Role of CI Adoption in Mobile App Success: An Empirical Study of Open-Source Android Projects

Xiaoxin Zhou
Faculty of Information
University of Toronto
Toronto, Ontario, Canada
xiaoxin.zhou@utoronto.ca

Taher A. Ghaleb
Department of Computer Science
Trent University
Peterborough, Ontario, Canada
taherghaleb@trentu.ca

Safwat Hassan
Faculty of Information
University of Toronto
Toronto, Ontario, Ontario
safwat.hassan@utoronto.ca

Abstract

Mobile apps face strong pressure for fast and reliable updates. Continuous Integration (CI) helps automate builds, tests, and releases, but its impact on mobile development remains underexplored. Despite the widespread use of CI, little is known about how it affects development activity, release speed, and user-facing outcomes in mobile projects. Existing studies mostly focus on CI adoption in general-purpose software, providing limited insight into mobile-specific dynamics, such as app store visibility and user engagement. In this paper, we analyze open-source Android apps to (1) compare CI adopters and non-adopters, (2) characterize adoption patterns using activity and bug metrics, and (3) assess pre/post adoption changes and user-facing outcomes. We observe that CI adopters are larger and more active, with faster and more regular releases. CI adoption is concentrated in integration- and reliability-intensive categories (e.g., finance and productivity) and is associated with higher Google Play Store engagement (more downloads and reviews) without lower ratings. Overall, CI adoption aligns with practices that support sustained delivery, higher project visibility, and stronger user engagement in mobile ecosystems.

CCS Concepts

• **Software and its engineering → Software configuration management and version control systems**; • **Information systems → Mobile information processing systems**.

Keywords

Continuous Integration, CI adoption, Android apps, Empirical study, Mining software repositories

ACM Reference Format:
Xiaoxin Zhou, Taher A. Ghaleb, and Safwat Hassan. 2026. Role of CI Adoption in Mobile App Success: An Empirical Study of Open-Source Android Projects. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793302.3793343

1 Introduction

Mobile applications (hereafter referred to as mobile *apps*) are an integral part of daily life. To succeed, developers must rapidly deploy new releases that deliver requested features or fix reported bugs [25]. Continuous Integration (CI) has become a widespread practice in software development, including mobile apps, helping to automate building, testing, packaging, and deployment [2, 6, 9]. Previous work studied practices to improve build performance, such as reducing build execution time and minimizing build failures [3, 11, 12, 20]. Other research has examined CI adoption in Android apps, focusing primarily on configuration and maintenance practices. However, little is known about how CI adoption in mobile apps affects development productivity, release speed, and user-facing success metrics [10, 23, 35].

In this paper, we examine the association between CI adoption and various aspects of software development in 2,542 open-source Android projects. Specifically, we investigate how CI usage is associated with development practices, release patterns, and app success. To this end, we address the following research questions (RQs):

RQ1: How do CI adopters differ from non-adopters? We compare Android projects with and without CI adoption. CI adopters are larger and more active, with higher source lines of code (+24%), more contributors (+260%), more commits (+280%), and faster releasing (+41% tags/week, 18% shorter inter-tags). Both have similar Google Play Store listing rates ($\approx$ 33%), but CI adopters have $\sim 5\times$ more downloads and $\approx 2.2\times$ more reviews, especially in utility and integration-heavy categories.

RQ2: What are the distinct patterns of CI adopters? We focus on CI adopters with labeled bug issues (n = 442) and normalize activity metrics. Among 442 CI adopters with labeled bugs, we find that faster releasing does not increase bug density. The *"Best"* group has short tag lifetimes (8 days) and low bug density (0.23 issues/KLoC), while the *"Worst"* group has longer cycles (32 days) and higher bug density (2.73 issues/KLoC).

RQ3: What are the evolution characteristics of CI adoption? A before/after analysis of 172 projects shows post-adoption increases in commits (+59%), PRs opened (+75%), and PRs merged (+72%), but merges slowed (median ratio = 1.6). The User-facing metrics on Google Play show no significant improvement.

RQ4: What explains the variation in before- and after-adoption performance? Clustering 61 projects by before/after change patterns reveals three CI adoption trajectories: (1) higher activity with shorter releases, (2) lower activity with longer releases, and (3) higher activity with longer releases, reflecting diverse outcomes shaped by project practices and pipeline setups.

Overall, this paper makes the following contributions.

- A publicly available dataset [40] linking CI adoption, repository metrics, and Google Play Store outcomes.

MSR '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793343

- Models (Random Forest and k-means clustering) that reveal patterns in CI adoption across different metrics (e.g., commits, pull requests (PRs), release speed, bug density).
- A dataset and analysis of how CI adoption affects development throughput, release speed, and user outcomes.
- Insights into the variability of post-adoption performance, showing that higher activity does not necessarily correlate with faster releases.

We organize the rest of this paper as follows. Section 2 presents our study setup. Section 3 discusses our empirical analyses and results. Section 4 discusses the implications of our work. Section 5 describes possible threats to the validity of our study. Section 6 reviews work related to CI and mobile apps. Finally, Section 7 concludes our paper and suggests future directions.

2 Study Setup

This section presents the setup of our study dataset. We explain how we collect and preprocess the data to address our research questions. We break the analysis and data preprocessing into three steps: 1) Collect Data, 2) Data Preprocessing, and 3) CI Data Analysis, as shown in Figure 1 and described below. We make our final clean dataset publicly available on Figshare [40].

2.1 Data Collection

For this study, we collect a dataset of Android repositories from GitHub (in July 2025) using a source that has been widely adopted in prior research [10],

which focuses on open-source Android apps and provides publicly accessible information on CI configurations. Building on this foundation, we systematically gather and curate the repositories following the steps outlined below.

2.1.1 ***Collecting potential "Android" repositories.*** We collect Android projects from GitHub using the official GitHub REST API[1]. GitHub hosts more than 67 million repositories in total. To narrow down our dataset, we search for repositories with the `android` topic[2], or the word `android` is mentioned in the project title, description, or read-me files. We also search for projects with "Java", "Kotlin", or "Dart" as their primary programming language. In addition, we work on repositories with more than 100 stars to ensure that our dataset consists of mature and actively used projects. We follow established guidelines used in prior studies [7, 39], which also use a 100-star threshold to ensure selecting popular projects and avoid toy ones. This step, conducted in July 2025, resulted in the identification of 48,396 repositories.

2.1.2 Removing toy projects: To further refine the dataset, we follow prior work [10] to exclude projects without a user interface by verifying the presence of an `AndroidManifest.xml` file containing at least one `activity` tag. In Android, activities define the screens of an application; therefore, the existence of a manifest file with an activity indicates that the project represents a functional app rather than a demo or library project.

We then exclud toy projects (e.g., tutorials, sample apps, and library projects) that mimic the structure of real Android apps but do not represent actual mobile applications [10]. This filtration step ensures that the selected projects are indeed Android apps, thereby enhancing the accuracy and relevance of our dataset for analysis. At the end of this step, we obtain 5,214 repositories.

[1]https://docs.github.com/en/rest
[2]https://github.com/topics/android

2.2 Data Preprocessing

To ensure that our analysis captures meaningful signals of continuous integration and continuous delivery (CI) activity, we first preprocess the dataset of GitHub repositories. Our goal is to focus on repositories that demonstrate evidence of versioning, issue tracking, and sustained development over time. The following filtering steps are applied. We should note that we update our dataset in October 2025 to incorporate the most recent repository information.

2.2.1 Remove repositories with no tags: In GitHub repositories, tags can serve as mechanisms for tracking the history of software updates.

Since our analysis focuses on the speed of updates, we examine the publication dates associated with the project tags to estimate update frequency and, indirectly, contributor efficiency. To avoid noise from inactive or abandoned projects, we apply a filter requiring that each repository contain at least one tag (i.e., more than zero updates). After implementing this filtering process, we identify 2,690 repositories suitable for analysis.

