Improving the Robustness and Efficiency of Continuous Integration and Deployment

Keheliya Gallaba*
McGill University, Montréal, Canada
keheliya.gallaba@mail.mcgill.ca

***Abstract*—Modern software is developed at a rapid pace. To sustain that rapid pace, organizations rely heavily on automated build, test, and release steps. To that end, Continuous Integration and Continuous Deployment (CI/CD) services take the incremental codebase changes that are produced by developers, compile them, link, and package them into software deliverables, verify their functionality, and deliver them to end users.**

While CI/CD processes provide mission-critical features, if they are misconfigured or poorly operated, the pace of development may be slowed or even halted. To prevent such issues, in this thesis, we set out to study and improve the robustness and efficiency of CI/CD.

The thesis will include (1) conceptual contributions in the form of empirical studies of large samples of adopters of CI/CD tools to discover best practices and common limitations, as well as (2) technical contributions in the form of tools that support stakeholders to avoid common limitations (e.g., data misinterpretation issues, CI configuration mistakes).

I. Introduction

Continuous Integration (CI) [1] is a software development practice where changes to a codebase are integrated into upstream repositories after being built and verified by an automated workflow. Continuous Deployment (CD) takes this a step further, ensuring that the software can be reliably released at any time by automating the deployment and release workflows as well. Prior work [2]–[5] has shown that CI/CD is broadly adopted by open source and proprietary software teams (e.g., Mozilla, Google, Microsoft, and ING). The adoption of CI/CD has been linked to increased developer productivity [6], speeding up development [7], and improving software quality [4], [8].

Due to the popularity of CI/CD, cloud-based CI/CD service providers (e.g., Travis CI, CircleCI, Jenkins, and AppVeyor) have also emerged, making CI/CD available to the masses. However, teams often encounter difficulties when adopting CI/CD in their organizations leading to unstable software delivery pipelines and wasted resources. For example, suboptimal configuration of Mozilla's CI service was inflating the operating cost of their CI service by 16%.[1] Moreover, a typo in the CI specification of the *geoscixyz/gpgLabs*[2] project halted deployment of new releases.[3] In our work, we tackle three problems that relate to CI/CD services:

Misconfiguration of CI/CD Environments. Improperly configuring CI/CD environments may lead to suboptimal performance (e.g., violating the semantics of CI/CD specifications hinders the runtime optimizations that CI/CD service providers can perform) or conceal faults (e.g., misspelled properties and their associated commands are silently ignored by popular CI/CD service providers such as Travis CI). To study how real CI/CD workflows are configured, we conduct empirical studies of use and misuse of features in CI/CD specification files [9]. Based on trends from our empirical studies, we propose tools[4] to detect and remove instances of feature misuse. Furthermore, we offer recommendations to guide future development and maintenance of CI/CD specifications.

Misinterpretation of CI/CD Results. CI/CD results are often analyzed "off the shelf". However, unmitigated noise and variations in build complexity can hinder effective decision making in research and practice. In this aspect, we set out to analyze noise and characterize differences in CI/CD results [10]. We also provide tools[5] and recommendations for mitigating noise in CI/CD results.

Inefficient Use of CI/CD Resources. A key goal of CI/CD is to provide rapid feedback to software teams and releases to users throughout the development process. Software organizations invest resources in the operation and maintenance of CI/CD services in order to benefit from such a rapid feedback and release cycle. However, inefficient queues and long job durations can lead to wasted resources.

We are currently conducting empirical studies of historical CI/CD data to understand bottlenecks with the goal of evaluating improvements (e.g., intelligent queuing algorithms, build optimization).

Paper Organization. The remainder of the paper is organized as follows. Section II details how modern CI/CD services are configured and positions our research with prior work. Section III presents our research hypothesis. Section IV summarizes our work on improving the robustness of CI/CD services. Section V presents our plan to improve the efficiency of CI/CD services. Section VI draws conclusions.

*The author is advised by Dr. Shane McIntosh of McGill University.

