Empirical Software Engineering (EMSE) manuscript No.
(will be inserted by the editor)

An Empirical Characterization of Bad Practices in Continuous Integration

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

This is a pre-print of an article published in Empirical Software Engineering. The final authenticated version will be available online at: https://doi.org/10.1007/s10664-019-09785-8.

Abstract Continuous Integration (CI) has been claimed to introduce several benefits in software development, including high software quality and reliability. However, recent work pointed out challenges, barriers and bad practices characterizing its adoption. This paper empirically investigates what are the bad practices experienced by developers applying CI. The investigation has been conducted by leveraging semi-structured interviews of 13 experts and mining more than 2,300 Stack Overflow posts. As a result, we compiled a catalog of 79 CI bad smells belonging to 7 categories related to different dimensions of a CI pipeline management and process. We have also investigated the perceived importance of the identified bad smells through a survey involving 26 professional developers, and discussed how the results of our study relate to existing knowledge about CI bad practices. Whilst some results, such as the poor usage of branches, confirm existing literature, the study also highlights uncovered bad practices, *e.g.,* related to static analysis tools or the abuse of shell scripts, and contradict knowledge from existing literature, *e.g.,* about avoiding nightly builds. We discuss the implications of our catalog of CI bad smells for (i) practitioners, *e.g.,* favor specific, portable tools over hacking, and do not ignore nor hide build failures, (ii) educators, *e.g.,* teach CI culture, not just technology, and teach CI by providing examples of what not to do,

Fiorella Zampetti, Gerardo Canfora, Massimiliano Di Penta
University of Sannio, Via Traiano, 9, Benevento, Italy
E-mail: fiorella.zampetti@unisannio.it, canfora@unisannio.it, dipenta@unisannio.it

Carmine Vassallo, Harald C. Gall
University of Zurich, Binzmuehlestrasse 14, Zurich, Switzerland
E-mail: vassallo@ifi.uzh.ch, gall@ifi.uzh.ch

Sebastiano Panichella
Zurich University of Applied Sciences, Obere Kirchgasse 2, Winterthur, Switzerland
E-mail: panc@zhaw.ch

and (iii) researchers, *e.g.,* developing support for failure analysis, as well as automated CI bad smell detectors.

Keywords Continuous Integration · Empirical Study · Bad Practices · Survey · Interview

1 Introduction

Continuous Integration (CI) (Beck, 2000; Booch, 1991) entails an automated build process on dedicated server machines, with the main purpose of detecting integration errors as early as possible (Beller et al., 2017; Duvall et al., 2007; Ståhl and Bosch, 2014b). In certain circumstances, the next step is Continuous Delivery (CD), in which code changes are also released to production in short cycles, *i.e.,* daily or even hourly (Amazon, 2017; Humble and Farley, 2010).

Industrial organizations that moved to CI reported huge benefits, such as significant improvements in productivity, customer satisfaction, and the ability to release high-quality products through rapid iterations (Chen, 2017). The undisputed advantages of the CI process have motivated also many Open Source Software (OSS) contributors (Hilton et al., 2016; Ståhl and Bosch, 2014b) to adopt it, promoting CI as one of the most widely used software engineering practices (Chen, 2017; Hilton et al., 2016).

Despite its increasing adoption, the heavy use of automation in CI makes its introduction in established development contexts very challenging (Chen, 2017; Hilton et al., 2016). For this reason, in recent work researchers investigated the barriers (Hilton et al., 2017) and challenges (Chen, 2017) characterizing the migration to CI. They found that developers struggle with automating the build process as well as debugging build failures (Hilton et al., 2017).

Also, once CI is in place, it might be wrongly applied, thus, limiting its effectiveness. Previous work (Duvall et al., 2007; Duvall, 2010; Humble and Farley, 2010; Savor et al., 2016) highlights some bad practices in its exercise concerning commits frequency, management of built artifacts and overall build duration. In this context, Duvall (2011), by surveying related work in literature, created a catalog featuring 50 patterns (and their corresponding antipatterns) regarding several phases or relevant topics of the CI process.

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

This paper aims at empirically investigating what bad practices developers incur when using CI in their daily development activities. This is done through (i) semi-structured interviews with 13 practitioners directly involved in the management and use of CI pipelines of 6 medium/large companies, and (ii) manual analysis of over 2,300 Stack Overflow (SO) posts, all related to CI topics. By relying on such sources of information and using a card sorting approach (Spencer, 2009), we derived a catalog of 79 CI bad smells organized

into 7 categories spanning across the different dimensions of a CI pipeline management (*e.g.,* Repository, Build Process Organization, Quality Assurance, Delivery). Afterward, we have assessed the perceived importance of the 79 bad smells through a survey, which involved 26 developers working in 21 different companies.

To document the identified CI bad smells, we provide traceability (in our dataset) between the catalog and the SO posts and interviews' sentences. For example, the CI bad smell "Pipeline related resources (*e.g.,* configuration files, build script, test data) are not versioned" is originating from a SO post[1] where a user asked:

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

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

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

While a catalog of CI patterns and corresponding antipatterns exist (Duvall, 2011), this paper aims at empirically inferring and validating bad practices through a reproducible process, which has manifold advantages: (i) it allows us to investigate what is the current perception of CI bad practices (based on SO posts and interviews with experts), (ii) it allows others to replicate our process, and, possibly, to improve our catalog or even confute our results, and (iii) it allows us to perform an unbiased comparison between CI bad smells emerged from our study and what stated in Duvall's catalog. To investigate (i) the correspondence or contradiction between what is known in literature and what we observed in the reality, and (ii) cases in which developers follow a bad practice because they aim at pursuing trade-offs between conflicting goals, we have analyzed and discussed the relationships between the CI bad smells we identified, and the pattern/antipatterns of Duvall's catalog (Duvall, 2011). Finally, the study shows that different bad smells (including those matching Duvall's antipatterns) have a different degree of perceived importance.

The paper is further organized as follows. Section 2 provides the context of our study by discussing the related literature. Section 3 defines the study, its research questions, and details the study methodology. Section 4 reports and discusses the study results, while Section 5 discusses the threats to the study validity. The study implications are outlined in Section 6, while Section 7 concludes the paper.

[1] https://stackoverflow.com/questions/1351755

2 Related Work

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

- We use the term "bad smells" similarly to previous work to denote "symptoms of poor design or implementation choices" (Fowler et al., 1999b). In our case, "CI bad smells" are symptoms of poor choices in the application and enactment of CI principles.
- We use the term "bad practice" when we generically discuss the bad application of CI principles, without referring to a specific problem.
- Finally, when referring to the catalog by Duvall (2011), we use the terms "pattern"/"antipattern" to be consistent with the terminology used there. However, in this paper, Duvall's antipatterns and our CI bad smells have the same meaning.

2.1 Studies on Continuous Integration and Delivery Practice.

Many researchers have studied the CI/CD practices adopted in industry and open source projects (Deshpande and Riehle, 2008; Hilton et al., 2016; Ståhl and Bosch, 2014a,b; Vasilescu et al., 2015). Hilton et al. (2016) conducted an extensive study on the usage of CI infrastructure and found that CI is currently very popular in OSS. Ståhl and Bosch (2014a) pointed out that in industry there is not a uniform adoption of CI. More specifically, they have identified the presence of different variation points in the CI term usage. Similarly, we involved different companies, varying in size and domain, to guarantee diversity of respondents and reliability of our results. Other researchers have focused the attention on the impact of CI adoption on both code quality and developers' productivity. Vasilescu et al. (2015) showed how CI practices improve developers productivity without negatively impacting the overall code quality. From a different perspective, Vassallo et al. (2016) investigated, by surveying developers of a large financial organization, the adoption of the CI/CD pipeline during development activities, confirming what known from existing literature (*e.g.,* the execution of automated tests to improve the quality of their product), or confuting them (*e.g.,* the usage of refactoring activities during normal development).

While the studies mentioned above investigated CI practice and served as inception for our work (Section 3.2.1), our perspective is different *i.e.,* identifying and categorizing specific CI problems into a catalog of CI bad smells.

Concerning CD practices, Chen (2017) analyzed four years' CD adoption in a multi-billion-euro company and identified a list of challenges related to the CD adoption. Also, Chen identified six strategies to overcome those challenges such as (i) selling CD as a painkiller, (ii) starting with easy but important applications and (iii) visual CD pipeline skeleton. Savor et al. (2016), by analyzing the adoption of Continuous Deployment in two industrial (Internet)

companies, found that the CD adoption does not negatively impact developer productivity, even when the project increases in terms of size and complexity.