2.2.2 Remove repositories with no issues: In GitHub repositories, contributors and other interested users can report problems, bugs, or feature requests by creating issues. These issues serve as a communication channel between users and project maintainers, providing valuable feedback about software quality and functionality. To reduce ambiguity and focus only on repositories with observable issue-tracking activity, we apply a filter to exclude all repositories with zero reported issues. Combined with our earlier filter that excludes repositories without tags, this process left us with 2,560 repositories. This ensures that the dataset reflects projects with both measurable update histories and at least minimal evidence of issue reporting, allowing for a more meaningful analysis of development activity and code quality.

2.2.3 Remove repositories with no activities: For this particular analysis, we use October 1, 2025, as the cutoff date and select repositories that show activity, defined as having at least one commit, within the 90 days prior. This approach focuses the study on recently active repositories, ensuring that our conclusions are timely and relevant. After applying this criterion, we identify 2,542 Android repositories for inclusion in our study. The 90-day activity window follows prior work, including Moriconi et al. [27] and Khelifi et al. [21], which use a 90-day period to ensure selecting actively maintained projects.

2.3 Identifying CI adopters

In this phase, we analyze the final dataset of **2,542** Android repositories to (i) identify which projects adopt CI and which do not, and (ii) determine which projects are published on the Google Play Store. Following prior work [10], we identify projects that adopt CI by searching for configuration files associated with popular CI services (e.g., GitHub Actions). Each service defines its pipeline through one or more YAML files placed at specific paths in the

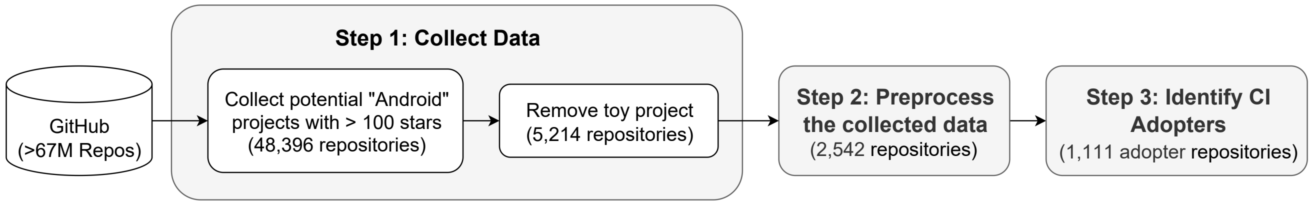
GitHub (>67M Repos)
Step 1: Collect Data
Collect potential "Android" projects with > 100 stars (48,396 repositories)
Remove toy project (5,214 repositories)
Step 2: Preprocess the collected data (2,542 repositories)
Step 3: Identify CI Adopters (1,111 adopter repositories)

Figure 1: Data collection and preprocessing pipeline for CI analysis.

repository. Table 1 summarizes these services and their corresponding file patterns. Using this mapping, we classify repositories into two groups: *CI adopters*, which contain at least one recognized configuration file, and *non-adopters*, which do not. As a result of this classification, we identify 1,111 CI adopters repositories and 1,431 non-adopters repositories that did not.

Table 1: List of CI services with their corresponding YAML file path patterns [10].

CI service	Path (regular expression)
GitHub Actions	`r'\.github/workflows/.*\.(yml|yaml)$'`
Travis CI	`r'\.travis\.yml$'`
CircleCI	`r'(\.circleci/config\.yml|circle\.yml)$'`
GitLab CI	`r'\.gitlab-ci\.yml$'`
Azure Pipelines	`r'azure-pipelines\.yml$'`
AppVeyor	`r'(\.appveyor\.yml|appveyor\.yml)$'`
Bitbucket	`r'bitbucket-pipelines\.yml$'`

In addition to contrasting CI adopters with non-adopters, we incorporate an app-distribution signal by verifying whether each repository corresponds to an application listed on the Google Play Store. Specifically, for every repository page, we apply a regular expression to extract candidate Google Play Store URLs from project metadata (e.g., README). We then programmatically request each candidate URL and mark the app as 'exists' only if the response indicates a valid listing (e.g., HTTP 200 and presence of canonical Play Store markers), thereby minimizing false positives from soft 404s or region-restricted entries.

3 Empirical Analysis and Results

In this section, we present the results of our empirical analyses across four research questions (RQs). For each RQ, we describe the motivation, outline our approach, and summarize the key findings.

3.1 RQ1: How do CI adopters differ from non-adopters?

3.1.1 ***Motivation.*** CI has been widely adopted in software development as a practice that can support more efficient workflows and maintainable code. However, it is not clear how repositories that adopt CI differ from those that do not in real-world open-source Android projects. Understanding these differences is important for characterizing CI adoption patterns and identifying traits associated with active or well-maintained projects. This RQ addresses these differences by examining the observable characteristics of CI adopters compared to non-adopters.

3.1.2 ***Approach:*** We collect and analyze repository-level metrics, including the number of issues, contributors, source lines of code (SLOC; computed using the open-source tool `sloc`[3]), project lifetime, median tag interval, issue density (issues per KLoC), commit frequency, and release/tag frequency, using a 90-day activity window preceding October 1, 2025.

Step 1: CI Data Collection. We detect and extract CI configurations (e.g., workflow files and run histories) and align them with each repository's activity timeline.

Step 2: Data Preprocessing. We deduplicate records, normalize units, and filter out repositories missing critical information (i.e., those without issue histories, with effectively zero lifetime, such as projects created via a single bulk import) or lacking tag/release records. Repositories with incomplete or inconsistent metadata are excluded to ensure valid comparisons.

Step 3: Classification and Google Play Store Linking. Repositories are classified into CI adopters (CI) and non-adopters (Non-CI), which we sometimes refer to interchangeably as CI vs. non-CI for brevity. We link each GitHub repository to its corresponding Google Play Store app (when available) to study user engagement metrics. For this analysis, we restrict our attention to repositories that are active within the 90 days preceding October 1, 2025, ensuring that engagement measures reflect recent activity. Additionally, for category-level analyses, we assign each app to its primary Google Play Store category (e.g., Finance) and focus only on categories containing at least five repositories to improve statistical reliability.

Step 4: Complexity and Evolution Analysis. Using the filtered dataset (df_{clean}), we compute four repository-level metrics to compare CI adopters and non-adopters, as follows:

(1) *Project lifetime.* For each repository (r), we take the project creation timestamp (created_at_r) and the last commit timestamp (last_commit_r), and compute

$$\text{ProjectLifetime}_r = \text{last\_commit}_r - \text{created\_at}_r \quad \text{(in days)}.$$

This measures the active development span observed in history.

(2) *Median tag (release) lifetime.* For each repository, we collect all release tag dates $\{t_1, \ldots, t_n\}$, sort them by time, form consecutive gaps $\Delta_i = t_i - t_{i-1}$ (days), and define

$$\text{MedianTagLifetime}_r = \text{median}\{\Delta_2, \ldots, \Delta_n\}.$$

If a repository has $n < 2$ tags, the quantity is undefined and we remove these repositories from our analysis. This captures a typical release speed while being robust to outliers.

[3]https://github.com/flosse/sloc

(3) *Issues density.* We proxy issue pressure by normalizing the total number of issues by code size (source lines of code, SLOC):

$$\text{IssuesDensity}_r = \frac{\text{NoIssues}_r}{\text{SLOC}_r} \times 1000 \quad \text{(issues per KLOC).}$$

If $\text{SLOC}_r = 0$, we set the metric to 0 and remove these repositories from our analysis.

(4) *Tag (release) frequency.* We measure how often releases occur over the observed lifetime. Let NoTags_r be the number of tags and ProjectLifetime_r in days; then

$$\text{ReleaseFreqWeek}_r = \frac{\text{NoTags}_r}{\text{ProjectLifetime}_r} \times 7 \quad \text{(tags/week).}$$

If $\text{ProjectLifetime}_r = 0$, we set 0 and remove these repositories from our analysis.

Step 5: Statistical comparison. To assess whether the distributions of these metrics differ significantly between CI adopters and non-adopters, we apply the Mann–Whitney U test [24], a nonparametric test for comparing two independent groups.

