Do Developers Fix Continuous Integration Smells?

Ayberk Yaşa*
ayberkyasa@gmail.com
Bilkent University
Ankara, Turkey

Ege Ergül*
egeergull2001@gmail.com
Bilkent University
Ankara, Turkey

Hakan Erdogmus
hakane@andrew.cmu.edu
Carnegie Mellon University
USA

Eray Tüzün
eraytuzun@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

ABSTRACT

Continuous Integration (CI) is a common software engineering practice in which the code changes are frequently merged into a software project repository after automated builds and tests have been successfully run. CI enables developers to quickly detect bugs, enhance the quality of the code, and shorten review times. However, developers may encounter some obstacles in following the CI principles. They may be unaware of them, they may follow the principles partially or they may even act against them. These behaviors result in CI smells. CI smells may in turn lessen the benefits of CI. Addressing CI smells rapidly allows software projects to fully reap the benefits of CI and increase its effectiveness. The main objective of this study is to investigate how frequently developers address CI smells. To achieve this objective, we first selected seven smells, then implemented scripts for detecting these smells automatically, and then ran the scripts in eight open-source software projects using GitHub Actions. To assess the resolution extent of CI smells by practitioners, we calculated the occurrences and time-to-resolution (TTR) of each smell. Our results suggest that *Skipped Job* smell has been fixed slightly more than other CI smells. The most frequently observed smell was *Long Build*, which was detected in an average of 19.03% of all CI builds. *Fake Success* smell does not get resolved in projects where it exists. Our study reveals that practitioners do not fix CI smells in practice. Further studies are needed to explore the underlying reasons behind this, in order to recommend more effective strategies for addressing these smells.

CCS CONCEPTS

• **Software and its engineering → Agile software development**; *Software maintenance tools*; Software testing and debugging; Software libraries and repositories.

*Both authors contributed equally to this research.

PROMISE '23, December 8, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0375-1/23/12.
https://doi.org/10.1145/3617555.3617870

KEYWORDS

Continuous Integration, Continuous Integration Smells, CI-Smell, Anti-Patterns, Smell Detection, Process Smells

ACM Reference Format:
Ayberk Yaşa, Ege Ergül, Hakan Erdogmus, and Eray Tüzün. 2023. Do Developers Fix Continuous Integration Smells?. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '23), December 8, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3617555.3617870

1 INTRODUCTION

Continuous Integration (CI) is a software engineering practice for having automated builds and applying automated tests to the code changes to be merged. Team members routinely integrate their work, often at least once per day [6]. To find integration errors as early as possible, each integration is validated by an automated build (including tests).

The term CI was coined in 1994 by Grady Booch [9]. Martin Fowler, who is one of the founding members of the Agile Alliance[1], also advocated for CI. Fowler [6] described the practices that make up an effective CI process in 2006. Fowler also discussed the benefits of CI, which can be reaped if these practices are strictly followed.

The adoption of CI practices has increased over the years as developers have introduced CI into their projects. There are advantages to strictly following CI principles and practices. As Fowler [6] discussed, CI allows developers to detect bugs earlier and easier, increases the quality of the code, shortens review times, and lets developers have a better understanding of what works and does not work in the project and get more rapid feedback on new features. To achieve these benefits, which support the quality of the software development process, a set of sub-practices were defined [4, 6, 11]. These include frequently committing code changes, building the software frequently, and performing the tests automatically. However, when these practices are not followed as prescribed, or shortcuts are taken, software projects can be negatively affected. These poor practices are called CI smells.

Previous works categorized CI smells, building on what was originally suggested by Fowler [6], and proposed different taxonomies of these smells [5, 19]. Some researchers developed CI-support tools to examine the CI process from different perspectives and to detect anti-patterns [14, 15, 20]. The impact of CI smells on productivity and quality of projects, development practices, and code reviews

[1]https://www.agilealliance.org/

was also investigated [1, 8, 21]. Zampetti et al. [18] proposed a categorization of actions for restructuring CI pipelines to avoid and remove CI smells.

Although categorizations of actions to fix smells exist, to what extent developers fix CI smells has not been investigated to the best of our knowledge. Knowing how developers follow the best practices is valuable since these CI practices offer several advantages, as examined by Fowler [6]. Being aware of the most commonly seen and resolved smells would allow us to devise automated tools and solutions to avoid these smells in the first place. Therefore, the main goal of this study is to determine how frequently smells are addressed by the developers. In this study, we mined a sample of open-source repositories to obtain this information, focusing on the following research question.

RQ - To what extent do developers fix CI smells?

To answer this research question, we first selected seven smells, then implemented the detection strategy for each smell, and ran these strategies in eight open-source software projects that use GitHub Actions. Finally, we measured each smell's occurrences and average time-to-resolution (TTR) in every project.

The paper is organized as follows. The next section explains the research methodology. Section 3 provides the results, and Section 4 discusses and interprets the findings. Section 5 addresses the validity threats of our investigation. Section 6 discusses the related work. Finally, Section 7 provides the conclusions and discusses future work.

2 METHODOLOGY

Following the snowballing approach suggested by Wohlin [16], we initiated our literature review with three studies [5, 6, 19], which ultimately yielded 18 academic papers. To establish a strong foundation for this study, we deliberately selected these three studies authored by recognized pioneers in the field. These papers provide essential information that serves as a basis for our research focus, as highlighted in Section 1. Next, we identified the most feasible ways to measure how much practitioners fix CI smells, selected a subset of smells, and devised a detection strategy for each smell. Then, we selected open-source projects that are hosted on GitHub and use GitHub Actions on which to run our CI smell detection strategies. Figure 1 gives an overview of our methodology.

2.1 Metrics for Measuring the Extent of Fixing CI Smells

Based on our review, we found three ways to investigate to what extent practitioners resolve CI smells. The first way is to survey experienced developers who are actively responsible for the CI process in their company to get their perceived importance [7, 10].

Secondly, we can take advantage of the folder where the GitHub Actions CI pipelines are stored. The full path of this folder is "*/.github/workflows*" from the repository's root. Each YAML file in this folder contains repository-specific pipelines serving a different purpose[2]. We obtain the change history of each YAML file using the GitHub REST API. We run our smell detection strategies for each change in the history of each file. For the smells that are

[2]Note that in a YAML file, the flow of scripted instructions is called a *workflow*. Thus, the term *workflow* will be used interchangeably with the YAML file defining it.

