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

Carmine Vassallo
University of Zurich
Zurich, Switzerland
vassallo@ifi.uzh.ch

Sebastian Proksch
Delft University of Technology
Delft, The Netherlands
s.proksch@tudelft.nl

Anna Jancso
University of Zurich
Zurich, Switzerland
anna.jancso@uzh.ch

Harald C. Gall
University of Zurich
Zurich, Switzerland
gall@ifi.uzh.ch

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

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.

CCS CONCEPTS

• **Software and its engineering → Agile software development**; *Automated static analysis*; Software libraries and repositories; Software testing and debugging.

KEYWORDS

Continuous Delivery, Continuous Integration, DevOps, Anti-pattern, Configuration, Linter.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

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 [19], increased developer productivity [43], and fast release cycles [5]. 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 [29]. Several catalogs of CI/CD best practices exist [5, 20, 28, 37]. However, while their adoption has been advocated in research papers, white papers, and books, developers have difficulties to apply them in practice [18], deviating from principles and generating anti-patterns [44].

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 [44] 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 GitLab [11] can validate their configuration files using online linters [12]. 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., [9] achieved a high user acceptance when pointing out the misuse of four different configuration options, like executing commands in the wrong build step. Rahman et al. [36] focused on security-related issues and Sharma et al. [38] 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 avoid violations of accepted CD principles when configuring CD pipelines. 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 [20, 28]. 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 a replication package [45].

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:

RQ_1 Are the CD Smells Detected by CD-Linter Relevant to Developers?
RQ_2 How Accurate Is CD-Linter?
RQ_3 How Frequent Are the Investigated CD Smells in Practice?

Figure 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 4. Inspired by existing literature in this area, we started by selecting four CD smells that affect the definition of CD pipelines (1) (Section 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 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 opening 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 5.1, we will answer RQ_1 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 RQ_2 (9), which will be discussed in Section 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 5.3.

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 [7]. By supporting GitLab, CD-Linter targets industrial and open-source projects alike.

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 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 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

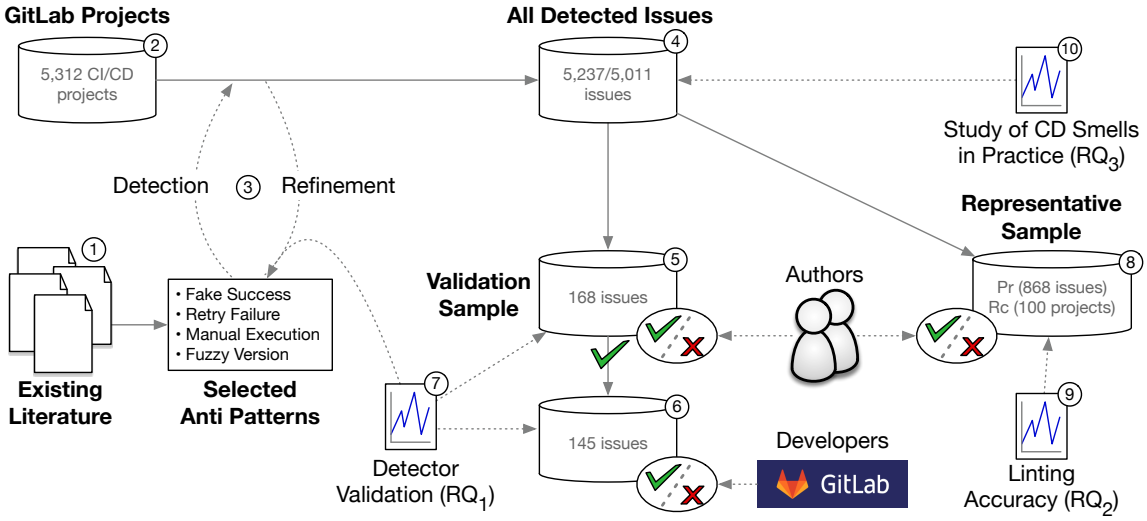

Figure 1: Methodology overview

```
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 2: Example excerpt of GitLab configuration

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.

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 [28] 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 [20], 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 [44]. 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 [45] 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 [21, 26], 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 codebase 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 manually 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

```
<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 3: Example of a *pom.xml*

major version is allowed; or (iv) *Any Upper Version*: every equal or higher version can be used.

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 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, such as the dependency resolution and the automated download from a centralized repository. Figure 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.