2.2 Continuous Integration Bad Practices and Barriers.

Duvall et al. (2007) identified the risks that can be encountered when using CI, *e.g.,* lack of project visibility or the inability to create deployable software. Such risks highlighted the need for (i) a fully automated build process, (ii) a centralized dependencies management to reduce class-path and transitive dependencies' problems, (iii) running private builds and (iv) the existence of different target environments on which deploy candidate releases.

Hilton et al. (2017) investigated what barriers developers face when moving to CI, involving different and orthogonal dimensions namely: assurance, security, and flexibility. For instance, they found that developers do not have the same access to the environment as when they debug locally and their productivity decreases when dealing with blocking build failures.

Humble and Farley (2010), instead, set out the principles of software delivery and provided suggestions on how to construct a delivery pipeline (including the integration pipeline), by using proper tools, automating deployment and testing activities. Based on such principles, Olsson et al. (2012) explored the barriers companies face when moving towards CD. More specifically, they identified as barriers: (i) the complexity of environments (in particular of network environments) where software is deployed; (ii) the need for shortening the internal verification to ensure a timely delivery; and (iii) the need for addressing the lack of transparency caused by an incomplete overview of the current status of development projects.

Our work shares with previous work the challenges encountered when applying CI, but also CD. However, the observation perspective and the stage in which the problems are observed are different. Indeed, we do not look at problems encountered in the transition to CI or CD, while we infer CI bad smells when CI is already being applied. However, in some cases, the consequences of the bad smells we inferred share commonalities with the barriers affecting the transition (Olsson et al., 2012). As an example, we foresee some CI bad smells concerning (i) deployment of artifacts that have been generated in a local environment, and (ii) deployment without a previous verification in a representative, production-like environment. Also, our catalog features a bad smell named "Authentication data is hardcoded (in clear) under VCS", which considers the lack of an appropriate and secure authentication when deploying from a CI server. In summary, our findings indicate that, even when organizations have performed a transition towards CI/CD, some issues previously identified as barriers still arise, and some more can occur.

Duvall defined a comprehensive set of 50 patterns and related antipatterns regarding several phases or relevant topics in the CI/CD process (Duvall, 2011). In the catalog, it is possible to find bad habits concerning the versioned resources, the way developers use to trigger a new build, test scheduling poli-

cies, and the use of feature branches. Duvall also highlighted the need for a fully automated pipeline having a proper dependency management, a roll-back release strategy and pre-production environments where the release candidate should be tested. As we explained in the introduction, while it is not our intent to show whether our catalog is better or worse than the one by Duvall, we (i) define and apply an empirical methodology to derive CI bad smells, (ii) study the developers' perception of such bad smells, and (iii) finally, we discuss the differences between the two catalogs, as well as cases in which a bad smell present in both catalogs was considered as not particularly relevant by the study respondents.

Vassallo et al. (2019a) proposed CI-Odor, an approach that detects the presence of four CI antipatterns (slow build, broken master, skip failed tests, and late merging) inspired by Duvall (2011) catalog. While they focused on the automated detection of on some Duvall's antipatterns, our aim is to investigate what bad practices (beyond those by Duvall) are relevant for practitioners, also to develop further detection strategies.

The phenomenon of slow builds was also investigated by Ghaleb et al. (2019), finding that long builds (*e.g.,* exceeding the 10 minutes rule-of-thumb (Duvall et al., 2007)) do not necessarily depend on the project size/complexity, but may also be related to build configuration issues, such as multiple (and failed) build attempts. To this extent, our catalog includes various kinds of build configuration issues that can result in such a side effect. Also, Abdalkareem et al. (2019) propose to optimize the build process by determining, through a tool named CI-Skipper, which commits can be skipped. While this is not necessarily related to the presence of CI bad smells, it may be a mechanism useful to solve problems related to slow builds.

The work by Gallaba and McIntosh (2018) investigates configuration smells for CI infrastructure scripts in Travis-CI (*i.e.,* `.travis.yml` files), and proposes Hansel and Gretel, two tools to identify and remove four types of antipatterns in Travis-CI configuration scripts. Such antipatterns are related to (i) redirecting scripts into interpreters, (ii) bypassing security checks, (iii) having unused properties in `.travis.yml` files, and (iv) having unrelated commands in build phases. Also in this case, our work is complementary to such specific detectors as it provides a comprehensive, empirically-derived catalog of CI bad smells along with developers' perception of such bad smells.

3 Empirical Study Definition and Planning

In the following, we define our study according to the Goal Question Metric (GQM) paradigm (Basili, 1992).

The goal of this study is to identify the bad smells developers incur when adopting CI and assess the perceived importance of such bad smells. The quality focus is the overall improvement of the CI process and its associated outcomes, *e.g.,* improving developers' productivity and software reliability. The perspective is of researchers interested, in the short term, to compile a catalog

of CI bad smells and use them for education and technology transfer purposes, and in the long term to develop monitoring systems aimed at automatically identifying CI bad smells, whenever this is possible. The context from which we have inferred the catalog of CI bad smells consists of six companies (where we interviewed 13 experts), and 2,322 discussions sampled from SO. To assess the perceived importance of the identified CI bad smells, we have surveyed 26 CI practitioners belonging to 21 companies, that are not involved in the previous phase of the study.

3.1 Research Questions

The study aims at addressing the following research questions:

- **RQ$_1$:** *What are the bad practices encountered by practitioners when adopting CI?* This research question addresses the main goal of the study, which is the empirical identification and categorization of CI bad practices. The output of this categorization is a catalog of CI bad smells, grouped into categories. As explained in the introduction, while a catalog of CI patterns and related antipatterns already exists (Duvall, 2011), our goal is to infer bad practices from pieces of evidence, *i.e.,* SO posts or semi-structured interviews.
- **RQ$_2$:** *How relevant are the identified CI bad smells for developers working in CI?* While in the previous research question we derive a catalog of CI bad smells by interviewing experts and analyzing SO discussions, it could be possible that different bad smells might have a different degree of perceived importance. By surveying developers, this research question aims at assessing the importance of the bad smells identified in **RQ$_1$**, and, therefore, at performing an external validation of the compiled catalog.
- **RQ$_3$:** *How our pieces of evidence confirm/contradict/complement the existing CI pattern/antipattern catalog by Duvall (2011)?* This research question aims at comparing our catalog of CI bad smells with those from literature. This is done by performing a mapping between our catalog and the one by Duvall (2011). It is important to remark that it is not our goal to determine which catalog is better. Instead, we want to determine the extent to which the antipatterns defined by Duvall (2011) are reflected by problems occurring in real practice, and whether there are problems not considered by Duvall (2011).

3.2 Study Methodology

Fig. 1 shows the overall methodology we followed to create and validate our catalog of CI bad smells. The methodology comprises a set of steps to create the catalog and further steps to validate it. Also, the figure reports which steps produce results for our study research questions. Table 1, instead, provides

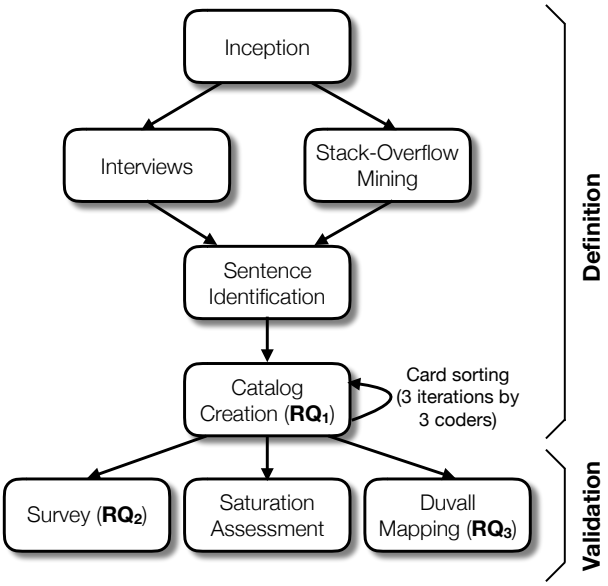


Fig. 1 Process for CI Bad Smell Catalog Creation and Validation.

Table 1 Summary of the study data.

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

essential information about data processed/collected during our interviews and SO mining.