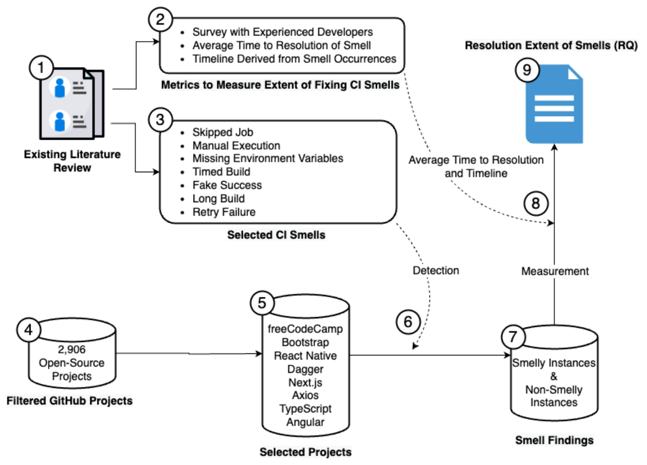
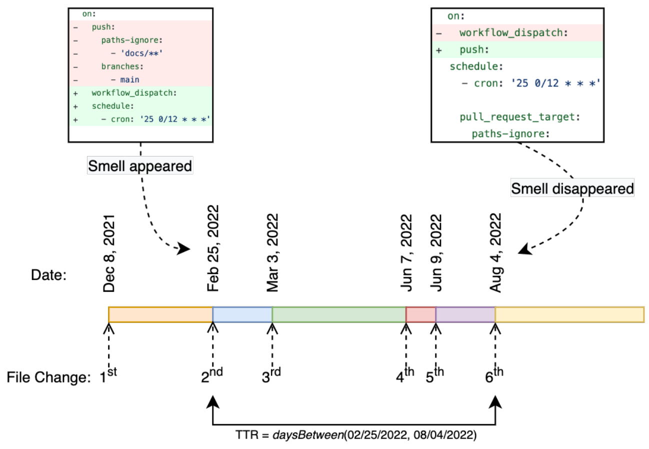

1
Existing Literature Review
2
Survey with Experienced Developers
Average Time to Resolution of Smell
Timeline Derived from Smell Occurrences
Metrics to Measure Extent of Fixing CI Smells
3
Skipped Job
Manual Execution
Missing Environment Variables
Timed Build
Fake Success
Long Build
Retry Failure
Selected CI Smells
Resolution Extent of Smells (RQ)
9
Average Time to Resolution and Timeline
8
Detection
Measurement
4
2,906 Open-Source Projects
Filtered GitHub Projects
5
freeCodeCamp
Bootstrap
React Native
Dagger
Next.js
Axios
TypeScript
Angular
Selected Projects
6
7
Smelly Instances & Non-Smelly Instances
Smell Findings

Figure 1: Methodology overview

not present in the n^{th} file change, but exist since the $(n-k)^{th}$ file change (where $k > 0$), we record the time between these two file changes as the TTR of that smell in days. In this way, we measure the average TTR of each smell in days, as a proxy for the urgency of fixing CI smells (See Figure 2 for an example).

Thirdly, the GitHub API provides log data about the repository and CI pipelines. The entire commit history of a repository can be obtained, as well as data about each CI pipeline run. We can detect smells using this log data as well. In this way, we build a timeline that depicts the change in the frequency of smells over time.

The first method can be applied to all kinds of smells. The second method can be applied to the smells whose detection strategies include YAML file mining, and the third one can be applied to smells detected with the log data. In this study, we will be focusing on the second and third strategies, and we are planning to apply the first method in our future work.


```
  on:
-   push:
-     paths-ignore:
-       - 'docs/**'
-     branches:
-       - main
+   workflow_dispatch:
+   schedule:
+     - cron: '25 0/12 * * *'
```
```
  on:
-   workflow_dispatch:
+   push:
   schedule:
     - cron: '25 0/12 * * *'

    pull_request_target:
      paths-ignore:
```

Figure 2: An example of TTR calculation of Manual Execution smell

2.2 Smell Selection

Zampetti et al. [19] and Duvall [5] proposed 79 CI smells grouped into seven categories and 50 smells grouped into 10 categories, respectively. While some of the smells in these two studies are the same, others are different. Automatically detecting some of them is not practicable since they are based on infrastructure choices, quality assurance processes, delivery processes, or company culture.

Initially, we created a combined list of the smells introduced by Duvall and Zampetti et al. [5, 19]. Then, we classified the smells as feasible or not by inspecting the GitHub Actions pipeline configuration files. That is, we determined whether each smell is measurable and detectable. This was a required step as most of the smells were relatively subjective (e.g. "Generated artifacts are versioned, while they should not" is a decision that may vary from organization to organization), hence immeasurable. Most of the remaining smells were not automatically detectable (e.g. "Build scripts are highly dependent upon the IDE" is a smell that cannot be automatically detected). After this independent classification of the smells by the first two authors, in 91.5% of the cases, there was an agreement on whether a smell was feasible to detect or not. To ensure the agreement was not by chance, we computed Cohen's κ [2] as 0.79 (moderate). For the cases where authors disagreed on whether a smell is feasible to detect or not, the author who claimed the smell to be detectable wrote detection scripts to prove that the smell is detectable. In this way, it was proven to the other author that these smells could be detected. Then, we finalized the list of smells. This step yielded 7 measurable and automatically detectable smells, whereas the rest of the smells were either immeasurable, not automatically detectable, or both. The summary of the selected smells is given in Table 1. To understand how each smell was searched in YAML files, first, the smells need to be explained in more detail.

Pipeline steps/stages are skipped arbitrarily: Zampetti et al. [19] classify this smell under the category of Build Process Organization. This smell is seen when developers force-skip the execution of the workflows. An example would be a developer directly committing local changes to the project by skipping the automated tests. In the rest of the paper, this smell will be referred to as **Skipped Job**.

Some pipeline tasks are started manually: This smell was first introduced by Duvall [5]. Zampetti et al. [19] also acknowledged this smell in their work and categorized it under Build Process Organization. This smell is seen when a workflow is executed manually, whereas it should have been executed automatically after each time a triggering event occurred. For example, if the script that initiates the project build needs manual triggering instead of being triggered every time a new commit arrives, this is considered an instance of this smell. In the rest of the paper, this smell will be referred to as **Manual Execution**.

