Mitigating Waste That Tacitly Accrues in Continuous Integration Pipelines

Nimmi Rashinika Weeraddana
University of Waterloo, Canada
nrweeraddana@uwaterloo.ca

***Abstract*—Continuous Integration (CI, i.e., the automatic build and test cycle applied to the change sets that development teams produce) has become a standard practice of modern software development. CI enables rapid feedback on code changes and fosters seamless integration in collaborative environments. While CI adoption enhances productivity and software quality, compute resources that enable CI are a shared commodity that organizations need to manage. As projects evolve, the complexity of CI pipelines introduces potential inefficiencies, such as prolonged CI build durations and frequent build restarts. Much of these inefficiencies are explicit, where developers are consciously aware of the overuse of CI resources, i.e., build time. However, there are also tacitly wasted CI resources that accumulate unnoticed. For example, inefficiencies in the CI environment, CI configurations, or dependencies can gradually extend CI build times and increase resource consumption. In this thesis, we focus on tacitly accrued CI waste, hypothesizing that neglecting such inefficiencies depletes CI resources substantially. In particular, we aim to quantify and characterize this waste by examining inefficiencies in the CI environment, CI configurations, and dependency configurations, and provide solutions to mitigate such waste.**

***Index Terms*—continuous integration, timeouts, ci smells, unused dependencies.**

I. Introduction

Continuous Integration (CI) is a software development practice where developers frequently merge their change sets into a shared codebase through a Version Control System (VCS), triggering automated builds and tests to ensure that the codebase remains stable [7], [18]. CI builds are automatically triggered by events such as code commits (i.e., submission of change sets to VCS), and they can also be initiated manually by developers. By using CI, project maintainers aim to mitigate the challenges of late-stage code integration, known as "integration hell," by enabling continuous feedback and supporting agile and DevOps methodologies [7], [42].

Fig. 1 illustrates the CI workflow. The workflow starts when a developer (or a bot) submits a change set to the VCS, triggering a CI build in the **CI environment**. This environment can either be an on-premise setup specific to a project or a cloud-based environment provided by a CI service provider (i.e., third-party companies). It includes the necessary infrastructure to execute a CI build, such as hardware, software, and network resources. For instance, hardware resources in a CI environment can include servers for executing tasks and storage for archiving build artifacts like binaries and logs. The settings and instructions for executing a CI build (e.g., which branches or events trigger builds, environment variables, and secure management of secrets) are specified in **CI configuration** files. For CircleCI (a popular CI provider), CI configurations have to be specified in the `configuration.yml` file within the `.circleci/` directory. Similarly, the CI configurations for GitHub Actions—another CI provider—are specified in `.yml` files in the `.github/workflows/` directory, with each file containing multiple steps to automate a CI build. A CI build usually includes tasks such as cloning the codebase, downloading dependencies that are listed in **dependency configuration** files, compiling the code, and running automated tests. A CI build may also include steps such as linting to enforce coding standards. When the tasks of a CI build are successfully completed a, the outcome of CI build is marked as `Passed`; otherwise, it is marked as `Failed`.

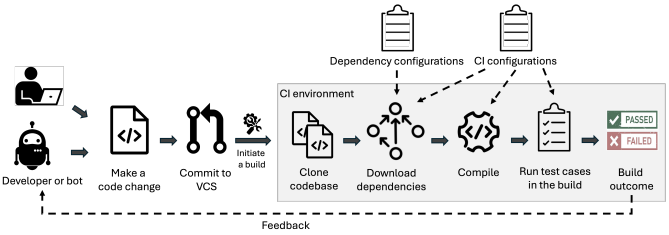

Fig. 1. An overview of a typical scenario where a CI build is triggered and the feedback is sent to the developer.

Due to the immediate feedback provided by CI on the change sets submitted to the VCS, CI is known to offer several benefits for its consumers, i.e., the maintainers and developers of projects that use CI [8], [17], [33], [42]. For example, CI is known to improve developer productivity [32] and software quality [17], [33].

