Does one CI-ze fit all? How Continuous Integration Performs in Different Contexts

Shujun Huang
Delft University of Technology
Delft, The Netherlands
s.h.huang@tudelft.nl

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

Sebastian Proksch
Delft University of Technology
Delft, The Netherlands
s.proksch@tudelft.nl

***Abstract*—Continuous Integration (CI) has become a fundamental practice of modern software engineering, widely adopted in both commercial and open-source projects to enhance development efficiency and software quality. Existing empirical research has examined the relationship between the effectiveness of CI adoption and project characteristics, but it often assumes CI as a uniform intervention and neglects the heterogeneity of its effects. In reality, the CI effectiveness is shaped by contextual factors such as project maturity, domain, programming language, or development practices. Moreover, many prior studies rely on convenience samples of open-source projects, leaving the influence of dataset composition largely unexamined. This study employs quota sampling to construct a diversified dataset, ensuring representation across product characteristics and project activities. Building on this foundation, we examine the adoption of CI both at the aggregate level and within stratified, quota-based subsets of the dataset. The results show that the impact of CI is sensitive to different dataset compositions. When the contextual distribution of projects changes, both the direction and magnitude of CI's effects can shift, emphasizing that empirical evidence on CI effectiveness is inherently context-dependent. These findings indicate that the effectiveness of CI cannot be generalized without considering project heterogeneity. Future research should explicitly control dataset composition to better isolate or explain contextual effects. Besides, applying suitable sampling strategies can also minimize contextual bias, and help generate more robust and generalizable evidence for CI studies.**

***Index Terms*—Continuous Integration, Contextual Factors, Project Heterogeneity, Quota Sampling, Dataset Composition**

I. Introduction

Continuous Integration (CI) has become a cornerstone of modern software engineering. Developers frequently merge their changes into a shared main branch and use CI automations to accelerate delivery, improve stability, and maintain software quality [1]. It has been widely adopted in both commercial [2] and open-source projects [3]. A substantial body of empirical work has examined the effects of CI [4, 5]. Several studies report improvements in productivity and delivery performance [6 – 8], while others emphasize challenges such as configuration complexity, maintenance overhead, and pipeline instability [9 – 11]. Crucially, earlier studies also suggest that CI's impact is not uniform, but strongly shaped by contextual factors, including project size, maturity, domain, programming language, and team composition [10, 12 – 17]. Not all projects benefit equally, some experience slower PR merging [7] or increased review workload that offsets productivity [18, 19]. From CI service selection to pipeline organization, the same practice can lead to contrasting outcomes across ecosystems [20 – 22]. Most prior studies focus on specific platforms (e.g., Travis CI) or narrow contexts, limiting the generalizability. In specialized domains such as machine learning [23] and high-performance computing [24], CI adoption remains underexplored and requires further investigation. All these prior studies seem to suggest that project context plays an important role, however, much of the available evidence is derived from a narrow set of open-source repositories, most prominently GitHub [25 – 27]. Given the platform's scale and practical resource constraints, researchers typically rely on purposive or convenience sampling [28]. While this strategy simplifies the dataset creation, it can limit representativeness and may lead to results that depend too heavily on specific project subsets and that do not generalize to the broader software ecosystem.

These limitations underscore a critical gap: *it remains unclear to what extent the effects of CI generalize and to what extent they are conditioned by contextual factors*. Although prior studies have acknowledged the moderating role of context [29], existing analyses are fragmented and largely restricted to single dimensions, offering little systematic comparison across multiple factors. To fill this gap, we systematically investigate how contextual diversity in a dataset influences empirical conclusions about CI. We address this through three research questions.

RQ1 How to build a contextually diverse dataset for empirical CI studies?

RQ2 How does the data distribution vary between the quota-based and stratified datasets?

RQ3 How does the impact of CI adoption across contexts differ from prior studies' findings?

We build a multi-context dataset using a quota sampling strategy. This dataset varies by project size, programming language, and other activity attributes. We compare its distributions at both aggregate and stratified levels to evaluate representativeness. We use this dataset to analyze the effects of CI on development activity and related measures and compare our results with prior studies to test their robustness and generalizability across contexts. This research design reveals the heterogeneous nature of CI's impact. It also offers practical

guidance on how future empirical studies can account for contextual diversity in both design and analysis.

Our results broadly confirm earlier findings but also demonstrate that context exerts a substantial moderating influence on multiple key indicators. In particular, programming language, quality level, and activity are shown to condition CI's effectiveness, yielding systematic differences in commit frequency, pull request and issue activity. These results indicate that context not only affects the magnitude of CI's impact but also determines its robustness and generalizability.

Overall, this paper presents the following main contributions:

- A combined approach for stratified and quota sampling, which provides a methodological reference for constructing datasets in other contexts for empirical studies.
- We show empirical evidence how contextual diversity systematically shapes the effects of CI, offering methodological guidance for the design of future studies or datasets.
- Building on the observed contextual diversity impact, we offer recommendations for the design and direction of future empirical experiments.

All data and scripts used in this study are available online [30].

II. Background and Related Work

Prior research in software engineering has already studied the practical aspects of dataset construction. In the following, we introduce the most relevant studies.

Empirical Studies. Empirical studies are essential for developing and validating knowledge in software engineering [31 – 33]. Most empirical research relies on large-scale datasets mined from open-source software (OSS) platforms such as GitHub, GitLab, and Bitbucket [34, 35]. These platforms provide versioned source code and valuable process data, like developer communication, CI activity, and review records [36]. Practical constraints (e.g., scope, resources, data accessibility) usually prevent researchers from analyzing the entire population, so they study subsets. This makes dataset construction a critical part, as it influences representativeness and generalizability of the findings [37]. Kitchenham et al. [33] proposed methodological guidelines for empirical software engineering, as aligning datasets with the research question is essential for reliable conclusions [37]. A dataset that inaccurately reflects the target population or real-world practices may introduce structural bias, which can raise concerns about the reliability of the findings [38]. Therefore, empirical studies not only ensure methodological rigor, but should also consider the representativeness and contextual relevance of their datasets.

Dataset Construction. Dataset construction is a crucial step in empirical software engineering, as the selected data directly impacts the reliability and external validity of the study [37]. It is often required to sample the available data, therefore, the chosen sampling strategy is an essential factor of empirical research design to reach external validity [28]. One major issue is the lack of comprehensive sampling frames, such as complete lists of software projects or developers, which prevent the application of probability-based sampling methods and lead to a *generality crisis* [28, 39]. As a result, many studies rely on convenience or purposive sampling [28]. A sampling strategy that is methodologically justified, clearly articulated, and appropriate for the study's objectives does not automatically invalidate research findings [39]. Nevertheless, beyond considering contextual effects in the analysis, dataset construction should also incorporate the diversity and address the potential impact of sampling bias on the results.