Use of nightly builds: This smell was first introduced by Duvall [5], and then classified by Zampetti et al. [19] under the category Build Process Organization. This smell is seen when a workflow is executed at predetermined times, whereas it should have been executed automatically after each time a triggering event occurred. For example, if a project is built nightly instead of being built every time a new commit arrives, this is considered as a smell. This specific case is called a nightly build. In the rest of the paper, this smell will be referred to as **Timed Build**.

A build succeeds even when a task is failed, or an error is thrown: This smell is first introduced by Duvall [5]. Then Zampetti et al. [19] also acknowledged it under the category Build Process Organization. As the name suggests, the smell is observed when a workflow completes successfully despite a failed job or a thrown error. In the rest of the paper, this smell will be referred to as **Fake Success**.

Environment variables are not used at all: This smell was first introduced by Duvall [5]. Later on, Zampetti et al. [19] categorized it under Build Maintainability. Although environment variables are needed, when they are not used, this is considered as a smell. In the rest of the paper, this smell will be referred to as **Missing Environment Variables**.

Failed tests are re-executed in the same build: Zampetti et al. [19] classify this smell under the Quality category. The smell is observed when tests are re-executed without any changes to the code or the CI pipeline, even though the tests have already failed before. In the rest of the paper, this smell will be referred to as **Retry Failure**.

Build time for the commit stage takes longer than 10 minutes: This smell was first introduced by Duvall [5], and then Zampetti et al. [19] categorized it under Build Process Organization. The smell is observed when the build of the entire project takes longer than 10 minutes when the pipeline is triggered. In the rest of the paper, this smell will be referred to as **Long Build**.

2.3 Detection Strategy for Selected Smells

We first conducted a review of the GitHub Actions documentation to have a better understanding of how this CI tool works and how we can detect the smells mentioned in Table 1. We checked to see if there is a predefined keyword or configurable parameter which can be added to the pipeline that would cause the practitioners to follow poor practices. If there is a predefined keyword or configurable parameter that can be used directly in GitHub Actions pipeline configuration files (*YAML files*) to detect the smell, we determined the presence of the keyword or the value of the parameter. Note that since there may be multiple CI-related YAML files in a project, the project is considered to have that smell as long as the smell is seen at least in one of these YAML files. The reasoning behind this decision is that in GitHub Actions, even though there may be several YAML files and many workflows, the CI concept as a whole is the combination of these separate workflows. Consequently, it is safe to decide that a project has a CI smell when the smell is observed in at least one of the YAML files.

If neither a relevant keyword nor a parameter exists, it is impossible to detect the smell from YAML files. In such a case, we used the log data of the repository and its CI pipelines through the GitHub REST API to detect the smell. To ensure the repeatability of this project, more detailed information on how to detect the selected smells is provided below according to the smell category.

Skipped Job: In GitHub Actions, a workflow can be skipped if the commit message includes one of the following[3]: `[skip ci]`,

[3]https://docs.github.com/en/actions/managing-workflow-runs/skipping-workflow-runs

Table 1: Summary Table of Selected Continuous Integration Smells

Smell Name	Description
Skipped Job	Pipeline steps/stages are skipped arbitrarily.
Manual Execution	Some pipeline tasks are started manually.
Timed Build	Nightly builds are used.
Fake Success	A build is succeeded when a task is failed, or an error is thrown.
Missing Environment Variables	Environment variables are not used at all.
Retry Failure	Failed tests are re-executed in the same build.
Long Build	Build time for the commit stage takes longer than 10 minutes.

`[ci skip]`, `[no ci]`, `[skip actions]`, `[actions skip]`. Alternatively, one can end the commit message with two empty lines followed by `skip-checks: true`. To detect this smell, each commit message is searched by a script.

Manual Execution: In GitHub Actions, if a YAML file contains the keyword `workflow_dispatch` then the workflow contains this smell[4]. Thus, to detect this smell, we used a script that iterates through each YAML file and searches this keyword. In the presence of this keyword, the project is labeled as containing this smell.

Timed Build: In GitHub Actions, if a YAML file contains the keyword `schedule` then the workflow contains this smell[5]. Thus, to detect this smell, we used a script that iterates each YAML file and search this keyword. In the presence of this keyword, the project is identified as containing this smell.

Fake Success: In GitHub Actions, if a YAML file contains the keyword `continue-on-error` then the workflow contains this smell[6]. Thus, to detect this smell, we used a script that iterated through each YAML file and search for this keyword. In the presence of this keyword, the project is identified as containing this smell.

Missing Environment Variables: To detect this smell, we used a script that iterates through each YAML file and searches the keyword of `env`[7]. In the absence of this keyword, the project is labeled as containing this smell.

Retry Failure: Usually, this smell is introduced due to flaky tests. Such cases are detected by using the GitHub REST API and GitHub Actions logs. In the workflow run histories of each project, if the value of `run_attempt` is greater than one for a workflow, the *Retry Failure* exists for that workflow.

Long Build: This smell is detected by using the log data for workflow runs fetched using the GitHub REST API. To detect this smell, the difference between the date value of `updated_at` and `run_started_at` timestamps are calculated. If this difference is greater than 10 minutes, this smell exists for the workflow.

The summary of the detection strategy for each selected smell is shown in Table 2.

[4]https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#workflow_dispatch
[5]https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#schedule
[6]https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idcontinue-on-error
[7]https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#env

3 RESULTS

In this section, the results that we found by applying the methodology we described are provided.

3.1 Open-Source Project Selection

We limited our investigation to projects that use GitHub Actions as their CI tool. Eight open-source GitHub projects were selected in the end, subject to the following criteria to investigate to what extent software practitioners resolve CI smells.

(1) **Projects should be open source:** It was important that the selected projects were open-source to make the best use of the GitHub REST API. For required access, the projects should be public with no need to obtain access tokens.
(2) **Projects should have at least 5,000 stars:** This way we ensured we were investigating popular enough projects. Our assumption is that popular projects are better maintained, use more streamlined processes, and they are more worthy of investigation.
(3) **Projects should use GitHub Actions and should have at least five workflow files:** Our pre-study investigation of GitHub projects revealed that if a project contained fewer than five workflow files, it was unlikely that any of them were related to CI. Therefore, we looked for projects with at least five workflow files.
(4) **Projects should have at least 100 contributors:** Since we are investigating software practices, our construct validity would improve with contributor diversity. If there were a limited number of contributors, construct validity could be compromised.
(5) **Project should be using GitHub Actions for at least two years:** This way, we ensured that the CI data were available for a sufficiently long time and the CI tool was used long enough to reveal underlying patterns.