Step 6: Preprocessing and Visualization. We remove $\pm\infty$, drop rows with `NaN` in computed fields, and (when plotting) apply $\log_{10}$ transforms with small constants to avoid $\log(0)$ issues (e.g., +1 for counts, +0.001 for densities/frequencies). We visualize each metric via split violin plots (CI vs. non-CI) with median overlays, using consistent hue ordering and sample sizes in the legend.

3.1.3 ***Findings:*** We present results from a comparative analysis of CI adopters and non-adopters across multiple repository metrics.

Observation 1.1: CI adopters are more mature projects in terms of size, contributors, and project age. Figure 2 shows log-scaled beanplots comparing Non-CI (n=200) and CI adopters (n=545), and Table 2 summarizes key repository-level metrics, reporting medians for each group, the absolute difference (Diff.), and relative change (Δ %) across metrics including SLOC, number of contributors, project age, commits, release frequency, median tag interval, and issue density. CI adopters are consistently larger and more active: median SLOC is +23.8% (p = 0.008), number of contributors +260.0% (p = 0.001), and project age slightly longer +4.0% (p = 0.529). Development intensity is higher, with +279.9% more commits (p = 0.001), +41.6% more tags/week (p = 0.001), and a shorter median inter-tag time −17.9% (p = 0.030). Issue density is also higher (+54.6%, p = 0.008), likely reflecting greater usage, richer reporting practices, or domain complexity rather than lower quality.

CI adopters stand out as larger, more active, and faster-releasing projects. The higher issue density appears driven by scale and observability rather than poor quality, highlighting that CI adoption correlates with stronger delivery discipline and greater development capacity in mobile software contexts.

Observation 1.2: CI-adopting apps show higher user engagement on Google Play Store.

Among repositories active within the 90 days preceding October 1, 2025, the share with a corresponding Google Play Store listing is similar for CI adopters and non-adopters (CI: 177/545 = 32.5%, Non-CI: 69/200 = 34.5%), indicating that CI adoption does not increase the likelihood of being listed. Instead, the advantages appear *post-listing*: CI adopters exhibit substantially higher user engagement. Figure 3 and Table 3 summarize metrics across ***246*** repositories (Non-CI n = 69, CI n = 177). Median downloads for CI adopters are *five times* higher (+400.0%, p = 0.039), and median ratings are over *double* (+121.7%, p = 0.019). Average star ratings are slightly higher for CI adopters (+1.7%), but not statistically significant (p = 0.211). These results suggest that CI-adopting apps achieve stronger uptake and user interaction post-publication.

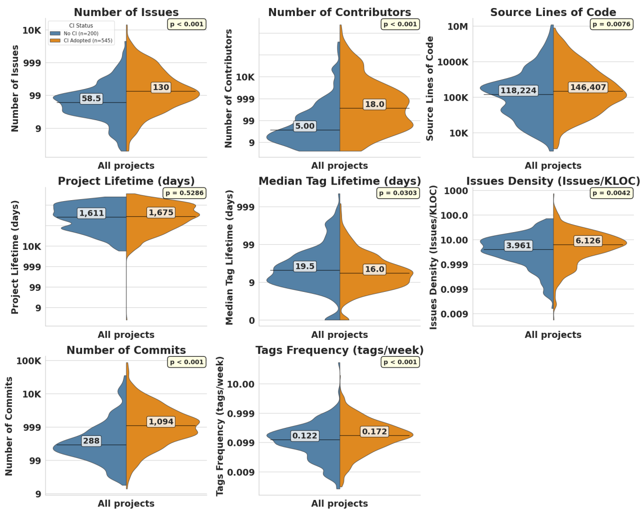

Figure 2: The 90-day activity repository metrics comparison: CI vs Non-CI projects.

Table 2: Repository Metrics: CI vs. Non-CI (Medians; $n_{\text{non-CI}}$=200, n_{CI}=545).

Metric	Non-CI	CI	Diff.	Δ (%)
Number of Issues	58.5	130	+71.5	+122.2
Number of Contributors	5	18	+13	+260.0
Source Lines of Code	118 224	146 407	+28 183	+23.8
Project Lifetime (days)	1611	1675	+64	+4.0
Median Tag Lifetime (days)	19.5	16.0	-3.5	-17.9
Issues Density (Issues/KLOC)	3.961	6.1	+2.2	+54.7
Number of Commits	288	1094	+806	+279.9
Tags Frequency (tags/week)	0.12	0.17	+0.05	+41.0

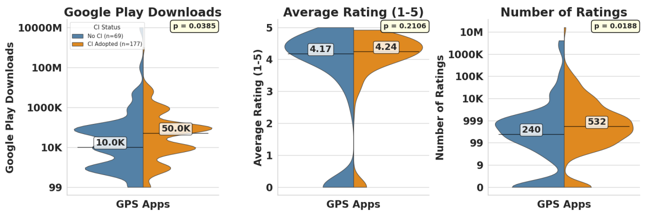

Figure 3: The 90-day activity Google Play Store (GPS) metrics for CI adopters vs Non-CI adopters.

Observation 1.3: CI adoption skews toward utility-style apps, particularly Tools and Productivity, and dominates categories where reliability and frequent updates are critical. Table 4

Table 3: Google Play Store (GPS) Metrics: CI vs. Non-CI (Medians; $n_{\text{non-CI}}$=69, n_{CI}=177).

Metric	Non-CI	CI	Diff.	Δ (%)
GPS Downloads	10 000	50 000	+40 000	+400
GPS Avg Rating (1–5)	4.17	4.24	+0.07	+1.7
GPS Number of Reviews	240	532	+292	+121.7

summarizes categories with at least five repositories (as described in Step 3). CI adopters are the majority in every category, averaging ≈71.5% overall (153/214). CI adoption is most prevalent in categories with frequent updates or reliability-critical features: *Finance* (10/10, 100%), *Maps & Navigation* (5/5, 100%), *Social* (6/6, 100%), *Communication* (10/12, 83.3%), and *Personalization* (7/8, 87.5%). Utility-heavy categories like *Tools* (64/96, 66.7%) and *Productivity* (21/33, 63.6%) also show clear majorities. Entertainment- and media-oriented categories are closer to parity: *Photography* (2/5, 40%) and *Health & Fitness* (3/5, 60%). Overall, the results can be explained as CI adoption is concentrated in apps that rely on frequent updates, strict security, or changing APIs, while content-focused apps tend to adopt CI less. These results encourage developers to adopt CI, especially in apps that require frequent updates, high reliability, or strong integration, indicating where CI practices may be most useful. To mitigate concerns about non-representative samples, we exclude categories with fewer than five apps, reducing the dataset from $n_{\text{non-CI}} = 69$, $n_{\text{CI}} = 177$ to $n_{\text{non-CI}} = 61$, $n_{\text{CI}} = 153$.

Table 4: Google Play Store Category Breakdown (sorted by total repositories): CI vs. Non-CI. Categories with the highest CI shares are marked *bold.*

Category	CI	Non-CI	Total	CI%
TOOLS	64 (26.0%)	32 (13.0%)	96 (39.0%)	66.7
PRODUCTIVITY	21 (8.5%)	12 (4.9%)	33 (13.4%)	63.6
GAME	10 (4.1%)	3 (1.2%)	13 (5.3%)	76.9
MUSIC_AND_AUDIO	9 (3.7%)	4 (1.6%)	13 (5.3%)	69.2
COMMUNICATION	10 (4.1%)	2 (0.8%)	12 (4.9%)	83.3
FINANCE	**10 (4.1%)**	**0 (0.0%)**	**10 (4.1%)**	**100.0**
ENTERTAINMENT	6 (2.4%)	2 (0.8%)	8 (3.3%)	75.0
PERSONALIZATION	7 (2.8%)	1 (0.4%)	8 (3.3%)	87.5
SOCIAL	**6 (2.4%)**	**0 (0.0%)**	**6 (2.4%)**	**100.0**
PHOTOGRAPHY	2 (0.8%)	3 (1.2%)	5 (2.0%)	40.0
MAPS_AND_NAV	**5 (2.0%)**	**0 (0.0%)**	**5 (2.0%)**	**100.0**
HEALTH_AND_FITNESS	3 (1.2%)	2 (0.8%)	5 (2.0%)	60.0
TOTAL	**153 (71.5%)**	**61 (28.5%)**	**214 (100%)**	**76.9**

RQ1 Summary: Among repositories active in the 90 days before October 1, 2025, CI adopters are larger, more active, and release more frequently. For Google Play Store apps, CI correlates with substantially higher user engagement, while listing likelihood is similar for adopters and non-adopters. CI adoption is particularly common in utility- and integration-heavy categories, highlighting where frequent updates and reliability matter most.