The Potential Impact of Contextual Factors. Existing research has increasingly highlighted the central role of contextual factors in shaping the effects of CI. Prior work has shown that elements such as programming language, project size, and team composition can substantially influence how CI manifests in practice, thereby defining its boundary conditions across projects [10, 12, 15]. These insights provide an important starting point for understanding CI's applicability. More recent efforts started to acknowledge the complexity of these influences. For example, Huang et al. identified a taxonomy for CI contexts with dimensions such as language, tooling, team structures, and quality levels, to systematically describe impact factors [17]. Rahman et al. found that CI's benefits differ between open-source and proprietary projects, suggesting that project type itself constitutes a salient contextual dimension [40]. Expanding on this, Silva et al. [41] recently investigated the relationship between CI and code review within ten closed-source projects, further highlighting how industrial constraints and internal team dynamics shape CI's impact on software quality differently than in open-source environments. Other studies further incorporate variables such as project age or total commit size into causal models as potential confounders [18], reflecting an awareness that lifecycle and activity baselines may condition the perceived impact of CI. Prior studies often examine single contextual dimensions, but their interplay shapes how CI works. We need systematic cross-context comparisons to uncover these mechanisms and to evaluate CI's robustness and generalizability across diverse settings. This approach aligns with cross-level analyses in information systems research and underscores the importance of contextual diversity in empirical CI [28, 42].

Representativeness and Generalizability. Research findings can be generalized beyond the original context only if the sample represents the target population and the study clearly defines the influencing factors and boundary conditions [42, 43]. Nagappan et al. [42] pose two critical questions for assessing generalizability: (1) Can findings from a small number of projects be extended to a broader range of projects? (2) Does a method or technique remain effective in different contexts? These reflect two dimensions of generalizability: horizontal transferability and contextual robustness. Both are essential for building knowledge that applies across settings. Tsang [43] defines three generalization types: *statistical*, *theoretical*, and *analytical* generalization through case/context comparison. In software engineering, theoretical and analytical generalization are more common, especially when probability sampling is not

feasible. The effectiveness of techniques varies due to the high context-specificity of SE practices and depends on organizational structure, team experience, programming languages, or project size [44]. Thus, even real-world project datasets cannot be assumed to be inherently representative [28]. Researchers must explicitly describe the sampling and clearly report on applicable contexts, boundary conditions, and assumptions to improve reproducibility and enable future replications.

Reproducibility in Empirical CI Research. Replication studies can build cumulative and verifiable knowledge in empirical research [32, 45 – 48]. Gómez et al. [49] classify replication into several types, including *Literal*, *Operational*, and *Conceptual* replication. Each type supports different validation goals and allows varying degrees of change in experimental design, data sources, or analysis methods. This classification shows that replication is not just repeating an experiment, it is a purposeful process that aims to identify which conditions influence the results. Furthermore, the integrity and quality of datasets largely determine whether a study is reproducible [50, 51]. In this study, we draw on prior designs and analytical frameworks but do not pursue a strict *literal replication*. The main reason lies in data availability and the evolution of CI platforms. While earlier work often focused on Travis CI [52, 53], the rise of GitHub Actions has introduced new data sources and practices [54]. This evolution made a more rigorous data selection necessary; as Santos et al. [55] argue, there is a critical need to monitor CI practices over time to ensure that datasets capture active and evolving CI usage rather than stale configurations. Their work underscores that the reliability of empirical findings depends on whether the sampled CI activities are representative of actual, ongoing development processes. Our study therefore constitutes a *conceptual replication*: we retain the overarching goal of examining CI's impact, but adapt data sources, sampling strategies, and analytical dimensions to fit current contexts. Differences across designs and datasets do not invalidate prior findings; instead, they reveal the conditions under which CI's effects vary. This perspective is essential for building cumulative knowledge.

III. Methodology

To systematically examine how dataset composition and contextual diversity influence empirical assessments of CI effectiveness, we divide our methodology into three distinct stages, an overview is presented in Figure 1. We started with constructing both a stratified and quota-based dataset (RQ_1) to represent contextual heterogeneity [56, 57]. We then extracted multiple development activity indicators informed by prior studies to analyse the overall CI effect. In an in-depth analysis, we then first compare the resulting datasets to assess their consistency and variation under different sampling strategies (RQ_2). Finally, we analyze the defined contextual strata and compare the results with prior empirical findings (RQ_3). The subsequent subsections will introduce the details for these distinct research phases.

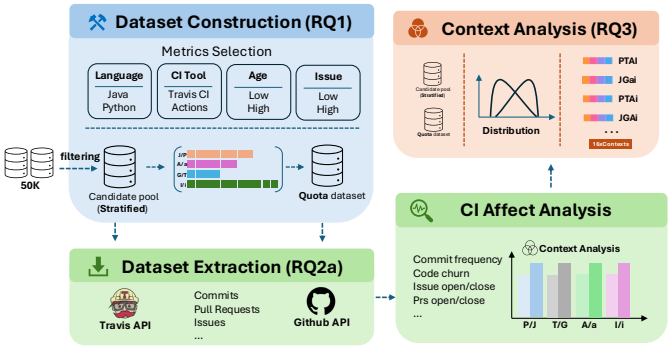


Fig. 1: Methodology

A. Dataset Construction

The dataset construction process involved the selection of sampling dimensions, data filtering, constructing candidate projects, and the implementation of sampling strategies.

Selection of Metrics. To represent context, we first needed to determine the metrics for stratification, which also function as the strata definition during sampling. The selection of context metrics was guided by the following principles: Firstly, the selected metrics should provide diversity while minimizing redundancy. For instance, commit frequency is highly correlated with project size, including both would reduce independence and hinder the feasibility of finding balanced samples. Secondly, we prioritized metrics that are widely recognized in prior literature, thereby enhancing comparability and interpretability. Moreover, we also considered different categories of contextual metrics identified in prior work [17]. Considering all of this, we selected four contextual metrics: *main programming language*, *CI tool type*, *project age*, *open issue ratio*. The first three have been individually examined in prior studies but rarely analyzed jointly in a systematic perspective. We selected Java and Python, as they represent contrasting development paradigms (compiled vs. interpreted), and Travis CI and GitHub Actions as two different CI tool types. The addition of *open issue ratio* is an exploratory metric that enables us to capture projects' responsiveness to issues and the quality focus. Together, these four metrics provide a balanced coverage of product, process, and quality perspectives, while reducing redundancy across dimensions. It is worth noting that while these metrics were selected to align with our study objectives and are not intended as fixed standards, researchers with different goals should appropriately adopt alternative dimensions of context.

Bucketing and Stratification. After defining the contextual dimensions, we established the sampling criteria. The same strategy applies to both stratified and quota sampling, with differences only in the sampling proportions across strata.

For each of the four dimensions, we constructed mutually exclusive strata. Categorical variables (programming language and CI tool type) were naturally divided into two categories. Numerical variables (project age and open issues ratio) were discretized into high and low categories using fixed-width

binning, which ensures consistent intervals independently of the sample distribution [58]. Together, these four axes define a stratification space of $2^4 = 16$ unique context combinations. Each project is unambiguously assigned to one stratum.

For ease of reference, we adopted a compact four-letter naming scheme to denote strata: the first letter indicates programming language (J = Java, P = Python), the second letter CI tool (T = Travis CI, G = GitHub Actions), the third letter project age (a = young, A = old), and the fourth letter for open issue ratio (i = low, I = high). For example, "JTAi" refers to a Java project using Travis CI that is young and exhibits short issue close times.

Candidate Pool and Filtering. We initially sampled 50,000 public repositories from GitHub (as of April 2, 2025 [59]). To construct the candidate pool, we applied the following filters:

- Excluded archived, forked, and mirror repositories;
- Required the use of GitHub Actions or Travis CI, with the main programming language restricted to Java or Python;
- Required an average frequency of at least one commit, release, or issue per month on the default branch;
- Required at least one active commit after 2024;
- Required at least one year activity before/after CI adoption.