3.2.1 Inception

As first step, we performed an inception phase to enrich our knowledge on the possible CI misuses and therefore be able to effectively conduct interviews and mine online discussions. To this aim, we relied on well-known books, together with the related online resources: the Duvall *et al.* book on CI (Duvall et al., 2007) and the book by Humble and Farley (2010). We intentionally did not use Duvall's catalog (Duvall, 2011) in this phase, because we wanted to proceed bottom-up from the problems experienced by developers adopting CI. This allows us to perform an unbiased comparison of our catalog with the one by Duvall in a subsequent phase of the study.

Table 2 Companies involved in the interviews.

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

In addition, we looked at previous research articles in the area: surveys about CI conducted in industrial context (Ståhl and Bosch, 2014b,a), as well as studies focusing on specific CI phases (Beller et al., 2017; Manuel Gerardo Orellana Cordero and Demeyer, 2017; Zampetti et al., 2017).

3.2.2 Semi-Structured Interviews

The goal of the semi-structured interviews was to understand and discuss with practitioners the problems they encountered when maintaining and using the whole CI pipeline in practice. We conducted interviews instead of a survey because (i) we expected that problems varied a lot depending on the context, and therefore it might have been worthwhile to deepen the discussion on specific CI areas; and (ii) interviews allowed us to discuss and capture specific situations the practitioners experienced doing CI activities. To conduct the interviews, we created an interview guide composed of the following sections:

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

The interviewed companies were identified based on the personal list of contacts of the authors. Thus, the interviewees were either our direct contacts, or we were forwarded to people specifically working on the CI pipeline. As shown at the top of Table 1, we identified in total 13 people from six different companies (in four cases we interviewed multiple people from the same company, working on rather different projects). The interviews were conducted either in person or using a video-conferencing system (Skype). In both cases, the audio of the interview was recorded. Each interview lasted between 30-45 minutes on average, depending on the participants' availability.

Table 3 Interviewed experts.

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

All the participants have more than eight years of experience in software development; all of them use Java; eight are also very familiar with JavaScript, and five are knowledgeable of Python. Table 2 summarizes the domain/size (number of employees) of each company, while Table 3 reports, for each interviewee, her role in the CI pipeline and also from when her company started to adopt the CI process. As shown in Table 2, the involved companies vary in terms of size and domain. More importantly, as shown in Table 3, the majority of our respondents declared to be actually in charge of configuring/maintaining the whole CI pipeline, while only two respondents are simply using the pipeline as is. Finally, each company started to use CI two or more years ago ("–" indicates that the respondent did not precisely know when). As an outcome, we obtained a set of transcribed sentences from the taped interviews, referring to possible CI bad practices to use in the subsequent phase to create the catalog. The sentences report either the current practice, (*e.g.,* "Our client dictates the quality gates. We have only to meet these gates"), a perceived bad practice (*e.g.,* "Hardcoded configurations are a CI smell") or, in some cases, both (*e.g.,* "Monitor the global technical debt ratio, without paying attention to a single developer activity. It's up to developers to decide when performing refactoring"). The complete set of collected sentences is available in our replication package (Zampetti et al., 2019).

3.2.3 Analysis of Stack Overflow Posts

As a first step, we needed to identify SO tags to use for retrieving candidate SO posts that could be relevant to our study. By scrutinizing the whole set of SO tags available[2], we identified four CI-related tags, namely *continuous-integration*, *jenkins*, *hudson*, and *travis-ci*, used to better contextualize the discussions (*i.e.,* issues/problems identification) to CI, since in some cases you could face the same problem in a different development practice. Then, we

[2] We have queried the SO database in May 2017

identified a total of 48 tags (a complete list is available in our online appendix (Zampetti et al., 2019)) that could relate to the specific activities and properties we were interested to investigate, such as Version Control Systems - VCS (*e.g.,branching-strategy, version-control*), build (*e.g.,batch-file, build-process*), testing (*e.g.,acceptance-testing, automated-tests*), performance (*e.g.,build-time*) and other related tags. In the end, we performed SO queries expecting at least one of the four CI-related tags, and one of the 48 specific tags (192 queries).

As a result, we downloaded 4,645 candidate, non-duplicated posts. We randomly divided such posts into two sets (ensuring proportions for each query), the first half used for inferring the catalog of CI bad smells, and the second half used for verifying the catalog saturation, as detailed in Section 3.2.6.

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

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

3.2.4 Identification of sentences reflecting symptoms of CI bad practices

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

a new short description in the list of possible ones, making it available for the upcoming annotations. As an example, the coders used the sentence "Test too long in the commit stage" to highlight the usage of a build process that does not adhere to the fast feedback practice Duvall (2011) or "Manual steps while triggering different stages in the pipeline" to point out the lack of full automation of a CI process.

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

3.2.5 Elicitation of the Catalog of CI bad smells

To address $\mathbf{RQ}_1$, and therefore provide a systematic classification of bad smells faced by developers when applying CI, we performed card-sorting (Spencer, 2009) starting from the symptoms identified in the previous phase. We relied on an online shared spreadsheet to perform the task.

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

1. We discarded bad smells related to problems in the development process that do not have a direct impact on CI. For instance, we excluded generic test smells (van Deursen et al., 2001) or code smells (Fowler et al., 1999a).
2. We discarded symptoms reflecting possible CI misuses that we recognized to be "bugs". As an example, we found some SO discussions in which developers discussed build failures due to the presence of bugs in third-party libraries included in the project. While the inclusion has a negative impact on the build process, this does not represent a CI misuse.
3. We merged related bad smells. For example, in some circumstances, a very similar bad practice was mentioned in two different CI activities, *e.g.,* arbitrarily skipping a failing static analysis check or a failing test case.

The process was iterated three times, until no further changes were applied to the catalog. In the end, the final version of the catalog, discussed in details in Section 4.1 (RQ_1), features a total of 79 bad smells grouped into 7 categories.

3.2.6 Catalog Validation on Unseen Stack Overflow Posts

To verify the catalog saturation, *i.e.,* its capability to cover bad smells not encountered in our manual analysis, we took the remaining set of 4,645-2,322=2,323 SO posts and extracted a statistically significant sample of 330 posts. Such a sample size was chosen to ensure a significance level of 95% and a margin of error of ±5%. Also, the 330 posts were sampled in proportion across the query tags. In other words, a stratified-random sampling was performed, where strata are represented by the set of posts returned by the

different queries (*i.e.,* tags). After that, two independent evaluators (two of the authors) selected the relevant posts and mapped them onto the 79 CI bad smells. Where the evaluators could not find a bad smell in the catalog matching the SO post, they added an annotation in the spreadsheet to be able to discuss these cases (8 cases in total). After the first round of independent classification, we found that the evaluators agreed in 76% of the cases on whether a post was related to CI bad smells or not. While this percentage seems high, it is still possible that they could have agreed by chance. Therefore, we computed the Cohen's k (Cohen, 1960), which resulted to be 0.46 (moderate). After that, a third author identified the inconsistent classifications and discussed with the evaluators the reasons why this occurred, *e.g.,* somebody classified as relevant SO posts that were seeking technical information (howto) about given pieces of technology. Such posts were not questioning an appropriate adoption practice, but, instead, seeking technical details, *e.g.,* installation or usage instructions. After that, the two evaluators reworked again on the inconsistent cases. After the re-coding, the Cohen's k increased to 0.79 (substantial agreement), and the agreement rate to 91%. Also, we computed the agreement rate in terms of the kind of smell each evaluator associated to the post. In this case, we used the Krippendorff's α reliability coefficient (Krippendorff, 1980) as the labeling was incomplete (*i.e.,* for a post an evaluator could have indicated a bad smell, and another none). We obtained a reliability coefficient $\alpha = 0.67$, which is considered an acceptable agreement.

In the end, 131 out of 330 posts were classified as related to CI bad smells by at least one of the evaluators, out of which only 1 was not included in the previous version of the catalog. More specifically, one SO post pointed out cases where the build fails as soon as the first test case fails. This implies having a great number of build failures without having a clear vision about the whole changes being implemented and pushed.

