Then, we identified a random set of CD smells in a way to achieve (i) a balanced set of CD smells of each type, and (ii) at most one CD smell per project owner, to avoid flooding the same owner with many issues. For Manual Execution we could detect a maximum of 42 smells across owners, and we ended up detecting a total of 168 CD smells.

Once the detected CD smells were uploaded to the CD-Linter web-based platform, which can automatically report issues, each issue was shown to two independent evaluators (two of the authors, one of which was not involved in the CD-Linter implementation) to remove false positives (object of a different study in Section 6.5.2). Each evaluator could read the report generated by CD-Linter, browse the file in which the CD smell was found, and, if needed, browse the entire repository and its history through a GitLab link. Once two evaluators reported a positive assessment, the issue was automatically posted and opened on GitLab. The disagreement cases were discussed and, in case of positive agreement, an issue was also opened. In total, we opened 145 issues.

We have monitored the issues over a period of 6 months (from August 2019 to February 2020). During this period, we collected 64 reactions, counting issues that have been upvoted/downvoted, commented, assigned, or closed. 59 projects did not show any activity during the observation period, so we decided to ignore them in our analysis. The response rate of the remaining, active projects was 74%. We performed a card sorting [111] of the received 120 comments to identify agreements and disagreements with our issues and their motivations. The card sorting was performed by two authors that, after a first round of independent tagging, met and merged their annotations.

In addition to the reactions, we checked the source code to see whether a reported smell has been removed or reintroduced in the observation period. In some cases, the smell has been fixed despite a negative reaction or without any reaction to the issue whatsoever.

Based on the developers' reaction, issues have been classified into the following 5 categories.

Ignored The issue has been closed without any further reaction.

Rejected The issue has been closed with a majority of downvotes or with negative feedback from the comments.

Pending The issue is still open and under discussion among the maintainers without a clear agreement/disagreement.

Accepted The issue has been assigned for fix, or has a majority of upvotes or positive feedback.

Fixed The smell reported in the issue has been removed.

To address **RQ₁**, we report and discuss the responses to the opened issues for each smell type. We report the number and percentages of positive and negative reactions, the rationale for rejecting the issues, and provide examples of positive feedback and false positives. The results of **RQ₁** directly improved CD-Linter (see Section 6.3.4).

6.4.3 Manual Validation of CD Smells

We executed the enhanced version of CD-Linter on the 5,312 projects, which resulted in the detection of 5,011 CD smells. Then, we formed a sample to be manually validated. We selected, for each owner, one CD smell of each type, if detected. Since for Fuzzy Version we have four sub-categories, we considered one of each sub-category (Missing Version, Only Major Version, Any Minor Version, and Any Upper Version), if present. We obtained as result a sample that consists of 868 issues and achieves an error margin of $\pm 3\%$ (setting a confidence level of 95% and a percentage of 50%). Then, similarly to what was done in **RQ₁**, each issue was independently validated by two authors. After each annotator concluded the tagging, we measured the Cohen's kappa inter-rater agreement (k) [18]. We obtained $k = 0.76$, i.e., a high agreement, therefore no re-coding was necessary. Finally, the two annotators discussed and solved the disagreement cases. To address **RQ₂**, we report the overall precision of CD-Linter on the validated sample, defined as $TP/(TP + FP)$, where TP : true positives and FP : false positives. We also computed the recall, defined as TP/TTP (TTP : total true positives), using a randomly selected sample of 100 projects (methodology similar to Gallaba et

al. [34]), making sure that those projects were not the same used to calculate the precision.

6.4.4 Measurement of CD Smell Occurrences

To address **RQ₃**, we run CD-Linter on the latest snapshot of the 5,312 projects described in Section 6.4.1. The analysis has been performed on an Intel Xeon(R) CPU E5-2640 with 2.50GHz (4 cores) with 4GB of available main memory and took a total of 74 seconds. We report the number of CD smells of different types we detected, as well as the percentage of projects and owners affected by at least one CD smell of each type. The latter provides us with an idea of the diffuseness of the considered CD smells.

6.5 Empirical Study Results

In this section, we will answer the three research questions and report on the results of the study defined in Section 6.4.

6.5.1 Are the Four CD Smells Relevant to Developers?

During the observation period of 6 months, 64 projects reacted to our issues (response rate of 74%), Figure 6.4 illustrates the reactions. Overall, 53% of the project maintainers reacted positively to our issues: 9% acknowledged the presence of a problem and are about to solve it, and 44% fixed the reported CD smells. We have also verified that the fixes were not reverted later and could not find cases in which the reported CD smells were re-introduced.

Developers took on average 50 days to fix a CD smell, with a maximum of 5.5 months and a minimum period of 1 hour. This high variation is unsurprising for open-source projects because the activity level or the commitment of contributors strongly depends on each project. The mean resolution time for different kinds of smells was 31 days for Fake Success, 55 days for Retry Failure, 43 days for Manual Execution, and 65 days for Fuzzy Version.

Looking at the negative cases, 9% of the issues were closed without reactions (i.e., ignored issues) and 32% were rejected. Several project maintainers that

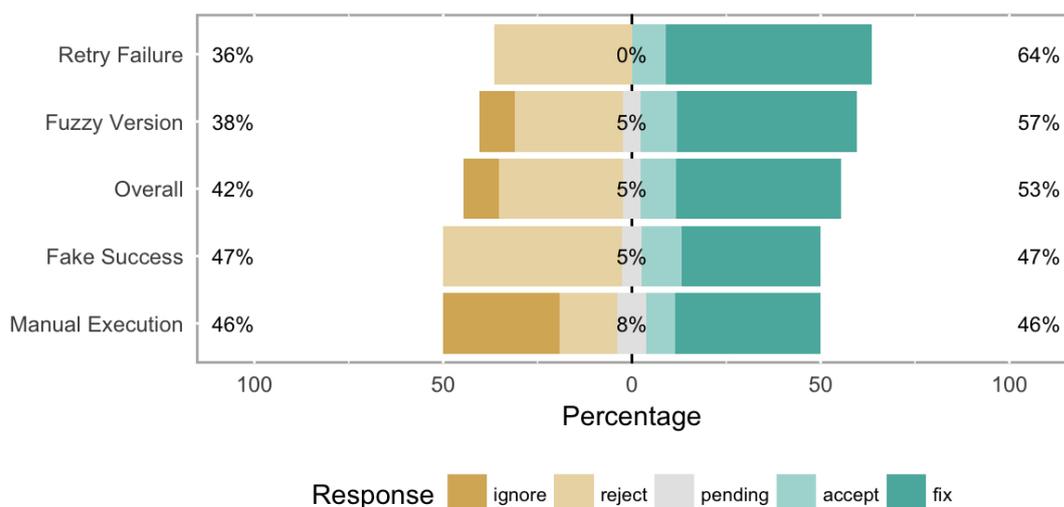


Figure 6.4: Reactions to the issues opened to evaluate CD-Linter

rejected issues provided us with reasons why they want to keep the CD smell. We found other cases in which developers rejected our issues simply because of a lack of trust in automated issue-reporting tools. In the following, we describe the reactions to each CD smell type (see Figure 6.4), the feedback we received when the issues were rejected, and how we refined our detection strategies based on the analyzed comments. We also report the percentage of false positives that we found during the assessment stage.

Fake Success We found only one false-positive case and opened 27 issues that report Fake Success cases, achieving a response rate of 70%. No issues were ignored, 10% of them were accepted, and the CD smell was removed in 37% of the projects. 9 opened issues (47% of the total) were instead rejected by the project maintainers. In several cases, developers generally agreed on our reported violation but decided to accept the CD smell nevertheless. Some developers prefer that non-essential jobs may fail, for example, checks for outdated dependencies, execution of static analysis tools, or external tools which might fail for unknown reasons. Other developers allow jobs, that are not fully implemented yet and, thus, should not impact the final build status, to fail. These projects typically state that they plan to remove the CD smell when the pipeline design is completed. Another developer, while

agreeing on the violation, did not fix the CD smell because `allow_failure:true` is recommended for certain jobs that run static and dynamic application security tests provided by GitLab. It is no longer necessary to use this configuration flag explicitly (it has been moved to a template and will be used implicitly through inheritance), but old tutorials still describe it as a best practice. While failing the overall build because of warnings raised by static analysis tools or errors in external tools is a key CD principle [25], we completely agree that projects should follow configuration recommendations of their CD provider. CD-Linter recognizes these cases and does not report them.

Retry Failure We found 7 false positives for Retry Failure issues, which corresponds to 16.7% of the validated sample. We reported the remaining 19 smells with a return rate of 58% (the lowest rate among all CD smell types). 9% of the maintainers confirmed the existence of the problem and 55% removed the CD smell from the configuration of their projects. Only 4 issues (36%) were rejected. This CD smell seems to be introduced to “hide” flakiness instead of solving it, thus, we decided to not modify our detection strategy. One developer mentioned that she deploys her application to a remote service that is randomly failing. Because the tool is out of her control, she decided to automatically retry the job multiple times hoping that it will succeed without breaking the overall build.

Manual Execution Manual Execution is the category where we found the largest percentage of false positives (26.2%), due to some periodic deployment jobs that CD-Linter did not recognize. We opened 16 issues and achieved a response rate of 81%. While 8% of the reported CD smells were accepted and 38% were fixed, 31% were ignored. Only 2 issues (15%) were rejected. In both cases, developers agreed on the importance of detecting this CD smell, but they also provided reasons for rejecting it. One of them set `when:manual` in a job executed in a stage that is not fully integrated yet with the rest of the pipeline. This can be addressed by allowing developers to directly configure CD-Linter and ignore jobs that are not part of the CD pipeline. The other developer rejected the issue because of a lack of trust on an automated reporting tool (CD-Linter). While this can be a threat to the study,

it does not constitute a problem in a usage scenario where a developer uses the tool herself.

Fuzzy Version We only found 3 Fuzzy Version false-positive instances, and we could open 24 issues achieving the highest response rate (87%) among all CD smell types. 9% of the reported CD smells were accepted and 48% fixed. While we cannot learn from the 9% of the issues that were ignored, we used the comments from the remaining 28% rejected issues to refine our detection strategy. Most complaints concern the reports about tools for which the version is left unspecified. In contrast to libraries, tools that are invoked in the pipeline (e.g., tools that compute code coverage) should always be updated to the latest version, especially because they might contain security improvements. Furthermore, tools are dependencies of the project rather than of the source code. Thus, in the case of uncontrolled updates, such tools would not affect the outcome of the build nor introduce errors, so we decided to incorporate this feedback and exclude tools from the detection of Fuzzy Version.

Answer to RQ₁

We received reactions from 74% of the projects. 53% of the project maintainers reacted positively to our issues, either accepting 9% or fixing 44% the reported CD smells. In the rejected issues, we received precious suggestions on how to improve CD-Linter, which we incorporated whenever possible.

6.5.2 How Accurate Is CD-Linter?

Table 6.2 reports the detection precision of CD-Linter. As the table shows, the detection precision varies between 73% of Manual Execution and 100% of Retry Failure and Fake Success, with an 81% for Fuzzy Version. Looking at the results of Fuzzy Version for the different sub-types of CD smells, Table 6.3 indicates that the detection precision is the lowest for the Missing Version category (77%), which, however, is the most common one. Instead, when a version is incompletely specified

Table 6.2: Detection precision for the four CD smells

CD Smell Type	TP	FP	Precision
Fuzzy Version	454	107	0.81
Fake Success	213	0	1.00
Manual Execution	27	10	0.73
Retry Failure	57	0	1.00
Overall	751	117	0.87

Table 6.3: Detection precision for the Fuzzy Version subtypes

Type	TP	FP	Precision
Missing Version	335	102	0.77
Any Upper Version	97	4	0.96
Only Major Version	20	1	0.95
Any Minor Version	2	0	1.00

(i.e., Any Upper Version, Only Major Version, or Any Minor Version CD smell), the CD-Linter accuracy raises to 95% or above.

In the following, we discuss false positives. As shown in Table 6.2, we found no false positives for Retry Failure and Fake Success. Note that this does not mean that CD-Linter would always be correct in such cases because developers might use these options for a specific, valid purpose.

For Manual Execution, false positives were mostly related to cases where the job name, content, or even comments added to the `.gitlab-ci.yml` file suggested that the job is related to a deployment activity that developers intentionally perform periodically, and therefore manually trigger (e.g., issuing a release). Despite filtering out jobs related to deployment, as explained in Section 6.3.4, we still encountered unforeseen cases. Examples include a job named `test-prerelase` (the typo in the job name made our filtering fail), but also a job named `push`, which was pushing Docker images to a repository (this case may or may not be fully automated). Also, the names of several jobs with the `when` parameter set to `manual` suggest that they should not be manually triggered. However, both the implementation and a comment left there indicate that developers intentionally configured a manual job triggering. Future work could improve CD-Linter by using Natural Language

Processing (NLP) techniques to analyze comments in CD configuration files to infer the rationale of choices made by developers.

