The impact of branching and merging strategies on KPIs in open-source software

Serban Ungureanu[1]

Responsible Professor: Sebastian Proksch[1]
Supervisor: Shujun Huang[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Serban Ungureanu
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang, Marco Zuniga

Abstract

Continuous Integration (CI) practices have become central to open-source software (OSS) development, yet the relationship between branching strategies, merge habits, and CI performance remains underexplored. Understanding their role is crucial for explaining the variation in CI outcomes and for refining development practices. We empirically examine how branching models (feature-based vs. trunk-based) and merge characteristics (size and frequency) affect key performance indicators (KPIs).

Using a dataset of 565 GitHub repositories, we analyze both short-term trends and long-term evolution of development strategies. We find that while feature branching is strongly associated with higher delivery frequency and lower defect counts, trunk-based workflows (though rare) sometimes outperform in lead time and recovery. Similarly, frequent merges correlate with faster delivery and shorter lead times, regardless of size. A longitudinal subset reveals that projects shift toward feature-based development over time, but do not consistently adopt smaller or more frequent merges.

We also highlight methodological limitations in mining GitHub. Future research should incorporate longitudinal repository tracking and developer surveys to capture workflows that are invisible to snapshot-based analysis. This study contributes to a nuanced understanding of how code management practices shape CI outcomes in collaborative OSS projects.

1 Introduction

Open-source software (OSS) underpins much of today's technology, from operating systems to cloud platforms, with an estimated value over $8.8 trillion. Without OSS, software costs would be 3.5 times higher [12]. As development has matured, Continuous Integration (CI) has become widely adopted in the OSS ecosystem [22; 24], enhancing the speed, reliability, and coordination of software delivery.

While prior research confirms the general benefits of adopting CI [22; 24], it often treats CI as a uniform practice. For instance, Baltes et al. find that CI adoption does not significantly change commit behavior [2], and Islam and Zibran observe a positive association between passing builds and code review activity [13]. However, these studies attribute outcomes to CI itself, without considering how differences in development workflows may shape CI performance, or by equating them solely with build success.

This is a notable gap. Branching and merging are fundamental to how teams coordinate changes, and directly influence delivery cadence, code review dynamics, and release reliability. Despite their central role in integration workflows, their impact on CI outcomes remains underexplored.

This study addresses that gap by examining whether branching and merging workflows provide measurable benefits in open-source CI projects. We focus on feature-based versus trunk-based development, and on the size and frequency of merges, asking whether these practices correlate with better outcomes on key CI performance indicators.

To that end, the following research questions are posed:

RQ1: *Does feature branch development show a clear improvement in CI KPIs compared to trunk-based development?*

RQ2: *Do frequent, small merges correlate with better CI KPIs compared to infrequent, large merges?*

RQ3: *How do code management strategies evolve over the lifetime of a project?*

To answer these questions, we evaluated repositories from GitHub on a set of KPIs: delivery frequency, delivery size, change failure rate, mean time to recovery, and change lead time, in line with industry standards [7]. The repositories are analyzed based on whether they follow feature or trunk-based development, and the size and frequency of merges. Based on these groupings, the KPIs are calculated for each category and compared to assess differences in performance.

Our findings show that while trunk-based development is relatively rare, it can outperform feature-based workflows in terms of change lead time and recovery speed. Conversely, feature-based development is more prevalent and associated with higher delivery frequency and lower defect counts. Regarding merging habits, frequent merges are consistently linked to faster delivery and shorter lead times, whereas merge size appears to have little effect. Notably, projects tend to adopt more structured feature branching over time, but do not significantly change their merge frequency or size. Finally, we reflect on key limitations of GitHub mining, especially in capturing rebased commits and implicit workflows, which future studies should address through longitudinal and developer-centric approaches.

2 Related Work

Continuous Integration (CI) has become a foundational practice in modern software development, prompting extensive research into its benefits, challenges, and impact. Existing work has examined CI from multiple perspectives, including its technical efficacy and social implications, but comparatively less attention has been paid to the structural development strategies that underlie CI practices. Our study builds upon this landscape by empirically linking project-level CI performance to branching strategies and merge behavior across a large sample of open-source repositories.

CI benefits As previously mentioned, many studies have already confirmed the popularity and general benefits of adopting CI [24; 22; 11]. These works serve as a foundation for understanding the value of CI, without differentiating how specific development workflows might influence these outcomes.

Detecting branching strategies The work of Gupta et al. [10] has been instrumental in detecting whether projects employ trunk-based development or feature branch development. Their methodology for identifying branching strategies served as a foundation for our methodology, which we

subsequently extended. However, they do not investigate the performance implications of these strategies, which our work aims to uncover.