3.4 Detection of the CD Smells

Having access to the parsed information in the meta-models, we can proceed to implement various strategies 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 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 [14, 16] that contain `allow_failure` set to `true` for the default job used in 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 2). 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 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 [13]. To build a comprehensive list of deploy stage names, we extracted the stage names from a random project sample of our dataset (see Section 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 [15, 17]. Furthermore, we excluded cases where the `action` parameter of `environment` is set to `stop`, which is a manual way to shut down an environment used in the build.

Table 1: Fuzzy Version syntax in Python and Maven

Fuzzy Version Type	Python	Maven
Correct	`==1.1.8`	`1.1.8`
Missing Version	*empty*	*empty*
Only Major Version	1	1
Any Minor Version	`1.*`	*n/a*
Any Upper Version	`>=1.1.8`	`[1.1.8,)`

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 1 shows a comparison of the version syntax.

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) an *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 1), we realized how some libraries, such as Spring Boot, self-manage dependency versioning [41]. 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 developers' feedback (RQ_1), 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 advisable. To this end, we compiled a list of tools used for Python and Java projects. These include, for example, pipenv [31], pytest [34], pylint [33], and pip [30] for Python, and JUnit [22], FindBugs [6], CheckStyle [1], and PMD [32] for Java (the complete list is in our replication package [45]).

4 EMPIRICAL STUDY DESIGN

The *goal* of this study is to evaluate CD-Linter and determine whether it can 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 2.

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 diverse CD adoption history.

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 4.1. Then, we identified a random set of 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, 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 5.2). Examples of false positives, which received a negative assessment, are manually-triggered deployment jobs that were erroneously reported as Manual Execution incidents. 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 [39] 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 was removed or reintroduced in the observation period. In some cases, the smell was fixed despite a negative reaction or without any reaction to the issue whatsoever.

Based on the developers' reactions, 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 a 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, has a majority of upvotes, or a positive feedback.
Fixed The smell reported in the issue has been removed.

To address RQ_1, 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 RQ_1 directly improved CD-Linter (see Section 3.4).

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. As result, we obtained a sample that consists of 868 issues and achieves an error margin of ±3% (setting a confidence level of 95% and a percentage of 50%). Then, similarly to what was done in RQ_1, each issue was independently validated by two authors. After each annotator concluded the tagging, we measured the Cohen's kappa inter-rater agreement (k) [3]. 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 RQ_2, 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. [9]), making sure that those projects were not the same used to calculate the precision.

4.4 Measurement of CD Smell Occurrences

To address RQ_3, we run CD-Linter on the latest snapshot of the 5,312 projects described in Section 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.

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 4.

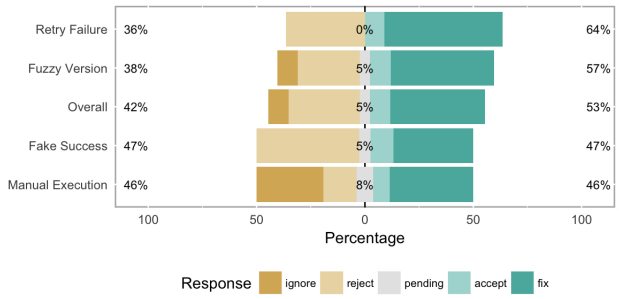

Figure 4: Reactions to the opened issues (over 6 months)

5.1 Are the CD Smells Detected by CD-Linter Relevant to Developers?

During the observation period of 6 months, 64 projects reacted to our issues (response rate of 74%), Figure 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 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 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 [5], 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 in an automated reporting tool. While this can be a threat to the study, it does not constitute a problem in a usage scenario where developers use the tool themselves.

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.

Table 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 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

RQ_1 Summary: 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.

5.2 How Accurate Is CD-Linter?

Table 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 different Fuzzy Version subtypes, Table 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 not fully specified (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 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 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. 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 RQ_1, 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 that 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 [2] (rule-based linter), SPHINX [40] (documentation generation), and WHEEL [35] (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 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.

RQ_2 Summary: 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 the current limitations of the tooling and can be addressed in the future.

