What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration

Xianhao Jin
Computer Science
Virginia Tech
xianhao8@vt.edu

Francisco Servant
Computer Science
Virginia Tech
fservant@vt.edu

***Abstract*—Continuous integration (CI) is a widely used practice in modern software engineering. Unfortunately, it is also an expensive practice — Google and Mozilla estimate their CI systems in millions of dollars. There are a number of techniques and tools designed to or having the potential to save the cost of CI or expand its benefit - reducing time to feedback. However, their benefits in some dimensions may also result in drawbacks in others. They may also be beneficial in other scenarios where they are not designed to help. In this paper, we perform the first exhaustive comparison of techniques to improve CI, evaluating 14 variants of 10 techniques using selection and prioritization strategies on build and test granularity. We evaluate their strengths and weaknesses with 10 different cost and time-to-feedback saving metrics on 100 real-world projects. We analyze the results of all techniques to understand the design decisions that helped different dimensions of benefit. We also synthesized those results to lay out a series of recommendations for the development of future research techniques to advance this area.**

***Index Terms*—continuous integration, software maintenance, empirical software engineering**

I. Introduction

Continuous Integration (CI) is a software development practice by which developers integrate code into a shared repository several times a day [13]. However, CI gains adoption in practice, difficulties *e.g.,* [43] and pain points *e.g.,* [61] have been discovered about it. As software companies adopt CI, they execute builds for many of projects, and they do so very frequently. As workload increases, two main problems appear: (1) the time to receive feedback from the build process increases, as software builds often outnumber the available computational resources — having to wait in build queues, and (2) the computational cost of running builds also becomes very high. Previous studies *e.g.,* [38] have highlighted the long time that developers have to wait to receive feedback about their builds. For example, at Google, developers must wait 45 minutes to 9 hours to receive testing results [33]. Even just the dependency-retrieval step of CI can take up to an hour per build [8]. Regarding the high cost of running builds, that is also highlighted in other studies [21], [23], [24], [43], [61]. The cost of CI reaches millions of dollars, *e.g.,* at Google [24] and Microsoft [21]. While other problems exist for CI, we focus on these two because they are the ones that most existing techniques have focused on addressing. They are also interrelated, since cost-reduction techniques may also reduce time-to-feedback — *e.g.,* skipping some tests may cause other tests to fail earlier.

Multiple techniques have been proposed to improve CI. Most of them have the goal of reducing either its **time-to-feedback** or its **computational cost**. All such techniques consider the observation of build failures to be more valuable than build passes, because failures provide actionable feedback, *i.e.,* they point to a problem that needs to be addressed. **Time-to-feedback-reduction** techniques aim to observe **failures earlier** — by **prioritizing** failing executions over passing ones. These techniques may operate in two different levels of granularity, by prioritizing: test executions *e.g.,* [11], or build executions *e.g.,* [33]. **Computational-cost-reduction** techniques aim to observe **failures only** — by **selectively executing** failing builds only, saving the cost of executing passing ones. They also may operate at two different levels of granularity, selecting: test executions *e.g.,* [36], or build executions *e.g.,* [2].

To the extent of our knowledge, the existing techniques to improve CI have been evaluated under different settings, making it hard to compare them. Previous studies used different software projects, different metrics, and rarely compared one technique to another. However, we expect that different choices of goal, granularity, and technique design will bring different trade-offs. For example, cost-reduction techniques at build-granularity may be more *risky* than a test-granularity one, *i.e.,* it may save more cost when it skips all the tests in a build, but it may also make more mistakes if it skips many failing tests in a build. However, the opposite may be true, if test-granularity cost-reduction techniques also skip a large ratio of full builds (*i.e.,* all the tests in the build). On another example, test-selection techniques may be a good alternative to test-prioritization techniques that also saves cost as an added benefit, or they may instead delay the observation of test failures if they mispredict too many of them. To the best of our knowledge, how these trade-offs manifest in practice is still mostly unknown. Empirically understanding these trade-offs will have valuable practical implications for the design of future techniques and for practitioners adopting them.

In this paper, we perform the first evaluation of the existing strategies to improve CI. We aim to understand the trade-offs between these techniques for three dimensions: (D1) computational-cost reduction, (D2) missed failure observation,

 The definitive Version of Record was published as: X. Jin and F. Servant, "What Helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021, pp. 213-225, DOI: https://doi.org/10.1109/ICSE43902.2021.00031

and (D3) early feedback.

For this goal, we performed a large-scale evaluation. We replicated and evaluated all the existing 10 CI-improving techniques from the research literature, representing the two goals (time-to-feedback and computational-cost reduction) and the two levels of granularity (build-level and test-level) for which such techniques have been proposed. We evaluated these techniques under the same settings, using the state-of-the-art dataset of continuous-integration data: TravisTorrent [5]. To be able to study all techniques, we extended TravisTorrent in multiple ways, mining additional Travis logs, Github commits, and building dependency graphs for all our studied projects. Finally, we measured the effectiveness of all techniques with 10 metrics in 3 dimensions. We included every metric that any previous evaluation of our studied techniques used (7), refitted 2 others and designed an additional one.

We analyzed the results obtained by all techniques on all metrics across all 3 dimensions, and we synthesized our observations, to understand which design decisions helped and which ones did not for each dimension. Finally, we further reflect on our results to provide a wide set of recommendations for the design of future techniques in this research area.

The main contributions of this paper are: (1) the first comprehensive evaluation of CI-improving techniques; (2) a collection of metrics to measure the performance of CI-improving techniques over various dimensions; (3) an extended Travis Torrent dataset with: detailed test and commit, and dependencies information; (4) the replication of 14 variants of 10 CI-improving techniques; (5) evidence for researchers to design future CI-improving techniques.

II. Approaches to Improve Continuous Integration

We summarize technique families in Table I and discuss each technique in detail in §III-C. Figure 1 depicts a non-interventional example timeline of builds, a timeline in which a build-selection technique is applied, a timeline produced by build-prioritization technique, a timeline where a test-selection technique is applied, and a timeline with applying a test-prioritization approach. The example timeline shows a chronological numbered sequence of builds in CI. Each build is made up of at least one test. We depict each test suite as a rectangle with a test number (e.g., t1). Failing tests are then highlighted in gray. The length of the rectangle refers to the time duration for the test to be executed. We depict skipped tests with a dashed rectangle. In the most ideal cost-saving scenario, all of the passing tests would be skipped and all of the failing tests would be observed as soon as possible.

A. Computational-cost Reduction

1) Test-level granularity: Test-selection techniques [18], [21], [36], [38], [55], [64], [66] aim at automatically detect and label tests that are not going to fail. These test-level approaches collect information from test history and project dependency along with the current commit and use some heuristic models to detect failing tests and skip the others. Figure 1 also illustrates how this type of techniques works in the simulation timeline. After a test-selection approach is activated, it selects a subset of tests (e.g., t2 in build #2, t4 in build #4) that it predicts to have a possibility to fail and decides to skip the others (e.g., t3 in build #1, t1 in build #5). For those tests that are not selected in the timeline and get skipped, we depict them as dashed rectangles. In this paper, we consider it can skip some builds when it selects no test in those builds.