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

3.2.7 Evaluating the Catalog through a Survey

To address $\mathbf{RQ}_2$, we conducted a survey involving practitioners adopting CI in their organization. To ensure a good generalizability of the validation and to encourage participation, we adopted the snowball (Goodman, 1961) sampling. That is, we shared the survey link to some contact points, and encouraged them to indicate us further participants, or people in the company better suited to participate in the survey. We followed this strategy because, while we had personal knowledge with a relatively limited set of contacts, snowballing helped to reach the relevant people (*i.e.,* those involved in CI) and, in general, to favor participation. The online survey presented to the participants had:

1. An introduction explaining the meaning of our CI bad smells, as well as some basic terminology definitions for avoiding misunderstandings;

2. A demographic section similar to the one described in Section 3.2.2.
3. A set of 7 sections in which bad smells belonging to each category are evaluated.

We asked each respondent to evaluate the relevance of each bad smell over a 5-level Likert scale (Oppenheim, 1992) (strongly agree, weakly agree, borderline, weakly disagree, strongly disagree), and we also gave the option to answer "Don't know". At the end of each section, we had an optional free comment field where the respondent could provide additional insights. The questionnaire has been administered through Survey Hero[3]. The link has been sent to the people using an invitation email, in which we encouraged to spread the links to other CI experts. We kept the questionnaire open for four weeks. Nobody reported to have particular issues (*e.g.,* privacy issues) with the used survey administration tool.

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

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

3.2.8 Mapping onto Duvall's Antipatterns

To address $\mathbf{RQ}_3$, we analyzed the overlap of our set of bad smells with those proposed by Duvall (2011) to (i) investigate the exhaustiveness of our catalog; (ii) determine whether the empirically derived set of bad smells are not covered in Duvall's catalog, pointing out also more specialized cases of CI bad smells and/or outlining trade-offs. Note that we consider the last version of Duvall's catalog (Duvall, 2011) and, in Section 4.3, we explain why some antipatterns are out of the scope of this study.

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

After the first round of annotation, the two authors agreed in the mapping with a Krippendorff α=0.65, which is just below the minimum acceptability

[3] https://www.surveyhero.com

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

of α=0.66 defined in the literature (Krippendorff, 1980). Similarly to what was done in the previous case (validation), a third author analyzed the disagreement cases and discussed them with the two annotators. As an example of inconsistent annotations, for the "Repository" pattern in Duvall's catalog (Duvall, 2011) the first annotator used the "Pipeline related resources are not versioned" while the second one mapped it onto the "Missing Artifact's Repository" bad smell. With the help of a different author, and looking at the whole description of the pattern (and related antipattern) in Duvall's catalog, the annotators converged on the fact that the antipattern is clearly highlighting the needs for having all the resources required to execute the build process under version control to avoid unnecessarily build failures. After that, the two annotators performed the mapping of the inconsistent cases again. The new mapping yielded an agreement rate of 80% and a Krippendorff α=0.84.

4 Empirical Study Results

Table 4-Table 10 provide an overview of the 79 CI bad smells emerged from our empirical investigation. As explained in Section 3.2.5, we grouped them into 7 categories related to different dimensions of a CI pipeline management.

The third column (D) of the tables reports whether or not the bad smell maps onto at least one of Duvall's antipatterns (Duvall, 2011) (RQ_3, Section 4.3). The fourth and fifth column report results related to the perception of CI bad smells (RQ_2, Section 4.2) and, specifically, the number of respondents that evaluated that specific bad smell (Resp.), and the evaluation results in form of asymmetric stacked bar charts. Note that each bar chart also reports three percentages: (i) respondents providing a disagree/strongly disagree answer, (ii) respondents providing a neutral answer, and (iii) respondents providing an agree/strongly agree answer.

4.1 Overview of the CI Bad Smell Catalog

In the following, we provide a general overview of the 7 categories of CI bad smells, without necessarily enumerating and describing all bad smells belonging to each category. For more detailed information, the complete catalog is available in our online appendix (Zampetti et al., 2019).

Repository groups bad smells concerning a poor repository organization, and misuse of version control system (VCS) in the context of CI (see Table 4). Some smells (R1–R3) deal with problems related to the repository structure which may affect the modularity of CI solutions (*e.g.,* to build different modules of a software project separately). Moreover, a poor project decomposition into sub-modules (R1) might make parallel work on branches more difficult, but also prevent from having (some) faster builds limited to certain modules only. Then, there are bad smells about branch misuses (R4–R7), *e.g.,* wrong choice between the use of feature branches and feature toggles (R6) or the use

Table 4 Results of Bad Smells' Perception - Repository (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

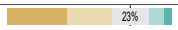
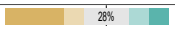
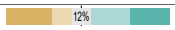
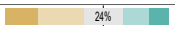
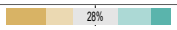
ID	CI Bad Smell	D	Resp.	Survey Results
R1	Project decomposition in the repository does not follow modularization principles	✗	26	30% 30% 39%
R2	Test cases are not organized in folders based on their purposes	✗	26	48% 30% 22%
R3	Local and remote workspace are not aligned	✗	25	30% 13% 57%
R4	Number of branches do not fit the project needs/characteristics	✓	26	41% 32% 27%
R5	A stable release branch is missing	✗	25	45% 9% 45%
R6	Feature branches are used instead of feature toggles	✓	24	30% 25% 45%
R7	Divergent Branches	✓	26	38% 14% 48%
R8	Generated artifacts are versioned, while they should not	✓	25	50% 32% 18%
R9	Blobs are unnecessarily checked-in at every build instead of being cached	✗	25	41% 36% 23%
R10	Pipeline related resources are not versioned	✓	25	18% 18% 64%

of an unbalanced number of branches that do not fit the project's characteristics (R4). Finally, some smells concern the poor choice of configuration items (R8–R10).

Infrastructure Choices groups bad smells related to a sub-optimal choice of hardware or software components while setting a CI pipeline (see Table 5). Hardware issues are mainly related to a poor allocation of the CI process across hardware nodes that could overload development machines or lose scalability (I1, I2). Software-related bad smells (I4–I7) concern poor tool choices and configuration, *e.g.,* use of inadequate plugins for certain tasks (I6), abuse of ad-hoc shell scripts (I7), as well as the use of external plugins with default configurations not suitable to the specific development scenario (I3). Indeed, each tool has to be configured according to the (i) developers' needs, (ii) final product requirements, and (iii) policies adopted by the organization.

Build Process Organization. This category, the one with the largest number of bad smells (29), features CI bad smells related to a poor configuration of the whole CI pipeline, as detailed in Table 6. Some of such bad smells are related to the CI environment's initialization. Specifically, inappropriate clean-up strategies (BP1) could have been chosen. This, on the one hand (aggressive clean-up), may unnecessarily slow-down the build, and, on the other hand (lack of clean-up where needed), would make the build less effective to reveal problems.

Table 5 Results of Bad Smells' Perception - Infrastructure Choices (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

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

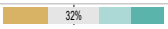
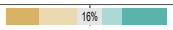
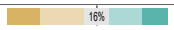
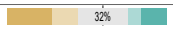
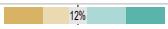
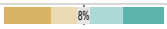
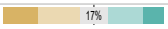
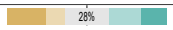
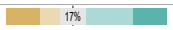
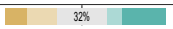
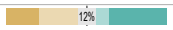
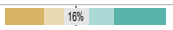
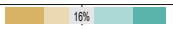
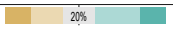
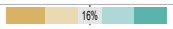
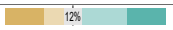
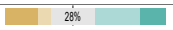
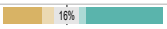
There are two CI bad smells dealing with cases in which either monolithic builds are used where they should not (BP4), and where there is a poor decomposition of build jobs (BP3), *e.g.,* including several activities in one single job or duplicating the same activities in multiple different jobs.

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

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

An inappropriate setting of the build outcome, *e.g.,* succeeding a build when a task is failed (BP16), is also considered a bad smell. The same applies when the outcome of a build depends on some flakiness in the execution (BP17). Although flakiness has been extensively studied in some specific context, *e.g.,* for testing (Bell et al., 2018; Luo et al., 2014; Palomba and Zaidman, 2017; Thorve et al., 2018), we focus on the extent to which different kinds of flakiness (related to testing, but also to the temporary unavailability of some online resources) affect the build outcome on a CI server. Furthermore, the build output needs to be properly configured (BP20–BP24). Here

Table 6 Results of Bad Smells' Perception - Build Process Organization (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

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

the bad smells are related to inadequate observability and low readability of logs/notifications.

Inadequate/wrong dependency management is also felt like a very important problem (BP18, BP19). Previous studies reported how the majority of build failures is due to these kinds of problems (Kerzazi et al., 2014; Seo et al., 2014; Vassallo et al., 2017).