Following prior work [18], we defined a project's CI adoption point as the timestamp of the first commit introducing a CI configuration file. For each project, we cloned the repository locally and traversed the commit history of the default branch to locate the first commit. We identified `.travis.yml` file for Travis CI and `.github/workflows/` directory for GitHub Actions. After filtering, we obtained a candidate pool of 1,580 projects, including 911 using GitHub Actions and 669 using Travis CI; 741 written in Python and 839 in Java.

Sampling. For stratified sampling, we preserved the natural distribution of the 1,580 candidate projects across all 16 contextual strata. For quota sampling, we applied equal allocation across strata, assigning *approximately* random 30 projects per stratum. The final quota sample that ensures context diversity comprises 407 projects, which represents about 30% of the original sample of 1,580 projects). Due to the difficulty of filling sufficient eligible projects in every stratum, we applied oversampling to strata containing less than 30 projects to ensure a balanced distribution across the dataset.

B. Data Extraction

After constructing the sample, we extracted data via the GitHub API [60]. For consistency, the analysis focused on the default branch, covering the full lifecycle of each project, from creation until April 2, 2025.

In the following, we will describe all metrics that were used to compare distributional shifts and relative changes before and after CI adoption, and to evaluate the robustness and moderating effects of context. Informed by prior work [18, 61], we extracted these indicators in one-month windows for both datasets. All timestamps were normalized and mapped to the beginning of each month to form the time bucket. Following prior work [18], we excluded a one-month transition window centered around the CI adoption date to reduce noise. The remaining timeline was labeled as *pre-CI* (months before the adoption bucket) and *post-CI* (the adoption month and subsequent months). To ensure comparability across contexts and sampling designs, we detail our statistical analysis procedures in the next section.

Commit Frequency. We first measured the commit activity by using the commit history of the default branch at a monthly granularity. We cloned each repository locally and retrieved its full default-branch history. For every commit, we use *author_date* to map the beginning of the corresponding calendar month to define its time buckets. For each project and month, we then count the total number of commits within the respective bucket. All commits recorded on the default branch are included. This measure provides a comprehensive estimate of the development rhythm and overall activity intensity. In the following, we denote this metric as *commit_frequency*.

Code Churn Size. We then measured the granularity of development activity by collecting the size of code changes per commit. Specifically, for each commit, we parse the corresponding diff file to extract the number of inserted (*insertions*) and deleted (`deletions`) lines, and define *code_churn_size* as: `churn` = `insertions` + `deletions`. Within each monthly time bucket, we compute the mean of churn values for all commits during that month. This metric reflects the average magnitude of changes per commit, providing an estimate of how intense the development activity is over time.

Modified Files. We use the number of modified files per commit to measure the scope and modularity of code changes. For each commit, we count the number of unique files that were added, deleted, or modified in its diff. At the monthly level, we compute the mean number of file changes, denoted as *modified_files*. This metric reflects the structural breadth of code modifications and complements the *code_churn_size* indicator: While code churn captures the line-level changes, the number of files changed represents the spread of modifications across the codebase. A smaller average number suggests more modular changes, whereas a larger value may indicate broader refactoring or cross-module adjustments.

Issue activity. We measure issue activity by aggregating issue opening and closing events on a monthly basis, denoted as *issue_open_count* and *issue_close_count*. For each issue, we record its creation time `created_at` and closing time `closed_at`. Since opening and closing an issue can occur in different months, the two counts are measured independently.

Pull request activity. Similar to issues, we aggregate the monthly counts for open/close pull requests (PRs) to measure the intensity of collaboration and integration activities, denoted as *prs_open_count* and *prs_close_count*. Specifically, for each pull request, we record its creation time `created_at` and closing time `closed_at`, and map them to the corresponding calendar month buckets. As opening and closing a PR can occur in different months, we collected them independently.

PR latency. We measure the efficiency of code review and integration by collecting the lifetime of PRs. For each PR closed within a given month, we compute its latency as the time interval between creation and closure, measured in hours. At the monthly level, we take the mean of latencies for all PRs closed in that month (*pr_latency*). This metric reflects the average response time of the review and integration process, showing how efficiently teams process incoming changes.

PR Merge Ratio. We quantify the integration success through the proportion of merged PRs among all closed PRs within a given month (*prs_merge_ratio*). A PR is considered merged if the platform metadata contains a valid *merged_at* timestamp. This metric reflects the success rate and stability of the review and integration process. A higher *prs_merge_ratio* indicates that code review and quality assurance procedures are effectively leading to integration, whereas a decline may suggest increased review stringency or coordination challenges.

Commit Merge Ratio. In contrast to *PR Merge Ratio*, which solely focuses on reviewed pull requests, *Commit Merge Ratio* provides a broader perspective on integration frequency. It measures the proportion of merge commits among all commits within a given month, capturing how often developers synchronize work from different branches into the mainline, denoted as *commit_merge_ratio*. A higher value typically reflects a more distributed workflow with frequent integration events, whereas a lower ratio suggests a more linear or centralized development pattern with fewer merges.

Releases Activity. We evaluate the delivery rhythm through its version publication activity, denoted as *releases_count*. Release dates are determined by the `published_at` timestamp, or `created_at` when unavailable, each timestamp is mapped to its corresponding calendar months. This indicator characterizes the regularity and responsiveness of software delivery. An increase after CI adoption suggests that the automation may contribute to shorter release cycles.

C. Data Analysis

For the collected datasets, we first compared their statistical distributions and then applied time-series modeling to analyze the effect of CI adoption.

Distribution Analysis. We first compared the raw data distribution of the two sampled datasets (*stratified* and *quota*) to examine potential differences introduced by distinct sampling strategies. We focused on the activity metrics collected in RQ2 and characterized the overall distributional patterns. If no significant differences observed, it would suggest that the data distribution of the two sampling strategies remains comparable. Conversely, discrepancies would indicate that the sampling design may introduce differences for particular project types, thereby influencing the data representativeness and the validity of subsequent analyses. To quantify these differences, we reported the key descriptive statistics (mean, median, and standard deviation). We further applied non-parametric Wilcoxon rank-sum tests [3, 7, 62] and report Cliff's Δ [7, 62] to capture both statistical detectability and practical significance.

Time Series Analysis Method. To examine the potential causal effects of CI adoption, we applied time-series modeling to each project's monthly activity metrics. As CI adoption usually occurs in the middle or later stages of a project's lifecycle, we can discern an intervention point that separates the pre-adoption and post-adoption phases. This design allows each project to be viewed as a natural experiment, where the CI introduction represents a clear intervention event. So we adopted the *Regression Discontinuity Design (RDD)* framework as a mixed-effects linear regression, which is widely used in empirical studies to estimate causal effects from observational time series data [18, 62], and we basically followed the existing practices [18], filtered out the 1% outliers and checked the variance inflation factors (VIF < 3) for multicollinearity, and report explanatory power via marginal (R^2_m) and conditional (R^2_c) coefficients of determination for mixed models.

This method assumes that, without intervention, the time series remains smooth around the cutoff. Therefore, any discontinuity observed at the time of CI adoption can be attributed to the effect. Specifically, we fit separate linear trends before and after the adoption point to capture the immediate level change (γ) and the change in slope (δ) induced by CI, thereby identifying its short-term and long-term impacts. Based on the monthly time series of each project, we estimate the following linear regression model following established practices [18]:

$$Y_t = \beta_0 + \beta_1 \cdot time_t + \beta_2 \cdot intervention_t + \beta_3 \cdot time\_after_t + \epsilon_t$$

where Y_t denotes the observed metric at time t; $time_t$ represents the number of months since the project's inception; $intervention_t$ is an indicator equal to 1 after CI adoption and 0 otherwise; $time\_after_t$ indicates the number of months since CI adoption; and ϵ_t is the error term. In this specification, β_0 captures the pre-adoption baseline level, β_1 the pre-adoption trend, β_2 the immediate level shift at adoption, and β_3 the post-adoption change in trend. Estimating these parameters enables us to quantify both the instantaneous and sustained effects of CI adoption on project development. We modeled both the *pre-* and *post-* adoption period in a unified regression framework, and then derived the slope difference (β_3) as the indicator of the directional effect of CI adoption.