2) Build-level granularity: Build-selection techniques [1], [2], [20], [27], [42] aim at automatically detect and label commits and builds that can be CI skipped. Some approaches [20], [27], [42] try to detect failing builds and skip those passing builds to achieve cost-saving. Others [1], [2] aim at identifying commits that can be CI skipped. Figure 1 illustrates how they work in the simulation timeline. As a build-level technique, when build-selection approach decides to skip a build (e.g., build #2, #4, #6), normally it skips all of the tests in that build. The inner test sequence is not changed and all of tests are run in an executed build.

B. Time-to-feedback Reduction

1) Test-level granularity: Test-prioritization techniques [11], [35], [37], [40], [57] try to give high priority to tests that are predicted to be failed so that developers could be informed in a shorter time. This family of approaches normally rearrange the execution order of tests within a build to make predicted-to-fail tests run earlier by analyzing information such as test failing history and test context. Figure 1 depicts an example of how this type of techniques works in the simulation timeline. With a test-level approach being activated, the CI system gives different tests different priorities and firstly executes those tests with a higher priority (e.g., t4 in build #2, t2 in build #3) as well as delays low-priority tests (e.g., t1 in build #3, t2 in build #6). The sequence of test executions in this timeline gets rearranged and the start-time for tests that are more likely to fail move ahead in time. Also, all tests are executed at last.

2) Build-level granularity: Build-prioritization techniques [33] aim at automatically prioritizes commits that are waiting for being executed. They favor builds with a larger percentage of test suites that have been found to fail recently and builds including test suites that have not been executed recently as an alternative path. Figure 1 also shows how this family of techniques works in the simulation timeline. Build-prioritization techniques will only be activated when there is a collision of builds (i.e., there are multiple builds waiting to occupy the limited resource). The technique is build-level so it will not change the inner order of the test executions and it will normally change the sequence of tests across builds when the approach is activated (e.g., build #4, #5). None of tests become dashed in this timeline because they all eventually execute.

III. Research Method

In this paper, we replicated and evaluated 14 variants of 10 CI-improving techniques, covering their two goals (time-to-feedback and computational-cost reduction) and their two levels of granularity (build-level and test-level) with 1 perfect

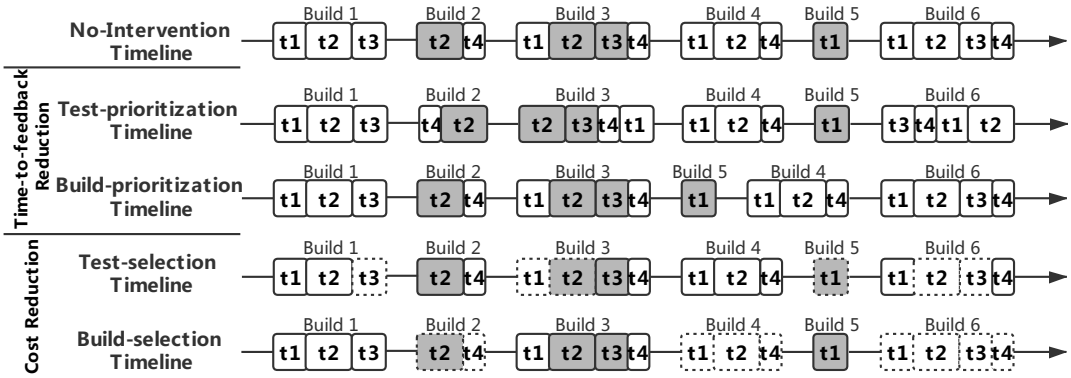

Fig. 1: Example timeline. Failing tests in gray. Build-selection runs builds fully when it predicts a failing build. Test-selection runs builds partially (for tests that would fail). Build-prioritization changes the build sequence. Test-prioritization changes the test sequence within a build.

technique for the ideal timeline. We evaluate them over 100 software projects in TravisTorrent, which we extended to be able to run all such kinds of techniques.

Our goal is to understand the trade-offs between existing CI-improving techniques, and between the metrics that have been used to evaluate them. We perform 2 empirical studies to analyze these trade-offs for the following 3 dimensions of CI-improving techniques, using 10 metrics. We only include selection techniques in Empirical Study 1 since prioritization techniques have no power in cost saving by nature. We involve selection and prioritization techniques in Empirical Study 2 because both of them can have an impact on fault detection, *e.g.,* wrongly-skipped failing builds by selection approaches can cause delay in fault detection.

Empirical Study 1: Cost Saving

D1: Computational-cost Reduction

D2: Missed Failure Observation

Empirical Study 2: Time-to-feedback Reduction

D3: Early Feedback

For each dimension, we study:

RQ1: What design decisions helped this dimension?

RQ2: What design decisions did not help this dimension?

A. Data Set

We perform our study over the Travis Torrent dataset [4], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. We remove "toy projects" from the data set by studying those that are more than one year old, and that have at least 200 builds and at least 1000 lines of source code, which is a criteria applied in multiple other works [42], [25]. To be able to evaluate test-granularity techniques, we also filter out those projects whose build logs do not contain any test information. We focused our study on builds with passing or failing result, rather than error or canceled — since they can be exceptions or may happen during the initialization and get aborted immediately before the real build starts. Besides, in Travis a single push or pull-request can trigger a build with multiple jobs, and each job corresponds to a configuration of the building step. We did a preliminary investigation of these builds and found that these jobs with the same build identifier normally share the same build result and build duration. Thus, as many existing papers have done [14], [44], [26], we considered these jobs as a single build. After this filtering process, we obtained 82,427 builds from 100 projects (13,464 failing builds).

To be able to execute all our studied techniques, we extended the information in TravisTorrent of these 100 projects in multiple ways. First of all, we needed to know the duration of each individual test for the comparison and replication. Also, to replicate some techniques, *e.g.,* [21], [11], we needed to capture the historical failure ratio for each individual test. To obtain these information, we built scripts to download the raw build logs from Travis and parse them to extract all of the information about test executions, such as test name, duration and outcome. Some techniques, *e.g.,* [36], [2], require additional information that TravisTorrent does not provide for builds, such as the content of commit messages, changed source lines and changed file names. For that, we also mined additional information about commits in the projects' code repositories through Github. Then, we matched each test with its corresponding test file in the project. Finally, to be able to run other techniques, *e.g.,* [18], [36], we built a dependency graph for the source code of each project using a static code analysis tool (Scitool Understand [48]) to determine the paths between the source files and test files.

B. Evaluation Process

We evaluate the techniques in a real-world scenario, to understand as best as possible the behavior that the techniques would show in practice. We take two measures for that.

First, we respect the original chronological order of build and test operations when training techniques. We achieve that by using an 11-fold, chronological variant of cross-validation. For each project, we split its chronological timeline into 11 folds. We use the first chronological fold only for testing, and we iteratively test the other 10 folds. For each testing fold,

TABLE I: Studied Techniques.

Goal	Approach	Granularity	Studied Technique
Time to Feed-back	Prioritization	Test	PT_Marijan13 [37]
			PT_Elbaum14 [11]
			PT_Thomas14 [57]
		Build	PB_Liang18 [32]
Comput-ational Cost	Selection	Test	ST_Gligoric15 [18]
			ST_Herzig15 [21]
			ST_Mach19 [36]
		Build	SB_Hassan17 [20]
			SB_Abd19 [2]
			SB_Jin20 [27]

we train on all the folds that precede it chronologically. This approach has been used in previous works *e.g.,* [7], [52] to avoid training with information that would not be available in practice, *i.e.,* it happens in the future.