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 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 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 4: CD smells in projects and owners

(# is "number of" and % is "percentage of" the analyzable instances)

		Projects		Owners	
Smell	**Occurrences**	#	%	#	%
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

Table 5: CD smells across different `.gitlab-ci.yml` sizes

(# stays for "number of" and % is the percentage respect to the total)

	.gitlab-ci.yml Size					
	Small		Medium		Long	
Smell	#	%	#	%	#	%
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 [20]. 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 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 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

Table 6: Break-down of Fuzzy Version smell

(YAML is `.gitlab-ci.yml`, POM is `pom.xml`, REQ is `requirements.txt`)

	File Type			
Category	YAML	POM	REQ	**Total**
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

Success and Manual Execution problems, and 1% of Retry Failure, 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 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). 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 5).

RQ_3 Summary: The most frequent CD smell is the Fuzzy Version (54.6% of the instances). Overall, CD smells affect 13% of the analyzed projects and 14% of the owners, mainly occurring in long configuration files.

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 RQ_2 mitigates this threat. In addition, the results of RQ_2, as well as the feedback provided by developers (RQ_1) 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 RQ_2 (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 (RQ_1), having two coders limited the subjectiveness of the results. The reactions we got in RQ_1 and the results of RQ_3 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 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 number of repositories are increasing. As explained in Section 4.1, GitLab gives the advantage of analyzing projects using the same CD infrastructure. Besides considering a sample (though relatively large) of projects, our evaluation is limited to GitLab configuration files, Maven builds, and Python dependencies. However, the detection principles explained in Section 3 can be applied to other pieces of technology and 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.

7 DISCUSSION

The empirical evaluation of CD-Linter, especially the developers' feedback collected in RQ_1, 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 their projects. 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 It Must Be Carefully Dosed. One problem we encountered in our empirical evaluation is that some developers are irritated by (and tend to discard) automatically-posted issues. While we tried to elaborate on the opened issues that these were the result of a manual review process, some developers still considered our issues a sort of spam, even when the suggestion was meaningful. Related research shows promising results when bots are used to aid software engineers [24, 25], 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, low precision of static analysis tools has already been reported as an adoption barrier multiple times [4, 23, 48]. 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 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 RQ_1 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.

8 RELATED WORK

This section describes related work about bad practices and their identification in CI/CD and infrastructure-as-code scripts.

8.1 Bad Practices in CI/CD

In their landmark books about CI [5] and CD [20], 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 [29]. Zampetti et al. [49] empirically characterized CI bad practices, finding commonalities but also differences with the ones advocated by Duvall [29]. Anti-patterns also occur because developers face several barriers when adopting CI/CD [18]. 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.

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. [9] 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. [44]. 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 [47] 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 [27, 42]. Another tool [46] improves the understandability of build failures through log summarization. Despite those approaches, developers still allow failures [8, 10]. 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. [38] 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. [36] 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.

9 SUMMARY

Previous work has introduced generic [44] or specialized [9, 36] linters that can help developers to improve their CD configurations. In contrast to previous work on CI smells that relies on historical information [44], 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 [48], 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 the developers' feedback that we have received for RQ_1 illustrated the usefulness of CD-Linter and 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.

10 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).