False positives of Fuzzy Version mostly relate to cases in which dependencies for pipeline-related tools were lacking a version number. As a consequence of the preliminary analysis conducted with developers in RQ₁, we argue that libraries used in production code should specify an exact version of a dependency, to avoid build failures or introducing bugs. However, this may not be strictly necessary for tools, because developers may want to always use the latest version to have fixed bugs and enhanced features. We have derived an initial list of such tools after the feedback from developers and exclude tools like `coala`¹⁷ (rule-based linter), `sphinx`¹⁸ (documentation generation), or `wheel`¹⁹ (packaging utility).

We estimated the recall of CD-Linter on a sample of 100 randomly-selected projects. Two authors individually inspected the configuration files and agreed on the presence of 90 CD smells. We applied CD-Linter to the same sample and could detect 85 of the manually identified incidents, achieving a recall of 94%. 3 false negatives were Fuzzy Version smells. Two of them were not occurring in `script` lines while the other affected a `requirements.pip` file, a configuration file that is not considered by our tool. The remaining 2 false negatives were Manual Execution smells. Those smells were not detected by our tool because their names contain deploy-related keywords. However, they were executed in the stages `metrics` and `build_unit_test`. Section 6.3.4 established name-based inclusion/exclusion criteria through inspecting a sample of projects, but it is not feasible to derive a simple heuristic that can cover all cases.

We believe that future iterations of CD-Linter can remove these false negatives by considering other features of the `.gitlab-ci.yml` (e.g., non-script lines in jobs) or other files that are currently not supported, and by enabling developers to configure their inclusion/exclusion criteria for job and stage names.

¹⁷<https://coala.io> ¹⁸<https://pypi.org/project/Sphinx>

¹⁹<https://pypi.org/project/wheel>

Answer to RQ₂

CD-Linter has a precision of 87% and a recall of 94%, with a perfect (100%) precision for Retry Failure and Fake Success. The false positives for Manual Execution and Fuzzy Version were caused by current limitations of the tooling and can be addressed in the future.

Table 6.4: CD smells in the analyzed projects and owners
(# is “number of” and % is “percentage of” analyzable instances)

Smell	Occurrences	Projects		Owners	
		#	%	#	%
Fuzzy Version	1,569 (54.6%)	320	37.1	242	37.0
Fake Success	633 (22.0%)	282	5.4	217	6.1
Retry Failure	532 (18.5%)	82	1.6	52	1.5
Manual Execution	140 (4.9%)	69	1.3	56	1.6
Overall	2,874	680	13.0	501	14.0

6.5.3 How Frequent Are the Investigated CD Smells in Practice?

To understand the frequency of CD smells in practice, we analyzed the latest snapshot of 5,312 projects (as described in Section 6.4.1). Among them, 863 projects are either written in Java (and built with Maven) or in Python and, therefore, qualify for an analysis of the existence of CD smells in the wild, including Fuzzy Version, which is the only language-specific smell. Note that 136 of the initially considered projects were then deleted and were not available for our analysis.

Table 6.4 illustrates the occurrence of CD smells in the analyzed projects. We detected 2,874 instances of CD smells that affect 13% of the projects (14% of the investigated owners). Fuzzy Version is the most common CD smell (54.6%) and is present in 37.1% of the analyzed projects. Fake Success and Retry Failure account for 22% and 18.5% of the identified CD smells respectively. While Fake Success occurs in 5.4% of the projects, Retry Failure is present in 1.6% of them. 4.9% of the CD smells were Manual Execution and affected 69 projects (1.3% of the total).

Table 6.5: CD smells across different `.gitlab-ci.yml` sizes
 (# stays for “number of” and % is the percentage with respect to the total)

Smell	.gitlab-ci.yml Size					
	Small		Medium		Long	
	#	%	#	%	#	%
Fuzzy Version	206	13.1	564	35.9	799	50.9
Fake Success	5	0.8	70	11.1	558	88.2
Retry Failure	2	0.4	5	0.9	525	98.7
Manual Execution	5	3.6	14	10.0	121	86.4
Overall	218	7.6	653	22.7	2,003	69.7
# Projects	67	9.9	208	30.6	405	59.6

Humble and Farley advocate that a CD pipeline should be composed of at least three separate stages, i.e., compile, test, and deploy [47]. However, organizations can call those stages differently, introduce additional stages, and they can define multiple jobs within one stage. This begs the question of whether more complex pipelines are also more prone to contain CD smells, which would make a tool like CD-Linter even more relevant. A qualitative insight from our manual analysis of Section 6.5.2 indicated that longer `.gitlab-ci.yml` files seem to contain more complex CD pipeline definitions. We decided to split the analysis and discuss the different subgroups separately.

We distinguish three groups, *small*, *medium*, and *long*, and define these categories through the first and third quartile over the length distribution of all `.gitlab-ci.yml` files. Small `.gitlab-ci.yml` files have up to 15 lines (9.9% of the smelly projects have them), while a long `.gitlab-ci.yml` file is of at least 55 lines (59.6% of the files with CD smells are long). The other projects (30.6%) are medium. In Table 6.5, we illustrate how CD smell instances are spread across the different clusters and it is immediately clear that the cluster of long `.gitlab-ci.yml` files contains most of the CD smells. The cluster with small `.gitlab-ci.yml` files includes 7.6% of the detected CD smells, with 13.1% of the total Fuzzy Version smells, 3.6% of the Manual Execution incidents, and a few of the other CD smell types. Projects with medium `.gitlab-ci.yml` sizes contain 35.9% of the Fuzzy Version smells, around 10% of the Fake Success and Manual Execution problems, and 1% of Retry Failure,

Table 6.6: Break-down of Fuzzy Version smell
(YAML is *.gitlab-ci.yml*, POM is *pom.xml*, REQ is *requirements.txt*)

Category	File Type			Total
	YAML	POM	REQ	
Missing Version	684 (48.4%)	120 (8.5%)	609 (43.1%)	1,413
Only Major Version	0 (0.0%)	6 (46.1%)	7 (53.9%)	13
Any Minor Version	0 (0.0%)	0 (0.0%)	3 (100%)	3
Any Upper Version	2 (1.4%)	0 (0.0%)	138 (98.6%)	140
Overall	686 (43.7%)	126 (8.1%)	757 (48.2%)	1,569
# Files	169 (44.6%)	43 (11.3%)	167 (44.1%)	379

achieving 653 smells (22.7% of the total). The last cluster with long `.gitlab-ci.yml` files contains the majority of the identify CD smells (69.7%). 88.2% of the Fake Success smells, 86.4% of Manual Execution incidents, and even 98.7% of the Retry Failure affect this cluster. More than half of the Fuzzy Version instances affect long `.gitlab-ci.yml` files. Within this cluster, we find that 40% of the projects have Fuzzy Version smells, 17% have Fake Success incidents, 6% are affected by Retry Failure and 4% contain Manual Execution. Overall, 31% of the projects are affected by at least one CD smell. While all other CD smells have more occurrences than Fuzzy Version, the density of Fuzzy Version in long files is similar to the density in the whole dataset (see Table 6.4).

Being the most common smell, we further analyzed Fuzzy Version and investigated its sub-categories concerning the different files that it can affect (Table 6.6). Overall, the Fuzzy Version incidents are mainly detected in `requirements.txt` files. Those files were affected by all Any Minor Version and (almost all) Any Upper Version that we found, while Only Major Version is also present in several `pom.xml` files. Missing Version is the most frequent Fuzzy Version smell (1,413) and it is spread across the different files. While `.gitlab-ci.yml` has the highest number of Missing Version occurrences (48.4% of the total), this Fuzzy Version type has 609 and 120 instances in `requirements.txt` and `pom.xml` respectively. Based on these results, Fuzzy Version very frequently affect files different from `.gitlab-ci.yml`. Thus, also CD pipelines that are not so complex (i.e., small `.gitlab-ci.yml`) can

contain several Fuzzy Version incidents, which explains why this CD smell is not only concentrated in long configuration files (Table 6.5).

Answer to RQ₃

The most frequent CD smell among the ones CD-Linter detects is the Fuzzy Version CD smell (54.6% of the instances). Overall, CD smells affect 13% of the analyzed projects and 14% of their owners, mainly occurring in long configuration files.

6.6 Threats to Validity

Threats to *construct validity* are related to possible imprecisions in our measurements. They can be mainly related to possible mistakes in the CD-Linter's implementation, beyond what we could discover by testing it. The extensive manual evaluation performed in RQ₂ mitigates this threat. In addition, results of RQ₂, as well as the feedback provided by developers (RQ₁) gave us indications on how to make CD-Linter more accurate.

Threats to *internal validity* concern factors, internal to our evaluation, that could influence the results. One threat is the subjectiveness of the manual validation of detected smells in RQ₂ (precision and recall). To limit this threat, we employed two evaluators, which discussed and resolved the cases of disagreement. Also for the coding of comments that developers posted on opened issues (RQ₁), having two coders limited the subjectiveness of the results. The reactions we got in RQ₁ and the results of RQ₃ may depend on the characteristics of the analyzed projects. In particular, projects with different degrees of maturity may adopt CD pipelines of different complexity, and may or may not adhere to CD principles and good practices. We have mitigated this threat through the project selection criteria illustrated in Section 6.4.1.

Threats to *external validity* concern the generalization of our findings. While we are aware that GitLab is not as popular as GitHub, its adoption and the number of repositories there is increasing. As explained in Section 6.4.1, it gives the advantage

of analyzing projects using the same CD infrastructure. Besides being limited to a sample (though relatively large) of projects, our evaluation, because of the current limitations of the CD-Linter's implementation, is limited to GitLab configuration files, Maven builds, and Python dependencies. While the detection principles explained in Section 6.3 can be applied to other pieces of technology, the underlying concepts would not change. In this paper, our purpose was to study the reaction of developers to the detection of CD smells, rather than coping with any possible technology.

6.7 Discussion

The empirical evaluation of CD-Linter, especially the developers' feedback collected in **RQ₁**, allowed us to distill useful lessons learned and formulate implications for future research in this area.

Linters Are Fast and Can Support the Pipeline Definition Undoubtedly, a paramount advantage of linters is that they are fast and that they can already be applied in early development phases. Our experiments have shown that CD-Linter can analyze configuration files from thousands of projects in the order of seconds. Many of the contacted developers have acknowledged (and often fixed) CD smells that we have pointed out in the project. We can conclude that using linters to support the pipeline definition and to catch smells early on is, indeed, a promising research direction.

Issue Reporting Is Useful, but Must Be Carefully Dosed One problem we experienced in our empirical evaluation is that some developers are irritated by (and tend to discard) automatically-posted issues. While we tried to elaborate in the opened issues that they were a result of a manual review process, some developers still considered them a sort of spam, even when the suggestion was meaningful. Recent works show promising results when bots are used to aid software engineers [60, 61], but we found that developers seem to be sensitive in the context of issue trackers. Despite some negative reactions, our efforts were generally well-received by developers though. To mitigate the negative effect described above, one

author followed-up on all comments on the opened issues, to explain the purpose of CD-Linter, justify the opened issue, and -most importantly- show that there was a human in the loop. Overall, involving open-source developers in our research was valuable for both sides, but it was crucial to take the time and talk to developers to show respect and emphasize the importance of the research.

Linters Are Intrinsically Imprecise A common issue of linters is their intrinsic imprecision. Not every deviation from an advocated principle is a smell and, often, a violation can only be assessed when the specific context is being considered. Such decisions have to be taken on a case by case basis for a project and go beyond the scope of static analysis tools. This phenomenon is not specific to CD-Linter though, the low precision of static analysis tools has already been reported as an adoption barrier multiple times [19, 52, 138]. In our case, CD-Linter seems to balance precision and recall well. Despite many rejected smell reports, the number of fixed reports and the generally positive feedback that we have received from developers indicate that developers appreciate the effort and that tools like CD-Linter can have a positive effect on CD practices.

Long and Complex CD Configurations Are Often Smelly While we find relatively few instances of the CD smells in simple configuration files, the density increases with the length (and complexity). One explanation could be that developers have to cope with phenomena such as flakiness, the need for manual job triggers, accepting failures from some jobs, or with special requirements for dependency management. For such reasons, we expect CD-Linter to be particularly beneficial for projects with a complex pipeline.

Findings Should Be Reported Quickly One of our lessons learned from **RQ₁** was that identified issues need to be reported timely, otherwise the issue may disappear or not be valid. In some cases, the CD smell was resolved already by the time we were done with validating it, so the reported issues were unnecessary. Generally, timely reporting is essential in case of issues that involve line numbers because these are fragile due to frequent source code changes and can be soon

outdated. In these cases, it might be helpful not to link to the latest version in the repository, but to the exact commit that has been analyzed for the issue.

Overall, this paper shows a promising future for linters of CI/CD pipelines. Future linters can extend the ideas in several ways, for example, not only considering dependency versions, but also other versioned entities in the build configuration, like build plugins or container images, in which the build is run. The results in this paper emphasize the need for more research on linters in this domain.