GitHub Search API was used to filter projects based on the criteria mentioned above (1, 2, and 4). This initial filtering resulted in a set of 2,609 open-source software projects hosted on GitHub. After this initial filtering, two authors manually inspected and selected the first eight open-source software projects that also satisfied the other two criteria (3 and 5). The details of the open-source projects selected are given in Table 3.

A replication package[8] is available, and includes the data fetched from the GitHub REST API, the source code, and the study results.

[8]https://figshare.com/s/5632179ddf84e6533865

Table 2: Summary Table of Detection Strategy for Each Selected Smell

Smell Name	Detection Strategy
Skipped Job	Log data of repository and CI pipelines
Manual Execution	Parameter of ***workflow_dispatch***
Timed Build	Parameter of ***schedule*** in the parameter of ***on***
Fake Success	Parameter of ***continue-on-error***
Missing Environment Variables	Keywords in the parameter of ***env***
Retry Failure	Log data of repository and CI pipelines
Long Build	Log data of repository and CI pipelines

Table 3: The Details of Selected Open-Source Projects

Project	Starting Year	Number of Contributors	Number of Commits	Number of Stars	Number of Workflow Files	Number of Workflow Runs
freeCodeCamp	2014	4678	32K	364K	20	144K
Bootstrap	2011	1346	22K	163K	13	108K
React Native	2015	2456	27K	109K	9	86K
Dagger	2021	134	6K	7.9K	11	33K
Next.js	2016	2576	15K	103K	11	103K
Axios	2014	405	1K	99K	7	3K
TypeScript	2014	699	34K	89K	15	41K
Angular	2014	1677	26K	87K	11	43K

3.2 Data Collection and Pre-Processing

Before data collection, the required data were identified, and the possible ways to access these data were researched. To detect the *Skipped Job* smell, the commit histories of projects were needed since a pipeline can be skipped by writing one of the special keywords to a commit message. To detect *Retry Failure*, workflow run histories of each project were needed since this smell could be detected if the `run_attempt` count is greater than one for a workflow run. To detect *Fake Success*, *Manual Execution*, *Missing Environment Variables* and *Timed Build* smells, the current and older versions of each YAML file for each project were needed. All four of these smells can be detected by searching a specific keyword on the YAML files (e.g., to detect a *Timed Build*, the `schedule` keyword can be searched in a YAML file). For the evolution of these smells, the version histories of the YAML files were extracted.

There can be more than one YAML file in a single project, and not all of these YAML files have to be related to CI. For example, in some projects, there was a YAML file that auto-closed stale pull requests. Although this YAML file automates a meaningful and required task, the task itself cannot be associated with CI. After the projects that used GitHub Actions were selected, the next step was to manually filter out the YAML files that could not be associated with CI. The first two authors performed this filtration process.

3.3 Evaluating to What Extent CI Smells are Addressed

Our research question investigates to what extent developers fix CI smells. To answer this question, the detection strategies of selected CI smells are executed on all versions of repository-specific pipeline files and the log data of the repository and CI pipelines. The results of four smells (*Manual Execution, Missing Environment Variables, Timed Build, and Fake Success*) and the remaining results are given in different subsections since they are being detected in different ways, and the results are in different formats.

3.3.1 Smells Detected with Pipeline Files. In Table 4, each smell has three fields. The first field (*#Resolved*) is the number of resolved smells that existed in YAML files: each occurrence means the smell materialized at some point in the history, and then disappeared (i.e., the smell has been fixed). To gather these numbers, we followed the metric shown in Figure 2. The second field (*#Unresolved*) represents the number of YAML files that contain the corresponding smells in the most recent version of that YAML file (i.e., the smell is still persisting). Note that the maximum value of this field for a project is the number of CI-related YAML files in that project. Finally, the third field (average TTR) represents the average time in days for the corresponding smell to be fixed. To calculate the average TTR values, the total TTR of that smell (i.e., the sum of every resolved instance of the related smell's TTR) is divided by that smell's resolution count (i.e., *#Resolved*). Note that the average TTR field can be infinite or N/A. When a non-zero total TTR value is divided by zero, the average TTR has the value infinite. This happens when there are smells existing in the project and none of them are resolved. If the *#Unresolved* and *#Resolved* are both zero, then the average TTR is recorded as N/A. That is, if the smell never happened in a project, then the average TTR is not available for that smell.

As shown in Table 4, *Manual Execution* smell's average TTR has infinite values in five projects and a N/A value in one project. Only Dagger and Angular have proper average TTR values for the *Manual Execution* smell, with the values 0.03 days and 19 days, respectively. For the *Missing Environment Variables* smell, five projects' average TTR values are all infinite. The average TTR values of freeCodeCamp, Next.js, and Angular are 246.75, 840.5, and 4 days, respectively. The *Timed Build* smell is almost never fixed in the investigated projects except for freeCodeCamp, which has an average TTR value of 321 days. Finally, half of the projects never fixed the *Fake Success* smell which explains the infinite values for this smell.

Table 4: The Results of Four Smells for Each Project

	Manual Execution			Missing Environment Variables			Timed Build			Fake Success		
	#Resolved	#Unresolved	Average TTR	#Resolved	#Unresolved	Average TTR	#Resolved	#Unresolved	Average TTR	#Resolved	#Unresolved	Average TTR
freeCodeCamp	0	10	infinite	8	12	246.75	1	10	321	0	1	infinite
Bootstrap	0	10	infinite	0	5	infinite	0	2	infinite	0	0	N/A
React Native	0	0	N/A	0	7	infinite	0	1	infinite	0	1	infinite
Dagger	1	5	0.03	0	11	infinite	0	1	infinite	0	7	infinite
Next.js	0	4	infinite	2	8	840.5	0	3	infinite	0	0	N/A
Axios	0	3	infinite	0	7	infinite	0	2	infinite	0	0	N/A
TypeScript	0	9	infinite	0	16	infinite	0	7	infinite	0	1	infinite
Angular	1	3	19	1	10	4	0	4	infinite	0	0	N/A

The remaining four projects never experienced the smell, hence their average TTR values are N/A.

