Automated Reporting of Anti-Patterns and Decay in Continuous Integration

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

Sebastian Proksch
Department of Informatics
University of Zurich
Zurich, Switzerland
proksch@ifi.uzh.ch

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

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

***Abstract*—Continuous Integration (CI) is a widely-used software engineering practice. The software is continuously built so that changes can be easily integrated and issues such as unmet quality goals or style inconsistencies get detected early. Unfortunately, it is not only hard to introduce CI into an existing project, but it is also challenging to live up to the CI principles when facing tough deadlines or business decisions. Previous work has identified common anti-patterns that reduce the promised benefits of CI. Typically, these anti-patterns slowly creep into a project over time before they are identified. We argue that automated detection can help with early identification and prevent such a process decay. In this work, we further analyze this assumption and survey 124 developers about CI anti-patterns. From the results, we build CI-ODOR, a reporting tool for CI processes that detects the existence of four relevant anti-patterns by analyzing regular build logs and repository information. In a study on the 18,474 build logs of 36 popular JAVA projects, we reveal the presence of 3,823 high-severity warnings spread across projects. We validate our reports in a survey among 13 original developers of these projects and through general feedback from 42 developers that confirm the relevance of our reports.**

***Index Terms*—Continuous Integration, Anti-Pattern, Detection, CI-Smell, CI-Decay**

I. INTRODUCTION

Continuous Integration (CI) is a common development practice and its great benefits on quality and productivity are widely accepted [29]. CI advocates full automation of all build steps (i.e., compilation, testing, and code quality assessment) to create a new version of the software [3]. The CI process is most effective when developers follow best practices, such as *commit often*, that reduce conflicts in the team and ensure that the build is continuously executable [3]. In practice, it is often challenging to live up to these standards and *anti-patterns* can be observed: common but ineffective solutions to a recurring problem that should be avoided. For example, failing tests are removed instead of fixing the root cause. Over the years, researchers have defined catalogs of CI anti-patterns [4], [13], which eventually become a threatening maintainability problem for a software project, if not properly addressed [3].

A creeping decay of quality has been described before in other contexts. Fowler [5] popularized the term *code smell* to describe a symptom that indicates the existence of a deeper problem in the source code. While the perception of smell is subjective, previous work could show that code smell intensity correlates with the likelihood of the existence of a deeper issue [19]. The smell metaphor was later adopted in other areas such as system design [17], configuration files [23], or spreadsheets [10]. Hence, the idea that a CI anti-pattern manifests itself as a *CI smell* as well. In contrast to anti-patterns in development artifacts, CI anti-patterns affect the software development process and provoke *CI decay*.

In this paper, we further study this phenomenon. Our results of a broad survey among 124 professional developers confirm that CI decay is indeed a relevant problem. Most participants confirm that deviations from CI best practices happen in practice, both intentionally and unintentionally, and that the benefits of CI diminish when many deviations exist in a project. The awareness about the presence of anti-patterns in the CI pipeline is key for an educated decision about whether a deviation needs to be fixed. Inspired by Duvall's catalog of Continuous Integration and Delivery (CD) anti-patterns [4], we built CI-ODOR, an automated detection and reporting tool that provides awareness about CI decay caused by four different anti-patterns. Through the analysis of build log and repository information our tool identifies (1) slow builds, and especially increasing trends of build time, (2) broken release branch, and the corresponding time-to-fix, (3) skipped failing tests, and (4) late merging of development branches.

By analyzing a total of 18,474 recent builds logs of 36 popular JAVA projects, we identified 3,823 high-severity anti-pattern instances, and 4,697 with medium severity, spread across all projects. To evaluate our tool, we have surveyed 13 original developers about the relevance of reports (containing recent instances of detected smells) generated for their project and 42 developers about the general usefulness of CI-ODOR. The reports are perceived as useful, relevant, and most participants would integrate CI-ODOR in their CI pipeline to increase their awareness about the CI process. We also find untapped potential for future detectors and that more work on CI anti-patterns is necessary to improve the handling of project specifics.

In summary, this paper presents the following contributions:

- Verification of the relevance of *CI decay* in practice;
- *Detectors* of four relevant CI anti-patterns;
- CI-ODOR, an automated *CI anti-patterns reporting tool*;
- An *empirical study* on the presence of CI decay and on the developers awareness about CI anti-patterns.

II. Methodology Overview

In this paper, we introduce CI-Odor, an automated reporting tool that can be integrated into CI pipelines to increase the awareness about anti-patterns in CI. Figure 1 illustrates our methodology to create and evaluate the tool.

This work is based on the existing anti-patterns catalog (1) of Duvall [4], which describes 50 patterns and anti-patterns that influence the effectiveness of a CI/CD pipeline. In an internal selection, we identified a subset of CI anti-patterns from this catalog that can be automatically detected by analyzing build log and versioning information (2). For each of these anti-patterns, we added an explanation and an illustration of the detection strategy and validated their selection in a survey among 124 professional software developers (3). We asked the participants about the relevance of the anti-patterns in practice and the suitability of our detection strategies. Based on the results of the survey, we eliminated several candidates, refined our detection strategies, and ended up implementing a set of four detectors (4). We integrated these detectors in a reporting tool, CI-Odor (5), that aggregates the different analysis results and that presents statistics such as a trend analyses to increase the awareness about the different anti-patterns.

We evaluated the usefulness and relevance of the reports created by CI-Odor in a second survey (6). For the survey, we conducted a case study, in which we analyzed the build logs of 36 projects (7). The resulting reports (8) are publicly available and we asked the original developers of these projects to rate them (9). In addition, we selected reports that illustrate the full capabilities of our reporting (e.g., there is at least one detected instance of each anti-pattern). We ask both the original developers and other developers with experience in CI (10) to rate these example reports.

III. Which Anti-Patterns to Detect, and How?

The existing anti-pattern catalog of Duvall [4] is extensive and contains several examples that go beyond the scope of automated tools, e.g., decisions regarding the deployment strategies. We started with selecting a subset of anti-patterns, for which we could develop appropriate detectors. In this section, we first introduce our pre-selected list of candidates and the survey that we used to validate and finalize our selection.

A. Pre-Selection

The rationale of our pre-selection was two-fold. We wanted to cover different aspects of the CI pipeline such as version control or build failure management and exclude others that are more related to CD. At the same time, we selected anti-patterns that can be detected using data that is typically produced by every CI pipeline independently from custom settings, i.e., build logs and repository. The following list introduces all anti-pattern candidates, proposes a detection strategy, and justifies their relevance for the CI process quality. For traceability, we include the name of Duvall's positive example [4].