6.8 Related Work

This section describes related work about bad practices and their identification in CI/CD and infrastructure-as-code scripts.

6.8.1 Bad Practices in CI/CD

In their landmark books about CI [25] and CD [47], previous researchers outlined wrong decisions while applying CI/CD. The lack of build automation and project visibility together with the inability to create deployable software are a few examples of those practices that prevent organizations from achieving the expected benefits. Duvall collected these and other bad practices in a catalog of 50 anti-patterns (and their corresponding patterns) that occur during several steps of a CI/CD pipeline [27]. Zampetti et al. [143] empirically characterized CI bad practices, finding commonalities but also differences with the ones advocated by Duvall [27]. Anti-patterns also occur because developers face several barriers when adopting CI/CD [45]. For instance, developers need to debug failures occurring on a remote server and maintain complex build infrastructures.

The catalogs of anti-patterns and the studies discussed above constitute the foundations of our work, as we use them to derive principles for which CD-Linter detects smells.

6.8.2 Detection of Smells in Development Workflows

Several researchers have proposed approaches to automate the identification, and, in some cases, the removal of problems arising in build and, more in general, Infrastructure as Code (IaC) scripts.

Gallaba et al. [34] developed an approach for detecting and eliminating misuses such as the presence of unused properties and bypassed security checks in Travis-CI build scripts. While we also statically analyze configuration files, our approach detects those anti-patterns that are violations of CI/CD principles.

Deviations from such principles have been also investigated by Vassallo et al. [129]. They proposed CI-Odor, a tool that analyzes artifacts produced during CI such as logs and revisions to detect anti-patterns (e.g., builds become slow, developers work on feature branches for a longer period) that occur over time and cause a CI decay. Differently from this work, we focus on the anti-patterns that can be statically detected in configuration files.

Troubleshooting build failures is challenging and often causes delays in the delivery process. A previous work [134] has proposed a taxonomy of build failures based on their root causes. Researchers have implemented solutions that automatically repair some of these build failure types [66, 120]. Another tool [133] improves the understandability of build failures through log summarization. Despite those approaches, developers still allow failures [33, 35]. This strengthens our motivation for including Fake Success in our linter.

Finally, other related works are devoted to the detection of smells in IaC scripts. Sharma et al. [108] leveraged best practices associated with code quality management to assess configuration code quality and derived a catalog of configuration smells for IaC scripts developed in Puppet. While those smells are more similar to traditional code smells (i.e., they concern with maintainability and understandability of Puppet code), CD-Linter detects smells specific to the CI/CD configuration where developers violate principles. Rahman et al. [94] implemented a linter that detects seven types of security problems in IaC scripts. Their work is complementary to ours as it deals with a very specific category of problems related to IaC scripts. Many of their security smells can also occur in CI/CD pipelines.

6.9 Summary

Previous work has introduced generic [34] or specialized [94] linters that can help developers to improve their CD configuration. In contrast to previous work on CI smells that relies on historical information [129], in this paper we proposed CD-Linter, a static analysis tool able to identify four types of CD smells in CD pipelines, right when they are introduced in the pipeline configuration. Our empirical evaluation has shown that the supported CD smells are relevant in practice, that CD-Linter is accurate, and that the supported smells frequently occur in the wild. Linters generally suffer from many false positives, sometimes up to 90% and more [138], but CD-Linter reaches a precision of 87% and recall of 94%, which represent acceptable results and a good compromise. In a large set of 5,312 projects, we found that 31% of pipelines with long configuration files are affected by at least one instance of the detected smells. The empirical evaluation of CD-Linter, and especially the developer feedback that we have received for **RQ₁** illustrated the usefulness of CD-Linter and it allowed us to distill useful insights that can foster the adoption of CD-Linter in practice and stimulate research on similar tools to further advance this area.

Acknowledgments

We would like to thank all the study participants. C. Vassallo and H. C. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275).

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, Sebastian Proksch, Harald C. Gall, Massimiliano Di Penta
Published in Proceedings of the 41st IEEE/ACM International Conference on Software Engineering, Technical Research Paper (ICSE), 2019
Contribution: survey design, CI-Odor design and implementation, experiment design, data collection, data analysis, and paper writing

Abstract

Continuous Integration (CI) is a widely-used software engineering practice. The software is continuously built so that changes can be easily integrated and issues such as unmet quality goals or style inconsistencies get detected early. Unfortunately, it is not only hard to introduce CI into an existing project, but it is also challenging to live up to the CI principles when facing tough deadlines or business decisions. Previous work has identified common anti-patterns that reduce the promised benefits of CI. Typically, these anti-patterns slowly creep into a project over time before they are identified. We argue that automated detection can help with early identification and prevent such a process decay. In this work, we further analyze this assumption and survey 124 developers about CI anti-patterns. From the results, we build CI-Odor, a reporting tool for CI processes that detects the existence of four relevant anti-patterns by analyzing regular build logs and repository information. In a study on the 18,474 build logs of 36 popular Java projects, we reveal the

presence of 3,823 high-severity warnings spread across projects. We validate our reports in a survey among 13 original developers of these projects and through general feedback from 42 developers that confirm the relevance of our reports.

7.1 Introduction

Continuous Integration (CI) is a common development practice and its great benefits on quality and productivity are widely accepted [124]. CI advocates full automation of all build steps (i.e., compilation, testing, and code quality assessment) to create a new version of the software [25]. The CI process is most effective when developers follow best practices, such as *commit often*, that reduce conflicts in the team and ensure that the build is continuously executable [25]. In practice, it is often challenging to live up to these standards and *anti-patterns* can be observed: common but ineffective solutions to a recurring problem that should be avoided. For example, failing tests are removed instead of fixing the root cause. Over the years, researchers have defined catalogs of CI anti-patterns [27, 47], which eventually become a threatening maintainability problem for a software project, if not properly addressed [25].

A creeping decay of quality has been described before in other contexts. Fowler [30] popularized the term *code smell* to describe a symptom that indicates the existence of a deeper problem in the source code. While the perception of smell is subjective, previous work could show that code smell intensity correlates with the likelihood of the existence of a deeper issue [85]. The smell metaphor was later adopted in other areas such as system design [76], configuration files [108], or spreadsheets [44]. Hence, the idea that a CI anti-pattern manifests itself as a *CI smell* as well. In contrast to anti-patterns in development artifacts, CI anti-patterns affect the software development process and provoke *CI decay*.

In this paper, we further study this phenomenon. Our results of a broad survey among 124 professional developers confirm that CI decay is indeed a relevant problem. Most participants confirm that deviations from CI best practices happen in practice, both intentionally and unintentionally, and that the benefits of CI diminish when many deviations exist in a project. The awareness about the presence of anti-patterns in the CI pipeline is key for an educated decision about whether a

deviation needs to be fixed. Inspired by Duvall’s catalog of Continuous Integration and Delivery (CD) anti-patterns [27], we built CI-Odor, an automated detection and reporting tool that provides awareness about CI decay caused by four different anti-patterns. Through the analysis of build log and repository information our tool identifies (1) slow builds, and especially increasing trends of build time, (2) broken release branch, and the corresponding time-to-fix, (3) skipped failing tests, and (4) late merging of development branches.

By analyzing a total of 18,474 recent builds logs of 36 popular Java projects, we identified 3,823 high-severity anti-pattern instances, and 4,697 with medium severity, spread across all projects. To evaluate our tool, we have surveyed 13 original developers about the relevance of reports (containing recent instances of detected smells) generated for their project and 42 developers about the general usefulness of CI-Odor. The reports are perceived as useful, relevant, and most participants would integrate CI-Odor in their CI pipeline to increase their awareness about the CI process. We also find untapped potential for future detectors and that more work on CI anti-patterns is necessary to improve the handling of project specifics.

In summary, this paper presents the following contributions:

- Verification of the relevance of *CI decay* in practice;
- *Detectors* of four relevant CI anti-patterns;
- CI-Odor, an automated *CI anti-patterns reporting tool*;
- An *empirical study* on the presence of CI decay and on the developers awareness about CI anti-patterns.

7.2 Methodology Overview

In this paper, we introduce CI-Odor, an automated reporting tool that can be integrated into CI pipelines to increase the awareness about anti-patterns in CI. Figure 7.1 illustrates our methodology to create and evaluate the tool.

This work is based on the existing anti-patterns catalog (1) of Duvall [27], which describes 50 patterns and anti-patterns that influence the effectiveness of a CI/CD pipeline. In an internal selection, we identified a subset of CI anti-patterns from this

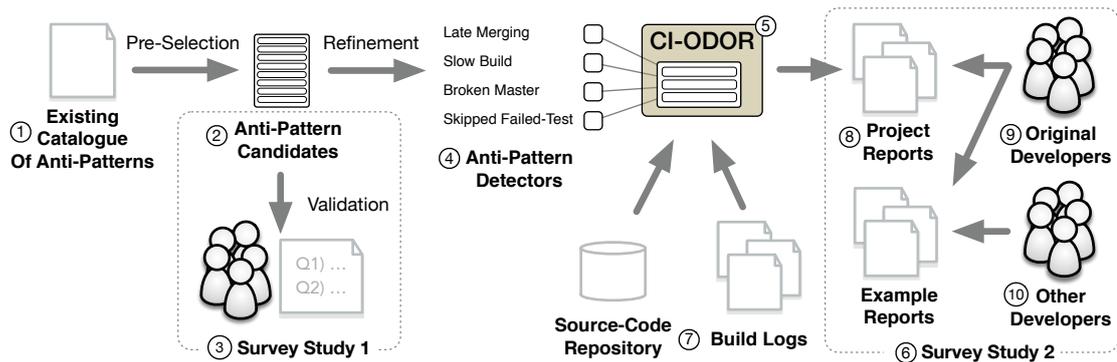


Figure 7.1: Overview of the methodology used to build and evaluate CI-Odor

catalog that can be automatically detected by analyzing build log and versioning information (2). For each of these anti-patterns, we added an explanation and an illustration of the detection strategy and validated their selection in a survey among 124 professional software developers (3). We asked the participants about the relevance of the anti-patterns in practice and the suitability of our detection strategies. Based on the results of the survey, we eliminated several candidates, refined our detection strategies, and ended up implementing a set of four detectors (4). We integrated these detectors in a reporting tool, CI-Odor (5), that aggregates the different analysis results and that presents statistics such as a trend analyses to increase the awareness about the different anti-patterns.

We evaluated the usefulness and relevance of the reports created by CI-Odor in a second survey (6). For the survey, we conducted a case study, in which we analyzed the build logs of 36 projects (7). The resulting reports (8) are publicly available and we asked the original developers of these projects to rate them (9). In addition, we selected reports that illustrate the full capabilities of our reporting (e.g., there is at least one detected instance of each anti-pattern). We ask both the original developers and other developers with experience in CI (10) to rate these example reports.

7.3 Which Anti-Patterns to Detect, and how?

The existing anti-pattern catalog of Duvall [27] is extensive and contains several examples that go beyond the scope of automated tools, e.g., decisions regarding

the deployment strategies. We started with selecting a subset of anti-patterns, for which we could develop appropriate detectors. In this section, we first introduce our pre-selected list of candidates and the survey that we used to validate and finalize our selection.

7.3.1 Pre-Selection

The rationale of our pre-selection was two-fold. We wanted to cover different aspects of the CI pipeline such as version control or build failure management and exclude others that are more related to CD. At the same time, we selected anti-patterns that can be detected using data that is typically produced by every CI pipeline independently from custom settings, i.e., build logs and repository. The following list introduces all anti-pattern candidates, proposes a detection strategy, and justifies their relevance for the CI process quality. For traceability, we include the name of Duvall's positive example [27].

Late Merging (Merge Daily) Agile teams often develop in feature branches. Integration effort and conflict potential increase if completed features are not integrated timely. We propose to warn about cases in which the last commit of a branch is older than a predefined threshold.

Aged Branches (Short-Lived Branches) Infrequently synced feature branches substantially diverge over time and end up being very hard to integrate. We propose to warn when an open branch has not been merged into master for a release.

Broken Release Branch (Stop the Line) A broken build that is not fixed timely prevents the CI pipeline from properly assessing the effect of new changes. We propose to warn when a build stays broken for longer than usual.

Bloated Repository (Repository) Artifacts that can be created in a build or fetched through provisioning mechanisms should not be committed to the version control system. We propose to warn when binaries can be found in the repository.

Scheduled Builds (Continuous Integration) A scheduled build either (unnecessarily) builds a change a second time or is a sign that a change is not automatically built, which breaks the idea of always ensuring a working system. We propose to warn about build configurations that schedule builds.

Absent Feedback (Continuous Feedback) Developers are missing out on required feedback, when they are not automatically notified about relevant build events, especially build failures. We propose to warn about configuration files that do not enable any notification channels.