[1] https://oduinn.com/2013/12/13/the-financial-cost-of-a-checkin-part-2/

[2] https://github.com/geoscixyz/gpgLabs

[3] https://github.com/geoscixyz/gpgLabs/issues/72

[4] https://github.com/software-rebels/hansel_and_gretel

[5] https://github.com/software-rebels/bbchch

II. Background and Related Work

In this section, we define the concepts that are needed to understand our work. In this paper, we follow the definitions of the popular Travis CI service provider.[6] However, the problems that we discuss and the solutions that we propose are general, and will likely apply to other CI/CD environments.

A **build** is comprised of **jobs**, each of which may (1) target a different variant of the development or runtime environment of the project; or (2) execute an independent set of tasks to achieve parallelization. Once all of the jobs in the build are finished, the build is also finished.

Each job is comprised of three main **phases**: *install*, *script*, and *deploy*. Each phase may be preceded by a before sub-phase or followed by an after sub-phase. These sub-phases are used to ensure that all of the pre-conditions are satisfied before the main phase is executed (*before install, before script, before deploy*), and all of the post-conditions are met after executing the main phase (*after success, after failure, after deploy*).

Jobs in Travis CI conclude with one of four outcomes:

- **Passed.** The project was built successfully and passed all tests. All phases terminated with an exit code of zero.
- **Errored.** Any of the commands that are specified in the *before_install, install,* or *before_script* phases of the build terminated with a non-zero exit code. When this occurs, the job terminates immediately.
- **Failed.** A command in the *script* phase terminated with a non-zero exit code. When this occurs, the job switches to the *after_failure* phase and then terminates.
- **Cancelled.** A Travis CI user with sufficient permissions aborted the build using the web interface or the API.

Projects that use the Travis CI service inform Travis CI about how jobs are to be executed using a `.travis.yml` specification file. The properties that are set in this configuration file specify which revisions will initiate builds, how the build environments are to be configured for executing builds, and how different teams or team members should be notified about the outcome of the build. Furthermore, the file specifies which tools are required during the build process and the order in which these tools need to be executed.

These configuration options can be mainly divided into two sections: *node configuration*, which specifies how nodes in the CI service should be prepared before building commences; and *build process configuration*, which specifies each step in the execution of its build jobs.

Although CI/CD provides many benefits, software teams often encounter various difficulties when adopting CI/CD in their organizations. Like other software artifacts, CI/CD-related artifacts also suffer from quality issues that are identified in existing software quality evaluation standards (i.e., ISO/IEC 9126 [11], SQuaRE [12]). Due to their importance to software organizations, we choose to focus on robustness and efficiency aspects in this thesis, leaving other quality aspects (e.g., usability, security, portability) to future work.

[6] https://docs.travis-ci.com/user/for-beginners/

A. Ensuring Robustness in CI/CD Services

Prior work suggests various techniques for detecting and repairing unstable configurations that are indirectly related to CI/CD. Macho et al. [13] propose a tool to automatically repair builds that break due to dependency-related issues. Hassan and Wang [14] use existing build script fixes to resolve new build failures by recommending fix patterns. By applying text mining techniques and qualitative analysis, Rahman and Williams [15] identify properties that characterize defective configuration scripts used for CD. Vassallo et al. [16] propose a tool that summarizes the reasons when a build fails and suggests possible solutions found on the Internet. Sharma et al. [17] catalog smells that are related to Infrastructure as Code (IaC).

To the best of our knowledge, our work on detecting and removing CI configuration anti-patterns is the first of its kind. In more recent work, Vassallo et al. [18] have complemented our static anti-pattern list with process-related anti-patterns.

B. Identifying and Fixing CI/CD Resource Bottlenecks

Ghaleb et al. [19] study characteristics of CI/CD builds that may be associated with the long build durations and recommend caching content that rarely changes to speed up builds. This motivates our work on reducing the execution time of builds.

Abdalkareem et al. [20] report on savings that can be achieved by automatically skipping the CI/CD process entirely for changes that are unlikely to impact the build outcome. Schermann et al. [21] discuss how multiple experiments can be run efficiently in parallel in CD pipelines by formulating the scenario as an optimization problem and using a genetic algorithm. Esfahani et al. [22] propose using content-based caching to run build-related tasks only when needed. Our proposed approach for reducing build execution time accelerates builds using both caching and task parallelization.