Late Merging (Merge Daily) Agile teams often develop in features branches. Integration effort and conflict potential increase if completed features are not integrated timely. We propose to warn about cases in which the last commit of a branch is older than a predefined threshold.

Aged Branches (Short-Lived Branches) Infrequently synced feature branches substantially diverge over time and end up being very hard to integrate. We propose to warn when an open branch has not been merged into master for a release.

Broken Release Branch (Stop the Line) A broken build that is not fixed timely prevents the CI pipeline from properly assessing the effect of new changes. We propose to warn when a build stays broken for longer than usual.

Bloated Repository (Repository) Artifacts that can be created in a build or fetched through provisioning mechanisms should not be committed to the version control system. We propose to warn when binaries can be found in the repository.

Scheduled Builds (Continuous Integration) A scheduled build either (unnecessarily) builds a change a second time or is a sign that a change is not automatically built, which breaks the idea of always ensuring a working system. We propose to warn about build configurations that schedule builds.

Absent Feedback (Continuous Feedback) Developers are missing out on required feedback, when they are not automatically notified about relevant build events, especially build failures. We propose to warn about configuration files that do not enable any notification channels.

Email-Only Notifications (Visible Dashboards) According to Duvall [3], email is inappropriate as a single notification channel, because developers might not have access or notifications get lost among other messages. We propose to warn about configurations files that only notify by email.

Skip Failed Tests (Automate Tests) Skipping a failed test can fix a broken build but addresses a symptom rather than fixing the cause. It threatens the safety provided by the test suite. We propose to warn about cases in which a previously failed test does no longer occur in the next (fixed) build.

Slow Build (Fast Builds) A slow build, caused by a coding issue or by a high workload of build server, produces waiting times for developers and adds overhead to the CI process. We propose to warn about significant build slow-downs.

Please note that these descriptions are shortened introductions from our survey. A complete version of the first survey is available on our artifact page [30].

B. Survey on the Practical Relevance

We conducted a survey to validate the relevance of the selected anti-patterns and the proposed detection rules.

Survey Design. The survey contains three sections. The first section is about the perceived severity of the problem of deviations from CI best practices. The second section has a focus on the anti-patterns. We introduced each one with an elaborated description that includes explanatory images and asked participants to evaluate the anti-pattern relevance and our proposed detection strategy. Finally, we asked for a general validation of the idea of anti-patterns detection.

All survey questions were optional and had Likert scales [18] with either five (*Strong Disagree* to *Strong Agree*) or four levels (*None* to *High*). The survey also contained open questions for

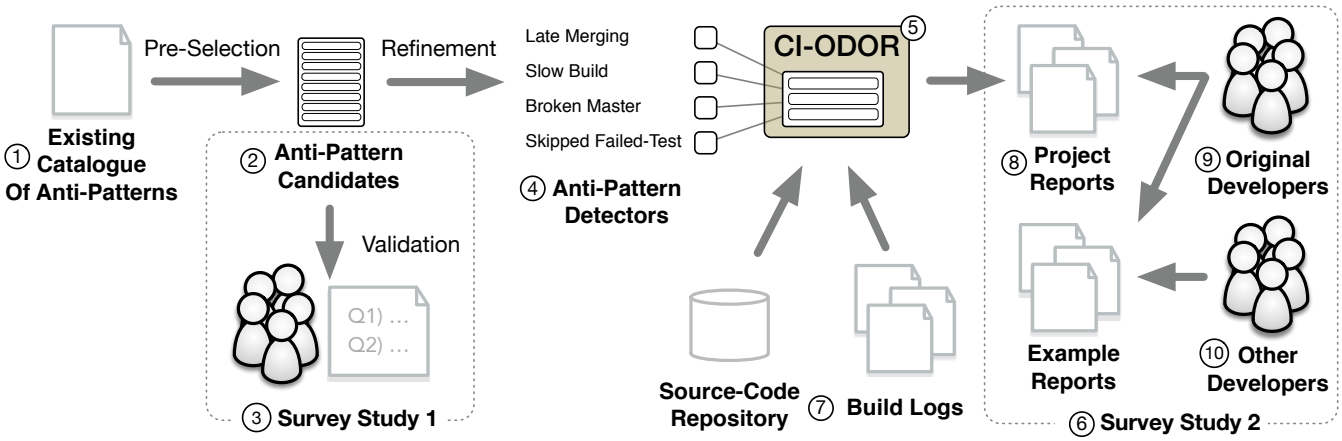


Fig. 1: Overview over the Different Parts of This Paper

feedback in all sections. Figure 2 includes an excerpt of our survey; a complete export is available on our artifact page [30].

Advertisement. We advertised the survey on social media (i.e., TWITTER, REDDIT sub-forums dedicated to DEVOPS and CONTINUOUS INTEGRATION), in CI-related newsletters, and by sending it to personal contacts. We promised to raffle off two vouchers to reward participation.

Demographics. In the end, 605 developers opened our questionnaire, out of which 144 finished all questions, leaving us with a completion rate of 23.8%. We included three control questions in the survey that asked about the proficiency in programming, CI theory, and CI practice. We excluded responses from participants that reported less than *moderate* experience in any of these questions. After the filtering, we ended up with 124 qualified participants. Most of our participants (79.8%) report that they got in contact with CI in an industry position. 68% of our participants hold an academic degree related to computer science (32.3% Bachelor, 28.2% Master, 8% Ph.D.). The large majority reports a *high* level of experience in programming (76.6%), CI theory (69.4%), and CI practice (59.7%).

Data-Analysis Methodology. To analyze the Likert-scale answers, we create asymmetric stacked bar charts with proportions for various agreement levels that are shown in Figure 3. We performed card sorting to analyze the open answers [25]. We started by splitting the answers into individual statements, grouped common arguments, and finally organized these arguments hierarchically.

Problem Statement

- CI best-practices are no strict rules, they can be adapted for a project.
- One can deviate from CI best-practices unintentionally.
- The benefit of using CI diminishes, the more best-practice deviations exist.

Relevance and Sufficiency of Smell Detection (for all anti-patterns)

- Anti-pattern X is relevant in a typical CI pipeline.
- The detection strategy is sufficient to identify occurrences of an anti-pattern.

General Validation of Idea

- I would integrate such a tool in my CI pipeline.

Fig. 2: Questions of First Survey (Shortened)

Problem Statement. The survey results show that deviations from CI best practices happen in practice. Most participants state that a project can intentionally deviate from CI best-practices (67.7%) and even more participants agree that a deviation might be unintentional (77.3%). The majority of participants (77.4%) agree that the CI benefits diminish when many best-practice deviations exist. These answers confirm our conjecture that CI decay is a relevant problem in practice.