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

Finally, although we did not focus on security-related issues, as there is a specific work by Rahman et al. (2019), we got interview responses as well as SO discussions related to the presence of security-sensitive data (*e.g.,* authentication information) hard-coded in the VCS (BP29).

Table 7 Results of Bad Smells' Perception - Build Maintainability (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

ID	CI Bad Smell	D	Resp.	Survey Results
BM1	Absolute/machine-dependent paths are used	✗	26	31% 4% 65%
BM2	Build scripts are highly dependent upon the IDE	✓	26	27% 15% 58%
BM3	Environment variables are not used at all	✓	26	35% 35% 31%
BM4	Build configurations are cloned in different environments	✓	24	50% 21% 29%
BM5	Build jobs are not parametrized	✗	25	36% 28% 36%
BM6	Lengthy build scripts	✗	25	40% 32% 28%
BM7	Missing smoke test, set of tests to verify the testability of the build	✓	25	24% 32% 44%
BM8	Missing/Poor strict naming convention for build jobs	✗	26	15% 38% 46%

Build Maintainability. Since build configuration files often change over time and their changes induce more relative churn than source code changes (McIntosh et al., 2011), their maintainability is also an important concern. As reported in Table 7, problems can arise when a build configuration is coupled with a specific workspace (see BM1–BM3, *e.g.,* environment variables are not used when they should), or when the build script is poorly commented, uses meaningless variable names, and modularity is not used when it should be. While a recent work by Gallaba and McIntosh (2018) deals with specific problems related to the maintainability of Travis-CI `.travis.yml` files, our bad smells are technology-independent and cover problems that can occur in all scripts involved in a build pipeline.

Quality Assurance. This category relates to CI bad smells that are linkable to testing and static analysis phases (see Table 8). Bad smells related

Table 8 Results of Bad Smells' Perception - Quality Assurance (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

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

to testing are due to the lack of optimization for testing tasks within a CI pipeline[5]: for example a branch is not tested before merging it (Q5), or all permutations of features toggles are tested (Q6) and, as a consequence, the build gets slow or fails without a reason.

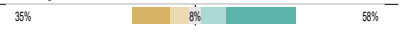
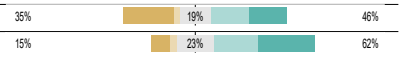
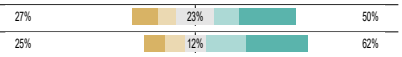
Moreover, this category features smells related to how test coverage thresholds (resulting in build failures, when not reached) are set (Q3, Q4), or the lack of a clear separation between test suites related to different testing activities (Q10) (Manuel Gerardo Orellana Cordero and Demeyer, 2017; Vassallo et al., 2017). Our catalog also includes problems related to (i) the use of production resources during testing operations (Q7), (ii) the test suite not being fully automated provoking a non-reproducible build (Q8), and (iii) the presence of flaky tests that may unnecessarily fail the build (Q9). Note that we have ex-

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

cluded from our analysis the presence of test smells (van Deursen et al., 2001), since they can negatively impact the understandability and maintainability of the product under development independently from the use of CI.

Bad smells related to static analysis are mostly due to how tools are configured and used within a CI pipeline. More in detail, they include failing to select appropriate checks given the project characteristics (Q14), or setting quality gates not representative of what is relevant for developers and/or customers (Q12).

Table 9 Results of Bad Smells' Perception - Delivery Process (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

ID	CI Bad Smell	D	Resp.	Survey Results
D1	Artifacts locally generated are deployed	✗	26	
D2	Missing Artifacts' repository	✓	26	
D3	Missing rollback strategy	✓	26	
D4	Release tag strategy is missing	✗	26	
D5	Missing check for deliverables	✗	24	

Delivery Process. This category of bad smells concerns the storage of artifacts related to a project release. As reported in Table 9, they are related to poor/lack of usage of artifact repositories giving the possibility of rollback of deployed artifacts (D2, D3), or to the adoption of bad deployment strategies (D1), *e.g.,* deployment of locally-generated artifacts.

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

Culture. We found bad smells whose symptoms might not be inferred by observing the CI pipeline, but are more human-related (see Table 10). They deal with the lack of a shared culture on how developers should behave when adopting CI. These include (i) bad push/pull practices (C1, C2), *e.g.,* pushing changes before a previous build failure is being fixed; (ii) poor prioritization of CI-related activities (C5, C6), including the fixing of build failures, and (iii) Dev/Ops separation (C3, C4), *i.e.,* developers and operators roles are kept separate, which is against the DevOps practice and, among other negative effects, increases the burden when fixing build failures affecting different stages of a CI pipeline.

4.2 Perceived Importance of CI Bad Smells

In the following, we describe examples of CI bad smells perceived as very relevant, or, on the contrary, not particularly relevant by the respondents to

Table 10 Results of Bad Smells' Perception - Culture (mapping with Duvall antipatterns (D), # of respondents that evaluated the smell (Resp), and the asymmetric stacked bar chart with 3 percentages: strongly disagree/disagree answer, neutral answer, and agree/strongly agree answer).

ID	CI Bad Smell	D	Resp.	Survey Results
C1	Changes are pulled before fixing a previous build failure	✗	23	30% 26% 43%
C2	Team meeting/discussion is performed just before pushing on the master branch	✗	22	32% 32% 36%
C3	Developers and Operators are kept as separate roles	✓	25	24% 32% 44%
C4	Developers do not have a complete control of the environment	✓	26	19% 35% 46%
C5	Build failures are not fixed immediately giving priority to other changes	✓	26	12% 31% 58%
C6	Issue notifications are ignored	✓	25	16% 20% 64%

our survey. As previously explained, a summary of the perceived relevance is depicted as asymmetric stacked bar charts on the right-side of Table 4-Table 10.

Overall, our results indicate that:

1. 26 bad smells received a strongly agree/agree assessment by over 50% of the respondents (*i.e.,* the third percentage in the asymmetric stacked bar charts is greater than 50%). Such bad smells are listed in Table 11.
2. 26 bad smells had more strongly agree/agree (but $\leq$ 50%) than strongly disagree/disagree assessments.
3. 14 bad smells had more strongly disagree/disagree (but $\leq$ 50%) than strongly agree/agree assessments.
4. 7 bad smells received a strongly disagree/disagree assessment by over 50% of the respondents (*i.e.,* the first percentage in the asymmetric stacked bar charts is greater than 50%). Such bad smells are listed in Table 12.
5. 5 bad smells received an equal number of agreement and disagreement assessments.

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

As regards the **Repository** organization (see Table 4), out of 10 CI bad smells, 3 were considered relevant by the majority of respondents, and other 3 received more positive than negative assessments. Respondents found particularly relevant the lack of alignment between the local (developers') workspace and the CI server workspace (R3), but also the lack of a stable release branch (R5). This was also remarked by one of our interviewees: *"Production builds have to be "shiny", the development ones can be "cloudy". The last build in production must not be a failed build."*. If there is a stable release branch, the development team always has a software product ready to be released.

Table 11 Bad smells considered relevant by the majority of respondents.

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

Table 12 Bad smells considered as not relevant by the majority of respondents.

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

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

Most of the bad smells belonging to the **Infrastructure Choices**, see Table 5, received more negative than positive assessments. While the original book on CI by Duvall et al. (2007) stressed this aspect, nowadays the availability of adequate hardware to support a CI server is generally not an issue, unless one has to deal with very specific contexts, such as cyber-physical systems for which the pipeline requires to be connected with hardware-in-the-loop or simulators. We would have expected positive feedback about bad smells related to software infrastructure choices, but this was not usually the case. In particular, the distribution of resources related to the same pipeline stage over multiple servers (I1) was not considered a problem, likely because, at least for some pipeline stages (*i.e.,* the ones different from the commit stage), the overhead due to the download of required resources from different servers may still be acceptable. Only one bad smell received more positive than negative assessments. Such a bad smell is related to the use of multiple plugin versions (I4), possibly causing conflicts or inconsistencies. One SO post, while identifying the root cause of a build failure, remarked that *"When there are multiple version of [a tool] installed side by side in Build Server, ensure right version of [the tool] is used by Build Server to execute the unit test"*. Truly, an inconsistency may not necessarily result in a build failure, but may produce slightly different outputs that might confuse developers or even fail external tools consuming such outputs. The usage of default configurations in build scripts (I3) received an equal proportion of positive and negative assessments. While previous research has highly motivated the need for properly configuring tools used in the build (*e.g.,* static analysis tools, see the work of Zampetti et al. (2017)), it was also found that developers rarely pay attention to that (Beller et al., 2016; Zampetti et al., 2017).