Pull requests in the context of CI Prior work has also examined how pull request workflows interact with CI. Zampetti et al. [23] found that CI failures are frequently discussed in PR threads, but do not consistently block merges. Vale et al. [21] showed that pull requests reduce the incidence of merge conflicts, though communication activity alone is not a strong predictor. These findings emphasize the social and procedural dimensions of modern development practices, which our work complements through a large-scale KPI-focused analysis of branching strategies.

Code review dynamics Complementing these, Cassee et al. [6] assess how the introduction of CI affects the volume of code review communication itself. Their regression-discontinuity analysis over 685 GitHub projects shows that the number of general and line-level review comments per pull request decreases significantly after adopting Travis-CI, while the number of follow-up code changes remains stable. These findings suggest that CI offloads part of the review burden, functioning as a "silent helper" by automating tasks that would otherwise require manual comments.

Effects on builds Islam and Zibran [13] conducted a large-scale empirical study analyzing over 3.6 million builds from 1,090 open-source projects to investigate which development factors affect individual CI build outcomes. Their findings indicate that the number of changed lines, modified files, and built commits per task significantly influence build success or failure, while factors such as contribution model and branching strategy do not. Unlike our work, which focuses on project-level performance metrics, their study centers on build-level reliability, offering complementary insights into the operational behavior of CI systems.

Linking workflows to CI performance While prior studies have investigated CI outcomes from angles such as build stability, review behavior, and communication dynamics, few have systematically linked these outcomes to the underlying development strategies that shape them. Our work addresses this gap by analyzing how branching models and merge habits influence project-level CI KPIs across 565 repositories. By extending classification techniques and focusing on measurable delivery metrics, we offer an empirical bridge between structural workflows and their practical impact on continuous integration performance.

3 Methodology

Our methodology had three main phases: selection and collection, grouping based on exhibited behaviors, and measuring the KPIs.

Dataset selection Our data collection criteria were primarily informed by the work done by Beller et al. [3] on the *TravisTorrent* project, which offers reasonable baselines for selecting projects for evaluation:

> *[...] we restricted our project space using established filtering criteria to all non-fork, non-toy, somewhat popular (>10 watchers on GITHUB) projects with a history of TRAVIS CI use (>50 builds) in Ruby (898) or Java (402)*

While our criteria are not identical (for example we do not care about programming language), this served as a valuable initial reference point.

The selection process used the SEART GitHub Search Engine [8], which allowed for the discovery of repositories that meet the agreed criteria. Specifically, repositories were required to have between 50 and 10,000 commits, be created before 1 May 2024, and have at least one commit between 1 April 2025 and 1 May 2025. Additional filters excluded forks and retained only repositories with at least 10 watchers, an open-source license, open issues, and an active pull request history.

We applied these search criteria to ensure the selected repositories exhibit sustained development activity and real-world relevance. By filtering for a minimum number of commits, recent activity, and community engagement (e.g., watchers), as well as requiring the use of GitHub-native features like issues, pull requests, and licensing, we target projects that are actively maintained and reflective of typical development workflows.

This query returned approximately 13,700 repositories. Out of this set, a total of 90 repositories were picked uniformly at random, by shuffling the full list and selecting the first 90. This number was picked as a compromise between sample size and time constraints. Out of the 90 repositories, 2 proved to have issues during data collection, and only 88 were kept.

In addition to the initially selected repositories, a secondary dataset of 477 more was incorporated, for a total of 565. This dataset, collected independently by fellow TU Delft student Atanas Buntov [5], follows similar selection criteria and was merged with the primary dataset to enhance coverage. As both studies are being published simultaneously, no direct cross-referencing is possible.

Our initially collected 88 repositories will be referred to as the *native* dataset, and the combined dataset of all 565 repositories will be referred to as the *extended* dataset. We make both datasets available at Zenodo [17; 19].