However, as projects evolve and their CI pipelines grow in complexity and scale, CI pipelines can become susceptible to inefficiencies that negatively impact the effectiveness of CI pipelines. For example, flakiness is an inefficiency in CI, where CI builds triggered on the same change set intermittently pass or fail without a consistent outcome [15], [30]. To address the uncertainty caused by flaky builds, developers often restart CI builds after a failure in an attempt to determine whether the failure is caused by flakiness [6], [24]. While such restarts are intended to ensure the consistency of the build outcome, if the flakiness is not treated, restarted builds contribute to additional waste of CI build time and resources [24].

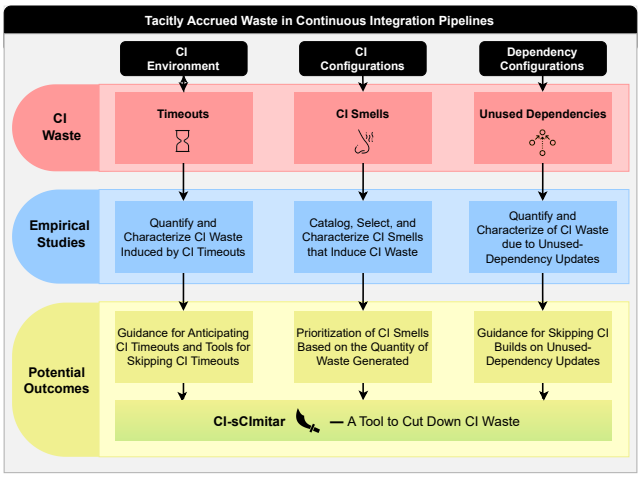

Fig. 2. An overview of the proposed thesis.

Much of the inefficiencies in CI pipelines that have previously been explored are explicit, where developers are aware of the waste of CI build time (e.g., restarted CI builds). However, there is also waste that tacitly (i.e., silently) accrued in CI pipelines due to inefficiencies that arise without developers' direct involvement. This thesis aims to understand the causes and implications of tacitly accrued waste in CI pipelines. Thus, we will use historical CI data to evaluate the following research hypothesis.

Research Hypothesis

Neglecting tacitly accrued CI waste leads to a substantial depletion of CI resources. By quantifying and characterizing forms of such CI waste, strategies can be developed to reduce CI waste and enhance the overall effectiveness of CI pipelines.

To evaluate this hypothesis, we explore tacitly accrued CI waste by examining inefficiencies in CI environments (SectionII), CI configurations (SectionIII), and dependency configurations (SectionIV). By systematically exploring these aspects, we aim to uncover the specific factors contributing to inefficiencies in Open Source Software (OSS), and provide insights for mitigating waste in CI. Fig. 2 shows an overview of the proposed thesis. Lastly, we hope to leverage the insights collected by the end of this exploration to propose a multi-faceted approach, CI-sCIMITAR (SectionV), to assist developers in cutting down tacitly accrued waste in CI pipelines.

II. CI Waste Due to Inefficiencies in CI Environments

CI builds may run into unforeseen issues in a CI environment, such as network problems[1] and prolonged waiting on external services. Such inefficiencies increase resource consumption. To mitigate this, CI providers often impose time limits on CI builds to prevent erroneous builds from consuming excessive resources in the CI environment. However, when a CI build exceeds this time limit, it is automatically terminated,[2] often after consuming a substantial amount of CI resources. While these limits are meant to protect the CI environment from abuse, they can result in the maximum allowable resources being consumed without providing any feedback on whether the build would have passed or failed [10], [13]. In this section, we explore the **prevalence** and **characteristics** of builds that lead to **CI timeouts**. This analysis would provide insight into potential improvements for CI pipelines to mitigate CI timeouts. Below, we provide a preview of our approach and results. An extensive version of this section appears in our published work [36].

A. Prevalence of CI timeouts

Approach. We begin with a dataset curated by Gallaba et al. [10], which contains CircleCI builds spanning 7,795 GitHub projects. We estimate the prevalence of CI timeouts by calculating their frequencies and comparing the durations of CI timeouts to signal-generating builds (i.e., CI builds with a `passed` or `failed` outcome).