As expected, we received positive feedback by the majority of respondents for several (11 out of 29) bad smells belonging to the **Build Process Organization**, *i.e.,* on how all steps of a build are configured through build automation scripts, as detailed in Table 6. In this context, a relevant bad smell is the lack of a proper package management mechanism (BP2) in the build automation. Specifically, if the CI pipeline does not include a package manager, developers might wrongly assume the presence of resources no longer available or use an outdated version of them. Furthermore, it becomes difficult to manage tools included in the pipeline, in terms of installing, upgrading, configuring, and removing them. By using a package manager it is possible to

specify when checking for updates, thus avoiding to download all dependencies at each build. The CI bad smell discussed above originates from an interviewee who clearly stated that: *"Sometimes we don't clean the dependencies (that are not useful anymore) in our project. As a result, we have huge packages that are time-consuming to deploy"*.

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

The way the build output is reported is also particularly important. First of all, respondents believe that ignoring the outcome of a task when determining the build status (BP16) defeats the primary purpose of CI. These kinds of smells may occur when, for example, static analysis tools produce high-severity warnings without failing a build. While a previous study found that this practice is indeed adopted for tools that may produce a high number of false positives (Wedyan et al., 2009), one SO post remarked that *"...if the build fails when a potential bug is introduced, the amount of time required to fix it is reduced."*, and a different user in the same discussion highlighted that *"If you want to use static analysis do it right, fix the problem when it occurs, don't let an error propagate further into the system."*. A related bad smell judged as relevant is the lack of an explicit notification of the build outcome (BP23) to developers through emails or other channels. In other words, having the build status only reported in the CI dashboard is not particularly effective, because developers might not realize that a build has failed.

Furthermore, it can be troublesome if tools do not (clearly) report results in the build logs (BP20, BP21) since their output might be difficult or impossible to access when the tool is executed on a CI server. At the same time, a very verbose build log (BP24) containing unnecessary details (this depends on how different tools/plugins are configured) would make the build result difficult to browse and understand. Quite surprisingly, even if an interviewee remarked that *"To skip tests is always an antipattern "*, our survey results report the same proportion of agreement and disagreement on this bad smell (BP8). This is a surprising result since it is difficult to imagine some circumstances in which hiding the presence of issues/problems in the build is a good practice.

Concerning **Build Maintainability**, out of eight bad smells, two received a positive assessment by the majority of respondents and other two received more positive than negative responses (see Table 7). The two most positively-assessed bad smells were related to the usage of absolute paths in the build (BM1), and the coupling between the build and the IDE (BM2). The high perceived relevance of such smells is justified considering that their presence

will unavoidably limit the portability of the build resulting in statements such as *"but it works on my machine"*. The need for performing smoke-testing (BM7) was also considered relatively important, as well as the usage of suitable naming conventions inside the build scripts (BM8).

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

About **Quality Assurance**, out of 14 bad smells, five received a positive assessment by the majority of respondents, and six more positive than negative assessments (see Table 8). Respondents considered as relevant the lack of testing in a production-like environment (Q1), while at the same time they considered dangerous the use of production resources for testing purposes (Q7). The latter has been also highly discussed in SO. Indeed, in a SO post, a user searching *"...for the CI server to be useful, my thoughts are that it needs to be run in production mode with as close-as-possible a mirror of the actual production environment (without touching the production DB, obviously)."*. By analyzing the provided answers on SO, we found that *"Testing environment should be (configured) as close as it gets to the Production."* concluding the discussion by highlighting that *"The best solution is to mimic the production environment as much as possible but not on the same physical hardware."*.

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

Concerning the usage of static analysis, the majority of respondents considered a problem the use of CI (and static analysis tools) to monitor specific developers (Q13) since, as also highlighted in our interviews, this kind of monitoring only negatively impacts the overall developers' productivity and their ability to work in a team. Moreover, one interviewee also highlighted that

"Monitor the global technical debt ratio, without paying attention to single developer activity. It's up to developers to decide when to perform refactoring". As regards the definition of quality gates only based on customers' guidelines (Q12), we got more positive than negative judgments. Indeed, customers could not have enough knowledge about the software development process and the implemented source code. This might result in an excessive number of build failures, due to some quality checks suggested by developers (Johnson et al., 2013) (*e.g.,* on a too low cyclomatic-complexity threshold or too high test-coverage threshold) or, on the contrary, might result in the omission of some relevant checks that only developers might consider, *e.g.,* because they know details about the source code.

In general, we received particularly positive feedback for the bad smells related to the categories **Delivery Process** and **Culture**. All bad smells belonging to the **Delivery Process** category obtained a positive assessment either by the majority of respondents or, at least, they received more positive than negative assessments (see Table 9). The most relevant bad smells were related to the deployment of artifacts generated locally (D1) (*i.e.,* on the developers' machine rather than on the CI infrastructure), to the absence of an artifact repository where to deploy released products (D2), and to the lack of a rollback mechanism (D3).

About the **Culture** category, the majority of respondents considered relevant bad smells related to not trying to fix a build failure immediately (C5), and ignoring CI notifications (C6), see Table 10. Other bad smells related to how work is organized received more positive than negative feedback. They concern (i) pulling from the repository when a build has failed (C1), (ii) organizing a meeting just before pushing to the master (C2), (iii) keeping separate the developer and operator roles (C3), and (iv) developers that do not have complete control of the CI environment (C4). This indicates how the lack of a shared view on how developers (and teams) should behave when adopting CI would diminish the advantages introduced by putting in place a suitable (and in some cases complex) CI infrastructure.

4.3 Mapping and Comparison with Duvall's Antipatterns

Table 13 reports the mapping between Duvall's patterns and our CI bad smells. Note that we use the pattern name, because Duvall, under each pattern, reports a brief description of the good practice (pattern) and the corresponding bad practice (antipattern). For brevity, in Table 13 and in the remainder of this section, we refer pattern/antipattern using the pattern's name, however, we discuss the comparison considering Duvall's antipatterns and our CI bad smells. Where necessary, we also report the corresponding antipattern in our text. The last column summarizes the perception of the CI bad smell (detailed values are in Tables 4-10) corresponding to each Duvall's pattern. In particular, we use (i) ↑↑ to indicate bad smells that received a positive assessment by the majority of respondents; (ii) ↑ for bad smells that received more positive

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

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

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

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

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

1. "Blue-Green Deployments", highlighting the need for deploying software to a non-production environment while production continues to run;

Table 13 Mapping between Duvall's patterns (and their antipatterns) and CI smells.

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

2. "Immune System", that emphasizes the need for deploying software one instance at a time while conducting Behavior-Driven Monitoring, and
3. "Decouple Database", aimed at ensuring that the application is backward and forward compatible with the database giving the possibility of independent deployment activities.

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

From a different perspective, looking at the second column in Table 4-Table 10, only 35 of our CI bad smells are covered by Duvall, while 44 are completely uncovered. More specifically, our catalog includes specific CI bad smells related to (i) the way the CI infrastructure is chosen and organized, as well as (ii) how a build process is organized and configured, and (iii) testing and quality checks.

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

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

Looking at the second column in Table 8, we can notice that three CI bad smells related to the definition/usage of quality gates are not mapped onto Duvall's catalog, and, at the same time, are highly relevant for our survey participants. More in detail, the "Quality gates are defined without developers considering only what dictated by the customer" (Q12) highlights that developers need to be involved in the definition of the quality gates, since customers do not have enough knowledge of software development and implementation details. The "Use quality gates in order to monitor the activity of specific developers without using them for measuring the overall software quality" (Q13), instead, emphasizes the fact that the overall team spirit will be negatively impacted in presence of monitoring activities. Finally, the "Unnecessary static analysis checks are included in the build process" (Q14) may unnecessarily slow down the build duration as well as may decrease the developers' productivity, since developers will waste their time trying to fix violated checks that do not completely fit the need of their organization.

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

Nightly builds are a particular type of scheduled builds that are usually performed overnight (*i.e.,* out of working hours). Duvall reports their usage as an antipattern corresponding to the "Continuous Integration" pattern ("Scheduled builds, nightly builds, building periodically, building exclusively on developers machines, not building at all."). Indeed, Duvall highlights as good practice the need for building and testing software with every change committed to a project VCS (Duvall, 2011).