We follow this approach for all the techniques based on machine learning, *e.g.,* [36]. For techniques that do not require training, *e.g.,* [2], we simply execute them over the same last 10 folds. For techniques that train on data from other projects, *i.e.,* for cross-project technique variants, we also executed them over the same last-10-fold timeline — and we divided them into 10 *project* folds to do cross-project cross-validation, *i.e.,* for each project, the technique is trained on 90 other projects and tested on its last 10 fold data.

Second, we respect the real-world availability of information. That is, for selection-based techniques, when a build or test is skipped, the technique will not know its outcome. For techniques that rely on the last build or test outcome *e.g.,* [19], we only inform them of the outcome of the last *executed* build or test. Additionally, when builds are skipped, we accumulate their code changes into the subsequent build.

C. Replicated Techniques

We replicated and studied all the techniques that have been proposed to improve CI by reducing the time to feedback or reducing its cost. In addition to these, there are other techniques that were proposed before CI and that could also be applied for these two goals: test prioritization techniques, and test selection techniques. Therefore, we also replicated and studied a state-of-the-art technique in each of these two categories that were not originally proposed for CI. We summarize all our studied techniques in Table I.

In total, we studied 10 techniques, across two goals (reducing time to feedback and cost) and two granularities (test and build levels). Since we also studied multiple variants of some techniques, our evaluation included 14 total technique variants. To provide a reference point, we also studied a perfect technique: *Perfect Technique*. It achieves the goal of each metric perfectly — it predicts which tests or builds will fail with 100% accuracy, prioritizing or selecting them perfectly.

We include the detailed description for each technique in §IV-A and §V-A.

IV. Empirical Study 1: Cost Saving

A. Studied Techniques

1) Test-selection Techniques: We replicated all the test-selection techniques that were proposed for improving CI: ST_Mach19 [36] and ST_Herzig15 [21]. To provide even more context for our study, we also evaluate a state-of-the-art test-selection technique: ST_Gligoric15 [18] — since test-selection techniques have also been proposed outside the context of CI, *e.g.,* [64], [18], [63], [62], [46], [45].

ST_Gligoric15 [18] skips tests that cannot reach the changed files, by tracking dynamic dependencies of tests on files. A test can be skipped in the new revision if none of its dependent files changed. The rationale is that tests that cannot reach changed files cannot detect faults in them.

ST_Herzig15 [21] is based on a cost model, which dynamically skips tests when the expected cost of running the test exceeds the expected cost of removing it, considering both the machine cost and human inspection cost [3], [22]. This technique tends to skip tests that mostly passed in the past or that have long runtime.

ST_Mach19 [36] proposes a Machine Learning algorithm with combined features of commit changes and test historical information. We studied two variants of it: one is trained in the past builds within the same project in which it is applied (*ST_Mach19_W*), and the other is trained in the builds of different software projects than the one in which it will be applied (*ST_Mach19_C*). It uses the following features: file extensions, change history, failure rates, project name, number of tests and minimal distance.

2) Build-selection Techniques: We then replicated all build-selection techniques that jave been proposed for improving CI: SB_Abd19 [2], and SB_Jin20 [27]. To provide even more context for our study, we also replicated a state-of-the-art build-prediction technique: SB_Hassan17 [20].

SB_Hassan17 [20] predicts every build's outcome based on the information from last build. Builds can be skipped when they are predicted to pass. In our study, information from the previous build is blinded if the build does not get executed. We study two variants of this technique (*SB_Hassan17_W* and *SB_Hassan17_C*) as we did for *ST_Mach19*.

SB_Abd19 [2] uses a rule-based approach to skip commits that only have *safe* changes, *e.g.,* changes on configuration or document files. This technique is expected to capture most failing builds since it only skips builds considered safe to skip.

SB_Jin20 [27] aims at saving CI cost by skipping passing builds. Their strategy is to capture the first failing build in a subsequence of failing builds and continuously build until a passing build appears. We replicated this technique under the configuration that provided the optimal effectiveness [27]. We studied three variants of this technique: *SB_Jin20_W* & *SB_Jin20_C* as we did previously, and also a rule-of-thumb variant (SB_Jin20_S) that skips builds with < 4 changed files.

B. D1: Computational-cost Reduction

We studied four metrics for D1. We plot the result of each metric in a box plot where each box represents the distribution

of values for all the studied projects.

1) Studied Metrics: **Build time saved** measures the proportion of total build time that is skipped among all build time per project. It was covered in SB_Abd19 [2].

Test time saved measures the same as the previous metric but in terms of test time. The previous work ST_Gligoric15 [18] used this metric in its evaluation. It shows how much time applying a technique could save during the phase of test executions.

Builds number saved measures the proportion of builds that are saved among all builds. It was studied by SB_Abd19 [2] and SB_Jin20 [27]. It represents how many resources could be saved as the number of builds.

Tests number saved measures the same as the previous metric but in term of tests. Previous papers [18], [21] studied this metric. It represents how many resources could be saved during test executions.

2) Analysis of Results: **Comparing Metrics.** When we compare the techniques' test number vs. test time saved, most of them saved a very similar ratio of test time than ratio of tests (except ST_Herzig15).

When comparing build number vs. build time, build-granularity techniques saved a very similar ratio of build time as of builds. Also, test-granularity techniques saved a larger ratio of build time than of builds. This means that test-granularity techniques save build time when they skip builds partially — when they skipped some of their tests.

When comparing test number vs. build number, build-granularity techniques saved a very similar ratio of builds and tests. Also, test-granularity techniques saved a much lower ratio of builds than of tests — some dramatically so (ST_Herzig15 and ST_Mach19_C). This means that test-granularity techniques saved a low ratio of full builds.

When comparing test time vs. build time, build-granularity techniques saved very similar ratios of test time and build time. Also, test-granularity techniques saved a much lower ratio of build time than of test time. This observation extends our earlier one: every build that these techniques did not skip fully, and thus did not skip its build-preparation time, reduced their ability to save build time to an important extent.

Comparing Granularities. By comparing test vs. build-granularity techniques, build-granularity techniques generally saved higher build-time cost — except for SB_Abd19. Build-granularity techniques have the advantage of skipping both test-execution and build-preparation time, while test-granularity techniques have the advantage of skipping tests spread over many builds, not only on those that get fully skipped. Our observation implies that skipping full builds was a better strategy for saving cost.

Comparing Techniques. We first observed that SB_Mach19_C and SB_Jin20_C skipped fewer builds than their counterparts that were trained only with data within the same project (SB_Mach19_W, SB_Jin20_W). After having been trained with a more diverse set of build and tests (across many projects), these techniques became less confident to skip them. ST_Herzig15 saved very low ratio of build time despite saving a large ratio of tests. This is because it very rarely skips tests that failed many times in the past — regardless of the code changes in the build. So, within each build, it very rarely skipped the tests with the most past failures — thus very rarely skipping builds fully. SB_Abd19 saved a median 21% build time, which is a relatively high amount, considering that it only skipped builds with non-executable changes, *e.g.,* that only changed formatting or comments. ST_Mach19_W and ST_Gligoric15 skipped a relatively high ratio of build time (competitively with build selection techniques) because they skipped many full builds. This is because they analyze the relationship between code changes and tests inside a build. ST_Gligoric15 skips all tests that cannot execute the code changes, and ST_Mach19_W considers the distance between the changes and the tests in its predictor. This allows both techniques to fully skip those builds in which no test can execute the code changes — *i.e.,* when only non-executable code was changed, or when no tests exist to execute the changes. SB_Jin20_W and SB_Jin20_S saved high ratios of build time, since they both focused on skipping full builds. While SB_Jin20_S provided higher savings, we expect it to also skip a higher ratio of skipped failing builds (see §IV-C) — SB_Jin20_S simply skips builds with <4 commits. Finally, SB_Hassan17_W and SB_Hassan17_C skipped too much build time (higher than the perfect baseline). This is because they mostly rely on the status of the previous build, which is unknown if skipped. So, as soon as they observe a passing build, they recurrently skip all subsequent builds.