Email-Only Notifications (Visible Dashboards) According to Duvall [25], email is inappropriate as a single notification channel, because developers might not have access or notifications get lost among other messages. We propose to warn about configurations files that only notify by email.

Skip Failed Tests (Automate Tests) Skipping a failed test can fix a broken build but addresses a symptom rather than fixing the cause. It threatens the safety provided by the test suite. We propose to warn about cases in which a previously failed test does no longer occur in the next (fixed) build.

Slow Build (Fast Builds) A slow build, caused by a coding issue or by a high workload of build server, produces waiting times for developers and adds overhead to the CI process. We propose to warn about significant build slow-downs.

Please note that these descriptions are shortened introductions from our survey. A complete version of the first survey is available on our artifact page [128].

Problem Statement

- CI best-practices are no strict rules, they can be adapted for a project.
- One can deviate from CI best-practices unintentionally.
- The benefit of using CI diminishes, the more best-practice deviations exist.

Relevance and Sufficiency of Smell Detection (for all anti-patterns)

- Anti-pattern X is relevant in a typical CI pipeline.
- The detection strategy is sufficient to identify occurrences of an anti-pattern.

General Validation of Idea

- I would integrate such a tool in my CI pipeline.
-

Figure 7.2: Main questions of the survey on the relevance of CI anti-patterns

7.3.2 Survey on the Practical Relevance

We conducted a survey to validate the relevance of the selected anti-patterns and the proposed detection rules.

Survey Design. The survey contains three sections. The first section is about the perceived severity of the problem of deviations from CI best practices. The second section has a focus on the anti-patterns. We introduced each one with an elaborated description that includes explanatory images and asked participants to evaluate the anti-pattern relevance and our proposed detection strategy. Finally, we asked for a general validation of the idea of anti-patterns detection.

All survey questions were optional and had Likert scales [83] with either five (*Strong Disagree* to *Strong Agree*) or four levels (*None* to *High*). The survey also contained open questions for feedback in all sections. Figure 7.2 includes an excerpt of our survey; a complete export is available on our artifact page [128].

Advertisement. We advertised the survey on social media (i.e., Twitter, Reddit sub-forums dedicated to DevOps and Continuous Integration), in CI-related newsletters, and by sending it to personal contacts. We promised to raffle off two vouchers to reward participation.

Demographics. In the end, 605 developers opened our questionnaire, out of which 144 finished all questions, leaving us with a completion rate of 23.8%. We included three control questions in the survey that asked about the proficiency in programming, CI theory, and CI practice. We excluded responses from participants that reported less than *moderate* experience in any of these questions. After the filtering, we ended up with 124 qualified participants. Most of our participants (79.8%) report that they got in contact with CI in an industry position. 68% of our participants hold an academic degree related to computer science (32.3% Bachelor, 28.2% Master, 8% Ph.D.). The large majority reports a *high* level of experience in programming (76.6%), CI theory (69.4%), and CI practice (59.7%).

Data-Analysis Methodology. To analyze the Likert-scale answers, we create asymmetric stacked bar charts with proportions for various agreement levels that are shown in Figure 7.3. We performed card sorting to analyze the open answers [111]. We started by splitting the answers into individual statements, grouped common arguments, and finally organized these arguments hierarchically.

Problem Statement. The survey results show that deviations from CI best practices happen in practice. Most participants state that a project can intentionally deviate from CI best-practices (67.7%) and even more participants agree that a deviation

might be unintentional (77.3%). The majority of participants (77.4%) agree that the CI benefits diminish when many best-practice deviations exist. These answers confirm our conjecture that CI decay is a relevant problem in practice.

Relevance & Detection. Our survey contains questions about the practical relevance of each anti-pattern and we received a very high level of agreement. Six detectors have an agreement of >75% and other two have an agreement of >60%. The only notable exception is *Email-Only Notifications* for which 30.3% participants disagree with its relevance. These results confirm that we had pre-selected relevant anti-patterns.

We asked our participants to rate the sufficiency of our detection strategy for all anti-patterns. Also here, the agreement is very high (8, >60%, 2, >75%), with the notable exception being *Late Merging* (54.1%), for which many participants point out specific ways to use version control that would have not been detected. The high agreement makes us confident that we have successfully identified the “common case” for our detection. However, several people made use of the open question for each anti-pattern to provide feedback on the detection strategies, such as pointing out alternate development processes that we did not cover so far.

Revised Detection Strategies. On average, 43 participants answered the open question about each anti-pattern and we carefully analyzed these answers to revise or exclude some detectors. Next, we discuss the results of our open card sorting and how we revised our detection strategies based on the suggestions from the survey.

Late Merging & Aged Branches Many participants point out similarities between both anti-patterns. Also, the suggestions for improving the detection strategies overlap in our answers. As a result, we decided to merge both anti-patterns. The feedback contains valuable suggestions to improve our simplistic detection strategies: 1. a Git-based detector must support both `merge` and `rebase` commands; 2. a feature branch should not be considered aged when no changes in other branches occur; 3. the age of a branch is irrelevant, as long as it is frequently synced with the master; 4. branches marked with release names, e.g., `re1-1.2.3`, should not be reported; 5. projects can release multiple times per day, so open branches

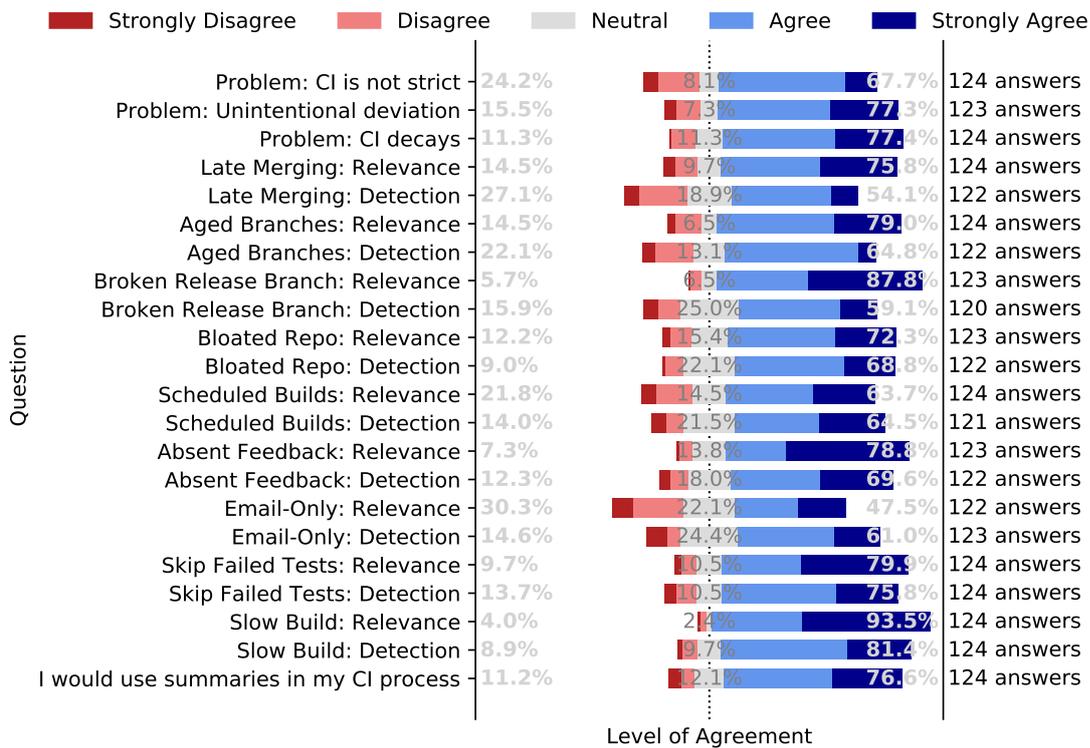


Figure 7.3: Likert-scale answers to the survey on the relevance of CI anti-patterns

frequently span several releases. Overall, we decided to revise our strategy to incorporate these suggestions and keep the *Late Merging* anti-pattern.

Broken Release Branch The survey participants broadly agree to the detection of the anti-pattern (87.8%). Several participants point out that a broken release branch should never happen, so everybody would be aware of it even without notification. In addition, we got some minor comments on our detection strategy, e.g., 1. providing an overview for incidents over time, 2. avoiding to consider the average time between commits as the commit frequency varies a lot. We decided to incorporate these suggestions into our detection strategy and to keep *Broken Release Branch*.

Bloated Repository Despite the high agreement among the participants (68.8%), we received several comments against the detection of this anti-pattern: 1. offending files are language and project specific, but can be easily fixed with a well-defined `.gitignore` file; 2. this anti-pattern is not CI specific; 3. several participants mention good reasons to include binaries, e.g., the availability, reliability, and

convenience of provisioning sources. We agreed with these concerns and eliminated the anti-pattern candidate.

Scheduled Builds Despite a high agreement with the proposed detection strategy (64.5%), many survey participants point out good reasons for scheduled builds. These include, for example, running extensive performance tests, UI tests, or frequently asserting the compatibility with a changing environment. We could not distinguish between good and bad cases of scheduled builds, so we decided to drop this candidate.

Absent Feedback & Email-Only Notifications The agreement rate to the detection strategy of both anti-patterns is high (69.6% and 61.0% respectively). However, many participants state doubts regarding the detection feasibility: 1. feedback might be delivered in ways that cannot be automatically checked, e.g., physical build lights; 2. it is impossible to validate successful notification delivery; 3. the best notification channel is a personal preference. We agreed with these concerns and decided to drop both candidates.

Skip Failed Tests The detection strategy for this anti-pattern has a very high agreement rate (75.8%), but several participants mention good reasons to remove a test, e.g., removal of functionality. We believe that in these scenarios tests would either be removed together with production code, or the build would fail due to a compilation error. Both scenarios would not trigger our detector. Other participants point out that removing, commenting, and skipping tests have the same effect, so we should cover all of these cases. Apart from this, we did not receive further suggestions for improvement. We decided to keep this anti-pattern.

Slow Build Most participants agree with the detection strategy for this anti-pattern (81.4%). The main concern mentioned by several participants is the threshold that is used to identify slow builds. At the same time, previous work mention that a slow *creep* is the *worst-case scenario* for build times [45]. We kept this anti-pattern.

Overall, we received valuable feedback on all presented anti-pattern candidates. Following the suggestions of our participants, we dropped *Schedule Builds*, *Absent Feedback*, *Email-Only Notifications*, and *Bloated Repository* for the reasons mentioned above. We revised the detection strategies of the remaining anti-patterns

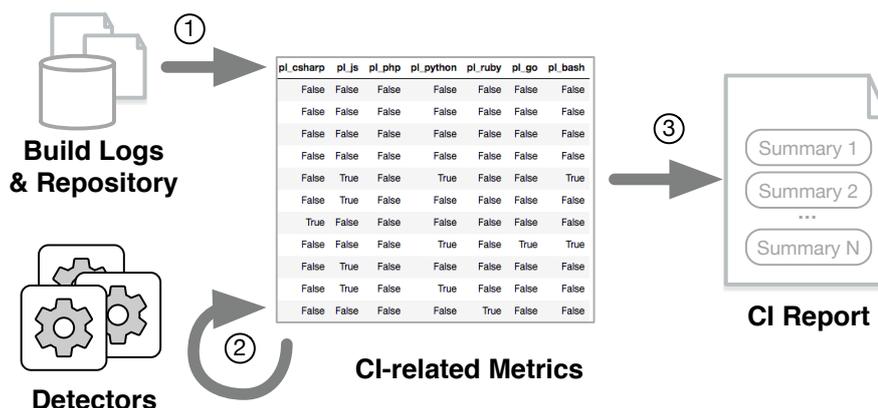


Figure 7.4: CI-reporting process in CI-Odor

Late Merging (which is now merged with *Aged Branches*), *Slow Build*, *Broken Release Branch*, and *Skip Failed Test* and kept them for the remainder of the paper. *General Feedback*. The last part of the survey contains a general open question to provide feedback on the whole CI-Odor idea, which was filled by all 124 participants. Most of them (30%) mention CI-Odor as useful for CI training and for learning to adopt CI best practices rigorously also when developers are not familiar with CI yet [46]. Furthermore, 13% of our participants believe that CI-Odor can reduce maintenance effort and improve reliability on the CI pipeline. As stated by 18%, some anti-patterns might go unnoticed without such a tool, which can be useful to monitor the CI health and take countermeasures when needed. 12% of the participants suggest to have highly-configurable detectors to support team/organization specifics in CI pipelines. Finally, 18% of the participants are quite skeptical about our detectors. As it happens with many quality check tools [138], their main concern is the likelihood of generating several false positives.