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

Having an adequate branching strategy is another key point in the CI process. Our catalog provides four different CI bad smells dealing with the adoption of a poor branch management strategy in the CI process. Among them, the "Feature branches are used instead of feature toggles" (R6) and "Number of branches do not fit the project needs/characteristics" (R4) identify situations in which the right decision depends upon the organization needs. More specifically, the bad smell (R6) discourages the use of branches when developing features, and it is in agreement with what stated by Duvall. However, a recent study by Rahman et al. (2016) found that feature toggles, despite their advantages, could introduce technical debt involving maintenance effort for developers. To this regard, we found quite controversial opinions in SO posts. For example, in one SO post, the preferred choice is the feature toggles even if "... requires very strict discipline as broke/dark code is making it to production". The discussion ends with the following suggestion: *"I do not believe in a better choice in all the cases"*.

Furthermore, the "Number of branches do not fit the project needs/characteristics" (R4) bad smell, mapped onto Duvall's antipattern "Multiple branches per project" (related to the "Mainline" pattern), occurs when a project has several branches that might not be needed. While Duvall indicates as an antipattern the "Feature Branching", our CI bad smell suggests the usage of a proper number of branches according to a well-defined strategy. An interviewee of our study agrees with the usage of different branches evolving in parallel (*"We use a branch for each service and usually 3/4 developers work on it. There is a straight separation between each branch."*), while some SO

discussions discourage to exceed in the number of branches: *"You must have your changes included in the main trunk so you can reduce the number of conflicts related to merge operations".*

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

Finally, Duvall considers as an antipattern the execution of dependent build jobs/tests in parallel. This bad smell has been found in many SO discussions, however, as already detailed in Section 3.2, we discarded it because we consider this to be a bug rather than a bad smell.

5 Threats to Validity

Threats to *construct validity* relate to the relationship between theory and experimentation. These are mainly due to imprecision in our measurements. We relied on SO tags to filter SO discussions. While it is possible that we could have missed some relevant posts because their tags were not directly related to CI, we at least made sure to pick all tags of interest by performing a manual, exhaustive analysis of all SO tags. Concerning the protocol used for the semi-structured interviews, it is possible that the incompleteness of our interview structure could have lead to some CI bad smells. However, to mitigate this threat, the questions being asked in the last part of our interview took into account what we learned from the existing literature (Duvall et al., 2007; Duvall, 2011; Humble and Farley, 2010). Concerning the way we collected bad practices relevance through the survey, we used a 5-level Likert scale (Oppenheim, 1992) to collect the perceived relevance of each CI practice. To limit random answers, we added a "Don't know" option and the opportunity to explain the answers with a free comment field.

Threats to *internal validity* are related to confounding factors, internal to the study, that can affect our results. Internal validity threats can, in particular, affect the extent to which the protocol followed to build the catalog could have influenced our results. As detailed in Section 3.2, we used different coun-

termeasures to limit the influence of our subjectiveness. As for the analysis of SO posts, it was not possible to afford two independent taggers per post. However, the independent re-check on the "Yes"/"'Maybe" limited the false positives, though could not mitigate possible false negatives. The mapping onto Duvall's antipatterns and the validation of the catalog were performed by two independent evaluators, inter-rater agreement was computed, assignment conflicts were resolved through a discussion, and inconsistent cases were re-coded again to improve the inter-rater agreement. For what concerns the creation of the catalog itself, we interleaved multiple iterations done by discussions of comments each author raised during each iteration.

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

Threats to *external validity* concern the generalizability of our findings. These are mainly due to (i) the choice of SO as a source for mining CI-related discussions, and (ii) the set of interviewed people. Concerning SO, it is the most widely used development discussion forum to date. Although specific forums for CI exist (*e.g.,* DZone[6]), such forums are more portals where white papers (*e.g.,* the one with Duvall's antipatterns (Duvall, 2011)) are posted rather than a Question & Answer (Q&A) forum. The number of participants (and companies) to the interviews and survey is admittedly, limited. At the same time, the companies are pretty diverse in terms of domain and size, and the interviewed/surveyed people always had several years of experience. Nevertheless, we could still have missed some relevant problems, *e.g.,* we have shown (Section 4.3) that eight of Duvall's antipatterns did not emerge from our study.

6 Implications

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

6.1 Implications for Practitioners

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

In this context, the aim of the catalog is providing developers with concrete misuses of the CI pipeline. These misuses can occur at different stages,

[6] https://dzone.com

including: (i) setting up and configuring the Software Configuration Management (SCM) infrastructure, as well as the CI infrastructure and tooling, and (ii) performing daily activities in the context of a CI pipeline, which translates into creating and synchronizing/merging branches, pushing changes, interpreting and leveraging results of the CI builds to improve software quality.

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

Favor specific, portable tools over hacking. One of the CI bad smells we identified is related to a sub-optimal selection of tools composing the CI pipeline. Specifically, the adoption of a tool that does not provide the features needed by developers may cause delays or, even worse, force the adoption of "hacking solutions", *e.g.,* custom shell scripts. While such scripts can represent a quick-and-dirty solution for the immediate, in the long term they can exhibit maintainability issues, or even introduce portability problems on different machines. This CI bad smell was pointed out by one of our interviewees, saying that they *"replaced the old scripts with Jenkins"*. Our catalog includes the bad smell "Use shell scripts for a task for which there is a suitable plugin available" (I7) that covers these aspects. Moreover, developers should pay attention to avoid different versions of tools conflicting with each other, as indicated by the bad smell "Different releases of tools/plugins versions are installed on the same server" (I4).

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

Do not ignore nor hide build failures. When working with the CI pipeline, it is possible that developers do not give adequate priority and importance to warnings outputted in build logs, or even worse, to broken builds. As a consequence, they tend to hide actual problems by just disabling failing tests or quality checks instead of solving them. In other cases, developers might improperly use the features provided by the whole environment or by specific toolkits. A concrete example is reported in a SO post[7] in which a user highlighted that there are cases in which developers focus more on *". . . keep the build passing as opposed to ensuring we have high quality software."*. The latter translates into *". . . comment[ing] out the tests to make the build pass."*

[7] https://stackoverflow.com/questions/214695/

resulting in having a passed build while the overall software quality is going down.

Our catalog features the bad smell "Pipeline steps/stages are skipped arbitrarily" (BP8), pointing out that it is important to not skip pipeline tasks only to have a passed build. Also, the Culture category features bad smells occurring when "Build failures are not fixed immediately giving priority to other changes" (C5), or "Issue notifications are ignored" (C6).

6.2 Implications for Researchers

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

Also, a CI bad-smell recommender could go beyond that, and "observe" the activity carried out by developers through the pipeline, including branching strategies, push/pull frequency, distribution of build failures, fix time *etc.*. As an example, the CI bad smell "Pipeline steps/stages are skipped arbitrarily" (BP8) could be detected by analyzing the build history of the project, and detecting cases where a build that failed because of test cases becomes successful again without changes to the production code but, instead, because test cases were commented out or disabled. One relevant example of CI bad-smell detector is the CI-Odor tool by Vassallo et al. (2019a), which copes with four antipatterns from Duvall's catalog.

Support for failure analysis. A key element for quick fixing a build is the capability of developers to properly analyze the output of a build. Our CI bad smells provide indications of what should be avoided when configuring the build log (*e.g.,* "Build reports contain verbose, irrelevant information" (BP24) or "The output of different build tasks are mixed in the build output" (BP21)) or other notification mechanisms (*e.g.,* "Failures notifications are only sent to teams/developers that explicitly subscribed" (BP22)). Also, techniques similar to those used to summarize release notes (Moreno et al., 2017) or bug reports (Rastkar et al., 2014) could be used to extract and summarize relevant information from verbose and complex build logs (as done by Vassallo et al. (2019b)).

Keep the context into account. When building tools to identify CI bad smells, researchers should take into account the developers' context and, if possible, their feedback to past recommendations to properly tune the recommender and avoid overloading developers with irrelevant suggestions. This could be achieved by collecting a set of preferences about the CI practices

adopted in one organization (*e.g.,* when to open a branch, push/pull practices, testing, and static analysis needs), by also observing the past history of developers' activity. For example, self-admitted technical debt (Potdar and Shihab, 2014) can be used to learn bad smells relevant to developers, and consequently help to configure static analysis tools properly.