C. D2: Missed Failure Observation

1) Studied Metrics: **Proportion of skipped failing tests.** This metric measures the undesired side effect of cost-saving techniques skipping some of the failing test cases. It was used by ST_Herzig15 [21].

Proportion of skipped failing builds. This metric measures the proportion of failing builds that are skipped among all failing builds. It was covered in SB_Jin20 [27].

2) Analysis of Results: **Comparing Metrics.** All techniques generally skipped a very similar ratio of failing tests than builds, with small differences.

ST_Mach19_C, ST_Herzig15, ST_Gligoric15, SB_Jin20_S skipped a slightly higher ratio of failing tests than builds. This is explained by test-granularity techniques skipping partial builds in addition to full builds, and thus they also skipped a higher ratio of failing tests. The case of SB_Jin20_S is different: it skipped a higher ratio of tests because it skipped fewer builds with no failing tests — few changed < 4 files.

SB_Abd19, SB_Jin20_C, ST_Mach19_W and SB_Jin20_W skipped a slightly higher ratio of failing builds than tests. This means that these techniques skipped failing builds with lower than average (or no) failing tests, *e.g.,* failing due to configuration or compilation errors (which amount to 35% of

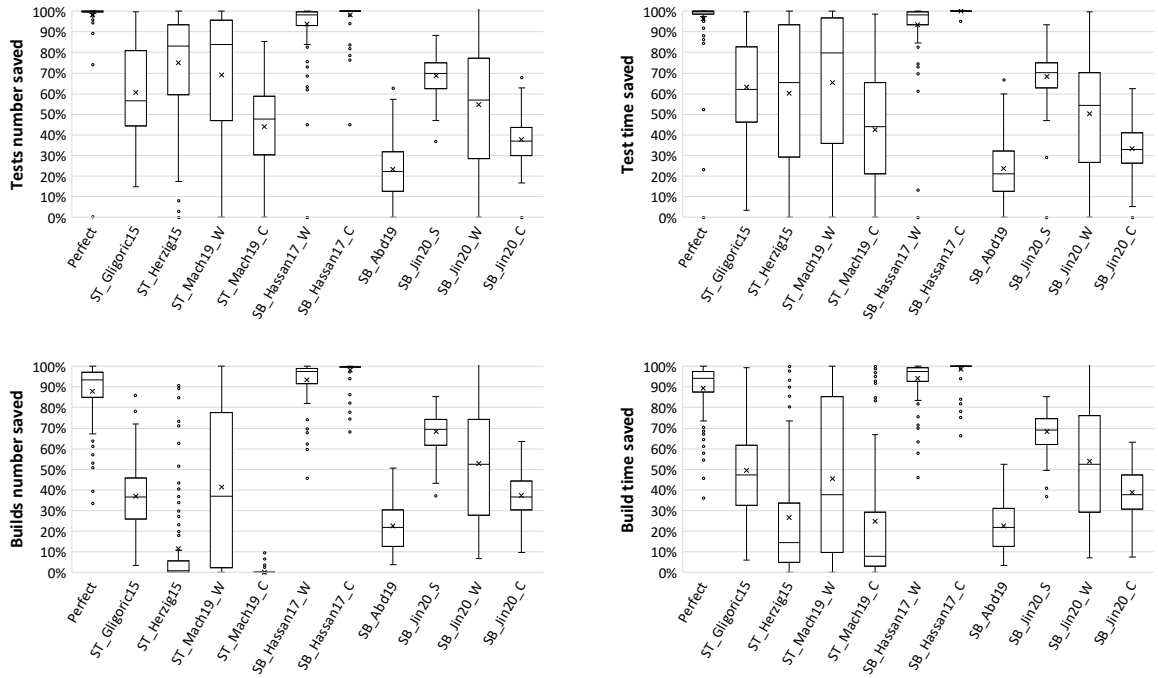

Fig. 2: Results for Cost Saving Metrics. Prioritization techniques not included, since they do not skip tests/builds.

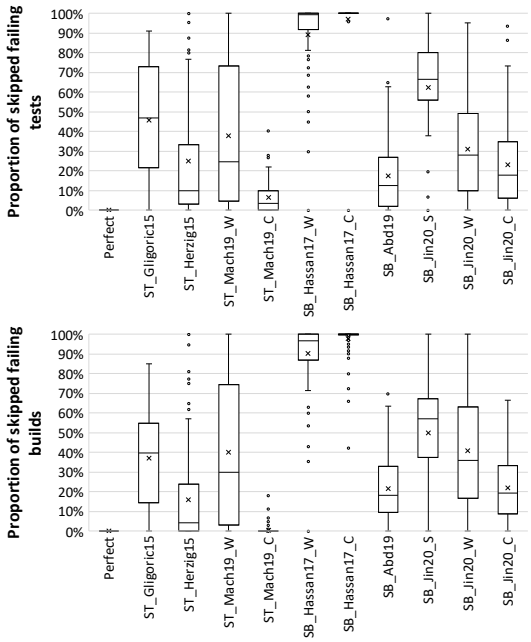

Fig. 3: Results for Missed Failure Observation Metrics. Prioritization techniques not included, since they do not skip tests/builds.

failing builds). Finally, SB_Hassan17_C and SB_Hassan17_W skipped most failing (and passing) tests and builds.

Comparing Granularities. Build-granularity techniques generally skipped higher ratios of failing builds and tests than test-granularity techniques — except for SB_Abd19. They generally skipped a higher ratio of all tests and builds.

Comparing Techniques. If we rank techniques on these two metrics of side-effect, we observe that they rank almost exactly in the opposite order as they would according to build time saved (for D1). This shows a clear trade-off between cost-saving and its side effect of skipping failures.

V. Empirical Study 2. D3: Time-to-feedback Reduction

In D3, we study how much prioritization techniques advance the observation of failures and how much the side effect in D2 will influence it. So, we study all the time-to-feedback and computational-cost reduction techniques.

A. Studied Techniques

We only describe here the techniques that we did not describe in earlier sections: prioritization techniques.

1) Test-prioritization Techniques: For this family of techniques, we replicated all the test-prioritization techniques that were proposed for improving CI: PT_Elbaum14 [11] and PT_Marijan13 [37]. To further extend this study, we also replicated the state-of-the-art test case prioritization (TCP) technique. We chose the technique that provided the highest effectiveness in the most recent evaluation of TCP techniques [35]: PT_Thomas14 [57]. TCP was a rich research area before CI became a common practice, *e.g.,* [40], [57], [10], [47]. We apply these techniques to prioritize tests within each build.

PT_Marijan13 [37] prioritizes tests that failed recently or have a shorter duration. Tests are ordered based on their historical failure data, test execution time and domain-specific heuristics.