IV. Creating a Representative Dataset (RQ1)

Sampling in empirical software engineering research often lacks true randomness [28, 39, 63]. Different sampling strategies can influence both the composition and quality of the resulting datasets. This research question aims to explore whether alternative sampling methods can be used to construct CI datasets with more diverse project contexts. To this end, we compare two sampling strategies: stratified sampling and quota sampling. Both methods utilize the same stratification variables but differ in the sampling approach.

Quota & Stratified distribution. Figure 2 compares the distributions of two continuous variables: *open issue ratio* and *project age* under stratified and quota sampling. The left panels display empirical cumulative distribution functions (ECDFs)

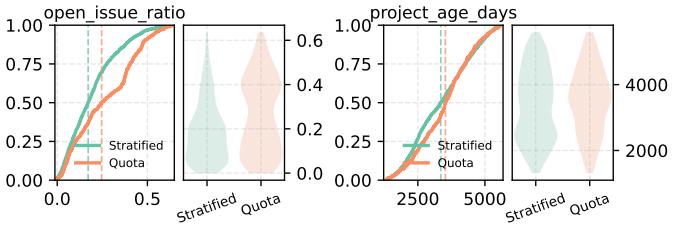


Fig. 2: Quota, Stratified sampling Comparison

with vertical dashed lines marking medians; the right panels show violin plots to visualize the density. For *open issue ratio*, the ECDF of the stratified sample is concentrated around shorter durations, with a steep rise and a lower median, suggesting that highly responsive projects are relatively over-represented. In contrast, the quota sample maintains a more balanced coverage across the full range and explicitly retains long-tail projects with a higher open issues ratio, thus capturing greater process heterogeneity. This difference is reinforced in the violin plots, where quota sampling exhibits broader vertical dispersion, while stratified sampling compresses tail distributions. For *project age*, the two samples show very similar medians, stratified sampling is slightly skewed toward younger projects, whereas quota sampling incorporates more mature repositories. The violin plots make this trend more evident: quota sampling produces wider distributions and heavier tails, mitigating the underrepresentation of older projects.

Regarding categorical variables, stratified sampling naturally preserves the proportions, while quota sampling balances the categories. In our case, the stratified sampling across languages and CI platforms results in 741/839 Python/Java projects and 911/669 projects using GitHub Actions/Travis CI. The quota sampling balances these dimensions. It is noteworthy that in practice, the distribution of languages and tools is not strongly skewed, this may come from the current sampling only including two languages and CI tools, which limits the representation of broader ecosystem trends.

Overall, the two sampling strategies serve complementary purposes. Stratified sampling preserves external validity by mirroring the population distribution, but may underrepresent rare or long-tail phenomena. In contrast, quota sampling enhances internal validity and contextual diversity by deliberately retaining extreme and boundary cases. Given that long-tail patterns often encode critical mechanisms of CI heterogeneity, quota sampling provides a stronger basis for identifying boundary conditions and context-dependent variations.

Context distribution. Table I presents the distribution of projects across the 16 contextual strata, revealing several systematic patterns that reflect the heterogeneity of the CI ecosystem. First, some combinations in strata (e.g., PGAI and JTaI) are heavily underrepresented, especially in high open issue ratio bucket. These rare combinations likely correspond to legacy systems with prolonged maintenance cycles and limited development agility, which are relatively uncommon in modern open-source environments. In contrast, strata

TABLE I: Strata distribution overview

Stratum	Description	Count	Cum.%
PGai	Python, GitHub Actions, young, low	304	19.2
JGai	Java, GitHub Actions, young, low	290	37.6
JTAi	Java, Travis CI, old, low	263	54.2
PTAi	Python, Travis CI, old, low	204	67.2
JGAi	Java, GitHub Actions, old, low	106	73.9
PGaI	Python, GitHub Actions, young, high	78	78.9
PTai	Python, Travis CI, young, low	60	82.5
PGAi	Python, GitHub Actions, old, low	51	85.8
JGaI	Java, GitHub Actions, young, high	51	89.1
JTai	Java, Travis CI, young, low	48	92.3
JTAI	Java, Travis CI, old, high	47	95.2
JGAI	Java, GitHub Actions, old, high	24	96.6
PTAI	Python, Travis CI, old, high	24	98.2
PTaI	Python, Travis CI, young, high	12	99.0
JTaI	Java, Travis CI, young, high	10	99.6
PGAI	Python, GitHub Actions, old, high	7	100.0

representing younger and highly efficient projects, such as PGai (304 projects) and JGai (290 projects), dominate the dataset. This skew toward "young and responsive" combinations indicates that modern CI adoption is concentrated among active communities that rapidly integrate automation and feedback-driven development. Second, two CI tools show pronounced asymmetries. GitHub Actions exhibits substantial representation in younger and low-latency strata (e.g., PGai and JGai), reflecting its rapid adoption by newer projects that prioritize automation flexibility and tighter integration with the GitHub ecosystem. Conversely, Travis CI appears more frequently in older yet still efficient contexts (e.g., PTAi with 204 projects and JTAi with 263 projects), suggesting that long-lived repositories are more likely to adopt Travis CI at an early age. This also aligns with the evolution patterns of CI platforms: On the GitHub platform, GitHub Actions naturally become the mainstream CI tool for emerging projects due to its smooth platform integration and compatibility. Finally, the scarcity of high-issue-duration strata under GitHub Actions (e.g., PGAI, JTaI) implies open source projects tend to sustain higher development agility. Taken together, these distributional patterns demonstrate that the quota-based dataset achieves a comprehensive representation of both mainstream and boundary context projects.

V. Dataset Comparison (RQ2)

To illustrate the data distribution, we analyzed multiple activity-related metrics in both stratified and quota sample datasets. Table II presents the complete results.

The comparative statistics reveal a consistent and nuanced divergence between the stratified sampling S_d and the quota sampling Q_d across multiple dimensions of development activity. While the absolute magnitude of Cliff's Δ remains modest ($\Delta \leq 0.06$ for all significant metrics), the extremely low p-values obtained from the Wilcoxon rank-sum tests confirm that these are systematic rather than random fluctuations. This indicates that the sampling strategy influences the representation of project types in the dataset, and thereby shapes how the CI ecosystem is interpreted. From the development

activity perspective, the S_d dataset shows slightly higher values than the Q_d sample across most metrics, including *commit_frequency* ($\Delta = +0.04$, $p \approx 5.63 \times 10^{-40}$), *modified_files* ($\Delta = +0.02$, $p \approx 2.43 \times 10^{-13}$), and both *issue* and *PR* activities ($\Delta \approx +0.03$–0.06, $p < 10^{-20}$). These consistently positive trends suggest that S_d sampling tends to preserve projects with more frequent collaboration and maintenance activity, while the Q_d dilutes the representation of highly active projects, leading to a slightly more conservative picture of ecosystem vitality. Besides, there are no significant differences observed in *pr_latency* ($\Delta \approx 0.00$, $p \approx 0.98$) *commit_merge_ratio* ($\Delta \approx 0.00$, $p \approx 0.27$), while a small but significant negative difference is observed in *pr_merge_ratio* ($\Delta = -0.01$, $p \approx 3.6 \times 10^{-3}$), suggesting that quota sampling slightly reduces the merged PRs proportion.