REFERENCES

[1] Checkstyle Team. 2020. Checkstyle. Retrieved September 10, 2020 from http://checkstyle.sourceforge.net
[2] Coala Team. 2020. Coala - Linting and fixing for all languages. Retrieved September 10, 2020 from https://coala.io/
[3] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.
[4] Cesar Couto, João Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. 2011. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal* 21 (2011), 241–257.
[5] P.M. Duvall, S. Matyas, and A. Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.
[6] FindBugs Team. 2020. FindBugs. Retrieved September 10, 2020 from http://findbugs.sourceforge.net/
[7] Forrester Team. 2019. The 2019 Forrester Wave Report. Retrieved September 10, 2020 from https://about.gitlab.com/analysts/forrester-cloudci19/
[8] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In *ASE*. ACM, 87–97.
[9] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Trans. Software Eng.* 46, 1 (2020), 33–50.
[10] T. A. Ghaleb, D. Alencar da Costa, Y. Zou, and A. E. Hassan. 2019. Studying the Impact of Noises in Build Breakage Data. *IEEE Transactions on Software Engineering* (2019), 1–1.
[11] GitLab Team. 2020. GitLab. Retrieved September 10, 2020 from https://about.gitlab.com
[12] GitLab Team. 2020. GitLab-CI Linter. Retrieved September 10, 2020 from https://docs.gitlab.com/ce/ci/yaml/README.html#validate-the-gitlab-ciyml
[13] GitLab Team. 2020. GitLab CI/CD Pipeline Configuration Reference. Retrieved September 10, 2020 from https://docs.gitlab.com/ee/ci/yaml/
[14] GitLab Team. 2020. GitLab DAST Template. Retrieved September 10, 2020 from https://gitlab.com/gitlab-org/gitlab-ee/blob/master/lib/gitlab/ci/templates/Security/DAST.gitlab-ci.yml
[15] GitLab Team. 2020. GitLab Review Apps. Retrieved September 10, 2020 from https://docs.gitlab.com/ee/ci/review_apps/
[16] GitLab Team. 2020. GitLab SAST Template. Retrieved September 10, 2020 from https://gitlab.com/gitlab-org/gitlab-ee/blob/master/lib/gitlab/ci/templates/Security/SAST.gitlab-ci.yml
[17] GitLab Team. 2020. GitLab Triage Template. Retrieved September 10, 2020 from https://gitlab.com/gitlab-org/gitlab-triage/blob/master/.gitlab-ci.yml
[18] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *ESEC/SIGSOFT FSE*. ACM, 197–207.
[19] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*. ACM, 426–437.
[20] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional.
[21] John Micco. 2016. Flaky tests at Google and how we mitigate them. Retrieved September 10, 2020 from https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[22] JUnit Team. 2020. JUnit. Retrieved September 10, 2020 from https://junit.org/junit5/
[23] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *ESEC/SIGSOFT FSE*. ACM, 45–54.
[24] Carlene Lebeuf, Margaret-Anne D. Storey, and Alexey Zagalsky. 2018. Software Bots. *IEEE Software* 35, 1 (2018), 18–23.
[25] Carlene Lebeuf, Alexey Zagalsky, Matthieu Foucault, and Margaret-Anne D. Storey. 2019. Defining and classifying software bots: a faceted taxonomy. In *BotSE@ICSE*. IEEE / ACM, 1–6.
[26] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *SIGSOFT FSE*. ACM, 643–653.
[27] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *SANER*. IEEE Computer Society, 106–117.
[28] Paul M. Duvall. 2010. Continuous Integration. Patterns and Antipatterns. Retrieved September 10, 2020 from https://dzone.com/refcardz/continuous-integration?chapter=1
[29] Paul M. Duvall. 2011. Continuous Delivery: Patterns and Antipatterns in the Software Life Cycle. Retrieved September 10, 2020 from https://dzone.com/refcardz/continuous-delivery-patterns
[30] Pip. 2020. Pip. Retrieved September 10, 2020 from https://pypi.org/project/pip/
[31] Pip Team. 2020. Pipenv: Python Development Workflow for Humans. Retrieved September 10, 2020 from https://docs.pipenv.org/
[32] PMD Team. 2020. PMD. Retrieved September 10, 2020 from https://pmd.github.io/
[33] Pylint Team. 2020. Pylint. Retrieved September 10, 2020 from https://www.pylint.org/
[34] Pytest Team. 2020. Pytest. Retrieved September 10, 2020 from http://pytest.org/
[35] Python Wheel Team. 2020. Python Wheel. Retrieved September 10, 2020 from https://pypi.org/project/wheel/
[36] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175.
[37] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie A. Williams, Kent L. Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *ICSE (Companion Volume)*. ACM, 21–30.
[38] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *MSR*. ACM, 189–200.
[39] D. Spencer and J.J. Garrett. 2009. Card Sorting: Designing Usable Categories. (2009).
[40] Sphinx Team. 2020. Spinx Python Documentation Generator. Retrieved September 10, 2020 from http://www.sphinx-doc.org/
[41] Spring Boot Team. 2020. Dependency Management in Spring Boot. Retrieved September 10, 2020 from https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-dependency-management
[42] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 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* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. ACM, 10.
[43] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *ESEC/SIGSOFT FSE*. ACM, 805–816.
[44] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *ICSE*. IEEE / ACM, 105–115.
[45] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. 2020. 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.
[46] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2020. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering* 25, 3 (2020), 2218–2257.
[47] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *ICSME*. IEEE Computer Society, 183–193.
[48] Fadi Wedyan, Dalal Alrmuny, and James M. Bieman. 2009. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction. In *ICST*. IEEE Computer Society, 141–150.
[49] 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, 2 (2020), 1095–1135.