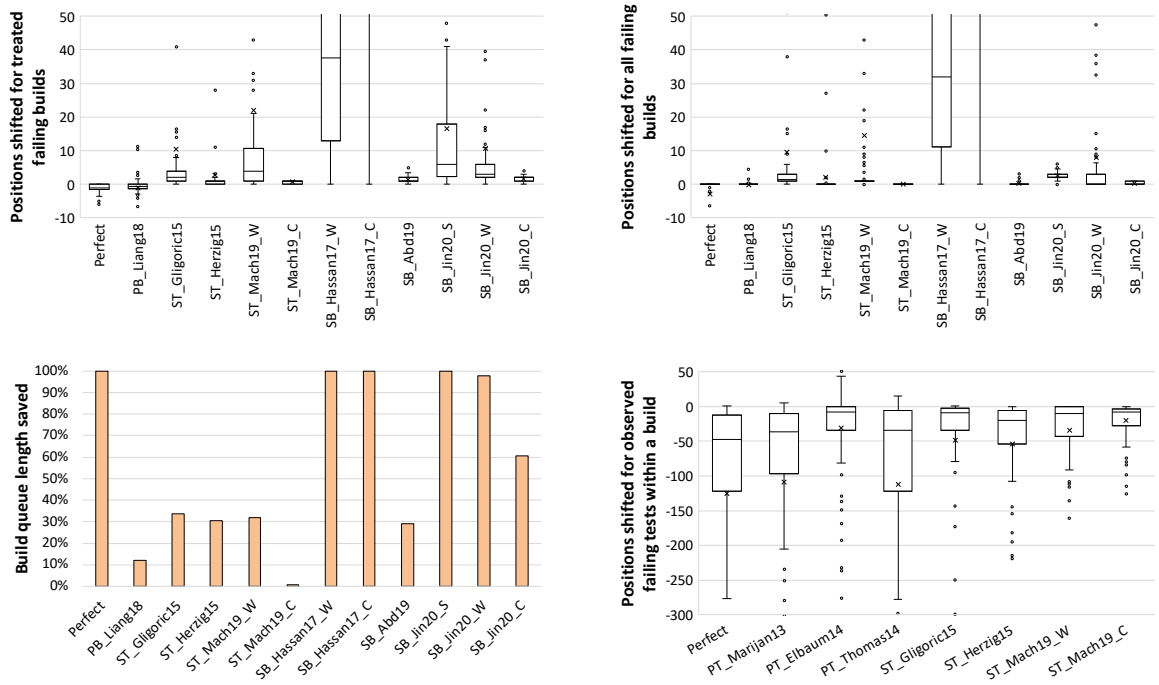

Fig. 4: Results for Time-to-feedback Reduction Metrics.

PT_Elbaum14 [11] favors tests that failed either recently or a long time ago.

PT_Thomas14 [57] uses topic modeling to diversity the tests that get executed earlier. Every prioritized test is selected if it contains the most different topics from the previous test in its identifiers and comments. The rationale behind this is that similar tests often find similar problems.

2) Build-Prioritization Techniques: To the extent of our knowledge, only one technique has been proposed to prioritize software builds, PB_Liang18 [33]. **PB_Liang18 [33]** executes builds containing a recently-failing and recently-non-executed test in a collision queue. We apply PB_Liang18 to prioritize builds within a build waiting queue, as its previous evaluation did [33]. Queues form when build executions overlap in time.

B. Studied Metrics

1) Positions shifted for observed failing tests within a build: measures the shifted positions for all observed failing tests (prioritized or not). A similar metric to this one was used in the evaluations of PT_Marijan13 [37], PT_Elbaum14 [11], and PT_Thomas14 [57]. For test-selection techniques, we measure the average number of shifted positions for all remaining tests — when a test is skipped, the next one can now run one position earlier.

2) Positions shifted for treated failing builds: measures the number of builds between every treated (delayed/advanced) failing build's original observation position and its new position. This metric was studied by SB_Jin20 [27]. For test-granularity techniques, this metric is not impacted, since the build is still executed in the same position. For build-selection techniques, we consider that when a build is skipped, it will run as the next build (its tests will run on it).

3) Positions shifted for all failing builds: measures the same as the previous one, but now across all failing builds. PB_Liang18 used a similar metric in its evaluation [33]. Through this metric, we can understand the impact of the previous metric over all builds.

4) Build-queue-length saved: This is a metric designed by us to measure how applying a technique could relieve the collision problem: when multiple builds are waiting to be executed within a limited resource. We follow the same configuration in PB_Liang18's paper. The build-queue-length refers to the median number of builds waiting ahead for each build in each project. With a pre-experiment on all projects, we find that for only one project - "Rails/Rails", the median value of every build's waiting queue is bigger than 0. Thus, we only report the result for this metric on that project.

C. Analysis of Results

Comparing Metrics. When comparing positions shifted for treated failing builds vs. all failing builds, for all techniques, the advance (PB_Liang18) or delay (others) that they introduce in the observation of failing builds is much lower when measured across the whole population of failing builds. The upside of this is that the undesired effect of most techniques (*i.e.,* delay of failure observation) is very low across all failing builds (median 0–2 builds). The downside is that the desired effect of PB_Liang18 (*i.e.,* advance of failure observation) is also very low across all failing builds (median 0 builds).

Next, we compare the performance of test selection techniques (*i.e.,* the only overlapping technique family) in the positions that observed failing tests shifted within a build vs. the positions that failing builds shifted across all builds.

We observe that test selection techniques provided some advancement in the observation of test failures (lower than most test prioritization techniques), while introducing a very low delay in observation of build failures (median 0–2).

Comparing Granularities. We did not observe a substantial difference when comparing granularities — we observed stronger differences when comparing techniques.

Comparing Technique Strategies. When comparing technique strategies (prioritization vs. selection), test-selection techniques provided some advancement in the observation of failing tests within a build, but test-prioritization techniques provided better results overall (except PT_Elbaum14).

Comparing Techniques. PT_Marijan3 and PT_Thomas14 behave very similarly — despite their different approaches to prioritization — and they are both close to perfect, prioritizing most tests correctly. PT_Elbaum14 provides a lower advancement of test failures (also lower than many test-selection techniques), since it uses a simpler criterion — prioritizing tests that were executed very recently or a long time ago. All test-selection techniques provided a very similar advancement of test-failure observation, except ST_Herzig15 which was slightly better. Interestingly, ST_Herzig15 was one of the techniques with the lowest delay in build-failure observation (median 0 for all failing builds). At the build-granularity, PB_Liang18 had a very low impact in prioritizing builds because builds very rarely occurred concurrently in our dataset — only the Rails project had a meaningful number of concurrent builds. An important metric in PB_Liang18's original evaluation was the savings in the build-queue length. We plot the results for all techniques for this metric in Figure 4. Interestingly, we also observed that test-selection and build-selection techniques also had a strong impact in this metric — less so for test-selection techniques and SB_Abd19 because they skip fewer full builds (see §IV-B2). Regarding build-selection techniques, those that saved more builds (see §IV-B2) also saved more in the build-queue-length metric, but also introduced higher delays in build-failure observation.

VI. Answers for Research Questions and Implications

We synthesize our observations and we lay out their implications to advance this area of research.