In contrast, the *release_count* shows that projects in S_d sample exhibit a significantly higher mean number of releases ($\Delta = +0.02$, $p \approx 5.8 \times 10^{-23}$), indicating that this sampling approach tends to capture more mature release cycles and complete development lifecycles. The stratification pattern in Table I explains this trend: in S_d, the *PGai* and *JGai* strata take the top two positions. These strata are relatively young projects and employ modern CI workflows, exhibit a high level of automation, and maintain a low open-issues ratio. Such projects benefit from shorter feedback loops and continuous integration processes, resulting in higher development activity and a more regular release rhythm. Their dominance within S_d largely explains the slightly elevated activity-related metrics, such as commit frequency, PR volume, and release count. In contrast, the Q_d provides a more balanced representation across contextual types. It includes a wider range of repositories, including those with uncommon characteristics. As a result, Q_d captures a broader but less active segment of the ecosystem, reflecting the diversity of development practices.

TABLE II: Statistics of development activity in Stratified/Quota dataset

Metric(m)	Cohort	Mean	Median	Std	Cliff's Δ	Mann–Whitney p
commit_frequency	Stratified	29.86	9.00	93.70	+0.04*	$5.63\text{x}10^{-40}$
	Quota	28.82	7.00	82.54		
code_churn_size	Stratified	1324.67	55.20	46822.75	+0.00	0.192
	Quota	1707.45	54.41	72795.16		
modified_files	Stratified	6.47	1.93	94.54	+0.02*	$2.43\text{x}10^{-13}$
	Quota	7.12	1.83	166.74		
issues_open_count	Stratified	5.37	1.00	26.33	+0.04*	$5.96\text{x}10^{-38}$
	Quota	4.81	1.00	16.54		
issues_close_count	Stratified	4.53	1.00	26.54	+0.06*	$9.16\text{x}10^{-89}$
	Quota	3.77	0.00	14.00		
prs_open_count	Stratified	7.68	2.00	21.46	+0.03*	$2.12\text{x}10^{-24}$
	Quota	7.83	1.00	29.58		
prs_close_count	Stratified	7.56	1.00	21.49	+0.03*	$1.99\text{x}10^{-26}$
	Quota	7.70	1.00	29.48		
prs_latency	Stratified	697.42	116.72	2385.04	+0.00	0.983
	Quota	751.42	118.59	2608.50		
prs_merge_ratio	Stratified	0.78	0.94	0.31	–0.01*	$3.59\text{x}10^{-3}$
	Quota	0.78	0.95	0.32		
commit_merge_ratio	Stratified	0.12	0.00	0.17	+0.00	0.271
	Quota	0.12	0.00	0.18		
releases_count	Stratified	0.70	0.00	3.07	+0.02*	$5.80\text{x}10^{-23}$
	Quota	0.64	0.00	2.77		

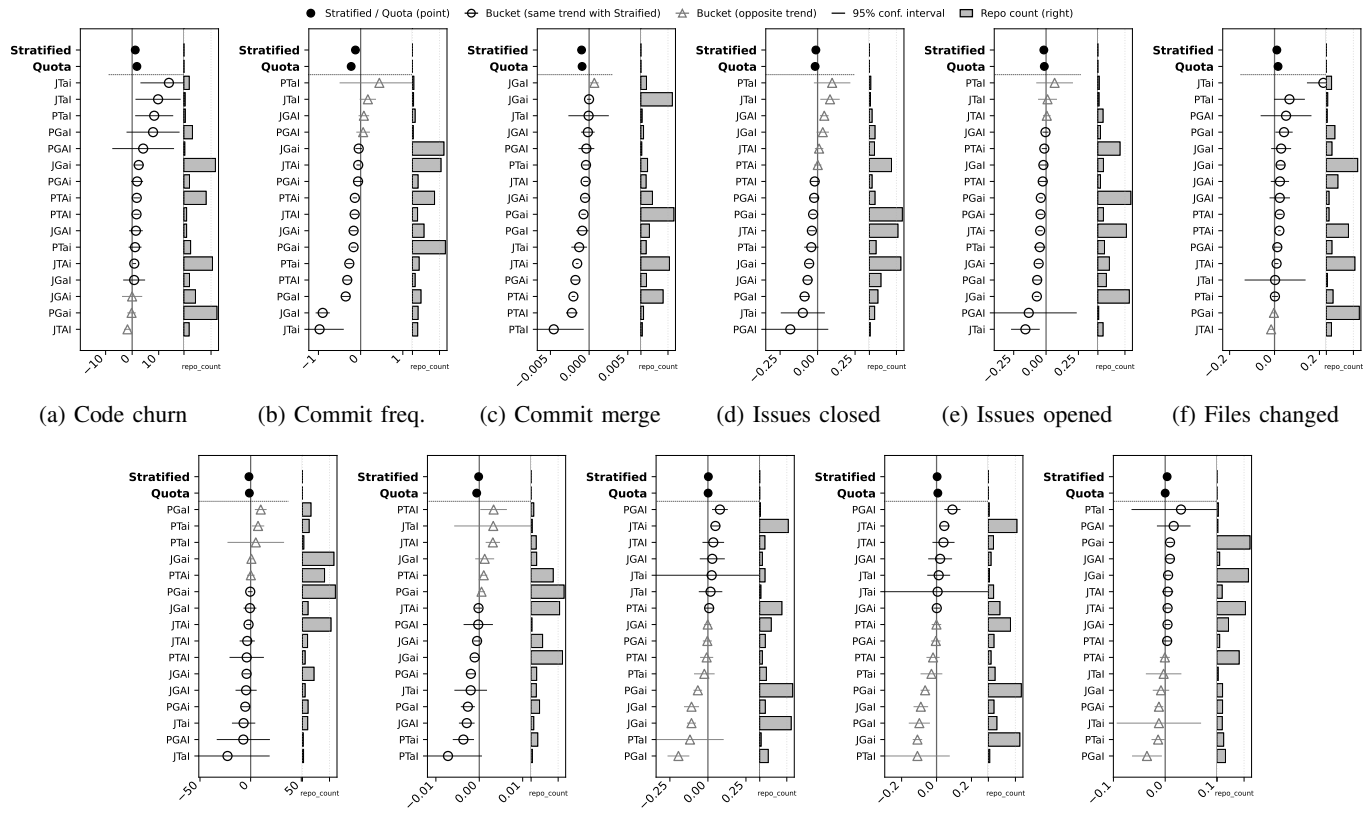
Discussion. A closer comparison shows that the two sampling strategies differ mainly in contextual composition, as each strategy determines which parts of the CI ecosystem are empirically visible. S_d adheres more closely to the natural distribution, thereby amplifying the visibility of dominant contexts, typically highly active or automation-intensive projects. While this preserves the authenticity of the ecosystem, it decreases the visibility of minority contexts, which may include legacy systems or some domains that diverge from the mainstream. In contrast, Q_d forces the balance across different context strata, rebalancing the dataset to increase the visibility of minority project types. This adjustment might not entirely align with the real distribution pattern, but it increases the visibility of uncommon or hidden contexts. As such, in Table II, these differences in contextual composition can translate into the measurable variations in activity metrics.