Relevance & Detection. Our survey contains questions about the practical relevance of each anti-pattern and we received a very high level of agreement. Six detectors have an agreement of >75% and other two have an agreement of >60%. The only notable exception is *Email-Only Notifications* for which 30.3% participants disagree with its relevance. These results confirm that we had pre-selected relevant anti-patterns.

We asked our participants to rate the sufficiency of our detection strategy for all anti-patterns. Also here, the agreement is very high (8, >60%, 2, >75%), with the notable exception being *Late Merging* (54.1%), for which many participants point out specific ways to use version control that would have not been detected. The high agreement makes us confident that we have successfully identified the "common case" for our detection. However, several people made use of the open question for each anti-pattern to provide feedback on the detection strategies, such as pointing out alternate development processes that we did not cover so far.

Revised Detection Strategies. On average, 43 participants answered the open question about each anti-pattern and we carefully analyzed these answers to revise or exclude some detectors. Next, we discuss the results of our open card sorting and how we revised our detection strategies based on the suggestions from the survey.

Late Merging & Aged Branches Many participants point out similarities between both anti-patterns. Also, the suggestions for improving the detection strategies overlap in our answers. As a result, we decided to merge both anti-patterns. The feedback contains valuable suggestions to improve our simplistic detection strategies: 1) a GIT-based detector must support both `merge` and `rebase` commands; 2) a feature

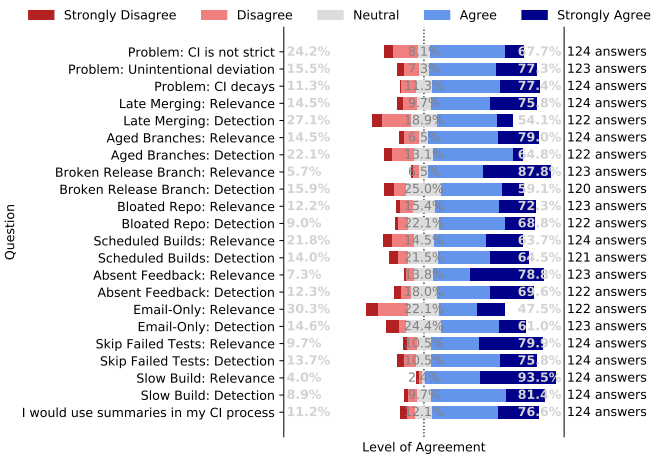


Fig. 3: Likert-Scale Answers to First Survey

branch should not be considered aged when no changes in other branches occur; 3) the age of a branch is irrelevant, as long as it is frequently synced with the master; 4) branches marked with release names, e.g., `rel-1.2.3`, should not be reported; 5) projects can release multiple times per day, so open branches frequently span several releases. Overall, we decided to revise our strategy to incorporate these suggestions and keep the *Late Merging* anti-pattern.

Broken Release Branch The survey participants broadly agree to the detection of the anti-pattern (87.8%). Several participants point out that a broken release branch should never happen, so everybody would be aware of it even without notification. In addition, we got some minor comments on our detection strategy, e.g., 1) providing an overview for incidents over time, 2) avoiding to consider the average time between commits as the commit frequency varies a lot. We decided to incorporate these suggestions into our detection strategy and to keep *Broken Release Branch*.

Bloated Repository Despite the high agreement among the participants (68.8%), we received several comments against the detection of this anti-pattern: 1) offending files are language and project specific, but can be easily fixed with a well-defined `.gitignore` file; 2) this anti-pattern is not CI specific; 3) several participants mention good reasons to include binaries, e.g., the availability, reliability, and convenience of provisioning sources. We agreed with these concerns and eliminated the anti-pattern candidate.

Scheduled Builds Despite a high agreement with the proposed detection strategy (64.5%), many survey participants point out good reasons for scheduled builds. These include, for example, running extensive performance tests, UI tests, or frequently asserting the compatibility with a changing environment. We could not distinguish between good and bad cases of scheduled builds, so we decided to drop this candidate.

Absent Feedback & Email-Only Notifications The agreement rate to the detection strategy of both anti-patterns is high (69.6% and 61.0% respectively). However, many participants state doubts regarding the detection feasibility: 1) feedback might be delivered in ways that cannot be automatically checked, e.g., physical build lights; 2) it is impossible to validate successful notification delivery; 3) the best notification channel is a personal preference. We agreed with these concerns and decided to drop both candidates.

Skip Failed Tests The detection strategy for this anti-pattern has a very high agreement rate (75.8%), but several participants mention good reasons to remove a test, e.g., removal of functionality. We believe that in these scenarios tests would either be removed together with production code, or the build would fail due to a compilation error. Both scenarios would not trigger our detector. Other participants point out that removing, commenting, and skipping tests have the same effect, so we should cover all of these cases. Apart from this, we did not receive further suggestions for improvement. We decided to keep this anti-pattern.

Slow Build Most participants agree with the detection strategy for this anti-pattern (81.4%). The main concern mentioned by several participants is the threshold that is used to identify slow builds. At the same time, previous work mention that a slow *creep* is the *worst-case scenario* for build times [11]. We kept this anti-pattern.

Overall, we received valuable feedback on all presented anti-pattern candidates. Following the suggestions of our participants, we dropped *Schedule Builds*, *Absent Feedback*, *Email-Only Notifications*, and *Bloated Repository* for the reasons mentioned above. We revised the detection strategies of the remaining anti-patterns *Late Merging* (which is now merged with *Aged Branches*), *Slow Build*, *Broken Release Branch*, and *Skip Failed Test* and kept them for the remainder of the paper.

General Feedback. The last part of the survey contains a general open question to provide feedback on the whole CI-ODOR idea, which was filled by all 124 participants. Most of them (30%) mention CI-ODOR as useful for CI training and for learning to adopt CI best practices rigorously also when developers are not familiar with CI yet [12]. Furthermore, 13% of our participants believe that CI-ODOR can reduce maintenance effort and improve reliability on the CI pipeline. As stated by 18%, some anti-patterns might go unnoticed without such a tool, which can be useful to monitor the CI health and take countermeasures when needed. 12% of the participants suggest to have highly-configurable detectors to support team/organization specifics in CI pipelines. Finally, 18% of the participants are quite skeptical about our detectors. As it happens with many quality check tools [33], their main concern is the likelihood of generating several false positives.