A. D1: Computational-cost Reduction

1) RQ1: What design decisions did not help?: First, we report on **missed opportunities** for saving more computational cost. Cost-saving techniques focused on skipping passing builds and tests, but they **did not specifically target those that would provide the highest savings**, *i.e.,* slower tests, slower builds, or all tests in a build (in the case of tests-selection). This is demonstrated by the fact that build-granularity techniques saved similar ratios of test number, test time, build number, and build time; and that test-granularity techniques saved similar ratios of test number and test time, and lower ratios of build time than test time.

We also learned that **training cost-saving techniques across projects** harmed their predictions. In other fields, training with data from multiple projects is considered to increase the accuracy of predictors. For cost-saving techniques, though, this exposed the techniques to more diverse sets of failures, making more builds/tests "look like a failure", resulting on the predictors saving less cost (being less inclined to skip builds and tests).

Test-selection techniques were also limited in the cost that they could save when **they did not target saving full builds** — ST_Mach19_C and ST_Herzig15 saved very low build time despite saving a high ratio of tests. An additional aspect that contributed to ST_Herzig15 saving limited build time (despite saving high number of tests) is that **it only used features characterizing the tests**, but not the code changes in the build — *e.g.,* missing the opportunity to skip full builds for no-code changes.

2) RQ2: What design decisions helped?: Other design decisions allowed techniques to save high cost. A particularly useful design decision was **trying to predict seemingly-safe builds and tests** — SB_Abd19 saved 21% builds simply by skipping builds with no-code changes, and ST_Gligoric15 saved 36% builds skipping tests that did not cover the code changed in the build.

Another decision that provided high cost savings was to **skip full builds instead of individual tests** — thus also saving build-preparation time. Skipping all tests in a build allows to skip the time to prepare the build (*i.e.,* compilation and other overhead like virtual machine preparation), and we observed that **build-preparation takes a large portion of build time**. An illustrative example is how ST_Gligoric15 and ST_Herzig15 saved about the same ratio of test time, but ST_Gligoric15 saved much higher build time because it saved a much higher ratio of full builds.

Test-selection techniques, however, performed really well in terms of saving a high ratio of tests (84% by ST_Herzig15 and 80% by ST_Machalica_W). This is because they could save some cost spread out across many builds — *i.e.,* **skipping partial builds achieved high cost savings**. However, the test-selection **techniques that skipped full builds also achieved high savings**. Intentionally or not, ST_Gligoric15 saved many full builds by simply skipping all tests that did not cover the changed code. ST_Mach19_W also saved many full builds by approximating the same idea: one of its predictor's features is the distance between the changed code and the test.

3) Implications for Future Techniques: Our results have multiple implications for the design of future techniques. First, we encourage future techniques to consider **hybrid approaches** to save both full builds and also partial builds, *i.e.,* to save cost at both build and test granularity. Future techniques should also leverage the beneficial factors that we already observed, such as **skipping full builds with no-code changes or no tests to cover them**. To save more full builds, novel prediction features could be designed, **targeting slower builds** if possible — which no existing technique attempts. To save more tests, existing techniques already

provide very useful features (saving a high ratio of tests), but other new features could be designed to target **saving more and slower tests**, and considering the **relationships between the tests and the code changes** in the build. Finally, our observations also show that **build time saved** is the metric that most comprehensively shows the cost saved by all existing techniques — even though cross-referencing multiple metrics allows for additional observations, as we did in this study.

B. D2: Missed Failure Observation

1) RQ1: What design decisions did not help?: In terms of the proportion of builds and tests that were skipped by cost-saving techniques, we generally observe that **the decisions that made techniques save higher cost also made them make more mistakes**, *i.e.,* skip higher ratios of failing builds and tests. It was also particularly interesting that **seemingly-safe techniques** — SB_Abd19 and ST_Gligoric15 — still **showed pretty high ratios of skipped failing builds and tests**. Our study thus shows that skipping builds with no-code changes or without tests to execute them is not enough to guarantee that they will not fail. A quick look discovered that the builds and tests skipped by these techniques failed for different reasons, such as configuration or compilation errors (present in 35% of failing builds).

2) RQ2: What design decisions helped?: One design decision that reduced the skipped failing tests and builds was **training techniques across projects**. All the _C variants skipped lower ratios than their _W counterparts (except SB_Hassan17_C). Also **test-granularity techniques generally skipped lower ratios of failing tests** than build-granularity techniques did of builds.

3) Implications for Future Techniques: These results imply multiple recommendations for future techniques. First, future techniques should design **novel features to predict failures that are caused by no-code changes**, *e.g.,* configuration changes, to avoid assuming that seemingly-safe builds will not fail. Second, future techniques should attempt to **break this trade-off between saving cost and skipping failures**. Existing techniques generally increase cost savings by also increasing missed failure observations. Future techniques should attempt to improve one of the two dimensions by keeping the other one fixed (or optimal). Finally, future studies should propose **new metrics to better assess the trade-off between cost-saving and skipped-failures** of various techniques — since most techniques succeed in one at the expense of the other. SB_Jin20 [27] proposed the harmonic mean of the two as a balanced metric, but further study is granted to understand whether both should be valued equally or in a weighted manner — particularly considering the much higher ratio of passes to failures in CI datasets.

C. D3: Time-to-feedback Reduction

1) RQ1: What design decisions did not help?: Unsurprisingly, **build-selection techniques did not advance the observation of build failures at all**, but at least they introduced very low delays in the observation of failing builds (and also saved some computational cost). Similarly, **test-selection techniques also introduced a small delay in the observation of test failures**. **Build-prioritization also showed very limited advancement in observing failing builds**, but that was mainly because only one of our studied projects (open-source) had some contention in the build queue. We expect that industrial software project would obtain a much higher benefit from this approach. Finally, we also observed that the build-selection techniques that produced **higher cost savings also introduced higher delays in build-failure observation**, showing again the tension between both goals.

2) RQ2: What design decisions helped?: **The best techniques to provide early feedback were test-prioritization techniques**. In fact, PT_Thomas14 provided near perfect results. We also found that **test-selection techniques provided lower, but competitive advancement of test failure observation**, while also providing some cost savings. For example, ST_Herzig15 provided high advancement of test-failure observation within a build, with very low delay of build-failure observation, while also saving some computational cost. Similarly, we observed that **build-selection techniques could also provide reductions in build-queue-length that were competitive with build prioritization**.

3) Implications for Future Techniques: For future techniques, we recommend to **combine test prioritization with test selection techniques** — since prioritization techniques could stop after the first failure is identified, and save the cost of running the remaining tests. We found that test-prioritization techniques already reached very high results (PT_Thomas14 is near perfect), so the features that they use could be also very useful for test selection to save cost. Conversely, existing test-selection techniques that already perform very well for cost-savings (*e.g.,* ST_Herzig15) could be improved in their ability to advance failure observation. Similarly, we recommend to further study the application of **build-selection techniques to provide early observation of build failures** by reducing the build queue via skipping builds in industrial projects in which parallel build requests are a larger issue. Finally, there is also space to develop new metrics that could capture the balance that techniques provide across all dimensions D1–D3.

D. Standing on the Shoulders of Giants

Our findings confirm and extend previous work:

1) D1: Beller *et al.* [4] observed that test time is a low proportion of build time. We extend this observation by finding that our studied test-selection techniques infrequently skipped full (all tests within) builds, which strongly limited their cost-saving ability. We thus recommend test-selection to incentivize skipping full builds to save higher cost in CI.