7.4 Reporting CI Practices

To implement a proof-of-concept of CI-Odor, our CI anti-pattern detector, we first chose supported technologies. We analyzed whether the perceived relevance of each smell varies across people working on different programming languages. Based on responses to our previous survey, a Kruskal-Wallis test [109] did not indicate, for any of the anti-patterns, a statistically significant difference among

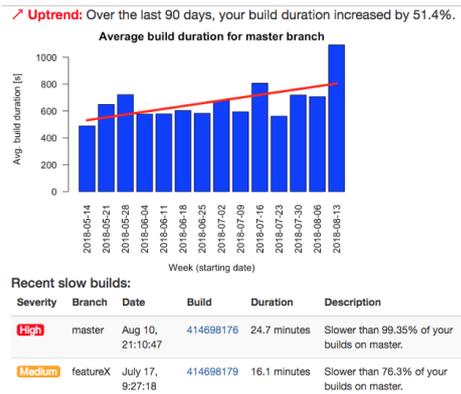
the four main programming languages the study participants reported as their main working language (p -value > 0.05), i.e., Java, JavaScript, Python, and Ruby. Thus, based on their popularity, we decided to focus on Java and Maven as our target programming language and respective build tool. To lower the likelihood of build-log deletion [124], we mined build data from TravisCI¹ and consequently repository data from GitHub.²

The overview of our reporting process is shown in Fig. 7.4. Given the TravisCI build logs of a particular project, we first extract CI-related *metrics* for every build (1). We then run detectors on top of this raw data (2) to derive additional metrics that either indicate the presence of a phenomenon (e.g., a test has been removed) or a change in a metric (e.g., a change in build time). From all metrics, we provide a reporting utility that visualizes several dimensions of the CI process, Figure 7.5 shows screenshots for the four different summaries that we include in our reports. Next, we discuss the details of the detection strategies for the four anti-patterns.

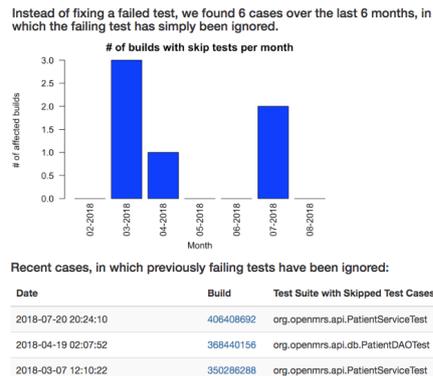
7.4.1 Slow Build

Figure 7.5a shows an example summary of *Slow Build* that contains the following items. 1. A bar chart highlighting the average build duration per week over the considered time window (3 months in our example and in the study of Section 7.5). 2. A linear regression trend line, along with a textual message highlighting whether the build time is increasing, stable, or decreasing over the observed period. We also experimented with the use of kernel smoothing, but the resulting trend did not drastically change, so we favored simplicity here. 3. A list of possible warnings for the last builds of each branch. Specifically, we report: (i) a *Medium-severity* warning when a build was slower than 75% of more builds on the master branch, i.e., it is in the fourth quartile; (ii) a *High-severity* warning when the build duration is an outlier with respect to the distribution of master builds in the observed time window. We used the box and whisker plot outlier definition [119], i.e., a build is an outlier when its duration is greater than $3Q + 1.5 \cdot IQR$, where $3Q$ is the third quartile and IQR the inter-quartile difference.

¹<https://travis-ci.org> ²<https://github.com>



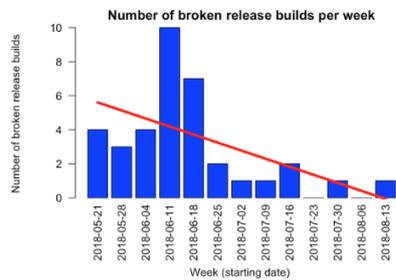
(a) Slow Build



(b) Skip Failed Tests

In the last 90 days, your master branch was broken 36 times and it took on average 14.8 hours to fix it again.

📉 **Downtrend:** Over the last 90 days, the number of broken release builds decreased by 93.1%.



(c) Broken Release Branch

High In your project, branches are typically synced with `master` every 2.8 days. However, branch `featureX` was last synced with `master` on Nov 23, 10:25 and branch `master` has commits that are 19 days newer than that.

Medium Your latest commits were performed on branch `featureX`. While you typically merge branches within 1.8 days in your project, branch `featureY` was last changed 3.7 days ago and has not been merged into `featureX` yet.

High In your project, branches typically do not run in parallel for more than 2.6 weeks. However, work on branch `featureX` started 5.1 weeks ago and the branch has not been synchronized with its parent since.

High Your feature branches are typically open for 2.3 weeks, however, you have been working on `featureX` for 7.3 weeks now.

Tip: Make sure that you do not forget to sync these branches from time to time.

Tip: Frequently synchronized branches are easier to integrate.

Tip: Break features into smaller tasks to finish them faster.

(d) Late Merging

Figure 7.5: Example summaries of the four anti-pattern detectors

In some cases, and this is especially true for the considered CI infrastructure (i.e., Travis-CI), the build time might depend on many external factors, including the priority given to the project (in Travis-CI projects with a free account get a low priority). However, we do not consider this a threat in our measurements, because even in these cases CI-Odor would highlight the need for using a better infrastructure.

7.4.2 Skip Failed Tests

We first extract the executions of all JUnit tests and their outcomes from each build log, i.e., the containing Maven module, the test suite name, the number of executed tests, the number of failed tests (incl. test errors), and the number of skipped test cases. We then derive a set of test-related CI metrics by matching tests run in jobs (with the same *id*) belonging to consecutive builds on the same branch. Specifically, we compute Δ_{Runs} , i.e., a change in the number of executed tests, Δ_{Breaks} , i.e., a change in the number of failed tests, and Δ_{Skipped} , i.e., a change in the number of skipped tests. To mark a test as skipped (in the next build), we evaluate whether the following expression is true:

$$(\Delta_{\text{Breaks}} < 0) \wedge (\Delta_{\text{Runs}} < 0 \vee \Delta_{\text{Skipped}} > 0)$$

Fig. 7.5b shows an example summary that contains: 1. a bar chart depicting the number of builds per month affected by skip failed tests (note that we adopted a granularity of one month for this smell, due to its lower frequency than *Slow Build*); 2. for each build where such an incident occurred, the list of test suites affected by the skip failed tests issue.

7.4.3 Broken Release Branch

To detect a broken release branch, we compute the final status of each `master` branch build, i.e., its *build status*, in the build history of a project. In particular, a build is *errored* when the *install* phase, which retrieves and installs the needed dependencies, returns a non-zero exit code. Instead, it is *failed* when any subsequent

phase returns a non-zero exit code. We determine the final status of each build and consider all the errored and failed builds as broken. Fig. 7.5c shows an example summary that reports: 1. the average time a release branch remains broken over the observed time period, considering consecutive broken builds; 2. a bar chart showing, for each week of the observed period, the number of broken builds; 3. a linear-regression line and a textual message highlighting the presence of an increasing or decreasing trend, if any.

7.4.4 Late Merging

We consider four different metrics about version control that help us to identify the *Late Merging* anti-pattern: *Missed Activity*, *Branch Deviation*, *Branch Activity*, and *Branch Age*. In the following, we introduce the different metrics using the example history of Fig. 7.6, which contains a `master` branch and `f1` with several merge commits.

Missed Activity (t_{MA}). Quantifies the amount of activity on other branches of the same repository since the current branch was last synced with the `master`, $t_{MA} = t_{LO} - t_{Sync}$, where t_{LO} is the date of the last commit on other branches and t_{Sync} is the date of the last merge commit. If t_{MA} grows, the potential integration effort increases. To allow for more specific warnings in the summary, we break this metric further down into its two components *Branch Deviation* and *Unsynced Activity*.

Branch Deviation (t_{BD}). Quantifies the amount of activity in other branches since the last change in the current branch, $t_{BD} = t_{LO} - t_{LC}$, where t_{LO} is the last commit date on other branches and t_{LC} is the last commit date on the current branch. If t_{BD} grows, other branches deviate from the current branch, which again increases the potential integration effort. Negative values mean that the current branch is ahead of other branches.

Unsynced Activity (t_{UA}). Quantifies the amount of activity in the current branch since the last sync with the `master`, $t_{UA} = t_{LC} - t_{Sync}$. A growing t_{UA} indicates a deviation from the `master`, and that the potential integration effort increases.

Total Activity (t_{TA}). Quantifies the total amount of activity on a branch since its creation, $t_{TA} = t_{LC} - t_{Fork}$, where t_{Fork} is the date of the branch creation. Feature

branches should be merged back into the `master` timely, a growing t_{TA} indicates a long-running deviation from the `master`.

When CI-Odor raises a warning. For each of the four metrics mentioned above, we compare their values with distributions in recent history and consider a *Medium-severity* warning if a value is above the third quartile, a *High-severity* warning if it is an outlier (using a similar approach to the one in Section 7.4.1).

History Rewrite. Git history can be rewritten, which makes it harder to analyze [13]. We include a second branch `f2` in our example to illustrate our handling. We detect rebasing in build logs by matching the meta-data of a commit that is built (*id, time, committer, message*) to meta-data of previous builds on the same branch. When all meta-data but the id can be matched to a previous commit, we mark this as a rebasing. In the example, the rebasing of (4) triggers a new build of (4') at the date t_{Sync} ; t_{Fork} is the date at which the first build of this branch was triggered. Now all derived metrics can be calculated as for the previous merge case.

Improved Detection Strategies. We consider two suggested improvements for the detection strategy. First, in addition to analyzing the build logs, we also analyze the current repository snapshot for every build to identify deleted branches that do not need to be reported anymore. Second, we filter out branches that mark releases, e.g., `rel-1.2`.

7.5 Empirical Assessment of the CI-Odor Summaries

We conducted a study on open source software projects to assess the accuracy and usefulness of our reporting. We first performed a selection of candidate projects. After detecting anti-patterns on such projects, we sent the generated summaries to the mailing lists/forums of said projects and we asked original developers to fill out a survey. In addition, we also asked participants to the study of Section 7.3 to answer the part of the survey concerning the usefulness of the summaries and of CI-Odor in general.

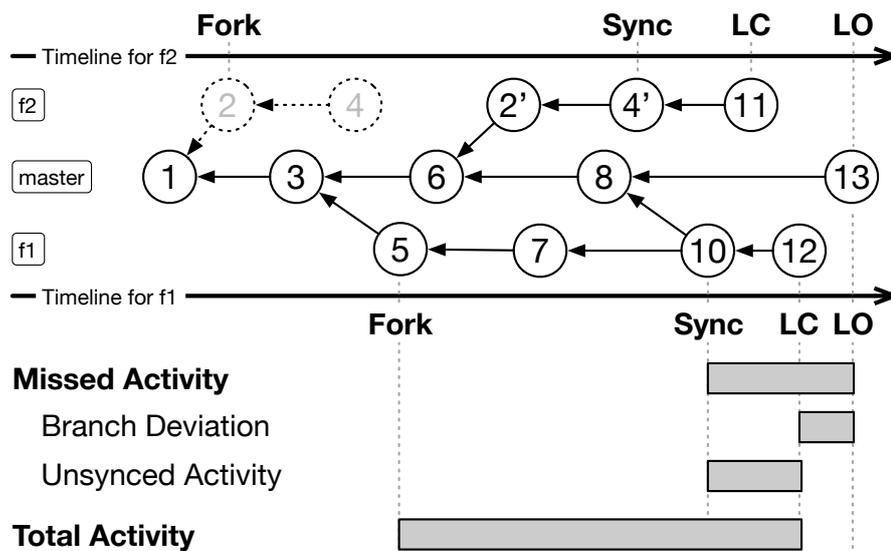


Figure 7.6: Example of different late-merging scenarios

7.5.1 Projects Selection

We used data from GHTorrent [39] (version Apr-01-2018) to identify suitable projects for our study. We filtered projects according to the following criteria: they are written in Java, have not been deleted, have at least one commit in 2018, have at least two project members, are no forks, and have been forked at least once. This initial filtering left us with a set of 2,155 project candidates.

To find projects in this set that perform CI and that are compatible with CI-Odor, we required the existence of configuration files for Maven (`pom.xml`) and TravisCI (`.travis.yml`), which further reduced our candidates to 467 projects. We then excluded projects with less than five project members to ensure a certain community size.

We then extracted build logs from TravisCI for the remaining 103 project candidates. For some projects, we could not access the logs or only found a very limited number. As CI-Odor is based on historical analyses and to ensure a minimum level of activity, we excluded the first quartile from the distribution of available build logs for these projects, which left us with 70 candidates that have had at least 54 builds in 2018.

As the final step, we manually identified the main communication channels for all remaining projects, because we need to contact the corresponding developers. Keeping only these projects, for which we found a public mailing list or could join a closed group channel (such as Google Groups or Slack), we ended up with a final selection of 36 projects for the validation. These projects cover various domains like business-oriented software, image processing, development tools and have a diverse sizes (from 2 thousand to 12 million LOC), ages (from 457 to 25 thousand commits), activity levels (from 60 to 2 thousand builds), team sizes (from 7 to 385 members with a median number of 60.5), and popularity (from 15 to 26 thousand GitHub stars). The full list of these projects on the artifact page of this paper [128].