Table 5: Repository-level metrics used in our analyses.

Repository-level metric	DT*	Description
Source_Lines	N	Source lines of code for the repository.
no_Contributors	N	Number of unique contributors who committed.
Project_lifetime (months)	N	Age of the project in months (project creation date to end of observation).
(Average # of pull requests) / month	N	Mean number of pull requests opened per month (PRs/month).
(Average # of commits) / month	N	Mean number of commits per month (commits/month).
# of CI services	N	Count of distinct CI services configured in the repository.
GitHub_Actions	C	Whether GitHub Actions is configured/used in the repository.
Travis	C	Whether Travis CI is configured/used in the repository.
CircleCI	C	Whether CircleCI is configured/used in the repository.
GitLab	C	Whether GitLab CI is configured/used in the repository.

DT: (C) Categorical; (N) Numeric.*

3.2 RQ2: What are the distinct patterns of CI adopters?

3.2.1 ***Motivation.*** Building on the RQ1 findings that CI adopters tend to be larger, more active, and more frequently updated projects, this research question explores *how* CI adopters differ internally. Understanding the specific patterns and configurations within CI-adopting projects helps explain variations in development efficiency (e.g., faster releasing or quicker issue resolution) and reveals which CI practices or project characteristics are associated with more effective software delivery.

3.2.2 ***Approach.*** We build on the RQ1 dataset to perform a finer-grained analysis of CI adopters. As summarized in Table 5, we extend the feature set by (i) identifying *labeled* issues (rather than all issues) and (ii) normalizing commit, pull request, and project lifetime counts to a *monthly* scale. We also detect the presence of common CI services, including Travis CI[4], CircleCI[5], GitLab CI[6], and GitHub Actions[7].

Step 1: Data Collection. Following Santos et al. [33], we identify bug-related issues as those labeled with any of {`defect`, `error`, `bug`, `issue`, `mistake`, `incorrect`, `fault`, `flaw`, `crash`, `regression`}. Repositories without such labels are excluded to ensure reliable defect-related metrics. The resulting dataset contains 442 CI-adopting repositories with at least one bug-labeled issue.

Step 2: Data Preprocessing. For each repository r, we define its active duration in months as:

$$M_r = \max\left(\varepsilon,\ \frac{\text{last\_activity\_date}_r - \text{first\_activity\_date}_r}{30.44}\right),$$

[4]https://www.travis-ci.com/
[5]https://circleci.com/
[6]https://about.gitlab.com/
[7]https://docs.github.com/en/actions

where $30.44 \approx 365.25/12$ converts days to average months, smoothing over month length variation. A small constant $\varepsilon > 0$ (e.g., 0.1) prevents division by zero for very short histories. For each count metric $X \in \{\#\text{commits}, \ \#\text{PRs opened}, \ \#\text{PRs merged}\}$,

$$X_r^{\text{per\_month}} = \frac{X_r^{\text{total}}}{M_r}.$$

Step 3: Quadrant-Based Analysis. We exclude repositories with zero labeled bug issues, and compute bug density as:

$$\text{bug\_density} \ = \ \frac{\texttt{bug\_issues\_labeled}}{\texttt{Source\_Lines/1000}}.$$

Repositories are then split by the medians of tag lifetime (months) and bug density to form four groups: *fast_low_bugs*, *slow_low_bugs*, *fast_high_bugs*, and *slow_high_bugs*, which we visualize in a quadrant plot to highlight contrasting CI adoption characteristics. The global medians are compute as follows:

$$\text{tag-release lifetime} = 15.5 \text{ days}, \quad \text{bug density} = 1.18 \text{ issues/KLoC}.$$

Step 4: Modeling and Feature Importance. We train a multiclass *Random Forest* classifier [4] on the filtered dataset ($N = 442$) to identify the project characteristics that best distinguish quadrant membership. The model uses 10 features capturing project scale, activity, CI service usage, and defect pressure: `Source_Lines`, `no_Contributors`, `Project_lifetime (months)`, `(Avg.# PRs)/month`, `(Avg.# commits)/month`, `# CI services`, `GitHub_Actions`, `Travis`, `CircleCI`, and `GitLab`. Binary columns are encoded as integers (e.g., `GitLab`$\in \{0, 1\}$). We apply a stratified 89.8%/10.2% train/test split (397/45) and fit the model with 500 trees, *max_depth* = 15, *min_samples_split* = 5, and *min_samples_leaf* = 2. After training the Random Forest model, we apply permutation feature importance[8] to identify the most important features for each quadrant. The top features are then aggregated and ranked using SK-ESD clustering [34], which provides a consolidated view of the key discriminators for quadrant membership. We should note that our data is nearly evenly distributed across the four classes (approximately 110 projects per class and 10 features used for modeling). Therefore, as proven by Peduzzi et al [30], an events-per-variable (EPV) ratio of 10 or greater is unlikely to cause overfitting.

3.2.3 *Findings:*

We present detailed results for CI adopters, reporting the median *tag release lifetime* and labeled *bug-issue density* in each quadrant.

Observation 2.1: Fast releasing do *not* increase bug density among CI adopters. Splitting the 442 CI-adopting repositories by tag-release lifetime and bug density medians yields four nearly equal groups: Fast/Low = 108, Slow/Low = 113, Fast/High = 113, Slow/High = 108 (Figure 4). Frequent releases supported by CI do not compromise code quality, suggesting that disciplined CI pipelines enable rapid delivery without increasing defects.

The Best quadrant (Fast/Low) shows a median bug density of 0.23 issues/KLoC and a tag-release lifetime of 8.0 days, whereas the Worst quadrant (Slow/High) reaches 2.73 issues/KLoC and 32.0 days (Mann–Whitney $U = 0$, $p < 10^{-6}$ for both metrics). Projects in the Best quadrant also sustain strong development activity (median commits = 1145), while other quadrants typically trade off speed or defect density. This indicates that maintaining fast releasing

[8]https://scikit-learn.org/stable/modules/permutation_importance.html

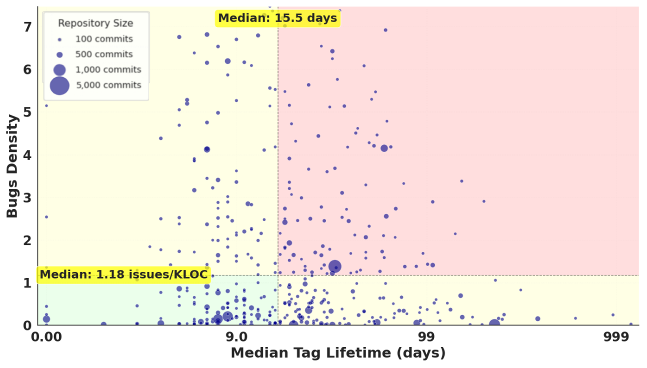

Figure 4: Quadrant plot for CI projects.

through CI does not inherently compromise software quality, as disciplined CI practices can support rapid, reliable delivery without increasing bug density.

Observation 2.2: Project lifetime and scale/activity primarily discriminate CI adoption patterns, while CI service adds modest marginal signal. As Table 6 depicts, `Project_lifetime (months)` consistently ranks as the top discriminator across all four quadrants. Measures of project size and throughput: `Source_Lines`, `no_Contributors`, and monthly activity (`(Avg # commits)/month`, `(Avg # PRs)/month`), are the Top–5 features (Table 7). Overall, CI service indicators (`GitHub_Actions`, `CircleCI`, `GitLab`) contribute little on average. In addition, the type of CI service can still provide informative discrimination. For example, in the `slow_low_bugs` quadrant, `Travis` and the number of CI services appear in the Top-5, and `GitHub_Actions` also ranks highly. This suggests that while project longevity and throughput are the main factors differentiating quadrants, specific CI services may help developers optimize maintenance-oriented or slow-moving projects.