2) D2: Jin and Servant [27] observed a trade-off of higher cost savings incurring more missed build failures in their technique. We extend this observation by finding that all our studied techniques were affected by that trade-off (techniques ranked equally by cost savings as by missed failures). We additionally identified clear strategies that made techniques miss fewer failures: training across projects, and operating

at test granularity. We also observed that a seemingly-safe technique [2] still missed a high ratio of failures. Finally, we elicited the need for better prediction of safe builds, and new metrics to compare trade-offs.

3) D3: Herzig *et al.* [21] found that their test-granularity technique incurs low delay in build-failure observations. We extend this observation by finding that all our other studied test-granularity techniques also incur low build-failure-observation delay, measured across all failing builds.

VII. Threats to Validity

A. Internal Validity

To guard internal validity, we carefully tested our evaluation tools on subsets of our dataset while developing them.

Our analysis could also be influenced by incorrect information in our analyzed dataset. For this, we studied a popular dataset that is prevalent among continuous integration studies: TravisTorrent [6]. Furthermore, many of our studied techniques [2], [20], [27], [33] were originally evaluated on TravisTorrent projects. Additionally, we extensively curated TravisTorrent, removing: toy projects following standard practice [25], [42], unusable projects for test-granularity techniques, and cancelled builds as in past work [14], [26], [44]. Finally, we also followed the advice in Gallaba *et al.*'s study [14] to consider the nuance in the TravisTorrent dataset. We did so in the following ways: (1) We considered passing builds with ignored failures as passing. Developers manually flag such failures to be ignored when they cannot officially support them [14], and thus should not represent the status of the build. (2) We considered builds that fail after another failure as correctly labeled, because they flag an unsolved problem, being informative for developers. (3) We considered failing builds with passing jobs as failing builds. If at least one job fails, it signals a problem, informing developers.

Our results may also be affected by flaky tests causing spurious failing builds. However, CI systems are expected to function even in the presence of flaky tests, since most companies do not consider it economically viable to remove them, e.g., [36], [39]. Besides, cross validation may make unrealistic use of chronological events To address this problem, we used time-based cross validation [7].

Our observed build and test runtimes may have been influenced by the load experienced in the build server at the time. However, we consider this potential impact to be very low, since we observed that the standard variance in test duration across builds was 0.5 seconds.

B. External Validity

To increase external validity, we selected the popular dataset TravisTorrent, which has been analyzed by many other research works. The projects we chose were all Java or Ruby projects, because there are no projects with other programming languages in the data set. Although these two programming languages are popular, different CI habits in other languages may provide slightly different results to the ones in this study. Our observations may slightly vary for separate software projects, but our goal was to derive general observations for a real-world population of software projects.

C. Construct Validity

A threat to construct validity is whether we studied software projects that are similar to those that suffer most accutely from high CI cost and delays in failure observation, e.g., the projects at Google [24] and Microsoft [21]. We studied the TravisTorrent dataset, which is the standard dataset used in the literature to evaluate techniques to save cost in CI [2], [27], [33], [9]. One of our studied projects (Rails) is particularly similar to industrial software projects. Rails was used alongside two other Google datasets to evaluate PB_Liang18 *et al.* [33], and it had similar magnitudes of test suites (thousands), test executions (millions) and test execution time (millions of seconds).

Nevertheless, early observation (or prediction) of build failures is beneficial, regardless of how much load a project's CI system experiences. It allows developers to not have to wait for builds to finish, which is the motivation of multiple previous works, e.g., [20], [2]. In particular, Abdalkareem *et al.* [2] found that developers from small projects — as small as 168 commits — also chose to manually skip commits in CI to save time. These savings can be substantial for the projects in our studied dataset: test-suite runtime varies from project to project (median 2.3 mins, 75th percentile 26 mins) but, more importantly, saving full builds could save much higher cost (median 14 mins, 75th percentile 52 mins). Also, many builds (20%) take longer than 30 minutes [4]. Test-selection could save higher cost if it leaned harder towards skipping full builds, but we found in this study that this incentive is not yet strongly leveraged by our studied test selection techniques.

VIII. Related Work

A. Empirical Studies of CI and its Cost and Benefit

Multiple researchers focused on understanding the practice of CI, studying both practitioners *e.g.,* [24] and software repositories [60]. Vasilescu *et al.* studied CI as a tool in social coding [59], and later studied its impact on software quality and productivity [60]. Zhao *et al.* studied the impact of CI in other development practices, like bug-fixing and testing [65]. Stahl *et al.* [56] and Hilton *et al.* [24] studied the benefits and costs of using CI, and the trade-offs between them [23]. Lepannen *et al.* similarly studied the costs and benefits of continuous delivery [31]. Felidré *et al.* [12] studied the adherence of projects to the original CI rules [13]. Other recent studies analyzed testing practices [16], difficulties [43] and pain points [61] in CI.

The high cost of running builds is highlighted by many empirical studies as an important problem in CI [24], [23], [43], [61], [21] — which reaches millions of dollars in large companies, *e.g.,* at Google [24] and Microsoft [21]. People [23], [60] believe that the benefit of CI is mainly lying in the early fault detection. Others [24], [31] find that projects adopting CI are able to adopt pull requests and release in a shorter time. Some also find that CI can help developer team

in other areas such as providing a common build environment [23] and increasing team communication [56].

B. Approaches to Reduce Time-to-feedback in CI

A related effort for improving CI aims at speeding up its feedback by prioritizing its tasks. The most common approach in this direction is to apply test case prioritization (TCP) techniques *e.g.,* [35], [40], [11], [37], [10], [47], [67] so that builds fail faster. These techniques, even though not designed to work in CI environment, have been claimed to have a potential to provide CI users earlier fault observation. Another similar approach achieves faster feedback by prioritizing builds instead of tests [33]. Their paper grants higher priority to those builds that are more likely to fail according to the historical failing information and works well for those projects that have a ton of collision issues. Naturally, these kinds of techniques don't provide benefit in saving the cost. In this paper, we study both test-prioritization techniques as well as build-prioritization techniques in terms of advancement of failure observation and compare them with selection techniques.

C. Approaches to Reduce Cost of CI

A popular effort to reduce the cost of CI focuses on understanding what causes long build durations *e.g.,* [17], [58]. Thus, most of the approaches that reduce the cost of CI aim at making builds faster by running fewer test cases on each build. It is found that a ton of passing tests could be saved in this way [29]. Some approaches use historical test failures to select tests [21], [11]. Others run tests with a small distance to code changes [38] or skip testing unchanged modules [55].

Recently, Machalica *et al.* predicted test case failures using a machine learning classifier [36]. These techniques are based on the broader field of regression test-selection (RTS) *e.g.,* [66], [64], [18], [63], [62], [46], [45]. While these techniques focus on making every build cheaper, other work addresses the cost of CI differently: by reducing the total number of builds that get executed. A related recent technique saves cost in CI by not building when builds only include non-code changes [2], [1]. They firstly create a rule-based selection technique and then take advantage of machine learning algorithm to improve the accuracy. Then Jin and Servant propose a build strategy that developing team should skip those less informative passing builds through build outcome prediction. Finally, other complementary efforts to reduce build duration have targeted speeding up the compilation process *e.g.,* [8] or the initiation of testing machines *e.g.,* [15]. In this paper, we refer cost-reduction techniques as selection techniques. We pick techniques in both build-selection techniques and test-selection techniques and examines their performance in different cost-saving and fault-observation metrics.