Cao et al. [23] propose analyzing an annotated build dependency graph to forecast build duration. Tufano et al. [24] propose a predictive model to preemptively alert developers on the extent to which their software changes may impact future building activities. While these techniques also aim to improve the transparency of CI/CD services, they are complementary to our focus of predicting queue time.

III. Research Hypothesis

In this thesis, we mine historical data from CI/CD services to understand limitations in and propose improvements to state-of-the-art CI/CD services. More specifically, we evaluate the following research hypothesis:

> *Defect-prone and slow CI/CD pipelines can lead to considerable amounts of wasted resources for CI/CD adopters and service providers. Specifications and outcome data from CI/CD services can be leveraged to increase the **robustness** and **efficiency** of CI/CD.*

As shown in Figure 1 we strive to improve the (1) robustness and (2) efficiency of CI/CD services. The details of our current progress and future plans for achieving these objectives are explained in the next two sections.

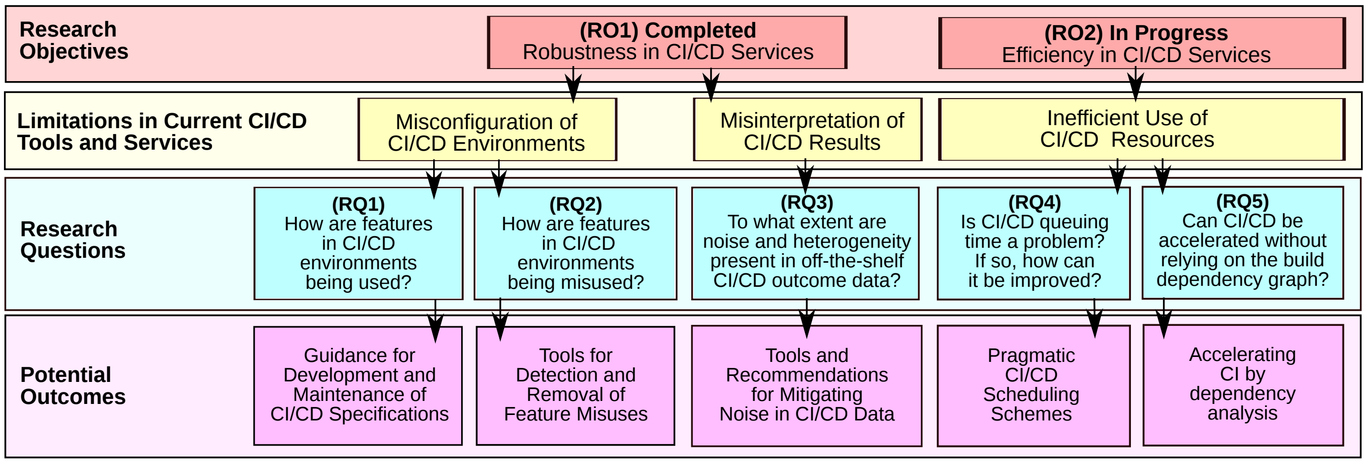
Research Objectives
(RO1) Completed
Robustness in CI/CD Services
(RO2) In Progress
Efficiency in CI/CD Services
Limitations in Current CI/CD Tools and Services
Misconfiguration of CI/CD Environments
Misinterpretation of CI/CD Results
Inefficient Use of CI/CD Resources
Research Questions
(RQ1) How are features in CI/CD environments being used?
(RQ2) How are features in CI/CD environments being misused?
(RQ3) To what extent are noise and heterogeneity present in off-the-shelf CI/CD outcome data?
(RQ4) Is CI/CD queuing time a problem? If so, how can it be improved?
(RQ5) Can CI/CD be accelerated without relying on the build dependency graph?
Potential Outcomes
Guidance for Development and Maintenance of CI/CD Specifications
Tools for Detection and Removal of Feature Misuses
Tools and Recommendations for Mitigating Noise in CI/CD Data
Pragmatic CI/CD Scheduling Schemes
Accelerating CI by dependency analysis

Fig. 1. An Overview of Our Proposed Thesis

IV. Improving the Robustness of CI/CD (RO1)

In this section, we explain the motivation, our research approach, and the obtained results for improving the robustness of CI specifications and CI outcome data.