Overall, these findings lead to an important insight: that sampling is not only a simple filtering process but a contextual lens that shapes what can be empirically generalized. As in stratified sampling, retaining the natural distribution reveals realism but may overlook the rare practices that could challenge general assumptions. In contrast, enforcing balancing in quota sampling increases the context coverage but may break the natural prevalence, possibly obscuring mainstream patterns. Recognizing this trade-off is essential for interpreting the empirical study findings, whether these findings reflect the core patterns of the ecosystem, or the boundaries of their generalizability across diverse project contexts.

VI. Context Effect (RQ3)

To further analyze how CI impacts vary across contexts, we applied the RDD model to both the stratified (S_d) and quota-based dataset (Q_d) across the 16 different context buckets. We compare the changes in the regression slopes *before and after CI adoption*, Figure 3 shows the results. Each plot represents one development dimension, and the vertical axis shows 16 contextual combinations (e.g., `JGAi`), jointly defined by four intersecting attributes: programming language (Java/Python), CI tool (Travis CI/GitHub Actions), project age (old/new), and open issue ratio (fast/slow). The two solid black circles denote the results for stratified and quota samples and serve as a reference for comparison. Hollow markers show context-specific results: open circles indicate contexts following the *same directional trend* as the stratified baseline, whereas light gray triangles denote *opposing trends* that contradict the findings from a stratified sample. The horizontal lines for each datapoint indicate the 95% confidence intervals to describe model uncertainty. The gray bars on the right reflect the number of repositories within each context, depicting the relative prevalence of this context in the ecosystem.

Overall trend. Overall, S_d and Q_d show broadly consistent post-adoption trends, indicating mildly positive or stable CI effects. Nevertheless, variations in magnitude and direction can be observed. For instance, in collaboration-related metrics such as *pr_latency* and *pr_closse_count*, Q_d partially diverges



(a) Code churn (b) Commit freq. (c) Commit merge (d) Issues closed (e) Issues opened (f) Files changed

(g) PR latency (h) PR merge (i) PRs closed (j) PRs opened (k) Releases

Fig. 3: Forest plots of the 11 CI-related metrics.

from S_d, suggesting the ecosystem structure and project composition moderate effects. S_d is biased by the prevalence of the different dimensions: Buckets with a higher number of projects have a stronger impact on the overall stratified trend and therefore dominant ecosystem characteristics. As such, S_d reflects mainstream patterns. Nonetheless, numerical dominance alone does not fully explain the observed variations. Statistically significant differences also persist across buckets, revealing that even large, mainstream projects do not always share consistent behavioral patterns. In some metrics, such as *pr_latency*, major buckets exhibit opposite slopes relative to the stratified baseline, suggesting that the contextual influences within these sub-ecosystems are heterogeneous and potentially counteracting. In addition to the trend analysis, it is important to consider the absolute counts of each bucket. Buckets such as `JGAi` and `PGai` contain 250–300 projects each. The generally higher count and activity levels found in the Java buckets might amplify the effect of the higher variance. In summary, high-volume context projects show a noticeable weighting effect on the aggregate outcome, but the extent of the shift ultimately depends on the directional coherence of their contextual behaviors.

Contextual Differences. By examining the varying context buckets in Figure 3 that either *confirm* (○), but particularly those that *contradict* (△) the overall trend, several patterns emerge. Travis-related subgroups exhibit a pronounced rightward shift, with most combinations showing above zero, indicating strong and more concentrated positive changes. In contrast, GitHub Actions (GHA) displays smaller, but more evenly distributed positive effects, suggesting steadier and more sustained improvements. While Travis tends to amplify collaboration and delivery dynamics, it also introduces greater variability, especially in issue activity, file modification, and release counts. This is likely due to its external integration, making its effectiveness more sensitive to configuration quality and project setup. GHA, as a native GitHub service, integrates continuous feedback directly into pull request workflows, leading to smaller but more stable improvements.

From the programming language perspective, most Java projects exhibit positive long-term trends, like in delivery and collaboration (PR merge ratio/issue activity). Java projects also deviate more often from the overall trend, which illustrates a higher heterogeneity in CI adaptation. In contrast, Python projects show smaller but more consistent effect improvements, typically concentrating near neutral-to-positive ranges. These patterns imply that lightweight ecosystems benefit from CI adoption more smoothly and homogeneously. Regarding project age, younger projects tend to show more positive trends in collaboration-related metrics, like PR latency and open issue count, indicating that early CI adoption facilitates faster feedback and issue resolution. In contrast, older projects display positive effects in code and release-related dimensions,

including modified files, PR open/close counts, and release frequency, which implies that more mature projects see stimulation of key development activities through CI. Interestingly, younger projects diverge more frequently from the overall trend, suggesting that early-phase CI adoption amplifies contextual variability: Younger projects adopt new CI tools and workflows faster, but the effects vary more. In contrast, older projects integrate CI as an incremental improvement to mature workflows, rather than driving radical changes.

Finally, projects with a higher ratio of open issues exhibit more sustained positive effects overall, which suggests that maintaining a larger backlog of unresolved issues fosters sustained team engagement and continuous interaction. At the same time, a high amount of parallel issues may lead to delayed responsiveness and slower issue resolution. For instance, high open-issue contexts show more negative effects in PR latency, implying that review and merging processes tend to slow down under coordination loads. These results imply that issue management dynamics intensify contextual variability, yielding divergent CI effects across ecosystems.

Comparison with Prior Studies. Prior research [18, 61, 62] generally agrees that CI can improve collaboration and delivery efficiency, but empirical results remain mixed, ranging from strong positive effects to negligible impact. At the ecosystem level, early studies primarily focus on Travis CI. Earlier work reported that CI adoption increased productivity [61] and integration frequency with smaller code churn size but may also increase the PR latency and workload [18], and the file modification size will increase [64], while other research [62] found no significant influence on development activity like merge ratio or code change size.

Our results may help to explain, when datasets are dominated by ecosystems such as Travis, that the distributions may show high variance, with some projects experiencing substantial gains, while others remain neutral or slightly negative. This internal heterogeneity dilutes aggregate patterns, leading to a “no effect” or "stronger effect" performance at the macro level. In collaboration and delivery metrics, prior studies offer similarly mixed results. While some report that CI adoption doubles release frequency and reduces PR merge time [3], others found that more than 71% of projects experienced slower PR merging after CI adoption [7]. Recent evidence also indicates that although rapid releases can accelerate issue fixing, they may prolong overall integration delays [19].

Our study further reveals that CI effects exhibit both coherence and divergent patterns across contexts. These discrepancies also stem from underlying differences in the dataset composition. When ecosystem and tool distributions are balanced, CI effects become more coherent, supporting that ecosystem dominance bias shapes aggregate outcomes. For productivity-related metrics such as commit frequency and code churn per commit, our findings align with previous work [62]: CI adoption does not necessarily increase individual productivity and may even slightly reduce it. This pattern is more pronounced in Travis-based projects. In contrast, GitHub Actions supports more stable individual productivity while improving coordination efficiency (e.g., reduced PR latency), reflecting a trade-off between coordination gains and individual output stabilization. Overall, although some of our results align with prior findings, they suggest that divergent conclusions across CI studies largely arise from contextual composition and ecosystem bias. When samples concentrate on specific languages, tools, or project types, observed effects may reflect ecological characteristics rather than general causal mechanisms. By systematically stratifying, our analysis reveals the heterogeneous structure of CI impact, that the contextual composition determines whether CI’s impact is visible or obscured in aggregate analyses.