Results. Out of 7,795 projects, 936 (12%) have at least one timeout build. The median number of timeout builds among these projects is four. However, 10% of the projects have 44 or more timeouts each, while an extreme 4% have 100 or more timeouts each, indicating a highly skewed distribution [27]. In addition, we find that the median timeout duration is 19.7 minutes, nearly five times longer than the median duration of signal-generating builds. This suggests that CI timeout builds are a substantial issue for GitHub projects.

B. Characteristics of CI timeouts

Approach. To investigate the factors that characterize the outcome of CI builds, we extract five families of features (such as build history, change set size, and timeout tendency) from 105,663 CI builds that span 24 most timeout-prone projects (accountable for 54% of all timeout builds) in Gallaba et al.'s dataset [10]. Then, we fit a statistical (logistic) regression model to our dataset, and analyze it. To assess the discriminatory power, we use the Area Under the Receiver Operating characteristic Curve (AUROC). We measure the explanatory power of each family of features by using Wald χ^2 tests [28].

Results. Our model has an AUROC of 0.939, which vastly surpasses that of a random predictor (AUROC = 0.5). Our Wald χ^2 tests reveal that the build history and timeout tendency features are strong indicators of CI timeouts, with the timeout status of the most recent build accounting for the largest proportion of the explanatory power. A longitudinal analysis (i.e., a study conducted over a period of time) shows that 64.03% of CI timeouts occur consecutively in projects, with a median interval of 24 hours before a passing or failing build. These findings suggest that build history and timeout tendencies can help anticipate CI timeouts.

[1]https://github.com/oblador/loki/issues/94

[2]https://support.circleci.com/hc/en-us/articles/360007188574

III. CI Waste Due to Inefficiencies in CI Configurations

CI configurations (i.e., the setup, build scripts, and settings)[3] are prone to anti-patterns, known as *CI smells* [3], [11], [34], [40]. CI smells can degrade the efficiency of CI builds, especially as projects scale. For example, configuring CI to use slow machines, disabling caching of large files and dependencies, and excessive logging are common CI smells. However, optimal CI configurations can vary across projects, as each has unique requirements and constraints. For example, while caching files can reduce CI build times, it may not be cost-effective for projects with tight budgets due to potential storage overages that incur additional charges by CI providers. In this section, we propose to measure and analyze the **impact of CI smells on CI build duration and cost**. Such a quantification would provide actionable insights to help developers optimize CI setups, enhancing resource utilization without compromising project budgets or scalability. Below, we discuss the proposed approaches and expected results.

A. Impact of CI smells on CI build durations

Proposed Approach. We aim to compare CI build durations before and after fixing CI smells in a set of open-source projects, and quantify the CI time improvement associated with each CI smell. Additionally, we hope to conduct a statistical regression analysis to assess how project characteristics, such as the average size of change sets and project activity levels, contribute to the CI build time savings achieved by fixing each smell.

Expected Results. We anticipate that addressing certain CI smells, like implementing optimized caching or reducing excessive logging, will significantly shorten build times. Through our regression analysis, we aim to identify project characteristics that drive greater CI time reductions after fixing these smells.

B. Impact of CI smells on the cost of CI

Proposed Approach. To examine the impact of CI smells on costs, we will estimate the build expenses before and after fixing CI smells. This analysis will consider variables such as the cost of using faster machines and caching. Then, we plan to analyze how CI costs can vary before and after fixing each CI smell that is considered in our study.

Expected Results. We expect that addressing certain CI smells may lead to reduced CI costs, while others may result in increased costs. For instance, using faster machines or implementing caching could initially increase costs—but would it ultimately reduce overall build expenses by decreasing build duration and frequency?

We hope that the findings of this proposed study on the impact of CI smells on build time and cost may provide insights into both universally applicable and project-specific CI configurations aimed at reducing build time and costs. We believe such recommendations would guide developers and project maintainers in making data-driven, cost-effective adjustments to their CI configurations based on project requirements.