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

6.3 Implications for Educators

Teach CI by providing examples of what not to do. The proposed catalog can be valuable for educators introducing CI/CD principles in software engineering curricula. A typical course about CI would introduce the topic, illustrate the main claimed advantages of CI, explain the technologies that can be used, and, importantly, provide principles to properly set up and use the CI pipeline. So far, such principles are typically being taught mainly based on the content of known books (Duvall et al., 2007; Humble and Farley, 2010) or by using catalogs/white-papers reflecting existing knowledge in this area coming from the available literature.

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

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

7 Conclusions and Future Work

The adoption of Continuous Integration and Delivery (CI/CD) as development practice is increasing. Studies conducted in industrial organizations and open source projects highlight benefits of CI/CD, including increased developer productivity, software quality, and reliability (Chen, 2017; Hilton et al., 2016; Vasilescu et al., 2015). Despite those advantages, the introduction of CI/CD in well-defined development contexts is still challenging, as also highlighted in previous work (Duvall, 2011; Hilton et al., 2017; Olsson et al., 2012).

This paper empirically investigates, by analyzing over 2,300 SO discussions and by interviewing 13 industry experts, on bad practices developers

encounter when adopting CI. As a result of the study, we compiled a catalog of 79 CI bad smells organized into 7 categories. The catalog has been validated through a survey involving 26 professional developers, indicating their perceived relevance of the 79 bad smells.

While a catalog of CI patterns and antipatterns exists (Duvall, 2011), this is, to the best of our knowledge, the first catalog empirically derived from concrete problems discussed in SO or highlighted by interviewed experts, and following a replicable methodology.

As we have shown in Section 4.3, while in some cases our results confirm what known from the existing literature, there are quite a few controversial cases, such as the appropriate balancing in the usage of nightly builds or feature branches. Moreover, while our catalog does not cover 11 of Duvall's antipatterns (though covering all categories), there are 44 of our CI bad smells not covered by Duvall, and they especially pertain to the CI infrastructure, the build configuration, as well as testing and quality checks. We also discuss cases in which the two catalogs agree and disagree, also highlighting examples of Duvall's antipatterns not perceived as important by our survey participants.

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

Interestingly, while some researchers previously investigated barriers towards CI (Hilton et al., 2017) and CD (Olsson et al., 2012) adoption, the inferred bad practices indicate that some of those issues still arise when the CI process has been adopted. They include, for example, long in-the-loop feedback during development, or network configuration issues.

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

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

References

Abdalkareem R, Mujahid S, Shihab E, Rilling J (2019) Which commits can be CI skipped? IEEE Transactions on Software Engineering pp 1–1

Amazon (2017) What is continuous delivery? URL https://aws.amazon.com/devops/continuous-delivery/

Basili VR (1992) Software modeling and measurement: The goal/question/metric paradigm. Tech. rep., College Park, MD, USA

Beck K (2000) Extreme programming explained: embrace change. Addison-Wesley Professional

Bell J, Legunsen O, Hilton M, Eloussi L, Yung T, Marinov D (2018) Deflaker: automatically detecting flaky tests. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 433–444

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)

Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: An explorative analysis of travis ci with github. In: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press

Booch G (1991) Object Oriented Design: With Applications. Benjamin Cummings

Chen L (2017) Continuous delivery: Overcoming adoption challenges. Journal of Systems and Software

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas

Deshpande A, Riehle D (2008) Continuous integration in open source software development. Open source development, communities and quality

Duvall P, Matyas SM, Glover A (2007) Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley

Duvall PM (2010) Continuous integration. patterns and antipatterns. DZone refcard #84 URL http://bit.ly/l8rfVS

Duvall PM (2011) Continuous delivery: Patterns and antipatterns in the software life cycle. DZone refcard #145 URL https://dzone.com/refcardz/continuous-delivery-patterns

Fowler M, Beck K, Brant J (1999a) Refactoring: improving the design of existing code. Addison-Wesley

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999b) Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional

Gallaba K, McIntosh S (2018) Use and misuse of continuous integration features: An empirical study of projects that (mis)use Travis CI. IEEE Transactions on Software Engineering (to appear):1–1, DOI 10.1109/TSE.2018.2838131

Ghaleb TA, da Costa DA, Zou Y (2019) An empirical study of the long duration of continuous integration builds. Empirical Software Engineering 24(4):2102–2139

Goodman LA (1961) Snowball sampling. The annals of mathematical statistics

Hilton M, Tunnell T, Huang K, Marinov D, Dig D (2016) Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)

Hilton M, Nelson N, Tunnell T, Marinov D, Dig D (2017) Trade-offs in continuous integration: Assurance, security, and flexibility. In: Proceedings of the

25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017

Humble J, Farley D (2010) Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Software Engineering (ICSE), 2013 35th International Conference on, IEEE

Kerzazi N, Khomh F, Adams B (2014) Why do automated builds break? an empirical study. In: 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE

Krippendorff K (1980) Content analysis: An introduction to its methodology. Sage

Luo Q, Hariri F, Eloussi L, Marinov D (2014) An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pp 643–653

Manuel Gerardo Orellana Cordero AM Gulsher Laghari, Demeyer S (2017) On the differences between unit and integration testing in the travistorrent dataset. In: Proceedings of the 14th working conference on mining software repositories

McIntosh S, Adams B, Nguyen TH, Kamei Y, Hassan AE (2011) An empirical study of build maintenance effort. In: Proceedings of the Int'l Conference on Software Engineering (ICSE)

Moreno L, Bavota G, Di Penta M, Oliveto R, Marcus A, Canfora G (2017) ARENA: an approach for the automated generation of release notes. IEEE Trans Software Eng 43(2):106–127

Olsson HH, Alahyari H, Bosch J (2012) Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '12

Oppenheim B (1992) Questionnaire Design, Interviewing and Attitude Measurement. Pinter Publishers

Palomba F, Zaidman A (2017) Does refactoring of test smells induce fixing flaky tests? In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, pp 1–12

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 30th IEEE International Conference on Software Maintenance and Evolution

Rahman A, Parnin C, Williams L (2019) The seven sins: security smells in infrastructure as code scripts. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp 164–175

Rahman MT, Querel LP, Rigby PC, Adams B (2016) Feature toggles: practitioner practices and a case study. In: Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on, IEEE

Rastkar S, Murphy GC, Murray G (2014) Automatic summarization of bug reports. IEEE Trans Software Eng 40(4):366–380

Savor T, Douglas M, Gentili M, Williams L, Beck K, Stumm M (2016) Continuous deployment at facebook and OANDA. In: Companion proceedings of the 38th International Conference on Software Engineering (ICSE Companion)

Seo H, Sadowski C, Elbaum SG, Aftandilian E, Bowdidge RW (2014) Programmers' build errors: a case study (at Google). In: Proc. Int'l Conf on Software Engineering (ICSE)

Spencer D (2009) Card sorting: Designing usable categories. Rosenfeld Media

Ståhl D, Bosch J (2014a) Automated software integration flows in industry: a multiple-case study. In: Companion Proceedings of the 36th International Conference on Software Engineering, ACM

Ståhl D, Bosch J (2014b) Modeling continuous integration practice differences in industry software development. J Syst Softw

Thorve S, Sreshtha C, Meng N (2018) An empirical study of flaky tests in android apps. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, pp 534–538

van Deursen A, Moonen L, Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)

Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM

Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Di Penta M, Zaidman A (2016) Continuous delivery practices in a large financial organization. In: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)

Vassallo C, Schermann G, Zampetti F, Romano D, Leitner P, Zaidman A, Di Penta M, Panichella S (2017) A tale of ci build failures: An open source and a financial organization perspective. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE

Vassallo C, Proksch S, Gall H, Di Penta M (2019a) Automated reporting of anti-patterns and decay in continuous integration. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, Canada, May 25 - 31, 2019, IEEE, p (to appear)

Vassallo C, Proksch S, Zemp T, Gall HC (2019b) Every build you break: Developer-oriented assistance for build failure resolution. Empirical Software Engineering (To appear)

Wedyan F, Alrmuny D, Bieman JM (2009) The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: Second

International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, pp 141–150

Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press

Zampetti F, Vassallo C, Panichella S, Canfora G, Gall H, Di Penta M (2019) An empirical characterization of bad practices in continuous delivery (online appendix). Tech. rep., URL http://home.ing.unisannio.it/fiorella.zampetti/datasets/CIBadPractices.zip