IV. Reporting CI Practices

To implement a proof-of-concept of CI-ODOR, our CI anti-pattern detector, we first chose supported technologies. We analyzed whether the perceived relevance of each smell varies across people working on different programming languages. Based on responses to our previous survey, a Kruskal-Wallis test [24] did not indicate, for any of the anti-patterns, a statistically significant difference among the four main programming languages the study participants reported as their main working

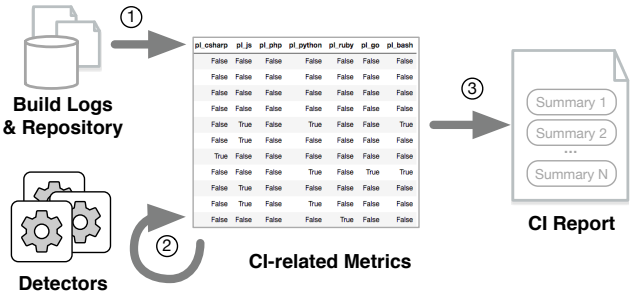


Fig. 4: CI-Reporting Process in CI-ODOR

language (p-value > 0.05), i.e., Java, JavaScript, Python, and Ruby. Thus, based on their popularity, we decided to focus on JAVA and MAVEN as our target programming language and respective build tool. To lower the likelihood of build-log deletion [29], we mined build data from TRAVISCI [27] and consequently repository data from GITHUB [7].

The overview of our reporting process is shown in Fig. 4. Given the TRAVISCI build logs of a particular project, we first extract CI-related *metrics* for every build (1). We then run detectors on top of this raw data (2) to derive additional metrics that either indicate the presence of a phenomenon (e.g., a test has been removed) or a change in a metric (e.g., a change in build time). From all metrics, we provide a reporting utility that visualizes several dimensions of the CI process, Figure 5 shows screenshots for the four different summaries that we include in our reports. Next, we discuss the details of the detection strategies for the four anti-patterns.

A. Slow Build

Figure 5a shows an example summary of *Slow Build* that contains the following items. 1) A bar chart highlighting the average build duration per week over the considered time window (3 months in our example and in the study of Section V). 2) A linear regression trend line, along with a textual message highlighting whether the build time is increasing, stable, or decreasing over the observed period. We also experimented with the use of kernel smoothing, but the resulting trend did not drastically change, so we favored simplicity here. 3) A list of possible warnings for the last builds of each branch. Specifically, we report: (i) a *Medium-severity* warning when a build was slower than 75% of more builds on the master branch, i.e., it is in the fourth quartile; (ii) a *High-severity* warning when the build duration is an outlier with respect to the distribution of master builds in the observed time window. We used the box and whisker plot outlier definition [28], i.e., a build is an outlier when its duration is greater than $3Q + 1.5 \cdot IQR$, where $3Q$ is the third quartile and IQR the inter-quartile difference.

In some cases, and this is especially true for the considered CI infrastructure (i.e.,TRAVIS-CI), the build time might depend on many external factors, including the priority given to the project (in TRAVIS-CI projects with a free account get a low priority). However, we do not consider this a threat in our measurements, because even in these cases CI-ODOR would highlight the need for using a better infrastructure.

B. Skip Failed Tests

We first extract the executions of all JUNIT tests and their outcomes from each build log, i.e., the containing MAVEN module, the test suite name, the number of executed tests, the number of failed tests (incl. test errors), and the number of skipped test cases. We then derive a set of test-related CI metrics by matching tests run in jobs (with the same *id*) belonging to consecutive builds on the same branch. Specifically, we compute Δ_{Runs}, i.e., a change in the number of executed tests, Δ_{Breaks}, i.e., a change in the number of failed tests, and Δ_{Skipped}, i.e., a change in the number of skipped tests. To mark a test as skipped (in the next build), we evaluate whether the following expression is true:

$$(\Delta_{\text{Breaks}} < 0) \wedge (\Delta_{\text{Runs}} < 0 \vee \Delta_{\text{Skipped}} > 0)$$

Fig. 5b shows an example summary that contains: 1) a bar chart depicting the number of builds per month affected by skip failed tests (note that we adopted a granularity of one month for this smell, due to its lower frequency than *Slow Build*); 2) for each build where such an incident occurred, the list of test suites affected by the skip failed tests issue.

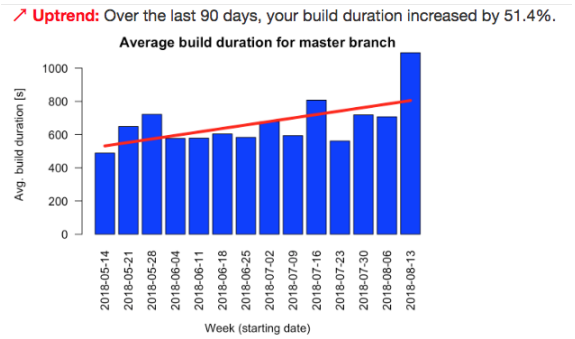
C. Broken Release Branch

To detect a broken release branch, we compute the final status of each `master` branch build, i.e., its *build status*, in the build history of a project. In particular, a build is *errored* when the *install* phase, which retrieves and installs the needed dependencies, returns a non-zero exit code. Instead, it is *failed* when any subsequent phase returns a non-zero exit code. We determine the final status of each build and consider all the errored and failed builds as broken. Fig. 5c shows an example summary that reports: 1) the average time a release branch remains broken over the observed time period, considering consecutive broken builds; 2) a bar chart showing, for each week of the observed period, the number of broken builds; 3) a linear-regression line and a textual message highlighting the presence of an increasing or decreasing trend, if any.

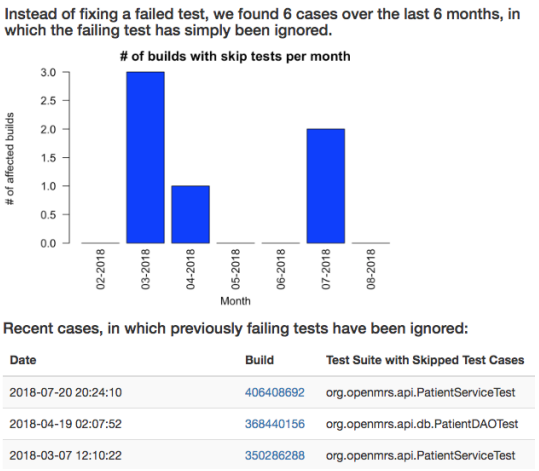
D. Late Merging

We consider four different metrics about version control that help us to identify the *Late Merging* anti-pattern: *Missed Activity*, *Branch Deviation*, *Branch Activity*, and *Branch Age*. In the following, we introduce the different metrics using the example history of Fig. 6, which contains a `master` branch and `f1` with several merge commits.