3.3.2 Smells Detected with Log Data. We applied the third method explained in Section 2.1 to produce a timeline showing the remaining three smells' occurrences. These are *Skipped Job*, *Long Build*, and *Retry Failure*. The results are given in Figures 3, 4, and 5. In Figures 4 and 5, the percentage of smell instances is provided. These values are calculated by dividing the number of observed smell instances by the total number of workflow runs. For example, freeCodeCamp has in total 144k workflow runs and 63k *Long Build* smell instances. Hence, the percentage of *Long Build* smell for freeCodeCamp is 43.74%. The average of the percentages of *Long Build* and *Retry Failure* smells are 19.03% and 2.26%, respectively. However, these percentage values cannot be calculated for the *Skipped Job* smell. Recall that *Skipped Job* smell is detected by checking the commit messages, and it is not possible to divide the number of smell instances by the number of workflow runs. Dividing the number of smell instances by the number of total commits after the adoption of CI would not work either because every YAML file is triggered by a different configuration (i.e., not every commit triggers every YAML file). For the correct percentage value, the number of *Skipped Job* instances should be divided by the number of commits that triggered the specific YAML file, which was not possible to detect.

Long Build smell is seen in all of the projects. However, in freeCodeCamp, Next.js, and TypeScript, there are intervals in which this smell existed almost continuously. Similarly, in React Native and Dagger, this smell is observed frequently but not as much as in the other projects. In contrast, in Bootstrap, Axios, and Angular, *Long Build* rarely occurs, forming an intermittent timeline. *Retry Failure* smell was pervasive in all projects ever since the projects started using GitHub Actions as their CI tool. Except for Axios, TypeScript, and Angular, this smell existed almost continuously. In these three projects, *Retry Failure* occurred rarely, forming an intermittent timeline. freeCodeCamp, React Native, Next.js, and Axios have encountered *Skipped Job* smell several times between 2018 and 2022, whereas Dagger, TypeScript, and Angular have never suffered from this smell. *Skipped Job* smell was a continuous problem for Bootstrap between 2014 and 2017, whereas Bootstrap dealt with that smell after 2017.

4 DISCUSSION

In this section, this study's results and implications are discussed.

4.1 Reflection on Findings

One of the main findings of this study is that many smells are never resolved or have high-resolution times. For example, this is the case for the *Fake Success* smell: All of the average TTR values that are not N/A are infinite. Similarly, in 7 out of 8 of the investigated projects, the *Timed Build* smell could never be resolved. Only freeCodeCamp's average TTR value for this smell is finite (321 days). Yet, the oddly high value of this average TTR may indicate that the smell was not deliberately fixed but rather resolved by chance. Both of the remaining two smells (*Manual Execution*, and *Missing Environment Variables*) have five infinite average TTR values (62.5% of the projects did not fix these smells). Some of the remaining average TTR values were surprisingly high (e.g. 840.5, and 246.75 days), which again may indicate that those smells were fixed by chance. Therefore, it can be said that these four smells are not frequently addressed by the investigated projects. This may indicate that developers may have been unaware of these smells, did not care about them, or these smells were hard to address. However, in order to draw concrete conclusions, we need further research.

According to the results of smells detected with log data, the frequency of the existence of *Long Build*, *Retry Failure*, and *Skipped Job* smells remained stable most of the time. Examining the timelines of the *Long Build* and *Retry Failure* smells in eight projects, we see that they are addressed less frequently when compared to the *Skipped Job* smell. *Skipped Job* is rarely encountered compared to the other two and is solved by the developers as soon as possible. This may have been because *Skipped Job* was easier to fix than the other two smells or because the developers were more aware of it or deemed it more important. However, this needs further research.

As it can be seen from Figure 5, *Retry Failure* is a commonly seen smell. This smell occurs when a pipeline is retried without addressing the cause of failure and succeeds on the subsequent attempt. A potential explanation for the prevalence of this smell might be the existence of flaky tests in the investigated projects, even if this needs further research. Flaky tests are nondeterministic and can yield either success or failure results inconsistently. As a result, when a developer faces a failure in the pipeline caused by a flaky test, they re-run the pipeline without changing anything on the codebase in the hope of a subsequently successful build, which results in a *Retry Failure*. Identifying and addressing flaky tests can help prevent this smell while improving the overall reliability of the software development process.

In Bootstrap, the occurrence of the *Skipped Job* smell suddenly stops after July 2017. To understand the possible reasons for this sudden disappearance, we inspected the history of both the contribution guidelines and documentation of this project, but could not