D. Evaluation frameworks for similar techniques

Multiple research works focus on comparing cross-tool performance with an evaluation framework. Zhu *et al.* [66] propose a regression test selection framework to check the output against rules inspired by existing test suites for three techniques. Leong *et al.* [30] propose a test selection algorithm evaluation method and evaluate five potential regression test selection algorithms, finding that the test selection problem remains largely open. Najafi *et al.* [41] studied the impact of test execution history on test selection and prioritization techniques. Luo *et al.* [35] conduct the first empirical study comparing the performance of eight test prioritization techniques applied to both real-world and mutation faults and find that the relative performance of the studied test prioritization techniques on mutants may not strongly correlate with performance on real faults. Lou *et al.* [34] systematically created a taxonomy of existing works in test-case prioritization, classifying them in: algorithms, criteria, measurements, constraints, scenarios, and empirical studies.

Differently to these works, our study in this paper specifically targets the context of CI, and it has a broader focus than test prioritization or selection. Our study is the first to compare all the techniques proposed to reduce time-to-feedback or cost in CI, including prioritization and selection techniques, at test and build granularities. We performed observations comparing across 2 goals, 3 dimensions, 10 metrics, 2 granularities, and 10 techniques. Most of our observations required comparisons at broad scope. For example: we revealed the need for a new incentive in test selection to skip full test suites (to also save build-preparation time), which would not be relevant in studies outside the scope of CI.

IX. Conclusions and Future work

In this article, we performed the most exhaustive evaluation of CI-improving techniques to date. We evaluated 14 variants of 10 CI-improving approaches from 4 families on 100 real-world projects. We compared their results across 10 metrics in 3 dimensions. We derived many observations from this evaluation, which we then synthesized to understand the design decisions that helped each dimension of metrics, as well as those that had a negative impact on it. Finally, we provide a set of recommendations for future techniques in this research area to take advantage of the factors that we observe were beneficial, and we lay out also future directions to improve on those factors that were not. We lay out plans to combine approaches at test and build granularities to save further costs, and to combine selection and prioritization approaches to improve on the early observation of failures while also saving some cost. Such techniques could consider additional history-based prediction features, such as the project's code-change history, *e.g.,* [50], [51], [49], [53], [54], since test-execution history was beneficial for some techniques, *e.g.,* [21]. We also discuss the need of future metrics to capture the various characteristics of these techniques in a more holistic way. In the future, we will work on designing a comprehensive technique that combines selection and prioritization as well as build and test granularities to maximize the benefit of CI while reducing its cost as much as possible.

X. Replication

We include a replication package for our paper [28].

REFERENCES

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

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

[3] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[4] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 356–367. IEEE, 2017.

[5] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

[6] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 447–450. IEEE, 2017.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance*, pages 337–345. IEEE, 2008.

[8] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric. Build system with lazy retrieval for java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–654. ACM, 2016.

[9] 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 *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53. IEEE, 2020.

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[11] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[12] W. Felidré, L. Furtado, D. A. Da Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10, 2019.

[13] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[14] 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 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 87–97. ACM, 2018.

[15] A. Gambi, Z. Rostyslav, and S. Dustdar. Improving cloud-based continuous integration environments. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 797–798. IEEE Press, 2015.

[16] A. Gautam, S. Vishwasrao, and F. Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 495–498. IEEE, 2017.

[17] T. A. Ghaleb, D. A. da Costa, and Y. Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, pages 1–38, 2019.

[18] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[19] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE, 2017.

[20] F. Hassan and X. Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162. IEEE, 2017.

[21] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 483–493. IEEE, 2015.

[22] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48. IEEE, 2015.

[23] 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*, pages 197–207. ACM, 2017.

[24] 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*, pages 426–437. ACM, 2016.

[25] M. R. Islam and M. F. Zibran. Insights into continuous integration build failures. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 467–470. IEEE, 2017.

[26] R. Jain, S. K. Singh, and B. Mishra. A brief study on build failures in continuous integration: Causation and effect. In *Progress in Advanced Computing and Intelligent Engineering*, pages 17–27. Springer, 2019.

[27] X. Jin and F. Servant. A cost-efficient approach to building in continuous integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 13–25. IEEE, 2020.

[28] X. Jin and F. Servant. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration, Mar. 2020. Available at https://doi.org/10.5281/zenodo.4372963.

[29] A. Labuschagne, L. Inozemtseva, and R. Holmes. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 821–830, 2017.

[30] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco. Assessing transition-based test selection algorithms at google. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 101–110. IEEE, 2019.

[31] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[32] J. Liang. Cost-effective techniques for continuous integration testing. 2018.

[33] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698, 2018.

[34] Y. Lou, J. Chen, L. Zhang, and D. Hao. A survey on regression test-case prioritization. In *Advances in Computers*, volume 113, pages 1–46. Elsevier, 2019.

[35] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. Assessing test case prioritization on real faults and mutants. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 240–251. IEEE, 2018.

[36] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE, 2019.

[37] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, 2013.

[38] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[39] J. Micco. The state of continuous integration testing@ google. 2017.

[40] S. Mostafa, X. Wang, and T. Xie. Perfranker: prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 23–34, 2017.

[41] A. Najafi, W. Shang, and P. C. Rigby. Improving test effectiveness using test executions history: An industrial experience report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 213–222. IEEE, 2019.

[42] A. Ni and M. Li. Cost-effective build outcome prediction using cascaded classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 455–458. IEEE, 2017.

[43] G. Pinto, M. Rebouças, and F. Castor. Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In *Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 74–77. IEEE Press, 2017.

[44] M. Rebouças, R. O. Santos, G. Pinto, and F. Castor. How does contributors' involvement influence the build status of an open-source software project? In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 475–478. IEEE Press, 2017.

[45] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[46] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[47] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[48] SciTools Understand. Understand static code analysis tool. https://scitools.com/, 2020. [Online; accessed 02-March-2020].

[49] F. Servant. Supporting bug investigation using history analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 754–757. IEEE, 2013.

[50] F. Servant and J. A. Jones. History slicing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 452–455. IEEE, 2011.

[51] F. Servant and J. A. Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[52] F. Servant and J. A. Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *International Conference on Software Engineering*, pages 36–46, 2012.

[53] F. Servant and J. A. Jones. Chronos: Visualizing slices of source-code history. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013.

[54] F. Servant and J. A. Jones. Fuzzy fine-grained code-history analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 746–757. IEEE, 2017.

[55] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 689–699. IEEE, 2017.

[56] D. Ståhl and J. Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, pages 736–743, 2013.

[57] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.

[58] M. Tufano, H. Sajnani, and K. Herzig. Towards predicting the impact of software changes on building activities. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, ICSE '19, 2019.

[59] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 401–405. IEEE, 2014.

[60] 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 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[61] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 647–658. ACM, 2019.

[62] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[63] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[64] L. Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209. IEEE, 2018.

[65] 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*, pages 60–71. IEEE Press, 2017.

[66] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. A framework for checking regression test selection tools. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 430–441. IEEE, 2019.

[67] Y. Zhu, E. Shihab, and P. C. Rigby. Test re-prioritization in continuous testing environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 69–79. IEEE, 2018.