Table 6: SK–ESD feature ranking (mean importance and dispersion).

Feature	Importance	Std	Rank
`Project_lifetime (months)`	0.340	0.058	1
`Source_Lines`	0.104	0.044	2
`(Average # of pull requests)/month`	0.070	0.039	3
`no_Contributors`	0.067	0.045	4
`GitHub_Actions`	0.000	0.000	5
`CircleCI`	0.000	0.000	6
`GitLab`	0.000	0.000	7
`# of CI services`	0.000	0.008	8
`(Average # of commits)/month`	−0.017	0.030	9
`Travis`	−0.018	0.009	10

Note: Importance is the SK–ESD average across clusters; Std is the across-cluster dispersion. Zeros indicate near-zero mean importance under this specification.

RQ2 Summary: Fast release cycles do not inherently increase defect density. Quadrant analysis reveals that CI enables fast releasing without sacrificing quality. Project lifetime and bug density are the key factors differentiating project performance, while CI service usage mainly distinguishes slower, more stable

Table 7: Top 5 features by quadrant in Random Forest.

	Metric	Imp.	Rank
FAST_HIGH_BUGS	`Project_lifetime (months)`	1.1340	1
	`Source_Lines`	0.3143	2
	`(Average # of commits) / month`	0.3081	3
	`Travis`	0.2939	4
	`# of CI services`	0.2913	5
FAST_LOW_BUGS	`Project_lifetime (months)`	1.2994	1
	`no_Contributors`	0.2961	2
	`Source_Lines`	0.2751	3
	`# of CI services`	0.2417	4
	`(Average # of commits) / month`	0.2260	5
SLOW_HIGH_BUGS	`Project_lifetime (months)`	1.1543	1
	`(Average # of commits) / month`	0.3024	2
	`Source_Lines`	0.2235	3
	`Travis`	0.2233	4
	`GitHub_Actions`	0.2021	5
SLOW_LOW_BUGS	`Project_lifetime (months)`	1.2653	1
	`Travis`	0.7582	2
	`# of CI services`	0.5684	3
	`GitHub_Actions`	0.5426	4
	`GitLab`	0.3306	5

projects. Overall, CI adoption can help maintain a disciplined release cycle and low defect load.

3.3 RQ3: What are the evolution characteristics of CI adoption?

3.3.1 ***Motivation***. Studying the impact of CI adoption is crucial for understanding how it affects development activity and user outcomes. While CI is often associated with improved efficiency, its effects can vary across repositories. By analyzing changes in commits, pull requests, and merge times, along with user-facing outcomes like reviews and ratings, we aim to identify which projects benefit most from CI adoption and whether it improves the user experience. This can help developers optimize their CI practices for better results.

3.3.2 ***Approach:*** We collect and analyze repository-level metrics for this research question. Building on the RQ2 setup, we focus on repositories active in the 90 days around CI adoption and also examine each project's full lifetime. For code-hosting outcomes, we compute before/after changes in (i) commits, (ii) pull requests opened, (iii) pull requests merged, and (iv) time to merge. We detect CI adoption from workflow-configuration history and align it with each repository's activity timeline. Data are obtained via the GitHub REST API; user-facing app outcomes (monthly review volume and average star rating) are retrieved from the Google Play Store using `google_play_scraper`[9].

Step 1: Data Collection. Focusing on repositories active in the 90-day window around CI adoption, we collect (i) commits, (ii) pull requests opened, (iii) pull requests merged, and (iv) time to merge from the GitHub REST API. For user-facing outcomes, we query the Google Play Store via `google_play_scraper` and obtain review histories, from which we compute total review counts and average star ratings over the pre- and post-adoption windows.

[9]https://pypi.org/project/google-play-scraper

Step 2: data Preprocessing. Some repositories adopted CI at project inception and thus lack a pre-adoption window; others had no early pull-request activity. Consequently, analyses are metric-specific (pairwise inclusion). From the original 177 repositories, the usable samples are: commits $n = 172$, PRs opened $n = 137$, PRs merged $n = 133$, and time to merge $n = 127$. For Google Play Store outcomes, some linked apps lack a valid pre-adoption window (the app launched after adoption or had no reviews/ratings recorded in the pre window). Thus, Play-based analyses are also metric-specific: review-volume rate ratio $n = 97$ and rating-change rate ratio $n = 95$.

Step 3: Data Analysis. We perform paired, within-repository comparisons using the full project history before vs. after CI adoption. The pre-adoption window spans from the first commit/PR to the CI adoption date; the post-adoption window spans from the adoption date to the last commit/PR. Metrics are month-normalized by dividing activity counts by the duration of each window in months (computed as days/30.44). For each repository r, we report four rate ratios:

(1) *Commits rate ratio.* Let C_r^{pre} and C_r^{post} be the commit counts in the pre- and post-adoption windows. Let M_r^{pre} and M_r^{post} be the window lengths in months. Define:

$$\text{commits\_increase\_rate}_r = \frac{C_r^{\text{post}}/M_r^{\text{post}}}{C_r^{\text{pre}}/M_r^{\text{pre}}}.$$

Values > 1 indicate higher commit throughput after adoption.

PRs-opened rate ratio. Let P_r^{pre} and P_r^{post} be the numbers of pull requests opened in each window. Then:

$$\text{prs\_increase\_rate}_r = \frac{P_r^{\text{post}}/M_r^{\text{post}}}{P_r^{\text{pre}}/M_r^{\text{pre}}}.$$

This captures the change in PR creation activity.

Merged-PRs rate ratio. Let N_r^{pre} and N_r^{post} be merged PR counts per window. We compute:

$$\text{merged\_prs\_increase\_rate}_r = \frac{N_r^{\text{post}}/M_r^{\text{post}}}{N_r^{\text{pre}}/M_r^{\text{pre}}}.$$

Larger values reflect more merges per month after adoption.

(2) *Merge-time speed-up.* For merged PRs, define merge time as `merged_at` − `created_at` (days). Let $\widetilde{T}_r^{\text{pre}}$ and $\widetilde{T}_r^{\text{post}}$ be the median merge times in the two windows. We report the inverse ratio (speed-up):

$$\text{merge\_time\_increase\_rate}_r = \frac{\widetilde{T}_r^{\text{post}}}{\widetilde{T}_r^{\text{pre}}},$$

so that values < 1 indicate faster merging after adoption.

Interpretation. Rates > 1.0 indicate an increase after CI; $= 1.0$ indicates no change; < 1.0 indicates a decrease. Ratios with undefined denominators are treated as missing and excluded, as no data before CI.

3.3.3 ***Findings:*** We present detailed before vs. after results around CI adoption.

Observation 3.1: Development activity (in terms of commits and PRs) improves after CI adoption. After adopting CI, development performance improves, with repositories seeing increases

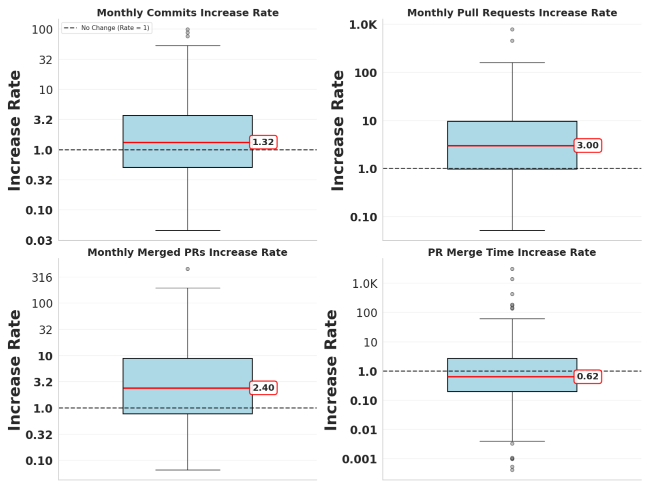

Figure 5: Before vs. after rate ratios around CI adoption.

Table 8: Before vs. After CI Adoption: Values > 1 indicate an increase after adoption; for merge time, the ratio is reported as a speed-up (before/after).