7.5.2 Quantification of the Phenomenon

This section provides a short overview of the anti-pattern instances and CI decay for the 36 projects for which we asked for feedback, with the goal of highlighting the magnitude of the investigated phenomenon. The analysis concerns a total of 18,474 builds from January 1, 2018 to August 15, 2018. This results in 8,520 detected incidents, 3,823 if we consider only high-severity warnings for *Slow Build* and *Late Merging*.

Concerning *Slow Build*, 20 projects exhibit an increasing trend in build time, whereas only 11 had a decrease, and 5 were stable. The percentage of cases in which a medium-severity warning could be generated is fairly high, with a median of 25% of the builds and a maximum (*authorjapps/zerocode*), where nearly all builds (93%) are slower than the third quartile of the previous time window. This indicates a slow increase in the build time, which can be normal project evolution. A single incident might not be worrisome *per se*, so we visualize the overall trend (see Fig. 7.5a). High-severity warnings (i.e., outliers) are not particularly frequent (75% of the projects have less than 3% of their builds exhibiting this warning), indicating that while the *Slow Build* phenomenon is quite pervasive, in most cases it manifests quite slowly over time.

Even though working on `master` is discouraged and a pull request paradigm has been advocated [40], we find *Broken Release Branch* in all projects: a median

Table 7.1: CI anti-patterns detected in the analyzed projects

SLOW BUILD						
Proj. with incr. trend						20
Proj. with decr. trend						11
Proj. with stable trend						5
Overall # of medium sev. warnings						4,634
Overall # of high sev. warnings						229
	Min	1Q	Median	3Q	Max	
% of medium sev. warn.	0%	16.80%	25.67%	35.76%	93.87%	
% of high sev. warn.	0%	0.44%	1.20%	2.95%	26.73%	
BROKEN RELEASE BRANCH						
Affected projects						36
Total # of incidents						3,423/18,474
Proj. with incr. trend						16
Proj. with decr. trend						19
	Min	1Q	Median	3Q	Max	
% of incidents	0.31%	6.46%	11.51%	28.69%	51.10%	
Fixing time	54.18 m	9.44 h	17.07 h	3.10 d	6.04 w	
SKIP FAILED TESTS						
Affected projects						15
Overall # of detected incidents						56
	Min	1Q	Median	3Q	Max	
% of affected builds	0.17%	0.24%	0.65%	1.35%	2.47%	
LATE MERGING						
Affected projects						35
# of medium severity warnings						63
# of high severity warnings						115
	Min	1Q	Median	3Q	Max	
# of affected branches	1	1.5	2	4	20	
% of affected branches	25.00%	46.28%	66.67%	100.00%	100.00%	

of 11.51% of the `master` builds are broken. At the same time, our data indicates that breaks are typically fixed within one day (i.e., the median is about 17h), even though the median fix time is above 3 days for the upper quartile of projects. We found one project (*rackerlabs/blueflood*) for which the `master` branch remained broken on average for over 6 weeks.

Skip Failed Test is the least prominent problem in the analyzed project histories. We found instances of this smell for 15 out of the 36 projects, in a total of 56 builds. The percentage of builds affected by this anti-pattern is below 2.5%. While instances of the anti-pattern can be found, developers seem to take failed tests seriously and do not skip them.

Concrete Questions About a Report (for original developers)

- The report is useful for my project and contains relevant warnings.
- I learned something about my project that I have not been aware of before.
- The results made me curious and I plan to investigate the different warnings.

General Questions About Usefulness of Examples

Slow Build, Failed-Test Skipping, Late Merging:

- This summary helps me to identify anti-pattern X.
- I know how to address the different warnings about anti-pattern X.
- High-severity warnings about anti-pattern X should fail the build.

Broken Release Branch:

- This summary improves awareness about...
 - ... the frequency of release-branch failures.
 - ... the time it takes to fix release-branch failures.
- I know how to improve the trend of this summary in the future.

General Validation of Tool and Idea

- The CI report provides information that is not available in any other tool.
- The reports provide a good overview of the CI practices used in a project.
- Frequent reports would have a positive influence on CI practices.
- I would like to integrate such a reporting in my own CI pipeline.

Figure 7.7: Questions of the survey on reports generated by CI-Odor

Concerning *Late Merging*, the anti-pattern affected nearly all projects (35 out of 36), and we raised a total of 115 high severity warnings, and 63 medium severity warnings. The median number of branches affected by a warning is 2, and only in one project *Evolveum/midpoint* the problem affected 20 branches, although the median percentage of affected branches is quite substantial (66.67%).

7.5.3 Survey on Generated Reports

We have conducted a second survey study to validate the usefulness of our generated reports.

Survey Design. To perform the study, we designed a questionnaire composed of a demographics section plus three sections, each one comprising Likert-scale questions and a field for open comments. In the first section, we asked original developers about the report that we have generated for their project. This part was automatically skipped for developers that have not seen a report. In the second section, we introduced the four different detector categories through an exemplary screenshot. We then ask questions about the understandability of the summary, its

actionability, and whether detected deviations should fail the build (if applicable). A final section of the survey contained general questions about the usefulness of the presented summaries. An excerpt (shortened questions, no demographics) of the survey questionnaire (the complete one is on our artifact page [128]), is depicted in Fig. 7.7.

Advertisement. We had two separate advertisement strategies to find study participants. To find *original developers*, we have created up-to-date summaries for our selected target projects and posted them on their corresponding communication channels, asking project members for feedback. At the same time, to receive enough *general feedback* about our summaries, we advertised the survey on Twitter, Reddit (targeting the same sub-forums of the previous survey) and we also sent a follow-up email to every participant of our first survey that allowed us to contact her again.

Demographics. In the end, 113 developers opened our survey, out of which 50 answered all questions (11 original and 39 general developers), resulting in a completion rate of 44.2%. We cannot calculate the return rate for the general population, but we know that we sent the reports out to 36 projects and we heard back from 7 projects (return rate of 19%). We included the control questions from our first survey again and excluded from the analysis 7 participants that indicated low experience. To increase the number of answers, we also kept partial answers. In total, we considered 42 answers as valid, out of which 13 were given by original developers.

Data-Analysis Methodology. As for the Likert-scale answers in the first survey (see Section 7.3), we present the results in asymmetric stacked bar charts (Fig. 7.8). We have received considerably less open answers, so we did not open-code all answers, but we will rather discuss the main points raised.

Report Rating. When we asked original developers about the usefulness of the reports for their projects, almost two-thirds (61.6%) of them agree that the report was useful and contained relevant information. We analyzed the open answers of the 23.1% that disagree and found that most either complain about project specifics that are not considered in the reports or about a bug that we had in the beginning. We got mixed answers about the novelty of information, the same amount (38.5%) of agreements and disagreements. This could be a sign that experienced developers

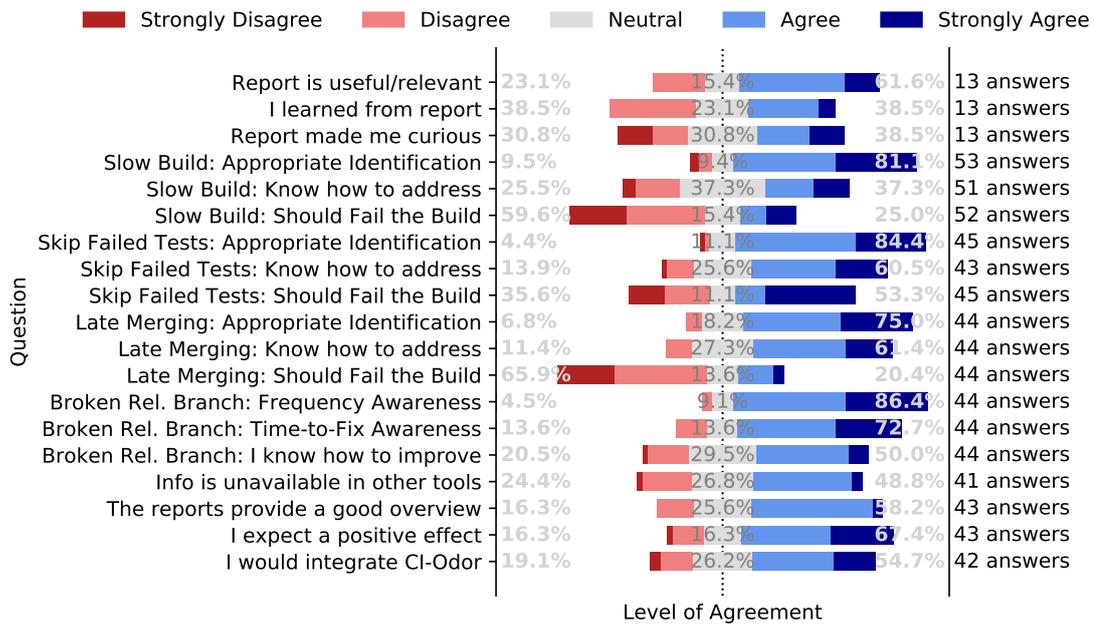


Figure 7.8: Likert-scale answers to the survey on reports generated by CI-Odor

are aware of these deviations in their project, but we did not have a question in the survey to back up this conjecture. We got a similar result when we asked about their reaction. 38.5% of original developers agree that the report made them curious and that they plan an investigation. Overall, we see these results as a sign that the reports are insightful for developers and introduce a minimal overhead.

Identification and Awareness. Across all anti-pattern detectors, the large majority (*Slow Build*: 81.1%, *Skip Failed Tests*: 84.4%, *Late Merging*: 75%, *Broken Release Branch*: 86.4% and 72.7%) of participants agree that the summaries are useful for anti-patterns identification and for increased awareness respectively. This makes us confident that we opted for the right statistics and picked good visualization strategies.

Actionability. The majority of participants agree that they know how to address the warnings for *Skip Failed Tests* (60.5%) and *Late Merging* (61.4%), which is unsurprising because the report points to concrete problems to fix. However, a considerable number of participants disagree for *Broken Release Branch* (20.5%) and especially for *Slow Build* (25.5%). Given the high agreement on *Identification and Awareness*, we think that the disagreement on actionability is a sign that the

report unveils the problem, but that deriving a fix is harder because it affects the process and team practices.

On Using CI-Odor to Fail Builds. The majority of participants disagree with the idea that a build should fail when it is slow (59.6%) or when signs of a *Late Merging* exist (65.9%) and only a small group agrees with this idea (25% and 20.4%). Although build failures provide feedback about issues in newly committed code such as bugs [10] or poor quality [142], our participants typically do not want detected anti-patterns in the CI process to break the build. The only exception is represented by *Skip Failed Tests*, where 53.3% are in favor of failing the build. From the open answers in the first survey, we know that some developers see this anti-pattern as a serious problem.

General Validation. The last part of the survey contained several statements about the validity of the reports and the general idea. 58.2% of the participants agree that the summaries are useful and that they contain relevant information for the project. 67.4% expect a positive effect from integrating our tool to their CI discipline and 54.7% are willing to integrate CI-Odor in their pipeline.

7.6 Discussion

This paper has introduced the idea that monitoring the CI process might be useful to discover the decay of best practices over time. Building a proof-of-concept implementation, CI-Odor, and surveying developers about the idea and our tool, we gained several valuable insights into the perceived or expected benefits of such an approach and actionable findings that have an impact on future work.

Positive Effect & Awareness. The first survey has shown us that anti-patterns are a relevant problem for CI. Best practices are not always being followed, and can even be accidentally broken. Almost two-thirds of the participants to the second survey expect that using such a reporting frequently would have a positive influence on their CI discipline.

Transparency. The study participants suggested that the tool should make the detection strategy fully transparent to increase the trust and acceptance among its users. We only briefly described the detection rules in the summary pages, to

allow study participants performing their task efficiently, but a production-ready tool could involve a fully-fledged description of the detectors.

Learnability. Participants of our first survey confirmed the usefulness of the proposed CI monitoring, especially in the early stages of CI adoption or to train project newcomers. Nearly half of the participants of our second survey were already aware of most of the highlight problems, but it is important to remark that our analysis excluded inexperienced developers about CI. The potential of a regular CI report can be seen by the 38.5% of participants that got curious from the report and started to investigate the reported issues.

Configurability. Our first survey indicated that since projects are very different, developers may want the reports and the detection thresholds to be customized based on their needs. While half of our participants are willing to integrate an anti-pattern detector in their pipeline, this percentage could increase by enhancing usability and reconfigurability, e.g., giving the freedom to enable/disable specific detectors or configure thresholds.

Ultimately, the important question is how useful a concrete report is, therefore we have asked original developers to rate the report that we have generated for their project. The low disagreement rate in this question (23%) and the high agreement for the integration in their own pipeline (55%) make us confident that the described reporting is a promising tool for software development teams that follow CI principles.