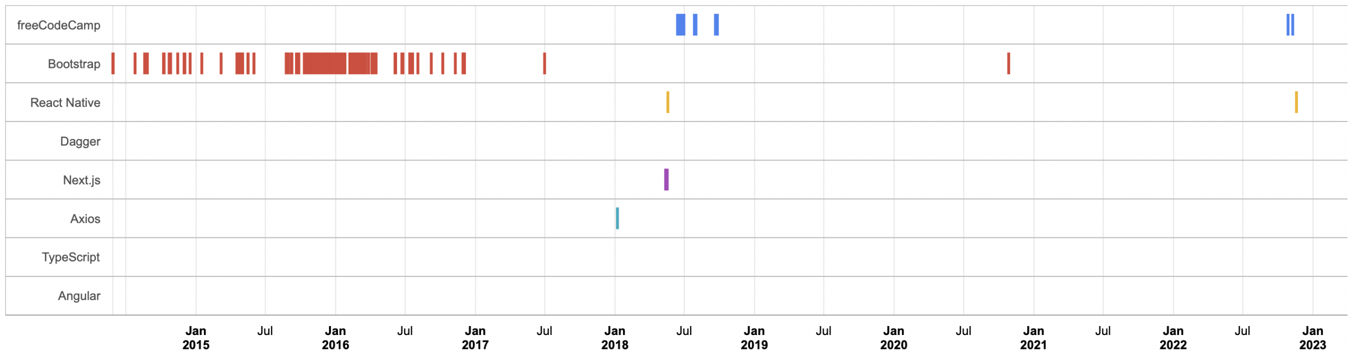


Figure 3: The Occurrences of Skipped Job Smell for Eight Projects

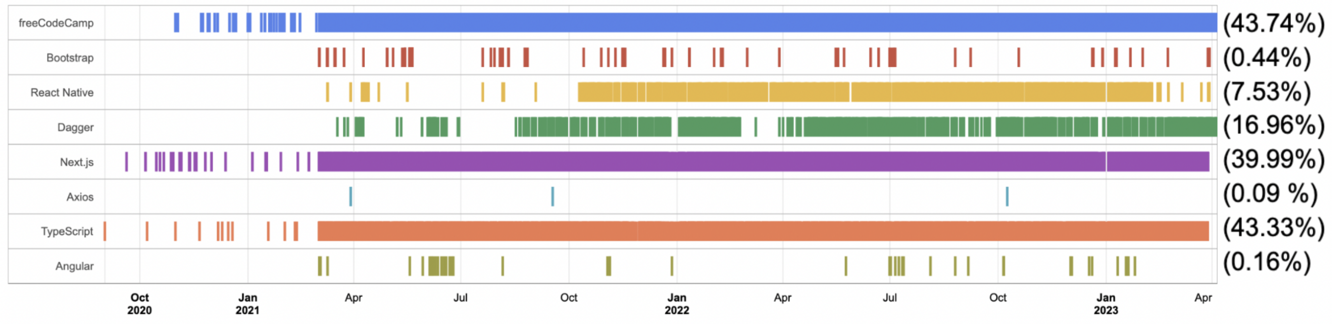


Figure 4: The Occurrences of Long Build Smell for Eight Projects

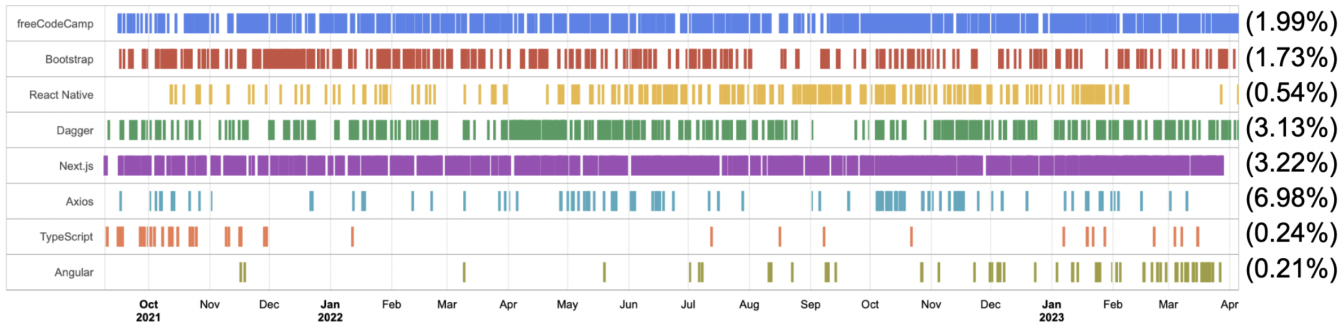


Figure 5: The Occurrences of Retry Failure Smell for Eight Projects

find a rule explaining this disappearance. Therefore, we opened a discussion item to ask about the possible reasons for this on GitHub Discussions[9]. Unfortunately, we have not received useful or definitive information about this issue in response to our post at the time of writing.

The *Long Build* smell was one of the least addressed smells. One possible explanation might be this: developers may not agree that pipeline builds that last longer than 10 minutes should be poor practice. Considering that pipeline build times depend on numerous factors such as the size of the project, the number of test cases, and the size of the CI pipeline, 10 minutes may not be a long time for a CI build. This may indicate that the 10-minute threshold in the *Long Build* smell is arbitrary, and should depend on the context.

[9]https://github.com/twbs/bootstrap/discussions/37693

4.2 Implications for Software Practitioners

This study has several implications for software practitioners. The investigated smells except for *Skipped Job* are not resolved by the software practitioners. There can be several reasons for this. Firstly, software practitioners may not be aware of these smells. That is, they are uninformed about what these smells are. In this case, software practitioners should be better informed about these smells.

Alternatively, software practitioners may know about these smells, but they are unaware that those smells exist in their projects. That is, they are overlooking those smells in their projects. In this case, software practitioners could be notified, e.g., by automated means of detection, so that they can be proactive about them. Lastly, software practitioners may be aware that these smells exist in their projects but do not mind them. In this case, it could be shown that these smells are impacting the project negatively. In future work, the impacts of these CI smells can be investigated and communicated to software practitioners.

If we could generalize from the small number of projects studied, developers are not following best CI practices, resulting in the prevalence of several CI smells in their projects. Automated mechanisms can help them be more aware of poor practices, or even patch the CI pipeline to remove the problematic patterns.

4.3 Implications for Researchers

Our study has several implications for researchers as well. The reason why investigated smells, with the rare exception of the *Skipped Job* smell, are not regularly addressed by software practitioners remains poorly understood. We speculated on the reasons, but an investigation of the root causes was outside the scope of this study. Future studies could pursue this line of work with a mix of qualitative and quantitative methods.

Another future area of study can be the development of tools that would prevent CI smells, similar to static analysis, but operating on the CI workflow specifications rather than code. Developers would then not need to be informed about or watchful of CI smells in advance, and instead be guided by the tools when their instances are detected in the pipelines before pipeline executions.

We mentioned that the existence of *Retry Failure* may be an indicator of flaky tests. Again, if automatic analysis of CI results can reveal this smell, correlate it with test runs, and alert the developers, developers may be prompted to fix flaky tests.

Finally, researchers can conduct a survey with experienced developers to learn about the perceived importance of different CI smells. Perhaps some categorized CI smells are superfluous, do not have important implications, or even serve a different purpose. This kind of research could help prioritize CI smells, delisting unhelpful ones, and guide automated smell detection tools to provide informed advice.

5 THREATS TO VALIDITY

In this section, we address potential threats to the validity of our study, outline the limitations within the scope of our research, and describe the mitigation strategies we employed. Additionally, we propose further mitigation strategies, when applicable, that can enhance the study design in future replications.

5.1 Internal Validity Threats

Context Dependence: Each organization has its own set of rules and best practices to follow. These may vary depending on the mission, domain, and the problem being solved. We have treated the selected projects uniformly, as if all CI smells need to be equally important to all projects. Future studies may take the context into account and treat the CI smells in proportion to their importance in the given context. Therefore, the practitioners working in this organization have to act by all these regulations. Hence, developers from different projects may fix CI smells on different levels.