[3]https://circleci.com/docs/configuration-reference/

IV. CI Waste Due to Inefficiencies in Dependency Configurations

Software projects frequently rely on numerous external dependencies listed in their dependency configurations, many of which may not be actively used in the project's codebase [21]. Although these unused dependencies are not essential for building and running the project, updating their versions in dependency configurations can still trigger CI builds. Such builds are wasteful because they do not impact the actual functioning of the software. In this section, we aim to quantify the **prevalence** and characterize the **sources** of CI waste due to **unused-dependency updates** to gain insights into mitigating such waste. Below, we provide a preview of our approach and results. An extensive description of the results of this study is available in our published work [37].

A. Prevalence of unused-dependency updates

Approach. We begin with a dataset of 13,991 JavaScript projects that use a dependency manager (i.e., `npm` in our case) and adopt CI. From these projects, we extract 121,453 commits that update dependencies in the `package.json` file (i.e., the dependency configuration file for `npm`). Next, by using the DepCheck tool,[4] we identify 49,731 of these as updating unused dependencies. Then, we retrieve CI data for these commits to analyze the impact of unused dependencies on build time. We use this dataset to quantify CI waste from the perspectives of CI providers, as well as CI consumers.

Results. We find that from the perspective of the CI provider, 55.88% (3,427 build hours) of the overall CI build time that is consumed by updates to `npm` dependency specifications in the studied projects is attributed to unused dependencies. At the project level, a median of 56.09% of CI build time is spent on updates to unused dependencies.

B. Sources of unused-dependency updates

Approach. We use our dataset of dependency updates to quantify CI waste by identifying *who* is generating unused dependency updates (i.e., bots or developers) and *which* types of dependencies tend to be affected (i.e., development or runtime).

Results. Our results show that a large proportion (92.93%) of the CI build time that is spent on unused dependencies is wasted due to bot-generated updates, with `Dependabot`[5] accounting for 74.52% of that wasted CI build time. With respect to the type of dependencies, the majority of the wasted CI build time (92.63%) occurs due to unused development dependencies, which are at lower risk of introducing field failures due to erroneous removal [5]. These findings suggest that CI waste can be reduced by targeting bot-generated updates and skipping wasteful builds, as the majority are triggered by unused, non-production dependencies.

[4]https://github.com/depcheck/depcheck

[5]https://github.com/dependabot

V. CI-sCIMITAR

In this section, we propose our tool, CI-sCIMITAR, which we plan to develop based on the insights gathered from our studies discussed in sections II, III, and IV. The aim of CI-sCIMITAR is to offer a comprehensive solution to enhance the efficiency and effectiveness of CI pipelines by minimizing waste at various stages of CI.

CI-sCIMITAR will be composed of three integral components. First, we introduce TIMEOUT-sCIMITAR, which will leverage machine learning to set suitable timeout limits for builds, and predict the likelihood of CI builds resulting in timeouts. Second, we introduce SMELL-sCIMITAR, where we hope to integrate the findings from our analysis of CI smells to detect smells that contribute to CI waste, and recommend optimized CI configurations, taking both CI time saved and CI cost spent into consideration. Lastly, we introduce DEP-sCIMITAR, a method designed to bypass unnecessary CI builds triggered by updates to unused dependencies made by developers as well as bots. We have successfully implemented this DEP-sCIMITAR component, and it is available as an `npm` package, and our published work [37] includes an evaluation.

Upon the full implementation of CI-sCIMITAR, we will select a significant sample of projects and re-execute the CI builds of their most stable branch with and without the tool. We will assess CI-sCIMITAR's effectiveness by measuring the build time saved. Besides, we will evaluate the additional build time savings achieved when CI-sCIMITAR is used alongside existing build acceleration tools [9].