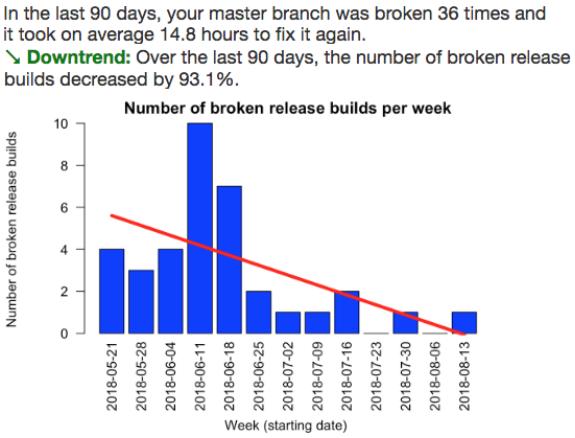
Missed Activity (t_{MA}). Quantifies the amount of activity on other branches of the same repository since the current branch was last synced with the `master`, $t_{\text{MA}} = t_{\text{LO}} - t_{\text{Sync}}$, where t_{LO} is the date of the last commit on other branches and t_{Sync} is the date of the last merge commit. If t_{MA} grows, the potential integration effort increases. To allow for more specific warnings in the summary, we break this metric further down into its two components *Branch Deviation* and *Unsynced Activity*.



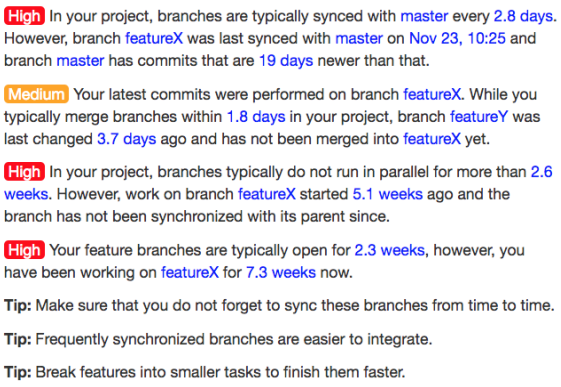
(a) Slow Build



(b) Skip Failed Tests



(c) Broken Release Branch



(d) Late Merging

Fig. 5: Example Summaries of the Four Anti-Pattern Detectors

Branch Deviation (t_{BD}). Quantifies the amount of activity in other branches since the last change in the current branch, $t_{\text{BD}} = t_{\text{LO}} - t_{\text{LC}}$, where t_{LO} is the last commit date on other branches and t_{LC} is the last commit date on the current branch. If t_{BD} grows, other branches deviate from the current branch, which again increases the potential integration effort. Negative values mean that the current branch is ahead of other branches.

Unsynced Activity (t_{UA}). Quantifies the amount of activity in the current branch since the last sync with the `master`, $t_{\text{UA}} = t_{\text{LC}} - t_{\text{Sync}}$. A growing t_{UA} indicates a deviation from the `master`, and that the potential integration effort increases.

Total Activity (t_{TA}). Quantifies the total amount of activity on a branch since its creation, $t_{\text{TA}} = t_{\text{LC}} - t_{\text{Fork}}$, where t_{Fork} is the date of the branch creation. Feature branches should be merged back into the `master` timely, a growing t_{TA} indicates a long-running deviation from the `master`.

When CI-ODOR *raises a warning.* For each of the four metrics mentioned above, we compare their values with distributions in recent history and consider a *Medium-severity* warning if a value is above the third quartile, a *High-severity* warning if it is an outlier (using a similar approach to the one in Section IV-A).

History Rewrite. GIT history can be rewritten, which makes it harder to analyze [2]. We include a second branch `f2` in our example to illustrate our handling. We detect rebasing in build logs by matching the meta-data of a commit that is built (*id*, *time*, *committer*, *message*) to meta-data of previous builds on the same branch. When all meta-data but the id can be matched to a previous commit, we mark this as a rebasing. In the example, the rebasing of (4) triggers a new build of (4') at the date t_{Sync}; t_{Fork} is the date at which the first build of this branch was triggered. Now all derived metrics can be calculated as for the previous merge case.

Improved Detection Strategies. We consider two suggested improvements for the detection strategy. First, in addition to analyzing the build logs, we also analyze the current repository snapshot for every build to identify deleted branches that do not need to be reported anymore. Second, we filter out branches that mark releases, e.g., `rel-1.2`.

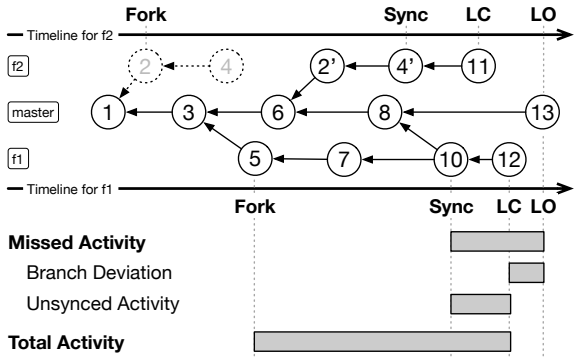


Fig. 6: Example of Different *Late Merging* Scenarios

V. Empirical Assessment of the CI-Odor Summaries

We conducted a study on open source software projects to assess the accuracy and usefulness of our reporting. We first performed a selection of candidate projects. After detecting anti-patterns on such projects, we sent the generated summaries to the mailing lists/forums of said projects and we asked original developers to fill out a survey. In addition, we also asked participants to the study of Section III to answer the part of the survey concerning the usefulness of the summaries and of CI-Odor in general.

A. Projects Selection

We used data from GHTorrent [8] (version Apr-01-2018) to identify suitable projects for our study. We filtered projects according to the following criteria: they are written in Java, have not been deleted, have at least one commit in 2018, have at least two project members, are no forks, and have been forked at least once. This initial filtering left us with a set of 2,155 project candidates.

To find projects in this set that perform CI and that are compatible with CI-Odor, we required the existence of configuration files for Maven (`pom.xml`) and TravisCI (`.travis.yml`), which further reduced our candidates to 467 projects. We then excluded projects with less than five project members to ensure a certain community size.

We then extracted build logs from TravisCI for the remaining 103 project candidates. For some projects, we could not access the logs or only found a very limited number. As CI-Odor is based on historical analyses and to ensure a minimum level of activity, we excluded the first quartile from the distribution of available build logs for these projects, which left us with 70 candidates that have had at least 54 builds in 2018.

As the final step, we manually identified the main communication channels for all remaining projects, because we need to contact the corresponding developers. Keeping only these projects, for which we found a public mailing list or could join a closed group channel (such as Google Groups or Slack), we ended up with a final selection of 36 projects for the validation. These projects cover various domains like business-oriented software, image processing, development tools and have a diverse sizes (from 2 thousand to 12 million LOC), ages (from 457 to 25 thousand commits), activity levels (from 60 to 2 thousand builds), team sizes (from 7 to 385 members with a median number of 60.5), and popularity (from 15 to 26 thousand GitHub stars). The full list of these projects on the artifact page of this paper [30].