VII. Discussion

The previous sections presented concrete results to the research questions. We will now reflect on their impacts on future work.

Broadening Contextual Exploration in CI studies. Our selection of four contextual dimensions was informed by prior studies and aimed to maximize the coverage of contextual heterogeneity while avoiding highly colinear metrics. Future research could further extend the analysis across a broader range of contextual dimensions. For instance, dimensions like team composition, team size, developer experience distribution, or organizational hierarchy can be included. Moreover, the inclusion of longitudinal or cross-project comparative analysis could help to systematically reveal how these characteristics interact and evolve over time. Such extensions would enable a deeper understanding of CI effectiveness, and provide more context-sensitive guidance for process improvements.

Lack of Benchmark Datasets. One observation from prior CI-related empirical studies is the widespread reliance on certain well-known datasets, like TravisTorrent [53]. While these resources facilitated early research, some of them are no longer actively maintained or accessible, limiting the reproducibility of prior findings. This issue stems from datasets being maintained by individual researchers without long-term sustainability guarantees. For future improvements, the research community should prioritize the development and long-term maintenance of benchmarks, e.g., through automated data collection. Establishing a clear framework for dataset contribution or curation would help to sustain collective ownership and encourage long-term community engagement. Together, these efforts can help CI research infrastructure transform into a stable, transparent, and collaborative basis for empirical work.

Data Stratification Challenges. During dataset construction, we observed that certain strata, such as projects with long issue resolution times, test coverage, or contribution volumes, are intrinsically rare. While some distributions align with intuitive assumptions, such as larger projects that tend to have more contributors and high activity, it remains difficult to identify large projects with both fewer contributors and lower activity. Increasing the strata size naturally leads to a decrease in the number of projects that fit these criteria, which is also our concern in deciding the strata size. In future improvements,

researchers should explicitly consider the representation of lower-frequency strata. We argue that rarity should not be conflated with irrelevance, as uncommon cases may reflect important patterns specific to highly complex projects or specific developments. Meanwhile, preserving the original distribution remains essential for uncovering the natural ecosystem-level trends. Therefore, future sample project selection should keep the balance between rare cases representation and maintaining the natural distribution to align with the research goal.

Insufficient Transparency in Replication. During our replication, we encountered challenges due to incomplete or underspecified dataset construction details in some studies. In many cases, papers do not present their filtering procedures, project selection criteria, or do not explain their rationale. For instance, we found studies that claim to select the *size* of projects without explaining what they are referring to and how they measure the metric, e.g., *source code size* versus *team size*. This lack of transparency introduces ambiguity and hinders reproducibility. We believe that future studies should including explicit definitions of key metrics, selection criteria, and any filtering steps of the dataset construction process.

Implicit Selection Bias. While dataset quality has been discussed in the related work section, we do not imply that data quality is solely an issue of completeness or accuracy. In CI research, we argue that data quality should also be defined relative to the study purpose. A dataset curated for evaluating test coverage may not be suitable for studies targeting sociotechnical factors. Furthermore, when project selection criteria are tightly coupled with theoretical assumptions, such as focusing on mature, active, or popular repositories, the resulting dataset ceases to be a neutral empirical sample. Instead, it becomes a theory-laden construct, shaped not only by the characteristics of the development ecosystem but also by the researcher's prior expectations. This raises a fundamental epistemological question: are we uncovering causal mechanisms that objectively exist in practice, or are we merely validating a conceptual framework we embedded in the sampling process? When datasets are implicitly preconditioned on factors like high activity or popularity, the observed outcomes, such as success rates or CI performance, may reflect the selection filter more than the general behavior of software projects. In such cases, empirical findings risk becoming self-fulfilling confirmations, reinforcing preconceptions rather than challenging or expanding them. We caution future studies to critically reflect on whether or how the sampling design can affect the empirical findings. In addition to a clear dataset documentation, researchers can conduct comparative or replication studies to evaluate the consistency and generalizability of conclusions.

VIII. Threats to Validity

Despite efforts to improve internal validity, several threats remain. We stratified projects by factors such as programming language and CI platform to mitigate confounding effects; however, selecting optimal control variables for specific research designs is beyond this study's scope. Our findings suggest that future work should more explicitly align control variables with project context, research objectives, and data sources in relative research. In addition, we use GitHub Actions build success or failure as a proxy for build quality. Although widely adopted, this binary measure provides only a coarse approximation as successful builds may still contain quality issues, while failures may result from transient infrastructure problems. These limitations may introduce bias and weaken the observed effects. Future studies could improve validity by incorporating richer quality indicators, such as test coverage, static analysis results, or historical failure patterns.

Our findings are also subject to potential threats to external validity. While we mitigate the risk of inherited design assumptions by replicating studies from peer-reviewed, top-tier venues, data source differences remain a key concern. Prior studies primarily use Travis CI, whereas we analyze GitHub Actions. Platform-level differences in build orchestration, scheduling, and error handling may influence observed outcomes independently of the targeted development practices. To address this concern, we compared our results from GitHub Actions with the corresponding Travis CI findings reported in the original studies. This comparison provides initial evidence of cross-platform reproducibility, but it does not guarantee full generalizability across CI platforms.

IX. Conclusion

Continuous Integration has become a fundamental practice of modern software engineering. Existing research often assumes CI to be a uniform intervention and neglects the heterogeneity of its effects. In reality, the effectiveness of CI is shaped by contextual factors such as project maturity, domain, programming language, or development practices. In this paper, we have examined the influence of the dataset composition both on the distribution of the resulting data and on the empirical conclusions that can be drawn from such a dataset.

Revisiting the Research Questions. We employed quota-based sampling to create a more diverse dataset (RQ_1). The observation of data distribution reveals that this approach puts more emphasis on less common project characteristics (RQ_2). Furthermore, we found that prior studies often relied on stratified sample datasets, which can potentially jeopardize the generalizability of previous results (RQ_3).

Implications. These findings indicate that sampling strategies can noticeably influence statistical results, as the observations can be shaped not only by the studied variables but also by the underlying dataset composition. Therefore, researchers should interpret experimental results with careful consideration of dataset composition to minimize potential bias. Also, future empirical studies are encouraged to conduct experiments across more diverse environments to enhance the generalizability and interpretability of conclusions regarding CI effects.

Data Availability Statement

All data and scripts used in this study have been made available in our online appendix [30].

REFERENCES

[1] R. Schneider, "Continuous integration: improving software quality and reducing risk," *Software Quality Professional*, vol. 10, no. 4, p. 51, 2008.

[2] M. Hilton, N. Nelson, D. Dig, T. Tunnell, and D. Marinov, "Continuous integration (CI) needs and wishes for developers of proprietary code," Corvallis, OR: Oregon State University, Dept. of Computer Science, Tech. Rep., 2016.

[3] 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*, 2016, pp. 426–437.

[4] E. Soares, G. Sizilio, J. Santos, D. A. Da Costa, and U. Kulesza, "The effects of continuous integration on software development: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 3, p. 78, 2022.

[5] J. Santos, D. Alencar da Costa, and U. Kulesza, "Investigating the impact of continuous integration practices on the productivity and quality of open-source projects," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 137–147.

[6] N. Cassee, B. Vasilescu, and A. Serebrenik, "The silent helper: the impact of continuous integration on code reviews," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 423–434.