A. More Robust CI/CD Specifications

Problem. Like programming languages, configuration languages also offer features, which can be used or misused. We first set out to study the ways in which CI/CD features are being used and misused.

Motivation. A typical CI/CD service has different nodes for *creating* build jobs, *processing* them, and *reporting* on the outcome. While configuring job creation and job reporting nodes is relatively simple (e.g., reporting only needs a contact method like an email address and a triggering event type like build failures), configuring job processing nodes is complex. Misconfiguration of CI/CD environments may waste resources or conceal faults. To address this, we conduct two empirical studies to answer the research questions, **(RQ1) How are features in CI/CD environments being used?** and **(RQ2) How are features in CI/CD environments being misused?**

Approach. In this work [10], to study how features in CI/CD specification files are being used, we analyze a curated sample of 9,312 open source projects that are hosted on GitHub and have adopted the popular Travis CI service. The sampled projects are selected based on size and activity, and filtered to remove forks and other duplicates. Furthermore, we define CI/CD misconfiguration patterns and create development tools[4] that can automatically detect and remove them.

Results. We find that explicit deployment code is rare—48% of the studied Travis CI specification code is instead associated with configuring job processing nodes. This shows that the developers rarely use CI/CD services for CD, despite CI/CD service providers supporting the deployment to many popular cloud services including AWS, Azure, and Heroku.

To analyze feature misuse, we propose Hansel—an anti-pattern detection tool for Travis CI specifications. We define four anti-patterns and Hansel detects anti-patterns in the Travis CI specifications of 894 projects in the corpus (10%), achieving a recall of 83% in a sample of 100 projects. Furthermore, we propose Gretel—an anti-pattern removal tool for Travis CI specifications, which can remove 70% of the most frequently occurring anti-pattern automatically. Using Gretel, we have produced 36 accepted pull requests that remove Travis CI anti-patterns automatically.

B. More Robust CI/CD Outcome Data

Problem. CI/CD outcome data is used by software practitioners and researchers when building tools and proposing techniques to solve software engineering problems. However, it may be harmful to use this data "off the shelf" without checking for noise and complexities. In this work, we set out to characterize CI/CD outcome data according to harmful assumptions that one may make about its cleanliness and homogeneity.

Motivation. Consider the example of the *zdavatz/spreadsheet*[7] project, which sets the `allow_failure` property in the initial build specification of the project to ignore failures when determining the final outcome of the build. However, this setting is never removed from the build specification during the five-year history of the project, violating the intended use of the feature.[8] Assuming the outcome of such builds to be entirely clean when there are many suppressed or ignored failures can yield incorrect inferences when analyzing CI/CD outcome data. In addition, builds vary in terms of the number of executed jobs and the number of supported build-time configurations. If prediction models are trained using data that treats these heterogeneous builds identically, model fitness may suffer and the insights that are derived from the models will likely be misleading. To better understand the extent to which noise and heterogeneity are present in CI/CD outcome data, we conduct an empirical study to answer the research question,

[7] https://github.com/zdavatz/spreadsheet
[8] https://github.com/zdavatz/spreadsheet/blame/master/.travis.yml

(RQ3) To what extent are noise and heterogeneity present in off-the-shelf CI/CD outcome data?

Approach. In this work [9], we analyze CI/CD outcome data from large software projects to quantify the noise and characterize their nuances. For this purpose, we use openly available project metadata and CI/CD results of 1,276 GITHUB projects that use TRAVIS CI.

Results. Passing build outcomes do not always indicate that the build was entirely clean. 12% of passing builds have an actively ignored failure. In 83% of branches from broken builds, the breakage persists. These breakages persist for up to 485 commits, suggesting that these breakages are not distracting.

One in every 7 to 11 builds is incorrectly labeled. This noise may influence analyses based on CI/CD outcome data, suggesting that noise needs to be filtered out before subsequent analyses of CI/CD outcome data.

V. IMPROVING THE EFFICIENCY OF CI/CD (RO2)

Developers adopt CI/CD with the intention of speeding up development [8]. However, the results of the CI/CD builds could be delayed due to waiting time in the queues of CI/CD service providers and bottlenecks in the execution of CI/CD jobs. We set out to study how queuing and job execution time can be reduced to improve overall CI/CD performance.