TABLE I: Detected CI Anti-patterns over the 36 Analyzed Projects

Slow Build					
Proj. with incr. trend					20
Proj. with decr. trend					11
Proj. with stable trend					5
Overall # of medium sev. warnings					4,634
Overall # of high sev. warnings					229
	Min	1Q	Median	3Q	Max
% of medium sev. warn.	0%	16.80%	25.67%	35.76%	93.87%
% of high sev. warn.	0%	0.44%	1.20%	2.95%	26.73%
Broken Release Branch					
Affected projects					36
Total # of incidents					3,423/18,474
Proj. with incr. trend					16
Proj. with decr. trend					19
	Min	1Q	Median	3Q	Max
% of incidents	0.31%	6.46%	11.51%	28.69%	51.10%
Fixing time	54.18 m	9.44 h	17.07 h	3.10 d	6.04 w
Skip Failed Tests					
Affected projects					15
Overall # of detected incidents					56
	Min	1Q	Median	3Q	Max
% of affected builds	0.17%	0.24%	0.65%	1.35%	2.47%
Late Merging					
Affected projects					35
# of medium severity warnings					63
# of high severity warnings					115
	Min	1Q	Median	3Q	Max
# of affected branches	1	1.5	2	4	20
% of affected branches	25.00%	46.28%	66.67%	100.00%	100.00%

B. Quantification of the Phenomenon

This section provides a short overview of the anti-pattern instances and CI decay for the 36 projects for which we asked for feedback, with the goal of highlighting the magnitude of the investigated phenomenon. The analysis concerns a total of 18,474 builds from January 1, 2018 to August 15, 2018. This results in 8,520 detected incidents, 3,823 if we consider only high-severity warnings for *Slow Build* and *Late Merging*.

Concerning *Slow Build*, 20 projects exhibit an increasing trend in build time, whereas only 11 had a decrease, and 5 were stable. The percentage of cases in which a medium-severity warning could be generated is fairly high, with a median of 25% of the builds and a maximum (*authorjapps/zerocode*), where nearly all builds (93%) are slower than the third quartile of the previous time window. This indicates a slow increase in the build time, which can be normal project evolution. A single incident might not be worrisome *per se*, so we visualize the overall trend (see Fig. 5a). High-severity warnings (i.e., outliers) are not particularly frequent (75% of the projects have less than 3% of their builds exhibiting this warning), indicating that while the *Slow Build* phenomenon is quite pervasive, in most cases it manifests quite slowly over time.

Even though working on `master` is discouraged and a pull request paradigm has been advocated [9], we find *Broken Release Branch* in all projects: a median of 11.51% of the `master` builds are broken. At the same time, our data indicates that breaks are typically fixed within one day (i.e., the median

Concrete Questions About a Report (for original developers)

- The report is useful for my project and contains relevant warnings.
- I learned something about my project that I have not been aware of before.
- The results made me curious and I plan to investigate the different warnings.

General Questions About Usefulness of Examples

Slow Build, Failed-Test Skipping, Late Merging:

- This summary helps me to identify anti-pattern X.
- I know how to address the different warnings about anti-pattern X.
- High-severity warnings about anti-pattern X should fail the build.

Broken Release Branch:

- This summary improves awareness about...
  - ... the frequency of release-branch failures.
  - ... the time it takes to fix release-branch failures.
- I know how to improve the trend of this summary in the future.

General Validation of Tool and Idea

- The CI report provides information that is not available in any other tool.
- The reports provide a good overview of the CI practices used in a project.
- Frequent reports would have a positive influence on CI practices.
- I would like to integrate such a reporting in my own CI pipeline.

Fig. 7: Questions of Second Survey (Shortened)

is about 17h), even though the median fix time is above 3 days for the upper quartile of projects. We found one project (*rackerlabs/blueflood*) for which the `master` branch remained broken on average for over 6 weeks.

Skip Failed Test is the least prominent problem in the analyzed project histories. We found instances of this smell for 15 out of the 36 projects, in a total of 56 builds. The percentage of builds affected by this anti-pattern is below 2.5%. While instances of the anti-pattern can be found, developers seem to take failed tests seriously and do not skip them.

Concerning *Late Merging*, the anti-pattern affected nearly all projects (35 out of 36), and we raised a total of 115 high severity warnings, and 63 medium severity warnings. The median number of branches affected by a warning is 2, and only in one project *Evolveum/midpoint* the problem affected 20 branches, although the median percentage of affected branches is quite substantial (66.67%).

C. Survey on Generated Reports

We have conducted a second survey study to validate the usefulness of our generated reports.

Survey Design. To perform the study, we designed a questionnaire composed of a demographics section plus three sections, each one comprising Likert-scale questions and a field for open comments. In the first section, we asked original developers about the report that we have generated for their project. This part was automatically skipped for developers that have not seen a report. In the second section, we introduced the four different detector categories through an exemplary screenshot. We then ask questions about the understandability of the summary, its actionability, and whether detected deviations should fail the build (if applicable). A final section of the survey contained general questions about the usefulness of the presented summaries. An excerpt (shortened questions, no demographics) of the survey questionnaire (the complete one is on our artifact page [30]), is depicted in Fig. 7.

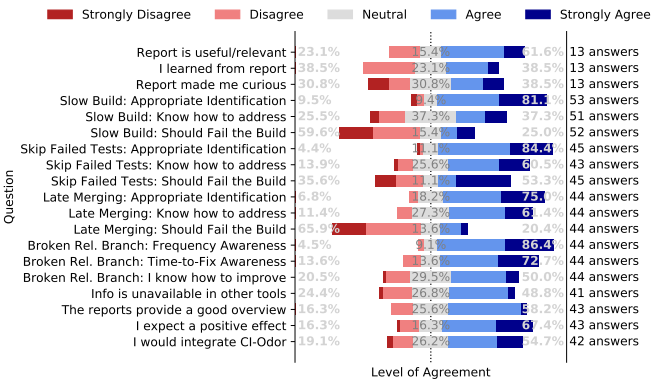


Fig. 8: Likert-Scale Answers to Second Survey

Advertisement. We had two separate advertisement strategies to find study participants. To find *original developers*, we have created up-to-date summaries for our selected target projects and posted them on their corresponding communication channels, asking project members for feedback. At the same time, to receive enough *general feedback* about our summaries, we advertised the survey on TWITTER, REDDIT (targeting the same sub-forums of the previous survey) and we also sent a follow-up email to every participant of our first survey that allowed us to contact her again.