Metric	Count	Median	Mean	Min	Max	Up / Down
Commits	172	1.32	5.39	0.05	98.94	102 (59.3%) / 70 (40.7%)
PRs Opened	137	3.00	20.02	0.00	777.00	102 (74.5%) / 35 (25.5%)
PRs Merged	133	2.40	13.67	0.00	447.00	96 (72.2%) / 37 (27.8%)
Merge Time	127	1.60	46.03	0.00	3065.52	75 (58.6%) /53 (41.4%)

in commits, pull requests (PRs) opened, and PRs merged. However, the introduction of more rigorous quality gates and increased PR concurrency leads to some trade-offs in merge speed. As shown in Figure 5 and Table 8, development activity generally increases post-CI adoption: 59.3% of repositories see more commits, 74.5% see more PRs opened, and 72.2% see more PRs merged. The median merge-time speed-up is 1.60, though only 41.4% of repositories merge faster after CI adoption, indicating that additional quality controls (such as status checks and required reviews) introduce some latency. The distributions of these metrics show heavy right tails, with means far exceeding medians, suggesting that while most repositories show moderate improvements, a few repositories experience substantial post-adoption surges (e.g., PRs opened up to 777 and merge-time ratio up to 3065.52). Therefore, developers should be prepared for trade-offs in merge speed due to added quality assurance measures. A balance between speed and rigor is recommended, especially for large projects with higher PR volumes, as the degree of improvement can vary significantly across projects.

Observation 3.2: No systematic improvement in user-facing outcomes after CI adoption. CI adoption does not appear to lead to systematic improvements in user-facing outcomes, such as Google Play Store reviews and star ratings, in the short run.As shown in Figure 6 and Table 9, there is no significant positive shift in median review volume or star ratings after CI adoption. In fact, the median review rate declines slightly to 0.83, with fewer than half of apps experiencing increases in reviews (41.2%). Similarly, median star ratings remain unchanged (0.97), with only 31.6% of apps showing improvements. The distributions are right-skewed, with means exceeding medians: reviews (1.21 vs. 0.83) and ratings (0.97 vs. 0.97), indicating that a small subset of apps saw large gains, but these do not significantly affect the overall trend. This suggests that developers should be cautious in expecting immediate or substantial improvements in user-facing outcomes, such as app ratings or review volume. Other factors, including app content, user experience, and marketing efforts, likely play a larger role in shaping user feedback on platforms like the Google Play Store.

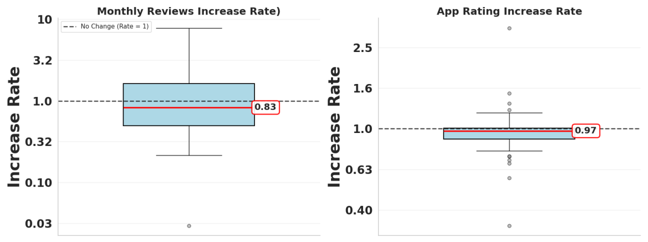

Figure 6: Google Play Store Outcomes Before vs. After CI Adoption.

Table 9: Google Play Store Outcomes Before vs. After CI Adoption: Rate Ratio Summary. Values > 1 indicate an increase after adoption.

Metric	Count	Median	Mean	Min	Max	Up / Down
Monthly Reviews	97	0.83	1.21	0.00	7.83	40 (41.2%) / 57 (58.8%)
Average Rating	95	0.97	0.97	0.33	3.13	30 (31.6%) / 64 (67.4%)

RQ3 Summary: CI adoption likely boosts development activity, with more commits and pull requests opened and merged, though median merge times may lengthen due to stricter pipelines. In contrast, user-facing metrics such as app reviews and ratings show no consistent short-term improvement, suggesting that CI primarily enhances internal development processes rather than immediate user outcomes.

3.4 RQ4: What explains the variation in before- and after-adoption performance?

3.4.1 ***Motivation.*** In RQ3, we find significant variability in the association of CI adoption with commit activity, pull requests, and merged PRs across different projects. To better understand which repositories benefit the most from CI adoption, it is important to examine the factors driving this variability. By conducting a pre/post analysis around each repository's CI adoption date, we can estimate relative changes in activity metrics and explore how these changes differ across adoption-lifetime quartiles, app categories, repository scale, and CI service configurations. This helps pinpoint key conditions associated with CI's effectiveness, offering practical insights into how developers can maximize the benefits of continuous integration.

3.4.2 ***Approach:*** Building upon the filtered dataset from RQ3, which consists of 177 active repositories, we conduct a temporal analysis to examine the CI adoption lifecycle. To quantify the duration of CI usage across repositories, we establish a time window by calculating the period between the CI adoption date and the repository's last commit activity.

Step 1: Data Preprocessing. Using the same data and preprocessing as in RQ3 (i.e., $n = 177$), we remove repositories with missing merge times, resulting in 127 repositories. We then standardize all variables using StandardScaler[10] to ensure that each feature contributes equally to distance-based analyses and to remove scale effects, which is important when comparing metrics with different units or ranges.

Step 2: Data Analysis. We compute *Adoption Lifetime* for each repository as the time between the CI creation date and the most recent activity date:

$$\text{AdoptionLifetimeMonths}_i = \frac{\text{LastActivityDate}_i - \text{CICreateDate}_i}{30.44}$$

We then apply k-means clustering [26] with a multi-dimensional feature space that captures changes in development and release activity after CI/CD adoption, including commit increase rate, pull request increase rate, merged pull requests increase rate, and release lifetime increase rate, to partition repositories into groups with similar characteristics.

3.4.3 *Findings:* We present detailed explanatory results on CI adoption

Observation 4.1: CI adoption lifetimes exhibit substantial heterogeneity. Repository lifetime shows considerable variation, with the median project age around 39.5 months (3.3 years). This heterogeneity helps explain the observed differences in post-CI adoption performance, as newer projects can experience sharp gains, while longer-term projects may see smaller incremental improvements. We observe that the median lifetime of repositories is 39.46 months (3.29 years), with the mean slightly higher at 41.15 months (3.43 years). The distribution is slightly right-skewed, with a range of approximately 131.87 months (from 1.45 months to 133.31 months) and an interquartile range (IQR) of about 36.33 months (3.03 years). This wide variation helps explain the performance differences observed in RQ3: newer projects can see rapid performance improvements from a low baseline, while projects with longer histories may only show marginal gains due to already having optimized practices. The long median repository age suggests that most projects have had sufficient time to realize CI benefits. However, developers should expect more significant performance gains from newer projects, while older projects may see more modest improvements due to prior optimizations.

Observation 4.2: Post CI-adoption development patterns vary by release timing and activity. As shown in Figure 7, clustering the before/after rate ratios reveals three primary post-adoption patterns across 61 repositories, as follows. In this, all the 4 metrics must not be NAN.

- **Cluster4** shows **increased** commits/PRs and **shorter** release lifetimes.
- **Cluster1** shows **decreased** commits/PRs with **longer** release lifetimes.

[10]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

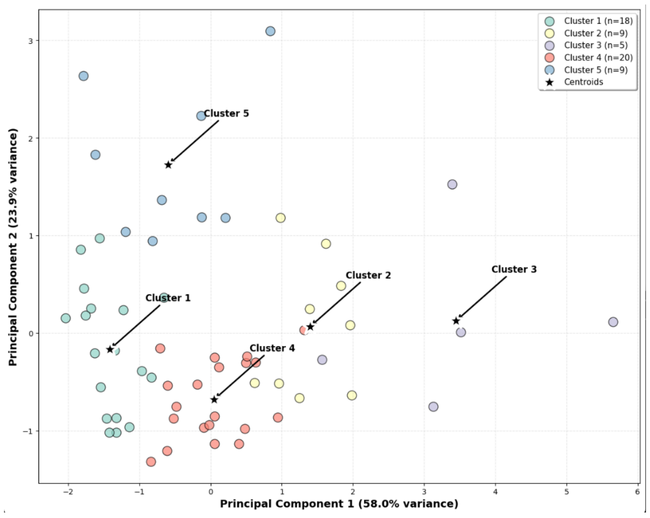

Figure 7: k-means clustering with $k = 5$ on repositories (log-transformed features). Points are colored by cluster; stars mark centroids.