A. Queuing Time

Problem. The time spent by builds waiting in processing queues can impact the perceived build duration. Since slow CI/CD feedback hinders developers and slows releases, long queuing time can adversely affect the user experience of CI/CD services and software team productivity.

Motivation. Builds that wait in queues for long periods before being processed generate waste for organizations that depend on rapid CI/CD cycles. If these delays are common, research into improvements of CI/CD queuing time would be valuable. Therefore, we are currently conducting an empirical study to investigate **(RQ4) Is CI/CD queuing time a problem? If so, how can it be improved?**

Envisioned Contribution. Historical queuing time data is being analyzed to better understand trends and tendencies. More pragmatic CI/CD scheduling algorithms that will drive queuing time down, improve CI/CD throughput, and/or optimize CI/CD infrastructure usage will be evaluated.

Proposed Approach. First, we will analyze the proportion of the perceived build duration that is spent in queues, and trends in that proportion over time. Next, assuming that queuing time accounts for a non-negligible proportion of the perceived build duration, we will use historical data to provide transparency for adopters of CI/CD services by estimating the expected queuing time of future builds. This will allow developers to plan their work more effectively. For example, if CI/CD results will be provided within a few minutes, a developer may choose to stay focused on that task, avoiding a costly context switch [25], [26]. Finally, we will improve queuing time by evaluating different queuing algorithms [27]–[30] under historical CI/CD service workload conditions.

B. Execution Time

Problem. Long running builds impact developer productivity and increase energy costs. Identifying cacheable content and parallelizable tasks can be used to accelerate CI builds [19], [22]. However, current build tools require explicit and complete declarations of dependency graphs and labeling of cacheable content to optimize execution.

Motivation. Software organizations invest resources into the operation and maintenance of CI/CD pipelines. Large organizations like Microsoft [22] and Google [3] leverage distributed caching and dependency parallelization to achieve faster organization-wide build performance. Distributed build caching and job parallelization rely upon the correctness of a hand-maintained build dependency graphs expressed within build files. Like any other software artifact, these build files may contain defects. These defects may take the form of over- or under-specified dependency graphs [31]. Over-specification would lead to suboptimal build parallelization, i.e., tasks that could have been executed in parallel may be executed sequentially due to an unnecessary dependency being respected. More concerningly, under-specification would introduce non-deterministic behavior, i.e., tasks that share an order dependency may be erroneously performed in parallel, leading to (false) build breakage. Therefore, we are conducting an empirical study in collaboration with MicroClusters Inc., a software company focusing on accelerating CI/CD, to answer the research question, **(RQ5) Can CI/CD be accelerated without relying on the build dependency graph?**

Envisioned Contribution. In this project, we aim to deliver a software solution that can safely accelerate CI/CD tasks by decomposing them into truly independent subtasks that can be executed in parallel. The key restriction that we operate under is that we cannot rely on the build files for a correct build dependency graph.

Proposed Approach. First, we will extract a concrete build flow graph by tracing the system calls during build execution. Second, we will design an analysis of the flow graph that can be used to safely accelerate future CI/CD builds. Finally, we will perform a large-scale empirical evaluation of the job decomposition solution by comparing the duration of accelerated CI/CD builds of a sample of projects to baselines from popular CI/CD service providers.

VI. CONCLUSION

When adopting Continuous Integration (CI) and Continuous Deployment (CD) practitioners often encounter problems: 1) Misconfiguration of CI/CD Environments, 2) Misinterpretation of CI/CD Results, and 3) Inefficient Use of CI/CD Resources. In this thesis, we set out to obtain actionable information from historical data in order to improve the robustness and efficiency of CI/CD tools. Our current progress and future plans will provide tools and guidelines for practitioners, and also recommendations for researchers that will help them to produce more robust configuration, more valid reports, and more efficiently use CI/CD resources.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.

[2] C. AtLee, L. Blakk, J. O'Duinn, and A. Z. Gasparnian, "Firefox release engineering," in *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, A. Brown and G. Wilson, Eds. Creative Commons, 2012, ch. 2. [Online]. Available: http://www.aosabook.org/en/ffreleng.html

[3] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at Google)," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014. doi: 10.1145/2568225.2568255 pp. 724–734.