Potentially Defective Analysis Scripts: Any bugs in the smell detection and data extraction scripts would cause the results to be incorrect. To mitigate this threat, we performed code reviews on the scripts and fixed potential bugs before using them.

Historical Events: Events internal to a project or organization unbeknown to the authors can affect the behavior of the developers. For example, a long-standing smell may be noticed by the team lead, and the developers may be warned about it, prompting them to suddenly fix the smell. Conversely, a rule may be changed in the contribution guidelines of an open-source project, causing the contributors to suddenly ignore a smell they cared about. A possible occurrence of this is the sudden disappearance of the *Skipped Job* smell in the Bootstrap project since 2017. To mitigate this threat, we posted a discussion on GitHub and asked the contributors of the project for reasons. If events that are known to change developer behavior are identified, their effects can be incorporated in the analysis, or they can help explain why certain smells are sometimes persistent and other times transient.

Erroneous Pre-Processing: We categorized the workflow files of each project as CI-related or not. This is because not all the workflow files were related to the CI process (e.g., there were automated flows that detected stale issues and automatically closed them). This categorization was manually performed by the authors without confirming with the developers and could have been wrong in certain cases. In future work, to eliminate this threat, the decision can be made by asking the developers or by using automated, more reliable, and more objective means.

Lack of Independent Validation of Smell Detection: We automated the smell detection and trusted our detection scripts. However, in some edge cases, we may have gotten false positives. Conversely, we may have missed real occurrences. For example, when our scripts detected that a commit message included a reserved keyword for skipping the CI pipeline, we automatically considered this instance as a smell. However, this behavior might have been intentional for legitimate reasons (e.g., to preserve CI resources), for example for a commit that only fixed typos in comments. Conversely, companies may have internal business reasons for ignoring a certain type of smell. For example, in addition to executing a build on each commit, a project may also use additional scheduled nightly builds, which would falsely trigger a smell instance if this rule is not known. Future work could try to obtain project- or organization-specific CI practices to be able to incorporate the underlying rules in the detection process to increase its accuracy.

Ambiguity in Tracking Smell Resolutions across File Changes: The smell detection strategies utilizing keywords in YAML files might result in false negatives in resolution counts. That is, the assumption that the same instance count indicates no remediation of smells may not hold true in all cases. For instance, in the *Fake Success* smell, where the `continue-on-error` keyword can appear in multiple locations within a YAML file, it is possible that this keyword was removed from one place and added elsewhere between changes, but the approach assumes no resolution. To mitigate this threat, we conducted further investigation by referring to the GitHub Actions documentation. We discovered that the ambiguity might only be relevant to the *Fake Success* smell. For other smells detected through keywords in YAML files, the possibility of occurrence due to the same keywords being in different locations within the file is ruled out. In these cases, the keywords responsible for these smells can only exist in one specific location within the YAML file. This clarification provides a more nuanced understanding of the potential limitations associated with tracking smells and reinforces the validity of the results for the majority of the investigated smells.

5.2 Construct Validity Threats

GitHub REST API documentation lacks explanations in terms of query parameters in the endpoints and keywords in the response body. Some keywords in the response body returned from the GitHub REST API were not self-explanatory. We opened a test repository and conducted some experiments to find out the purpose of the keywords we could not infer. This issue might affect construct validity. Before any replication, posts on public bulletin boards may try to clarify the mysterious GitHub API behavior.

5.3 External Validity Threats

In our study, we used a small convenience sample, limited by multiple criteria and research resources. Thus, the results may not generalize to every project. Our sample was not diverse. To overcome this threat, in the future, the number of investigated projects can be increased and selected from different domains.

In our study, all of the projects were open-source projects, and thus our findings are restricted to open-source development. The behavior of developers who work on proprietary or closed-source projects may be different than the developers who contribute to open-source projects.

In our study, we only examined poor practices with GitHub Actions as the CI tool. We have not included other popular CI tools such as Travis CI, CircleCI, Jenkins, and GitLab. The choice of the CI tool may have biased the results. For example, the use of GitHub Actions may increase the prevalence of certain smells. Duvall [5] stated that when a pipeline is failed, the entire team should be notified since fixing the buggy commit should be a top priority; and if the entire team is not notified, then this is also considered as a smell. In GitHub Actions, failed workflows are only notified to the developer who is responsible for that workflow run. There may be other unknown idiosyncrasies of the chosen CI that mask or amplify certain types of smells. In future work, projects that use other CI tools can be included in the sample, however, this requires data extraction and detection techniques and scripts to be customized to each tool.

In our study, we investigated projects which satisfied certain size, importance, and popularity-related criteria (e.g., projects should have at least 5,000 stars). Hence, our findings are biased toward larger, popular projects. The behavior of developers who work on smaller, less popular projects may very well be different. In the future, smaller and less popular projects can be included in the sample.

Finally, the number of smells we investigated in our study was limited to seven, which prevents this study from being generalized to all kinds of CI smells.

6 RELATED WORK

Previous researchers have investigated CI smells, which are poor practices in CI pipelines. Duvall [5] classified CI patterns and corresponding anti-patterns, with 50 smells grouped into ten categories. Zampetti et al. [19] empirically classified poor practices in CI, including those shared with Duvall [5]. Zampetti et al. [19] characterized 79 smells grouped into seven categories while investigating the perceived importance of the bad smells through a survey with developers. Note that the terms anti-pattern and smell will be used interchangeably. Our investigated CI smells are selected among the smells Duvall[5] and Zampetti[19] introduced.

Vasallo [15] proposed a linter for automatically identifying four different smells in pipeline configuration files by building on other studies. Vassalo et al. [14] built a reporting tool detecting four different anti-patterns by utilizing log data and repository data. Zhang et al. [20] proposed an automated tool detecting and repairing CI smells affecting the build performance. This tool resulted in a build performance improvement of 12.4% on average. These works informed us about the possible ways of automatically detecting CI smells. However, we investigated a different subset of CI smells than they did, and unlike them, we automatically detected CI smells on GitHub Actions.

Santos et al. [8] evaluated the impact of five CI sub-practices on the productivity and quality of open-source software projects. They revealed some correlations among sub-practices, productivity, and quality. Moreover, their analysis showed that projects attaching the most importance to CI sub-practices faced fewer problems related to CI. Zhao et al. [21] evaluated the impact of introducing CI in projects on other development practices. They used the number of merge commits, changed lines of code in a commit, the number of closed issues, the number of automated tests, and the number of closed pull requests as metrics to measure how the introduction of CI affects the software projects, all of which were in fact impacted after the introduction of CI. Cassee et al. [1] studied the impact of CI on socio-technical aspects of the software development process. This study's aim was to focus on code reviews. Their analysis revealed that CI impacts the complexity of pull requests by reducing review comments.