- **Cluster2**, **Cluster3**, **Cluster5** show **increased** activity with **longer** releases, but with substantial variability in release duration.

Cluster interpretations:

- **Cluster1 (n=18): Activity down, releases longer:** commits ↓ (0.57), PRs opened ↓ (0.53), PRs merged ↓ (0.54), release lifetime lengthens (3.82).
- **Cluster2 (n=9): PR-heavy gains, releases longer:** commits ↑ (1.26), PRs opened ↑ (4.81), PRs merged ↑ (3.63), release lifetime lengthens (2.28).
- **Cluster3 (n=5): Very large commit gains, releases longer:** commits ↑ (15.60), PRs opened ↑ (7.61), PRs merged ↑ (7.92), release lifetime lengthens (2.30).
- **Cluster4 (n=20): Increases with shorter releases:** commits ↑ (1.83), PRs opened ↑ (1.43), PRs merged ↑ (1.42), release lifetime shortens (0.93).
- **Cluster5 (n=9): Modest gains, releases extremely longer:** commits ↑ (1.29), PRs opened ↑ (1.67), PRs merged ↑ (1.51), release lifetime *much* longer (28.13).

This clustering reveals that Increased development activity does not always lead to faster releases. While Cluster4 (n=20) demonstrates a continuous delivery pattern with more commits/PRs and shorter release lifetimes, the combined groups of Cluster2, Cluster3, and Cluster5 (n=23) show that higher activity can coincide with longer release lifetimes. Notably, Cluster5 shows an extremely large release-lifetime ratio (28.13), which inflates the means and contributes to the wide dispersion observed in RQ3. This suggests that post-adoption pipelines may introduce additional quality gates, such as testing, reviews, and compliance checks, that intentionally extend release cycles to improve stability and quality. Developers should recognize that longer release lifetimes are not necessarily a sign of inefficiency, but rather a trade-off for enhanced rigor and quality in the release process.

Table 10: Cluster characteristics (original scale): *means* of before/after rate ratios. Release lifetime is *after/before* (< 1 = shorter).

Cluster	n	Commits	PRs opened	PRs merged	Release lifetime (after/before)
1	18	0.57	0.53	0.54	3.82
2	9	1.26	4.81	3.63	2.28
3	5	15.60	7.61	7.92	2.30
4	20	1.83	1.43	1.42	0.93
5	9	1.29	1.67	1.51	28.13

RQ4 summary: Three primary post-CI adoption patterns observed in our dataset: (1) increased activity with shorter release lifetimes, (2) decreased activity with longer release lifetimes, and (3) increased activity with longer release lifetimes. Interestingly, more development activity does not always lead to faster releases; longer lifetimes often reflect the addition of quality controls, such as testing and reviews, rather than inefficiency.

4 Implications

This section discusses the practical and research implications of our findings.

For developers. CI adopters are observed to be larger and more active, with faster releasing and higher user engagement (RQ1). Among adopters, faster releasing is not associated with higher bug density, indicating that well-structured CI can maintain quality even at higher development speed (RQ2). After adoption, repositories increase development activity (commits +59%, PRs +74%), but median merge times prolong, and Google Play Store metrics remain largely unchanged (RQ3). Developers should therefore complement CI with release and feedback practices to improve user-facing outcomes. The three observed adoption trajectories (RQ4) suggest that CI effects vary depending on project practices and pipeline configurations, emphasizing the need to tailor CI setups to team and project goals.

For researchers. The diversity in adoption outcomes (RQ4) points to open questions about how project characteristics, pipeline configurations, and team practices shape CI effectiveness. Future work could investigate longitudinal CI adoption, cross-domain comparisons (e.g., Android vs. ML projects), and socio-technical factors that can influence productivity, release speed, and user engagement.

5 Threats to Validity

Construct validity. (i) *CI adoption.* We infer adoption from repository workflow/configuration histories and release tags. Projects that enable CI via web dashboards or less common providers may be missed or have misdated adoption. (ii) *Metric proxies.* Repository signals (e.g., commit frequency, issue counts, SLOC) are proxies for productivity and quality; they may not fully capture latent constructs (e.g., developer effort, code health). In particular, using GitHub Issues as a proxy for bug density has limitations: though we search for bug-related keywords (e.g., "defect," "bug," "crash"), these heuristics are not fully robust and may miss relevant cases or include false positives. To detect CI adoption, we identify repositories that contain CI configuration (YAML) files. While this method confirms that developers have configured CI, it cannot verify whether the CI pipeline is actively executed in practice. Nevertheless, this approach is the standard and most practical proxy for CI adoption in large-scale repository mining studies [33].

Internal validity. Unobserved confounders (team size, domain, or governance model) may influence both CI adoption and outcomes. Though we control for observable repository characteristics, residual confounding may remain. Our 90-day activity window reduces long-run drift but may omit slower processes. We should note that our results do not imply causality; they rather summarize patterns and associations observed in the current data.

External validity. Our dataset focuses on Android apps and excludes iOS. We expect core development dynamics (e.g., commit frequency) to be broadly comparable across mobile operating systems, but this remains an assumption; platform-specific effects may limit generalizability. The study is limited to Android apps because app details are publicly available and crawlable on the Google Play Store, unlike other platforms such as the Apple App Store. Future work should extend this analysis to additional platforms. Results may also differ for non-mobile or closed-source projects.

6 Related Work

6.1 Mobile app release practices

Prior studies examined release behaviors and their effects on app quality and user satisfaction. For example, McIlroy et al. [25] found that frequent releases are associated with higher Google Play ratings, while Domínguez-Álvarez and Gorla [8] observed that Android apps release more often than iOS apps. Other studies analyzed emergency releases [16, 17, 32] and release communication through notes [37], showing diverse developer practices and challenges in managing updates. Nayebi et al. [28] and Nayebi et al. [29] surveyed developers and modeled release deployment likelihood, linking structured release processes with higher app success. Beyond mobile apps, ecosystem-level studies have examined how release activity evolves over project lifecycles and how sustained slowdowns or prolonged inactivity often precede project abandonment. Hasan et al. [15]. showed that declining release activity is a common signal of reduced maintenance in large software ecosystems, and that release frequency alone is insufficient to characterize project sustainability. Our work complements this line of research by connecting release behavior with CI adoption, offering empirical evidence on how CI supports faster, more regular, and higher-engagement releases.

6.2 CI adoption and practices

Several studies explored CI adoption and challenges. Hilton et al. [18] found limited CI usage on GitHub due to developers' lack of experience, and Chopra and Ghaleb [5] studied how projects gradually adopt and evolve multiple CI services over time. Several studies [1, 36, 38] identified CI pain points such as configuration smells, tool inconsistencies, and high complexity and heterogeneity in CI configurations. To address these challenges, recent work has examined automation and evolution in CI usage, including automatic generation of CI configurations [14] and CI service migration [19].

Bouzenia and Pradel [3] and Jin and Servant [20] highlighted CI's resource costs and feedback trade-offs, while Ghaleb et al. [11, 12, 13] studied configuration patterns and their effects on build duration and failure. Ghaleb et al. [11, 13] further showed that attempts to improve CI build duration or reliability often involve trade-offs, as common workarounds (e.g., retries or extended timeouts) can slow builds without guaranteeing success. Our work instead quantifies CI's broader associations with development activity, release speed, and user outcomes, focusing on open-source Android projects.

6.3 CI for mobile apps

CI research in mobile development remains limited. Polese et al. [31] and Liu et al. [22] showed that many Android projects do not build reliably, indicating difficulties in maintaining stable pipelines. Wang et al. [35] found that only 40% of apps use CI, mostly for unit testing. Similar to our study, Ghaleb et al. [10] investigated four CI services, including GitHub Actions, Travis CI, CircleCI, and GitLab CI, on a curated set of 2.5K Android apps from GitHub and F-Droid. They uncovered lack of commonalities among these services and identified trade-offs when selecting certain CI services. Our work complements theirs by analyzing the effects of CI adoption on development activity, release patterns, and user-facing outcomes, offering a more detailed view of CI usage in mobile apps.