[7] J. H. Bernardo, D. A. da Costa, and U. Kulesza, "Studying the impact of adopting continuous integration on the delivery time of pull requests," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 131–141.

[8] Y. Gupta, Y. Khan, K. Gallaba, and S. McIntosh, "The impact of the adoption of continuous integration on developer attraction and retention," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 491–494.

[9] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, pp. 1095–1135, 2020.

[10] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[11] A. Debbiche, M. Dienér, and R. Berntsson Svensson, "Challenges when adopting continuous integration: A case study," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2014, pp. 17–32.

[12] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, 2021.

[13] O. Elazhary, M. D. Storey, N. A. Ernst, and A. Zaidman, "Do as I do, not as I say: Do contribution guidelines match the GitHub contribution process?" in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 286–290.

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

[15] 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 *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 475–478.

[16] M. Wessel, J. Vargovich, M. A. Gerosa, and C. Treude, "GitHub actions: the impact on the pull request process," *Empirical Software Engineering*, vol. 28, no. 6, p. 131, 2023.

[17] S. Huang and S. Proksch, "A taxonomy of contextual factors in continuous integration processes," *IEEE Transactions on Software Engineering*, 2025.

[18] 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 *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71.

[19] D. A. d. Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan, "The impact of rapid release cycles on the integration delay of fixed issues," *Empirical Software Engineering*, vol. 23, no. 2, pp. 835–904, 2018.

[20] X. Jin and F. Servant, "What helped, and what did not? an evaluation of the strategies to improve continuous integration," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 213–225.

[21] D. Saraiva, D. A. Da Costa, U. Kulesza, G. Sizílio, J. G. Neto, R. Coelho, and M. Nagappan, "Unveiling the relationship between continuous integration and code coverage," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 247–259.

[22] I. Saidani, A. Ouni, M. W. Mkaouer, and F. Palomba, "On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent," *Information and Software Technology*, vol. 138, p. 106618, 2021.

[23] J. a. H. Bernardo, D. A. Da Costa, S. Q. d. Medeiros, and U. Kulesza, "How do Machine Learning Projects use Continuous Integration Practices? An Empirical Study on GitHub Actions," in *International Conference on Mining Software Repositories*, 2024.

[24] T. Gamblin and D. S. Katz, "Overcoming Challenges to Continuous Integration in HPC," *Computing in Science & Engineering*, 2022.

[25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, pp. 2035–2071, 2016.

[26] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "A systematic mapping study of software development with github," *Ieee access*, vol. 5, pp. 7173–7192, 2017.

[27] V. Cosentino, J. Luis, and J. Cabot, "Findings from github: methods, datasets and limitations," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 137–141.

[28] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022.

[29] J. Benjamin, J. Mathew, and R. T. Jose, "Study of the contextual factors of a project affecting the build performance in continuous integration," in *2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS)*. IEEE, 2023, pp. 1–6.

[30] "Online appendix," https://doi.org/10.5281/zenodo.18302337, 2025, zenodo.

[31] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 108–119.

[32] F. Q. Da Silva, M. Suassuna, A. C. C. França, A. M. Grubb, T. B. Gouveia, C. V. Monteiro, and I. E. dos Santos, "Replication of empirical studies in software engineering research: a systematic mapping study," *Empirical Software Engineering*, vol. 19, pp. 501–557, 2014.

[33] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[34] L. Zhang, J.-H. Tian, J. Jiang, Y.-J. Liu, M.-Y. Pu, and T. Yue, "Empirical research in software engineering—a literature survey," *Journal of Computer Science and Technology*, vol. 33, pp. 876–899, 2018.

[35] N. Eghbal, *Working in public: the making and maintenance of open source software*. Stripe Press, 2020.

[36] "GitHub - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/GitHub, [Accessed 24-05-2025].

[37] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," *Guide to advanced empirical software engineering*, pp. 285–311, 2008.

[38] M. F. Bosu and S. G. MacDonell, "A taxonomy of data quality challenges in empirical software engineering," in *2013 22nd Australian Software Engineering Conference*. IEEE, 2013, pp. 97–106.

[39] B. Amir and P. Ralph, "There is no random sampling in software engineering research," in *Proceedings of the 40th international conference on software engineering: companion proceeedings*, 2018, pp. 344–345.

[40] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran, "Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, 2018, pp. 8–14.

[41] R. Silva, P. Silva, C. Bezerra, A. Uchôa, and A. Garcia, "Unveiling the relationship between continuous integration and code review: A study with 10 closed-source projects," in *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 2025, pp. 393–404.

[42] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 466–476.

[43] E. W. Tsang, "Generalizing from research findings: The merits of case studies," *International Journal of Management Reviews*, vol. 16, no. 4, pp. 369–383, 2014.

[44] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, "The case for context-driven software engineering research: generalizability is overrated," *IEEE Software*, vol. 34, no. 5, pp. 72–75, 2017.

[45] K. Popper, *The logic of scientific discovery*. Routledge, 2005.

[46] R. M. Lindsay and A. S. Ehrenberg, "The design of replicated studies," *The American Statistician*, vol. 47, no. 3, pp. 217–228, 1993.

[47] M. Shepperd, N. Ajienka, and S. Counsell, "The role and value of replication in empirical software engineering results," *Information and Software Technology*, vol. 99, pp. 120–132, 2018.

[48] 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*. ACM, 2019, pp. 647–658.

[49] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[50] M. M. Rosli, E. Tempero, and A. Luxton-Reilly, "Evaluating the quality of datasets in software engineering," *Advanced Science Letters*, vol. 24, no. 10, pp. 7232–7239, 2018.

[51] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of management information systems*, vol. 12, no. 4, pp. 5–33, 1996.

[52] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: an explorative analysis of Travis CI with github," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[53] ——, "Travistorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 447–450.

[54] A. Khatami, C. Willekens, and A. Zaidman, "Catching smells in the act: A github actions workflow investigation," in *IEEE International Conference on Source Code Analysis and Manipulation, SCAM 2024*. IEEE, 2024, pp. 47–58.

[55] J. Santos, D. A. da Costa, S. McIntosh, and U. Kulesza, "On the need to monitor continuous integration practices," *Empiricalz Software Engineering*, vol. 30, no. 5, p. 125, 2025.

[56] A. S. Acharya, A. Prakash, P. Saxena, and A. Nigam, "Sampling: Why and how of it," *Indian journal of medical specialties*, vol. 4, no. 2, pp. 330–333, 2013.

[57] R. Iliyasu and I. Etikan, "Comparison of quota sampling and stratified random sampling," *Biom. Biostat. Int. J. Rev*, vol. 10, no. 1, pp. 24–27, 2021.

[58] S. Garcia, J. Luengo, J. A. Sáez, V. Lopez, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 734–750, 2012.

[59] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in GitHub for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[60] G. Docs, "REST API endpoints for GitHub Actions," https://docs.github.com/en/rest/actions?apiVersion=2022-11-28, accessed 28-05-2025.

[61] 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*, 2015, pp. 805–816.

[62] S. Baltes, J. Knack, D. Anastasiou, R. Tymann, and S. Diehl, "(no) influence of continuous integration on the commit activity in GitHub projects," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, 2018, pp. 1–7.

[63] N. Amir, F. Jabeen, and S. Niaz, "A brief review of conditions, circumstances and applicability of sampling techniques in computer science domain," in *2020 IEEE 23rd International Multitopic Conference (INMIC)*. IEEE, 2020, pp. 1–6.

[64] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality: The case of firefox," *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, 2015.