7.6.1 Threats to Validity

Threats to *construct validity* are related to the relationship between theory and observations. They mainly concern:

Implementation Bugs. Our implementation might contain bugs that cause the tool to report false positives or false negatives. While the absence of a labeled dataset made it hard to evaluate the detection strategies, we mitigated this threat through a manual review of a sample of the generated results and through a pilot study conducted before the second survey.

Build Cleanup. Travis-CI allows projects to delete their old build logs, e.g., as part of regular maintenance activities, and this could affect the historical analyses

our tool performs. However, previous work has shown that such deletions are unlikely [124], so we do not expect that our results are significantly affected.

Threats to *internal validity* are related to confounding factors internal to our study. In particular, the validation of our summaries could be affected by our selection of projects. We have mitigated this by selecting a diverse set of projects from GitHub and ensured a certain level of maturity by considering the popularity of a project. Unfortunately, our data source, GHTorrent, only approximates the number of committers in a project, which might result in a less diverse project sample.

Threats to *external validity* concern the work's generalizability and are related to:

Sample Size and Diversity. Our evaluation would surely benefit from the analysis of a larger sample of projects, as well as of the participation of more developers. In this work, the analysis of projects was limited to the ones for which we ask for feedback, and the technological limitations (Java, Maven) do not affect the relevance of the detected anti-patterns, as explained in Section 7.4.

Irrelevant Selection of Anti-Patterns. It is possible that our work missed anti-patterns that are highly-relevant for developers. We have reduced this threat through a first internal pre-selection of supported anti-patterns, which we validated in our first survey. Also, it is important to remark that it is not our goal to provide a comprehensive detector for all anti-patterns proposed by Duvall [27], but rather to propose and validate the idea — and its implementation in CI-Odor — of reporting CI decay to developers.

7.6.2 Future Work

Add Additional Detectors. While, as stated above, we validated the general CI-Odor perspective and four relevant anti-patterns, our future work primarily goes into providing additional detectors for new anti-patterns.

Consider More Contextual Information. Right now, we only leverage information from the Travis-CI logs and basic information from the code repositories. However, future work would integrate additional process-related metrics derived from other sources like bug trackers, task management systems, or communication platforms.

Support More Project-Specific Policies. A CI-supporting tool with smart capabilities could learn the problems/warnings in which developers are interested in, and personalize the recommendation consequently.

Derive Project-Specific Thresholds. While we considered thresholds based on consolidated statistics, future work could also consider adaptive, project-specific threshold learning and calibration.

7.7 Related Work

Researchers have investigated the CI adoption [46, 53, 59, 113], finding, in particular, numerous barriers for CI adoption [45], e.g., related to assurance, security, and flexibility in performing tasks such as source code debugging. In such a context, approaches like CI-Odor can be used to help developers understanding when they are not using CI properly.

Previous work has investigated best practices while using CI [145]. In their landmark work, Duvall et al. [25] identified principles and key practices of CI but also pointed out the risks deriving from the misuse of CI. Furthermore, Humble and Farley [47] performed a broader study, analyzing the key ingredients of a Continuous Delivery (CD) pipeline, as well as anti-patterns to be avoided. Such anti-patterns were better explained in the follow-up work by Duvall [27] where all the practices contained in books about CI [25] and CD [47] were condensed in a catalog of bad/good practices regarding the adoption of the whole CD pipeline with specific focus on the core part of CD, i.e., CI. Such a catalog is a comprehensive set of 50 patterns and anti-patterns regarding several phases or relevant topics in the CI/CD process. As explained in Section 7.3, Duvall's catalog constitutes the inception of our work.

One of the best practices associated with CI is the use of Infrastructure as Code (IaC) in order to implement the desired pipeline. Sharma et al., leveraged best practices associated with code quality management to assess configuration code quality and proposed a catalog of 13 implementation and 11 design configuration smells [108]. Recent work by Gallaba et al., [34] also investigated configuration smells and based on rules provided by linters (e.g., TravisLint) they measure smells

and derive automated fixes for them. Our smells have a different focus, because we look at process-related smells rather than at configuration issues.

Rahman and Williams [91, 95] proposed a text-mining approach to identify defective IaC scripts, focusing on security and privacy issues, e.g., related to file permissions or user accounts. Their work is complementary to ours as it deals with a very specific category of problems.

Studying and proposing automated fixes for build failures has also been a topic of investigation. Previous work has investigated the phenomenon of build failures [100, 134] from different perspectives, such as testing [10] and code analysis [142]. Also, researchers have proposed fixes for some kinds of build failures, e.g., broken dependencies related [66], or proposed approaches to augment the comprehensibility of build logs while inspecting the cause of such failures [131]. Our CI smells detector increases the awareness of developers about problems degrading their current CI practice. Given that our smells are just symptoms of bad practices we do not provide any automated fix for such issues but we let developers decide whether taking action or not.

7.8 Summary

This paper investigates the phenomenon that CI development practices decay over time. We survey 124 developers (80% from industry) to understand the problem. Beyond agreeing on the problem relevance, our respondents also confirm that CI anti-patterns are a major cause for the degradation of CI processes. To support developers in preventing CI pipeline from deteriorating we propose CI-Odor, an automated reporting tool of CI anti-patterns. We validate our approach, CI-Odor, by surveying 13 original developers about summaries for their projects, and by asking another 42 developers about the general usefulness. The results show that CI-Odor increases the awareness about CI anti-patterns, is perceived as useful, and that developers would integrate it into their pipeline.

Acknowledgments

We would like to thank all the study participants. C. Vassallo and H. C. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275). C. Vassallo also acknowledges the student sponsoring support by CHOOSE, the Swiss Group for Software Engineering.

Bibliography

- [1] Abdalkareem, R., Mujahid, S., Shihab, E., and Rilling, J. (2019). Which commits can be CI skipped? *IEEE Transactions on Software Engineering*, pages 1–1.
- [2] Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., and Tamburri, D. A. (2017). Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498.
- [3] Barr, E. T., Bird, C., Rigby, P. C., Hindle, A., Germán, D. M., and Devanbu, P. T. (2012). Cohesive and isolated development with branches. In *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 316–331. Springer.
- [4] Basili, V. R. (1992). Software modeling and measurement: The goal/question/metric paradigm. Technical report, University of Maryland at College Park, College Park, MD, USA.
- [5] Bavota, G., Gravino, C., Oliveto, R., De Lucia, A., Tortora, G., Genero, M., and Cruz-Lemus, J. A. (2011). Identifying the weaknesses of uml class diagrams during data model comprehension. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11*, pages 168–182, Berlin, Heidelberg. Springer-Verlag.
- [6] Beck, K. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

- [7] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). The agile manifesto.
- [8] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. (2018). Deflaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 433–444.
- [9] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. (2016). Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 470–481.
- [10] Beller, M., Gousios, G., and Zaidman, A. (2017a). Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 356–367. ACM.
- [11] Beller, M., Gousios, G., and Zaidman, A. (2017b). Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*.
- [12] Bevan, N. (1999). Quality in use: Meeting user needs for quality. *Journal of systems and software*, 49(1):89–96.
- [13] Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. (2009). The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10.
- [14] Bonaccorsi, A. and Rossi, C. (2004). Altruistic individuals, selfish firms? the structure of motivation in open source software. *First Monday*, 9(1).
- [15] Booch, G. (1991). *Object Oriented Design: With Applications*. Benjamin Cummings.
- [16] Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54.

- [17] Chen, L. (2017). Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72 – 86.
- [18] Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46.
- [19] Couto, C., Montandon, J. E., Silva, C., and Valente, M. T. (2011). Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21:241–257.
- [20] Dabbish, L. A., Stuart, H. C., Tsay, J., and Herbsleb, J. D. (2012). Social coding in github: transparency and collaboration in an open software repository. In *CSCW*, pages 1277–1286. ACM.
- [21] Désarmieux, C., Pekatikov, A., and McIntosh, S. (2016). The dispersion of build maintenance activity across maven lifecycle phases. In *Proc. Int’l Conference on Mining Software Repositories (MSR)*, pages 492–495.
- [22] Deshpande, A. and Riehle, D. (2008). Continuous integration in open source software development. *Open source development, communities and quality*, pages 273–280.
- [23] Downs, J., Plimmer, B., and Hosking, J. G. (2012). Ambient awareness of build status in collocated software teams. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 507–517.
- [24] Ducheneaut, N. (2005). Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368.
- [25] Duvall, P., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. A Martin Fowler signature book. Addison-Wesley.
- [26] Duvall, P. M. (2010). Continuous integration. patterns and antipatterns. *DZone refcard #84*.

- [27] Duvall, P. M. (2011). Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard #145*.
- [28] Ernst, N. A. and Mylopoulos, J. (2010). On the perception of software quality requirements during the project lifecycle. In *REFSQ*, volume 6182 of *Lecture Notes in Computer Science*, pages 143–157. Springer.
- [29] Everitt, B. (2002). *The Cambridge dictionary of statistics*. Cambridge University Press, Cambridge, UK; New York.
- [30] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
- [31] Fowler, M. and Foemmel, M. (2016). Continuous integration.
- [32] Fraser, G., Staats, M., McMinn, P., Arcuri, A., and Padberg, F. (2015). Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):23.
- [33] Gallaba, K., Macho, C., Pinzger, M., and McIntosh, S. (2018). Noise and heterogeneity in historical build data: An empirical study of travis ci. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–97.
- [34] Gallaba, K. and McIntosh, S. (2020). Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50.
- [35] Ghaleb, T. A., Alencar da Costa, D., Zou, Y., and Hassan, A. E. (2019). Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering*, pages 1–1.
- [36] Ghaleb, T. A., da Costa, D. A., and Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139.

- [37] Gibbs, S., Casais, E., Nierstrasz, O., Pintado, X., and Tschritzis, D. (1990). Class management for software communities. *Communications of the ACM*, 33(9):90–103.
- [38] Goodman, L. A. (1961). Snowball sampling. *The annals of mathematical statistics*, pages 148–170.
- [39] Gousios, G. (2013). The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 233–236.
- [40] Gousios, G., Storey, M.-A., and Bacchelli, A. (2016). Work practices and challenges in pull-based development: The contributor’s perspective. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 285–296, New York, NY, USA. ACM.
- [41] Haiduc, S., Aponte, J., and Marcus, A. (2010). Supporting program comprehension with source code summarization. In *ICSE (2)*.
- [42] Hartigan, J. A. and Wong, M. A. (1979). A k-means clustering algorithm. *Applied Statistics*, 28:100–108.
- [43] Hassan, F. and Wang, X. (2018). Hirebuild: an automatic approach to history-driven repair of build scripts. In *ICSE*, pages 1078–1089. ACM.
- [44] Hermans, F., Pinzger, M., and van Deursen, A. (2012). Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 34th International Conference on Software Engineering*, pages 441–451. IEEE Press.
- [45] Hilton, M., Nelson, N., Tunnell, T., Marinov, D., and Dig, D. (2017). Trade-offs in continuous integration: assurance, security, and flexibility. In *ESEC/SIGSOFT FSE*, pages 197–207. ACM.
- [46] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

- [47] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education.
- [48] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE.
- [49] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2016). An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071.
- [50] Kerzazi, N., Khomh, F., and Adams, B. (2014). Why do automated builds break? an empirical study. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.
- [51] Khomh, F., Adams, B., Dhaliwal, T., and Zou, Y. (2015). Understanding the impact of rapid releases on software quality - the case of firefox. *Empirical Software Engineering*, 20(2):336–373.
- [52] Kim, S. and Ernst, M. D. (2007). Which warnings should I fix first? In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 45–54.
- [53] Kim, S., Park, S., Yun, J., and Lee, Y. (2008). Automated continuous integration of component-based software: An industrial experience. In *ASE*, pages 423–426. IEEE Computer Society.
- [54] Kogan, J. (2007). *Introduction to Clustering Large and High-Dimensional Data*. Cambridge University Press, New York, NY, USA.
- [55] Krippendorff, K. (1980). *Content analysis: An introduction to its methodology*. Sage.

- [56] Kwan, I., Schroter, A., and Damian, D. (2011). Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, 37(3):307–324.
- [57] Lagerberg, L., Skude, T., Emanuelsson, P., Sandahl, K., and Stahl, D. (2013). The impact of agile principles and practices on large-scale software development projects: A multiple-case study of two projects at ericsson. In *ESEM*, pages 348–356. IEEE Computer Society.
- [58] LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 492–501, New York, NY, USA. ACM.
- [59] Laukkanen, E. I., Paasivaara, M., and Arvonen, T. (2015). Stakeholder perceptions of the adoption of continuous integration - A case study. In *AGILE*, pages 11–20. IEEE Computer Society.
- [60] Lebeuf, C., Storey, M. D., and Zagalsky, A. (2018). Software bots. *IEEE Software*, 35(1):18–23.
- [61] Lebeuf, C., Zagalsky, A., Foucault, M., and Storey, M. D. (2019). Defining and classifying software bots: a faceted taxonomy. In *Proceedings of the 1st International Workshop on Bots in Software Engineering, BotSE@ICSE 2019, Montreal, QC, Canada, May 28, 2019.*, pages 1–6.
- [62] Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology*.
- [63] Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J., and Kahkonen, T. (2004). Agile software development in large organizations. *Computer*, 37(12):26–34.
- [64] Lou, Y., Chen, J., Zhang, L., Hao, D., and Zhang, L. (2019). History-driven build failure fixing: how far are we? In *ISSTA*, pages 43–54. ACM.