Demographics. In the end, 113 developers opened our survey, out of which 50 answered all questions (11 original and 39 general developers), resulting in a completion rate of 44.2%. We cannot calculate the return rate for the general population, but we know that we sent the reports out to 36 projects and we heard back from 7 projects (return rate of 19%). We included the control questions from our first survey again and excluded from the analysis 7 participants that indicated low experience. To increase the number of answers, we also kept partial answers. In total, we considered 42 answers as valid, out of which 13 were given by original developers.

Data-Analysis Methodology. As for the Likert-scale answers in the first survey (see Section III), we present the results in asymmetric stacked bar charts (Fig. 8). We have received considerably less open answers, so we did not open-code all answers, but we will rather discuss the main points raised.

Report Rating. When we asked original developers about the usefulness of the reports for their projects, almost two-thirds (61.6%) of them agree that the report was useful and contained relevant information. We analyzed the open answers of the 23.1% that disagree and found that most either complain about project specifics that are not considered in the reports or about a bug that we had in the beginning. We got mixed answers about the novelty of information, the same amount (38.5%) of agreements and disagreements. This could be a sign that experienced developers are aware of these deviations in their project, but we did not have a question in the survey to back up this conjecture. We got a similar result when we asked about their reaction. 38.5% of original developers agree that the report made them curious and that they plan an investigation. Overall,

we see these results as a sign that the reports are insightful for developers and introduce a minimal overhead.

Identification and Awareness. Across all anti-pattern detectors, the large majority (*Slow Build*: 81.1%, *Skip Failed Tests*: 84.4%, *Late Merging*: 75%, *Broken Release Branch*: 86.4% and 72.7%) of participants agree that the summaries are useful for anti-patterns identification and for increased awareness respectively. This makes us confident that we opted for the right statistics and picked good visualization strategies.

Actionability. The majority of participants agree that they know how to address the warnings for *Skip Failed Tests* (60.5%) and *Late Merging* (61.4%), which is unsurprising because the report points to concrete problems to fix. However, a considerable number of participants disagree for *Broken Release Branch* (20.5%) and especially for *Slow Build* (25.5%). Given the high agreement on *Identification and Awareness*, we think that the disagreement on actionability is a sign that the report unveils the problem, but that deriving a fix is harder because it affects the process and team practices.

On Using CI-ODOR *to Fail Builds.* The majority of participants disagree with the idea that a build should fail when it is slow (59.6%) or when signs of a *Late Merging* exist (65.9%) and only a small group agrees with this idea (25% and 20.4%). Although build failures provide feedback about issues in newly committed code such as bugs [1] or poor quality [34], our participants typically do not want detected anti-patterns in the CI process to break the build. The only exception is represented by *Skip Failed Tests*, where 53.3% are in favor of failing the build. From the open answers in the first survey, we know that some developers see this anti-pattern as a serious problem.

General Validation. The last part of the survey contained several statements about the validity of the reports and the general idea. 58.2% of the participants agree that the summaries are useful and that they contain relevant information for the project. 67.4% expect a positive effect from integrating our tool to their CI discipline and 54.7% are willing to integrate CI-ODOR in their pipeline.

VI. DISCUSSION

This paper has introduced the idea that monitoring the CI process might be useful to discover the decay of best practices over time. Building a proof-of-concept implementation, CI-ODOR, and surveying developers about the idea and our tool, we gained several valuable insights into the perceived or expected benefits of such an approach and actionable findings that have an impact on future work.

Positive Effect & Awareness. The first survey has shown us that anti-patterns are a relevant problem for CI. Best practices are not always being followed, and can even be accidentally broken. Almost two-thirds of the participants to the second survey expect that using such a reporting frequently would have a positive influence on their CI discipline.

Transparency. The study participants suggested that the tool should make the detection strategy fully transparent to increase the trust and acceptance among its users. We only briefly described the detection rules in the summary pages, to allow study participants performing their task efficiently, but a production-ready tool could involve a fully-fledged description of the detectors.

Learnability. Participants of our first survey confirmed the usefulness of the proposed CI monitoring, especially in the early stages of CI adoption or to train project newcomers. Nearly half of the participants of our second survey were already aware of most of the highlight problems, but it is important to remark that our analysis excluded inexperienced developers about CI. The potential of a regular CI report can be seen by the 38.5% of participants that got curious from the report and started to investigate the reported issues.

Configurability. Our first survey indicated that since projects are very different, developers may want the reports and the detection thresholds to be customized based on their needs. While half of our participants are willing to integrate an anti-pattern detector in their pipeline, this percentage could increase by enhancing usability and reconfigurability, e.g., giving the freedom to enable/disable specific detectors or configure thresholds.

Ultimately, the important question is how useful a concrete report is, therefore we have asked original developers to rate the report that we have generated for their project. The low disagreement rate in this question (23%) and the high agreement for the integration in their own pipeline (55%) make us confident that the described reporting is a promising tool for software development teams that follow CI principles.

A. Threats to Validity

Threats to *construct validity* are related to the relationship between theory and observations. They mainly concern:

Implementation Bugs. Our implementation might contain bugs that cause the tool to report false positives or false negatives. While the absence of a labeled dataset made it hard to evaluate the detection strategies, we mitigated this threat through a manual review of a sample of the generated results and through a pilot study conducted before the second survey.

Build Cleanup. TRAVIS-CI allows projects to delete their old build logs, e.g., as part of regular maintenance activities, and this could affect the historical analyses our tool performs. However, previous work has shown that such deletions are unlikely [29], so we do not expect that our results are significantly affected.

Threats to *internal validity* are related to confounding factors internal to our study. In particular, the validation of our summaries could be affected by our selection of projects. We have mitigated this by selecting a diverse set of projects from GITHUB and ensured a certain level of maturity by considering the popularity of a project. Unfortunately, our data source, GHTORRENT, only approximates the number of committers in a project, which might result in a less diverse project sample.

Threats to *external validity* concern the work's generalizability and are related to:

Sample Size and Diversity. Our evaluation would surely benefit from the analysis of a larger sample of projects, as well as of the participation of more developers. In this work, the analysis of projects was limited to the ones for which we ask for feedback, and the technological limitations (Java, MAVEN) do not affect the relevance of the detected anti-patterns, as explained in Section IV.

Irrelevant Selection of Anti-Patterns. It is possible that our work missed anti-patterns that are highly-relevant for developers. We have reduced this threat through a first internal pre-selection of supported anti-patterns, which we validated in our first survey. Also, it is important to remark that it is not our goal to provide a comprehensive detector for all anti-patterns proposed by Duvall [4], but rather to propose and validate the idea — and its implementation in CI-ODOR — of reporting CI decay to developers.