VI. RELATED WORK

Long-duration CI builds are a common issue in maturing projects, leading to inefficiencies in the development process [9], [10], [12]. Ghaleb et al. [12] analyzed 104,442 CI builds from 67 open-source projects and found that builds that are re-executed multiple times are most prone to long durations. They also identified CI timeouts as a common consequence of lengthy builds, which is supported by other studies as well [10], [13]. CI timeouts waste resources and fail to provide useful feedback, which is explored in detail in this thesis.

CI smells, i.e., anti-patterns in CI configurations and processes are explored in other studies [11], [31], [34], [35], [39]–[41]. CI smells, if left unaddressed, can lead to degraded performance in CI pipelines. For example, Zhang et al. [41] cataloged five performance-related CI configuration smells in Travis CI, such as not using dependency caching and retrying failed steps within CI builds. While many of the existing studies catalog CI smells and propose fixes, quantifying the practical impact of these smells on CI waste would offer crucial insights into areas for improvement, which we hope to explore in this thesis.

Smells in dependency configurations may also waste CI time. While not caching dependencies, as suggested by Zhang et al. [41], is a smell in CI configurations, having unused dependencies listed in a project's dependency configurations is a smell in dependencies [21]. While caching dependencies can reduce the CI build time spent on installing unused dependencies [41], maintaining up-to-date versions of those dependencies may still trigger CI builds, wasting CI resources. In this thesis, we perform a detailed analysis of how unused dependencies impact CI processes.

Several approaches were proposed to **accelerate CI builds**. For example, incremental builds can accelerate the artifact-building phase by only rebuilding what is affected by code changes, such as Google's Bazel. However, they rely on manually specified build dependency graphs, which can drift out of sync or be incorrectly specified [9], [23], [25], [26]. Skipping CI builds also reduces CI build time [1], [2], [9], [20]. For example, Abdalkareem et al. [2] examined 1,813 commits where developers requested for CI builds to be skipped. This analysis identified reasons for skipping CI builds, such as changes only affecting non-compilable files like documentation. For these reasons, a rule-based method was proposed to automatically identify commits eligible for CI skipping. Other studies suggested predicting CI build outcomes to allow skipping likely-to-pass builds and offering early failure feedback before completion of a build [4], [19], [29], [38]. For example, Chen et al. [4] proposed BUILDFAST–an approach to predict CI build outcomes by using change-set and history-aware features, such as the status of the previous build and the total number of previously failed builds. Their predictor outperformed the state-of-the-art approaches by 47.5% in the F1 score for failed builds. Additionally, test-skipping methods [14], [16], [22] target specific tests without skipping entire builds, as shown by Gligoric et al. [14], who tracked test-file dependencies to skip tests unaffected by changes.

VII. CONCLUSION

In this thesis, we propose to investigate inefficiencies in CI that tacitly accrue waste, such as CI timeouts, CI configuration issues, and unused dependencies. The core hypothesis is that ignoring these wastes has a substantial drain on CI resources. To validate this, we analyze historical CI data to quantify these inefficiencies and propose strategies to mitigate them. Then, we propose CI-sCIMITAR, a tool designed to reduce CI waste and improve build efficiency, which will be evaluated for its impact on build times and cost.

This work advances the understanding of CI inefficiencies, offering actionable solutions for developers. Our findings on CI timeouts and unused dependencies are presented in sections II [36] and IV [37], respectively, while Section III proposes a study to measure the impact of CI smells on CI pipelines. Details of CI-sCIMITAR development are in Section V. I hope to finish the remainder of this research and complete my thesis by the end of August 2025. The timeline is outlined below.

Acknowledgment. I thank my supervisor, Prof. Shane McIntosh, for his immense guidance throughout this journey.

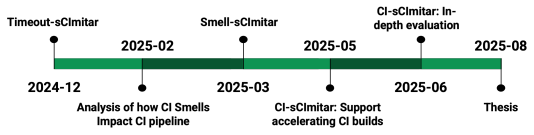

Fig. 3. Timeline for the completion of our study.

REFERENCES

[1] R. Abdalkareem, S. Mujahid, and E. Shihab, "A machine learning approach to improve the detection of ci skip commits," *Transactions on Software Engineering*, 2020.