- [65] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 643–653.
- [66] Macho, C., McIntosh, S., and Pinzger, M. (2018). Automatically repairing dependency-related build breakage. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.
- [67] Macqueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297.
- [68] Manuel Gerardo Orellana Cordero, Gulsher Laghari, A. M. and Demeyer, S. (2017). On the differences between unit and integration testing in the travistorrent dataset. In *Proceedings of the 14th working conference on mining software repositories*.
- [69] Maple, S. (2016). Java tools and technologies landscape report 2016. *Zero-Turnaround post*.
- [70] Marlow, J., Dabbish, L., and Herbsleb, J. D. (2013). Impression formation in online peer production: activity traces and personal profiles in github. In *CSCW*, pages 117–128. ACM.
- [71] Mcintosh, S., Adams, B., Nagappan, M., and Hassan, A. E. (2014). Mining co-change information to understand when build changes are necessary. In *Proc. Int’l Conf on Software Maintenance and Evolution (ICSME)*, pages 241–250. IEEE.
- [72] McIntosh, S., Adams, B., Nguyen, T. H., Kamei, Y., and Hassan, A. E. (2011). An empirical study of build maintenance effort. In *Proceedings of the Int’l Conference on Software Engineering (ICSE)*, pages 141–150.
- [73] McKee, S., Nelson, N., Sarma, A., and Dig, D. (2017). Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 467–478.

- [74] Menezes, G. G. L., Murta, L. G. P., Barros, M. O., and Van Der Hoek, A. (2018). On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, pages 1–1.
- [75] Miller, A. (2008). A hundred days of continuous integration. In *Proceedings of the Agile 2008*, AGILE '08, pages 289–293.
- [76] Moha, N., Guéhéneuc, Y., Duchien, L., and Meur, A. L. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36.
- [77] Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L. L., and Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for java classes. In *ICPC*, pages 23–32. IEEE Computer Society.
- [78] Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., and Canfora, G. (2017). ARENA: an approach for the automated generation of release notes. *IEEE Trans. Software Eng.*, 43(2):106–127.
- [79] Moreno, L., Bavota, G., Penta, M. D., Oliveto, R., and Marcus, A. (2015). How can I use this method? In *ICSE (1)*, pages 880–890. IEEE Computer Society.
- [80] Moreno, L. and Marcus, A. (2017). Automatic software summarization: the state of the art. In *ICSE (Companion Volume)*, pages 511–512. IEEE Computer Society.
- [81] Myers, G. J. (2004). *The art of software testing (2. ed.)*. Wiley.
- [82] Olsson, H. H., Alahyari, H., and Bosch, J. (2012). Climbing the “stairway to heaven” – a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '12, pages 392–399, Washington, DC, USA. IEEE Computer Society.

- [83] Oppenheim, B. (1992). *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers.
- [84] Orellana Cordero, M. G., Laghari, G., Murgia, A., and Demeyer, S. (2017). On the differences between unit and integration testing in the travistorrent dataset. In *International Conference on Mining Software Repositories, MSR'17*, pages 451–454, New York, NY, USA. ACM.
- [85] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. (2014). Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, pages 101–110. IEEE.
- [86] Palomba, F. and Zaidman, A. (2017). Does refactoring of test smells induce fixing flaky tests? In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 1–12.
- [87] Panichella, S., Panichella, A., Beller, M., Zaidman, A., and Gall, H. C. (2016). The impact of test case summaries on bug fixing performance: an empirical investigation. In *ICSE*, pages 547–558. ACM.
- [88] Ponzanelli, L., Bavota, G., Penta, M. D., Oliveto, R., and Lanza, M. (2014). Mining StackOverflow To Turn The IDE Into A Self-Confident Programming Prompter. In *MSR*.
- [89] Potdar, A. and Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 91–100.
- [90] R Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.
- [91] Rahman, A. (2018). Characteristics of defective infrastructure as code scripts in devops. In *Proceedings of the 40th International Conference on Software*

- Engineering: Companion Proceedings*, ICSE '18, pages 476–479, New York, NY, USA. ACM.
- [92] Rahman, A., Farhana, E., Parnin, C., and Williams, L. (2020a). Gang of eight: A defect taxonomy for infrastructure as code scripts. *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*.
- [93] Rahman, A., Farhana, E., and Williams, L. (2020b). The ‘as code’ activities: Development anti-patterns for infrastructure as code. *Empir. Softw. Eng.*, 25(5):3430–3467.
- [94] Rahman, A., Parnin, C., and Williams, L. (2019). The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, pages 164–175, Piscataway, NJ, USA. IEEE Press.
- [95] Rahman, A. and Williams, L. (2018). Characterizing defective configuration scripts used for continuous deployment. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 34–45.
- [96] Rahman, A. and Williams, L. (2019). Source code properties of defective infrastructure as code scripts. *Inf. Softw. Technol.*, 112:148–163.
- [97] Rahman, M. T., Querel, L.-P., Rigby, P. C., and Adams, B. (2016). Feature toggles: practitioner practices and a case study. In *Working Conference on Mining Software Repositories*.
- [98] Rastkar, S., Murphy, G. C., and Murray, G. (2010). Summarizing software artifacts: a case study of bug reports. In *ICSE (1)*, pages 505–514. ACM.
- [99] Rastkar, S., Murphy, G. C., and Murray, G. (2014). Automatic summarization of bug reports. *IEEE Trans. Software Eng.*, 40(4):366–380.
- [100] Rausch, T., Hummer, W., Leitner, P., and Schulte, S. (2017). An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *International Conference on Mining Software Repositories (MSR)*. ACM.

- [101] Robbins, N. B. and Heiberger, R. M. (2011). Plotting likert and other rating scales. In *Proceedings of the 2011 Joint Statistical Meeting*, pages 1058–1066.
- [102] Robinson, D. (2003). *An Introduction to Abstract Algebra*. De Gruyter textbook. Walter de Gruyter.
- [103] Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. (2016). Continuous deployment at facebook and OANDA. In *Companion proceedings of the 38th International Conference on Software Engineering (ICSE Companion)*, pages 21–30.
- [104] Schermann, G., Cito, J., and Leitner, P. (2018a). Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31.
- [105] Schermann, G., Cito, J., Leitner, P., and Gall, H. C. (2016). Towards quality gates in continuous delivery and deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4.
- [106] Schermann, G., Cito, J., Leitner, P., Zdun, U., and Gall, H. C. (2018b). We’re doing it live: A multi-method empirical study on continuous experimentation. *Inf. Softw. Technol.*, 99:41–57.
- [107] Seo, H., Sadowski, C., Elbaum, S. G., Aftandilian, E., and Bowdidge, R. W. (2014). Programmers’ build errors: a case study (at Google). In *International Conference on Software Engineering (ICSE)*.
- [108] Sharma, T., Fragkoulis, M., and Spinellis, D. (2016). Does your configuration code smell? In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 189–200.
- [109] Sheskin, D. J. (2007). *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All.
- [110] Sorbo, A. D., Panichella, S., Alexandru, C. V., Shimagaki, J., Visaggio, C. A., Canfora, G., and Gall, H. C. (2016). What would users change in my app? summarizing app reviews for recommending software changes. In *SIGSOFT FSE*, pages 499–510. ACM.

- [111] Spencer, D. (2009). *Card sorting: Designing usable categories*. Rosenfeld Media.
- [112] Staahl, D. and Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *J. Syst. Softw.*, 87:48–59.
- [113] Ståhl, D. and Bosch, J. (2014). Automated software integration flows in industry: a multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 54–63. ACM.
- [114] Tamburri, D. A., Kazman, R., and Fahimi, H. (2016). The architect’s role in community shepherding. *IEEE Software*, 33(6):70–79.
- [115] Thorve, S., Sreshtha, C., and Meng, N. (2018). An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 534–538.
- [116] Treude, C., Barzilay, O., and Storey, M.-A. (2011). How do programmers ask and answer questions on the web? (nier track). In *Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11*, pages 804–807, New York, NY, USA. ACM.
- [117] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2017a). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088.
- [118] Tufano, M., Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2017b). There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.*, 29(4).
- [119] Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley.
- [120] Urli, S., Yu, Z., Seinturier, L., and Monperrus, M. (2018). How to design a program repair bot?: Insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP ’18*. ACM.

- [121] van Deursen, A., Moonen, L., Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95.
- [122] Vasilescu, B. (2014). Human aspects, gamification, and social media in collaborative software engineering. In *ICSE Companion*, pages 646–649. ACM.
- [123] Vasilescu, B., Filkov, V., and Serebrenik, A. (2013). Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *SocialCom*, pages 188–195. IEEE Computer Society.
- [124] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM.
- [125] Vassallo, C., Palomba, F., Bacchelli, A., and Gall, H. C. (2018a). Continuous Code Quality: Are We (Really) Doing That? Online Appendix. <https://doi.org/10.5281/zenodo.1341015>.
- [126] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C. (2018b). Context is king: The developer perspective on the usage of static analysis tools. In *SANER*, pages 38–49. IEEE Computer Society.
- [127] Vassallo, C., Panichella, S., Penta, M. D., and Canfora, G. (2014). CODES: mining source code descriptions from developers discussions. In *ICPC*, pages 106–109. ACM.
- [128] Vassallo, C., Proksch, S., Gall, H. C., and Di Penta, M. (2019). Artifact Page - Automated Reporting of Anti-Patterns and Decay in Continuous Integration. <https://doi.org/10.5281/zenodo.2566032>.
- [129] Vassallo, C., Proksch, S., Gall, H. C., and Di Penta, M. (2019). Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 105–115.

- [130] Vassallo, C., Proksch, S., Jancso, A., Gall, H. C., and Di Penta, M. (2020a). Replication Package for “Configuration Smells in Continuous Delivery Pipelines: A Linter and A Six-Month Study on GitLab”. <https://doi.org/10.5281/zenodo.3861003>.
- [131] Vassallo, C., Proksch, S., Zemp, T., and Gall, H. C. (2018c). Un-Break My Build: Assisting Developers with Build Repair Hints. In *International Conference on Program Comprehension*.
- [132] Vassallo, C., Proksch, S., Zemp, T., and Gall, H. C. (2019). Replication Package for “Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution”. <https://doi.org/10.5281/zenodo.3346615>.
- [133] Vassallo, C., Proksch, S., Zemp, T., and Gall, H. C. (2020b). Every build you break: Developer-oriented assistance for build failure resolution. *Empirical Software Engineering (EMSE)*.
- [134] Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Penta, M. D., and Panichella, S. (2017). 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 2017, Shanghai, China, September 17-22, 2017*, pages 183–193.
- [135] Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Penta, M. D., and Zaidman, A. (2016). Continuous delivery practices in a large financial organization. In *ICSME*, pages 519–528. IEEE Computer Society.
- [136] Vos, T. E. J., Tonella, P., Prasetya, W., Kruse, P. M., Bagnato, A., Harman, M., and Shehory, O. (2014). FITTEST: A new continuous and automated testing process for future internet applications. In *CSMR-WCRE*, pages 407–410. IEEE Computer Society.
- [137] Walls, M. (2013). *Building a DevOps Culture*. O’Reilly Media.
- [138] Wedyan, F., Alrmony, D., and Bieman, J. M. (2009). The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In

- 2009 International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE.
- [139] Wong, E., Yang, J., and Tan, L. (2013). Autocomment: Mining question and answer sites for automatic comment generation. In *ASE*, pages 562–567. IEEE.
- [140] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–172.
- [141] Ying, A. T. T. and Robillard, M. P. (2013). Code fragment summarization. In *ESEC/SIGSOFT FSE*, pages 655–658. ACM.
- [142] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., and Penta, M. D. (2017). How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 334–344.
- [143] Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H., and Di Penta, M. (2020). An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering (EMSE)*.
- [144] Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H. C., and Di Penta, M. (2019). An Empirical Characterization of Bad Practices in Continuous Delivery (online appendix). <http://home.ing.unisannio.it/fiorella.zampetti/datasets/CIBadPractices.zip>.
- [145] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., and Vasilescu, B. (2017). The impact of continuous integration on other software development practices: A large-scale empirical study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Curriculum Vitae

Personal Details

Name: Carmine Vassallo
Nationality: Italy
Date of Birth: 13 December 1990

Education

September 2016 – October 2020	Doctoral Program at the University of Zurich, Department of Informatics, Chair of Software Engineering, Zurich, Switzerland
September 2013 – May 2016	Master of Science in Computer Engineering at the University of Sannio, Benevento, Italy
September 2009 – May 2013	Bachelor of Science in Computer Engineering at the University of Sannio, Benevento, Italy