7 Conclusion

This study analyzed 2,542 open-source Android projects, linking CI adoption with development activity and Google Play Store outcomes through statistical, machine learning, and clustering analyses. CI adoption aligns with project maturity and complexity rather than simplicity. Adopters are larger, more active, and concentrated in integration-heavy domains (e.g., Finance and Productivity), suggesting CI supports scaling and reliability needs. Faster releasing does not raise defect rates: top projects pair short 8-day releases with low bug density (0.23 issues/KLoC). Project lifetime and bug density dominate predictive importance, indicating disciplined CI sustains speed and quality. Post-adoption, activity rises (commits +59%, PRs +75%) but merges slow and user ratings remain stable, showing CI boosts development throughput but not necessarily delivery pace or user satisfaction. CI improves engineering efficiency and process control, but its impact on external success depends on complementary release and feedback practices.

Future work. We plan in future research to survey developers to understand motivations, perceived challenges, and organizational barriers in CI adoption. Future work should also compare and contrast CI adoption across domains such as Android, web, artificial intelligence, and machine learning projects, and analyze how evolving CI toolchains and automation depth influence delivery outcomes and developer experience.

Artifact Availability

A replication package, including scripts, data, and raw results, used to produce the findings of this study, is available online at Figshare [40].

Acknowledgments

Funding: We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC): [RGPIN-2021-03969] and [RGPIN-2025-05897].

References

[1] Edward Abrokwah and Taher A Ghaleb. 2025. An Empirical Study of Complexity, Heterogeneity, and Compliance of GitHub Actions Workflows. *arXiv preprint arXiv:2507.18062* (2025).

[2] Atlassian. [n. d.]. Continuous integration vs. continuous delivery vs. continuous deployment. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. Accessed: 2021-09-23.

[3] Islem Bouzenia and Michael Pradel. 2024. Resource usage and optimization opportunities in workflows of GitHub Actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[4] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[5] Nitika Chopra and Taher A Ghaleb. 2025. From First Use to Final Commit: Studying the Evolution of Multi-CI Service Adoption. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 773–778.

[6] CircleCI. [n. d.]. Continuous integration. https://circleci.com/continuous-integration/#what-is-the-difference-between-continuous-integration-continuous-delivery-and-continuous-deployment. Accessed: 2021-09-23.

[7] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 235–245. doi:10.1109/ICSME55016.2022.00029

[8] Daniel Domínguez-Álvarez and Alessandra Gorla. 2019. Release practices for iOS and Android apps. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics, WAMA*. ACM, 15–18.

[9] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf* (2006).

[10] Taher Ghaleb, Osamah Abduljalil, and Safwat Hassan. 2024. CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2024).

[11] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24 (2019), 2102–2139.

[12] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, Ying Zou, and Ahmed E Hassan. 2019. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1998–2011.

[13] Taher A. Ghaleb, Safwat Hassan, and Ying Zou. 2023. Studying the Interplay Between the Durations and Breakages of Continuous Integration Builds. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2476–2497. doi:10.1109/TSE.2022.3222160

[14] Taher A Ghaleb and Dulina Rathnayake. 2025. Can LLMs Write CI? A Study on Automatic Generation of GitHub Actions Configurations. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 767–772.

[15] Kazi Amit Hasan, Jerin Yasmin, Huizi Hao, Yuan Tian, Safwat Hassan, and Steven H. H. Ding. 2025. Understanding Abandonment and Slowdown Dynamics in the Maven Ecosystem. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 354–358. doi:10.1109/MSR66628.2025.00065

[16] Safwat Hassan, Cor-Paul Bezemer, and Ahmed E. Hassan. 2020. Studying Bad Updates of Top Free-to-Download Apps in the Google Play Store. *IEEE Transactions on Software Engineering* 46, 7 (2020), 773–793.

[17] Safwat Hassan, Weiyi Shang, and Ahmed E. Hassan. 2017. An empirical study of emergency updates for top Android mobile apps. *Empirical Software Engineering* 22, 1 (2017), 505–546.

[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE*. ACM, 426–437.

[19] Md Nazmul Hossain and Taher A Ghaleb. 2025. CIgrate: Automating CI Service Migration with Large Language Models. *arXiv preprint arXiv:2507.20402* (2025).

[20] Xianhao Jin and Francisco Servant. 2021. 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, 213–225.

[21] Jasem Khelifi, Yacine Benzina, Moataz Chouchen, Ali Ouni, Mohammed Sayagh, and Salah Bouktif. 2025. GHAminer: An Open Source Tool to Extract GitHub Actions Build Metrics. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 834–838. doi:10.1109/SANER64311.2025.00087

[22] Pei Liu, Li Li, Kui Liu, Shane McIntosh, and John C. Grundy. 2024. Understanding the quality and evolution of Android app build systems. *Journal of Software Evolution and Process* 36, 5 (2024).

[23] Pei Liu, Xiaoyu Sun, Yanjie Zhao, Yonghui Liu, John Grundy, and Li Li. 2022. A First Look at CI/CD Adoptions in Open-Source Android Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 201:1–201:6.

[24] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18, 1 (1947), 50–60. doi:10.1214/aoms/1177730491

[25] Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering* 21, 3 (2016), 1346–1370.

[26] James B McQueen. 1967. Some methods of classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symposium on Math. Stat. and Prob.* 281–297.

[27] Florent Moriconi, Thomas Durieux, Jean-Rémy Falleri, Raphaël Troncy, and Aurélien Francillon. 2025. GHALogs: Large-Scale Dataset of GitHub Actions Runs. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 669–673. doi:10.1109/MSR66628.2025.00104

[28] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. Release Practices for Mobile Apps–What do Users and Developers Think?. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*. IEEE Computer Society, 552–562.

[29] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. 2017. Which Version Should Be Released to App Store?. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*. IEEE Computer Society, 324–333.

[30] P. Peduzzi, J. Concato, E. Kemper, T. R. Holford, and A. R. Feinstein. 1996. A Simulation Study of the Number of Events Per Variable in Logistic Regression Analysis. *Journal of Clinical Epidemiology* 49, 12 (1996), 1373–1379. doi:10.1016/S0895-4356(96)00236-3

[31] Aidan Polese, Safwat Hassan, and Yuan Tian. 2022. Adoption of Third-party Libraries in Mobile Apps: A Case Study on Open-source Android Applications. In *9th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft ’22)*. IEEE, 125–135.

[32] Islem Saidani, Ali Ouni, Md Ahasanuzzaman, Safwat Hassan, Mohamed W. Mkaouer, and Ahmed E. Hassan. 2022. Tracking Bad Updates in Mobile Apps: A Search-based Approach. *Empirical Software Engineering* 27, 4 (2022), 81.

[33] Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. 2022. 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* (Helsinki, Finland) *(ESEM ’22)*. Association for Computing Machinery, New York, NY, USA, 137–147. doi:10.1145/3544902.3546244

[34] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[35] Dingbang Wang, Yu Zhao, Lu Xiao, and Tingting Yu. 2023. An Empirical Study of Regression Testing for Android Apps in Continuous Integration Environment. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–11.

[36] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. 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*. 647–658.

[37] Aidan Z. H. Yang, Safwat Hassan, Ying Zou, and Ahmed E. Hassan. 2022. An empirical study on release notes patterns of popular apps in the Google Play Store. *Empirical Software Engineering* 27, 2 (2022), 55.

[38] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25 (2020), 1095–1135.

[39] Shenyu Zheng, Bram Adams, and Ahmed E. Hassan. 2024. Does using Bazel help speed up continuous integration builds? *Empirical Softw. Engg.* 29, 5 (July 2024), 42 pages. doi:10.1007/s10664-024-10497-x

[40] Xiaoxin Zhou, Taher A. Ghaleb, and Safwat Hassan. 2026. CI Adoption and Mobile App Success: An Empirical Study of Open-Source Android Projects (Replication Package). https://figshare.com/articles/figure/CI_Adoption_and_Mobile_App_Success_An_Empirical_Study_of_Open-Source_Android_Projects_Replication_Package_/30815729?file=61299769.