[4] A. Miller, "A hundred days of continuous integration," in *Proceedings of the Agile Conference*, 2008, pp. 289–293.

[5] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella, "A tale of CI build failures: An open source and a financial organization perspective," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017. doi: 10.1109/icsme.2017.67 pp. 183–193.

[6] 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 Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015. doi: 10.1145/2786805.2786850 pp. 805–816.

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[8] G. Pinto, F. Castor, R. Bonifacio, and M. Rebouças, "Work practices and challenges in continuous integration: A survey with Travis CI users," *Software: Practice and Experience*, vol. 48, no. 12, pp. 2223–2236, 2018. doi: 10.1002/spe.2637

[9] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of Travis CI," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2018. doi: 10.1145/3238147.3238171 pp. 87–97.

[10] 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 (TSE)*, 2018. doi: 10.1109/tse.2018.2838131

[11] ISO/IEC, "ISO/IEC 9126:2001 software engineering – product quality," 2001. [Online]. Available: https://www.iso.org/standard/22749.html

[12] ——, "ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models," 2011. [Online]. Available: https://www.iso.org/standard/35733.html

[13] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2018. doi: 10.1109/SANER.2018.8330201 pp. 106–117.

[14] F. Hassan and X. Wang, "HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018. doi: 10.1145/3180155.3180181 pp. 1078–1089.

[15] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *Proceedings of International Conference on Software Testing, Validations, and Verification (ICST)*, 2018. doi: 10.1109/ICST.2018.00014 pp. 34–45.

[16] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall, "Un-break my build: Assisting developers with build repair hints," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2018. doi: 10.1145/3196321.3196350 pp. 41–51.

[17] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016. doi: 10.1145/2901739.2901761 pp. 189–200.

[18] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019. doi: 10.1109/ICSE.2019.00028 pp. 105–115.

[19] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering (EMSE)*, 2019. doi: 10.1007/s10664-019-09695-9

[20] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be CI skipped?" *IEEE Transactions on Software Engineering (TSE)*, 2019. doi: 10.1109/tse.2019.2897300

[21] G. Schermann and P. Leitner, "Search-based scheduling of experiments in continuous deployment," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2018. doi: 10.1109/icsme.2018.00059 pp. 485–495.

[22] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *Proceedings of the International Conference on Software Engineering Companion (ICSE)*, 2016. doi: 10.1145/2889160.2889222 pp. 11–20.

[23] Q. Cao, R. Wen, and S. McIntosh, "Forecasting the duration of incremental build jobs," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017. doi: 10.1109/icsme.2017.34 pp. 524–528.

[24] M. Tufano, H. Sajnani, and K. Herzig, "Towards predicting the impact of software changes on building activities," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019. doi: 10.1109/ICSE-NIER.2019.00021 pp. 49–52.

[25] M. Züger and T. Fritz, "Interruptibility of software developers and its prediction using psycho-physiological sensors," in *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, 2015. doi: 10.1145/2702123.2702593 pp. 2981–2990.

[26] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017. doi: 10.1109/tse.2017.2656886

[27] K. Lee and K. Lim, "Semi-online scheduling problems on a small number of machines," *Journal of Scheduling*, vol. 16, no. 5, pp. 461–477, 2013. doi: 10.1007/s10951-013-0329-x

[28] E. Lübbecke, O. Maurer, N. Megow, and A. Wiese, "A new approach to online scheduling," *ACM Transactions on Algorithms*, vol. 13, no. 1, pp. 1–34, 2016. doi: 10.1145/2996800

[29] J. Boyar, L. M. Favrholdt, C. Kudahl, K. S. Larsen, and J. W. Mikkelsen, "Online algorithms with advice: A survey," *ACM SIGACT News*, vol. 47, no. 3, pp. 93–129, 2016. doi: 10.1145/2993749.2993766

[30] S. Albers and M. Hellwig, "Online makespan minimization with parallel schedules," *Algorithmica*, vol. 78, no. 2, pp. 492–520, 2016. doi: 10.1007/s00453-016-0172-5

[31] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empirical Software Engineering (EMSE)*, vol. 22, no. 6, pp. 3117–3148, 2017. doi: 10.1007/s10664-017-9510-8