[2] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be ci skipped?" *Transactions on Software Engineering*, vol. 47, 2019.

[3] W. J. Brown, H. W. McCormick III, and S. H. Thomas, *AntiPatterns and patterns in software configuration management*. John Wiley & Sons, Inc., 1999.

[4] B. Chen, L. Chen, C. Zhang, and X. Peng, "Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *Proceedings of the 35th International Conference on Automated Software Engineering*, 2020.

[5] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018.

[6] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, "Empirical study of restarted and flaky builds on travis ci," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

[7] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*, 2007.

[8] A. Erlandsson and H. Lantz, "Improving feedback loop by two-step continuous integration."

[9] K. Gallaba, J. Ewart, Y. Junqueira, and S. Mcintosh, "Accelerating continuous integration by caching environments and inferring dependencies," *Transactions on Software Engineering*, 2020.

[10] K. Gallaba, M. Lamothe, and S. McIntosh, "Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI," in *Proc. of the International Conference on Software Engineering*, 2022.

[11] 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*, vol. 46, no. 1, 2018.

[12] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, 2019.

[13] T. A. Ghaleb, D. A. Da Costa, Y. Zou, and A. E. Hassan, "Studying the impact of noises in build breakage data," *IEEE Transactions on Software Engineering*, vol. 47, 2019.

[14] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[15] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Le Traon, "What made this test flake? pinpointing classes responsible for test flakiness," in *International Conference on Software Maintenance and Evolution*. IEEE, 2022.

[16] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, vol. 1, 2015.

[17] 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 international conference on automated software engineering*, 2016.

[18] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill, "Advantages and disadvantages of a monolithic repository: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018.

[19] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[20] ——, "Hybridcisave: A combined build and test selection approach in continuous integration," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, 2023.

[21] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in npm," in *37th International Conference on Automated Software Engineering*, 2022.

[22] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 2019.

[23] C. Macho, S. McIntosh, and M. Pinzger, "Predicting build co-changes with source code change and commit categories," in *23rd international conference on software analysis, evolution, and reengineering*, vol. 1. IEEE, 2016.

[24] R. Maipradit, D. Wang, P. Thongtanunam, R. G. Kula, Y. Kamei, and S. McIntosh, "Repeated Builds During Code Review: An Empirical Study of the OpenStack Community," in *Proc. of the International Conference on Automated Software Engineering*, 2023.

[25] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014.

[26] ——, "Identifying and understanding header file hotspots in c/c++ build processes," *Automated Software Engineering*, vol. 23, 2016.

[27] K. Pearson, "Contributions to the mathematical theory of evolution," *Philosophical Transactions of the Royal Society of London. A*, vol. 185, 1894.

[28] J. N. Rao and A. J. Scott, "The analysis of categorical data from complex sample surveys: chi-squared tests for goodness of fit and independence in two-way tables," *Journal of the American statistical association*, vol. 76, no. 374, 1981.

[29] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Information and Software Technology*, vol. 128, 2020.

[30] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[31] R. B. T. Silva and C. I. Bezerra, "Analyzing continuous integration bad practices in closed-source projects: An initial study," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020.

[32] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *Proceedings of the 12th IASTED International Conference on Software Engineering*, 2013.

[33] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 10th joint meeting on foundations of software engineering*, 2015.

[34] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *41st International Conference on Software Engineering*, 2019.

[35] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, "Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[36] N. Weeraddana, M. Alfadel, and S. McIntosh, "Characterizing timeout builds in continuous integration," *IEEE Transactions on Software Engineering*, 2024.

[37] N. R. Weeraddana, M. Alfadel, and S. McIntosh, "Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, 2024.

[38] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *31st international conference on software engineering*, 2009.

[39] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in *International Conference on Software Maintenance and Evolution*, 2021.

[40] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, 2020.

[41] C. Zhang, B. Chen, J. Hu, X. Peng, and W. Zhao, "Buildsonic: Detecting and repairing performance-related configuration smells for continuous integration builds," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[42] 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 *32nd International Conference on Automated Software Engineering*, 2017.