B. Future Work

Add Additional Detectors. While, as stated above, we validated the general CI-ODOR perspective and four relevant anti-patterns, our future work primarily goes into providing additional detectors for new anti-patterns.

Consider More Contextual Information. Right now, we only leverage information from the TRAVIS-CI logs and basic information from the code repositories. However, future work would integrate additional process-related metrics derived from other sources like bug trackers, task management systems, or communication platforms.

Support More Project-Specific Policies. A CI-supporting tool with smart capabilities could learn the problems/warnings in which developers are interested in, and personalize the recommendation consequently.

Derive Project-Specific Thresholds. While we considered thresholds based on consolidated statistics, future work could also consider adaptive, project-specific threshold learning and calibration.

VII. RELATED WORK

Researchers have investigated the CI adoption [12], [14], [15], [26], finding, in particular, numerous barriers for CI adoption [11], e.g., related to assurance, security, and flexibility in performing tasks such as source code debugging. In such a context, approaches like CI-ODOR can be used to help developers understanding when they are not using CI properly.

Previous work has investigated best practices while using CI [35]. In their landmark work, Duvall et al. [3] identified principles and key practices of CI but also pointed out the risks deriving from the misuse of CI. Furthermore, Humble and Farley [13] performed a broader study, analyzing the key ingredients of a Continuous Delivery (CD) pipeline, as well as anti-patterns to be avoided. Such anti-patterns were better explained in the follow-up work by Duvall [4] where all the practices contained in books about CI [3] and CD [13] were condensed in a catalog of bad/good practices regarding the adoption of the whole CD pipeline with specific focus on the core part of CD, i.e., CI. Such a catalog is a comprehensive set of 50 patterns and anti-patterns regarding several phases or relevant topics in the CI/CD process. As explained in Section III, Duvall's catalog constitutes the inception of our work.

One of the best practices associated with CI is the use of Infrastructure as Code (IaC) in order to implement the desired pipeline. Sharma et al., leveraged best practices associated with code quality management to assess configuration code quality and proposed a catalog of 13 implementation and 11 design configuration smells [23]. Recent work by Gallaba et al., [6] also investigated configuration smells and based on rules provided by linters (e.g., TRAVISLINT) they measure smells and derive automated fixes for them. Our smells have a different focus, because we look at process-related smells rather than at configuration issues.

Rahman and Williams [21], [20] proposed a text-mining approach to identify defective IaC scripts, focusing on security and privacy issues, e.g., related to file permissions or user accounts. Their work is complementary to ours as it deals with a very specific category of problems.

Studying and proposing automated fixed for build failures has also been a topic of investigation. Previous work has investigated the phenomenon of build failures [22], [32] from different perspectives, such as testing [1] and code analysis [34]. Also, researchers have proposed fixes for some kinds of build failures, e.g., broken dependencies related [16], or proposed approaches to augment the comprehensibility of build logs while inspecting the cause of such failures [31]. Our CI smells detector increases the awareness of developers about problems degrading their current CI practice. Given that our smells are just symptoms of bad practices we do not provide any automated fix for such issues but we let developers decide whether taking action or not.

VIII. SUMMARY

This paper investigates the phenomenon that CI development practices decay over time. We survey 124 developers (80% from industry) to understand the problem. Beyond agreeing on the problem relevance, our respondents also confirm that CI anti-patterns are a major cause for the degradation of CI processes. To support developers in preventing CI pipeline from deteriorating we propose CI-ODOR, an automated reporting tool of CI anti-patterns. We validate our approach, CI-ODOR, by surveying 13 original developers about summaries for their projects, and by asking another 42 developers about the general usefulness. The results show that CI-ODOR increases the awareness about CI anti-patterns, is perceived as useful, and that developers would integrate it into their pipeline.

IX. ACKNOWLEDGMENTS

We would like to thank all the study participants. C. Vassallo and H. C. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275). C. Vassallo also acknowledges the student sponsoring support by CHOOSE, the Swiss Group for Software Engineering.

References

[1] M. Beller, G. Gousios, and A. Zaidman. 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, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 356–367, 2017.

[2] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009.

[3] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[4] P. M. Duvall. Continuous delivery: Patterns and antipatterns in the software life cycle - https://dzone.com/refcardz/continuous-delivery-patterns. *DZone refcard #145*, 2011.

[5] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[6] K. Gallaba and S. McIntosh. 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, 2018.

[7] GitHub APIs. https://developer.github.com/v3/. Accessed: 2018-02-08.

[8] G. Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 233–236, 2013.

[9] G. Gousios, M.-A. Storey, and A. Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 285–296, New York, NY, USA, 2016. ACM.

[10] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 34th International Conference on Software Engineering*, pages 441–451. IEEE Press, 2012.

[11] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 197–207, 2017.

[12] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 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)*, pages 426–437, 2016.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[14] S. Kim, S. Park, J. Yun, and Y. Lee. Automated continuous integration of component-based software: An industrial experience. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 423–426. IEEE Computer Society, 2008.

[15] E. Laukkanen, M. Paasivaara, and T. Arvonen. Stakeholder perceptions of the adoption of continuous integration–a case study. In *Agile Conference (AGILE), 2015*, pages 11–20. IEEE, 2015.

[16] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 106–117, 2018.

[17] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[18] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.

[19] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? A study on developers' perception of bad code smells. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 101–110, 2014.

[20] A. Rahman. Characteristics of defective infrastructure as code scripts in devops. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 476–479, New York, NY, USA, 2018. ACM.

[21] A. Rahman and L. Williams. Characterizing defective configuration scripts used for continuous deployment. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 34–45, 2018.

[22] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 345–355, 2017.

[23] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 189–200, 2016.

[24] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[25] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[26] D. Ståhl and J. Bosch. Automated Software Integration Flows in Industry: A Multiple-case Study. In *International Conference on Software Engineering (Companion)*, pages 54–63, 2014.

[27] Travis-CI. https://travis-ci.org. Accessed: 2018-02-08.

[28] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[29] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 805–816, 2015.

[30] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta. Artifact Page - Automated Reporting of Anti-Patterns and Decay in Continuous Integration. https://doi.org/10.5281/zenodo.2566032.

[31] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Un-break my build: assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 41–51, 2018.

[32] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 183–193, 2017.

[33] F. Wedyan, D. Alrmuny, and J. M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *2009 International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE, 2009.

[34] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[35] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 60–71, Piscataway, NJ, USA, 2017. IEEE Press.