Vassallo [13] considered the barriers while adopting CI principles. This study provided developers with ways to instill a CI culture and improve the CI process.

Decan et al. [3] studied the use of GitHub Actions by inspecting 68K repositories on GitHub. They analyzed the types of workflows to be automated and the frequency of actions to be used in the workflows. This study showed that most of the workflows are used for development purposes and half of all steps in jobs are constructed by reusable actions provided by GitHub. This study enabled us to have a better understanding of the scope of GitHub Actions and helped us to decide whether we could investigate projects that use GitHub Actions or not.

Zampetti et al. [18] determined ways to restructure CI pipelines and track their evolution. This study categorized the actions to restructure CI pipelines. Some of these categories consist of actions to prevent and remove CI smells. Although the automatic detection methods of these actions are not investigated, it is a guide for us to decide what can be the possible metrics to investigate how much developers fix those smells.

To the best of our knowledge, our study is the first in terms of investigating the extent of resolving CI smells by practitioners. Yet, to what extent code smells are perceived as important was investigated before by Taibi et al. [12] and Aiko et al. These works played an important role in the formation of our research question. [17]. Taibi et al. [12] explored the perceived criticality of code smells among highly experienced coders. The investigation included presenting descriptions of code smells, infected code parts, and assessing the ability to recognize code smells. The findings revealed that, whereas developers regard code smells as hazardous in principle, they might not consider them to be significant in reality. Aiko et al. [17] also conducted a survey to investigate the knowledge about and interest in code smells, as well as their perceived criticality among professional software developers. They found that a significant proportion of respondents did not know about code smells, and those who were not concerned about code smells also lacked knowledge about them.

7 CONCLUSION

In this study, we used two methods to investigate to what extent developers fix CI smells. The first method uses the TTR of smells and the other uses the frequency of smell occurrences. We selected seven smells from a larger set of CI smells proposed in the literature. We conducted a quantitative analysis by first mining all versions of CI pipeline artifacts from selected open-source repositories using GitHub Actions. These artifacts included the workflow runs and commits. The selected projects were freeCodeCamp, Bootstrap, React Native, Dagger, Next.js, Axios, TypeScript, and Angular. We found that the developers do not tend to fix most types of smells, including *Manual Execution*, *Missing Environment Variables*, *Timed Build*, and *Fake Success*. We also found that certain types of smells are resolved more consistently than others, for example, the *Skipped Job* smell is resolved relatively more often than *Long Build* and *Retry Failure* smells.

For future work, we are planning to conduct a survey with experienced developers to ask them about their perceived importance of CI smells. This would help us triangulate our findings. We are also planning to expand our study's scope by increasing the number of open-source projects, including closed-source software projects and more CI tools in the sample, as well as expanding the set of CI smells with additional types. Moreover, we are planning to examine CI smells from an impact perspective: how do CI smells affect product quality, productivity, and the other software practices used? Finally, our ultimate goal is to build automated tools that prevent and automatically patch CI smells in a contextual way, focusing on the situations where they have the least desirable impacts in the context of the project and the organization.

REFERENCES

[1] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. 2020. The Silent Helper: The Impact of Continuous Integration on Code Reviews. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 423–434. https://ieeexplore.ieee.org/document/9054818

[2] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational And Psychological Measurement* 20 (4 1960), 37–46. https://doi.org/10.1177/001316446002000104

[3] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *Proceedings of the IEEE 38th International Conference on Software Maintenance and Evolution*. IEEE.

[4] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving software quality and reducing risk.* Addison-Wesley Professional.

[5] Paul M. Duvall. 2011. Continuous Delivery: Patterns and Antipatterns in the Software Lifecycle. https://dzone.com/refcardz/continuous-delivery-patterns.

[6] Martin Fowler. 2006. Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html.

[7] Vittoria Nardone, Biruk Asmare Muse, Mouna Abidi, Foutse Khomh, and Massimiliano Di Penta. 2022. Video Game Bad Smells: What They Are and How Developers Perceive Them. *ACM Transactions on Software Engineering and Methodology* 1 (9 2022), 1–34. https://dl.acm.org/doi/10.1145/3563214

[8] Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. 2022. Investigating the Impact of Continuous Integration Practices on the Productivity and Quality of Open-Source Projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Association for Computing Machinery, 137–147. https://dl.acm.org/doi/abs/10.1145/3544902.3546244

[9] Alek Sharma. 2018. A Brief History of DevOps, Part III: Automated Testing and Continuous Integration. https://circleci.com/blog/a-brief-history-of-devops-part-iii-automated-testing-and-continuous-integration/.

[10] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating Severity Thresholds for Test Smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, 311–321. https://doi.org/10.1145/3379597.3387453

[11] Daniel Stahla and Jan Boschb. 2014. Modeling Continuous Integration practice differences in industry software development. *Journal of Systems and Software* 87 (1 2014), 48–59. https://doi.org/10.1016/j.jss.2013.08.032

[12] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92 (9 2017), 223–235. https://www.sciencedirect.com/science/article/pii/S0950584916304128

[13] Carmine Vassallo. 2019. Enabling Continuous Improvement of a Continuous Integration Process. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1246–1249. https://ieeexplore.ieee.org/document/8952505

[14] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. 2019. Automated Reporting of Anti-Patterns and Decay in Continuous Integration. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 105–115. https://ieeexplore.ieee.org/document/8811921

[15] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. 2020. Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 327–337. https://dl.acm.org/doi/abs/10.1145/3368089.3409709

[16] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, 1–10. https://dl.acm.org/doi/abs/10.1145/2601248.2601268

[17] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, 242–251. https://ieeexplore.ieee.org/document/6671299

[18] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 471–482. https://ieeexplore.ieee.org/document/9609201

[19] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25 (1 2020), 1095–1135. https://link.springer.com/article/10.1007/s10664-019-09785-8

[20] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. 2022. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[21] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 60–71. https://ieeexplore.ieee.org/document/8115619

Received 2023-07-07; accepted 2023-07-28