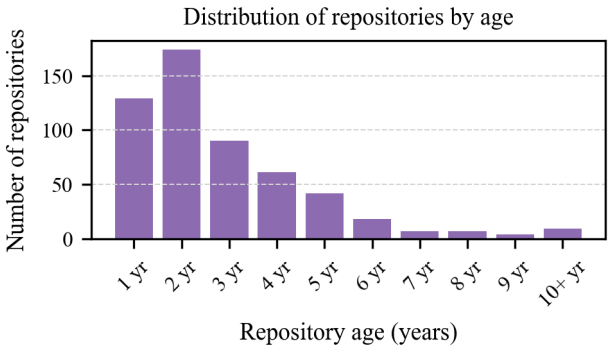

Figure 1: Age of collected repositories

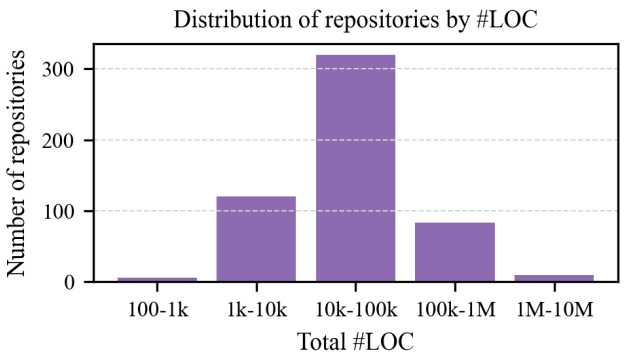

Figure 2: Repository sizes (measured in #LOC)

To ensure variety within the dataset, programming language was not included as a selection criterion. Similarly, project size and age were not filtering factors; repositories of various sizes (see Figure 2) and ages (see Figure 1) were collected. Only projects that appeared to be extremely small, resembling hobby work or personal abandonware, were excluded. Grouping of the projects was performed retrospectively to ensure a fair and representative sample of active software, excluding toy examples, hobby projects, and abandoned repositories.

Developed tools For data collection and KPI analysis, the research team developed `CI-tool-du`, a Java codebase. Additionally, a suite of Python scripts named `pyrp` was developed to support additional data collection and analysis. The specific tools and scripts used for each step of the analysis are detailed in the following sections. We make both tools available at Zenodo [16; 20].

Data collection Repository data is collected from the GitHub API. For the selected repositories, the following data was obtained: branches, commits, pull requests, issues[1], releases and deployments. User data was not obtained. The cutoff date for the data was set to 01/01/2024.

After data was collected using `CI-tool-du`, we still needed more data to perform our analysis. We enrich the original metadata with per-PR commit identifiers and line-level change statistics. For each pull request, we retrieve the list of associated commit SHAs from the GitHub API, as well as the total number of added and deleted lines. This information is integrated into the existing repository metadata files. Enrichment is only performed for merged pull requests and is skipped if the data is already present. This action is performed by the `pyrp/enrich.py` script.

Data analysis To evaluate the relationship between development strategies and CI performance, we adopt a time-series approach. For each repository, KPI values are calculated across fixed monthly intervals spanning the past 12 months. This temporal resolution enables a more stable comparison of trends across projects with varying levels of activity. After data collection, repositories are categorized according to their branching model, merge frequency, and merge size, based on the thresholds described in the following subsections. For each repository, KPIs are computed using a combination of GitHub API data and local Git clones, aggregated over fixed monthly intervals. Repositories are then independently grouped by branching model (feature-based or trunk-based) and by merge behavior (based on size and frequency). For each group, the average KPI values are computed, and performance is compared between groups: feature-based vs. trunk-based in one analysis, and between the four merge behavior categories in another.

Branching strategy To distinguish between trunk-based development and feature branch development, we compute the ratio of direct commits to total commits on the default branch over time. Direct commits are those made to the default branch without being part of a merged pull request, while feature commits are introduced via pull request merges. Using Git commit history and metadata from the GitHub API data, we classify each commit based on whether it is associated with a merged pull request (feature) or not (main).

Commits are grouped into fixed-size time intervals (e.g., monthly), and for each interval we record the number of direct main commits and merged pull request commits. The main commit ratio for each interval is defined as:

$$\text{main commit ratio} = \frac{\text{direct commits}}{\text{direct commits} + \text{merged PR commits}}$$

We compute the average ratio across all time intervals. If a repository has an average main commit ratio greater than 0.8, we classify it as following a trunk-based development model. Otherwise, it is considered feature-based [10]. This threshold reflects a strong preference for committing directly to the default branch, while allowing for moderate use of feature branches. This method enables a quantitative, time-aware classification of branching behavior across repositories.

One limitation of this approach is its inability to accurately classify projects that regularly rebase their feature branches instead of merging them. In such cases, all commits from the feature branch are rewritten as if they were made directly to the default branch, causing these projects to appear, as exclusively trunk-based.

To alleviate this, we extend upon the ratio-based method described by Gupta et al. [10]. We scan all non-default branches in each repository's Git history, and save the SHAs of all the commits found in those branches. Any commits found this way are treated as evidence of feature branching, regardless of whether it was later rebased or merged.

This is done by the `non_default_commits.py` and `branching.py` scripts from `pyrp`.

This is not a method for detecting rebasing directly, but it does serve as a correction to classification based solely on the default branch's commit history, by capturing evidence of feature branch activity that may otherwise be obscured. This does face limitations (such as branches being deleted after merging or rebasing) which will be expanded upon in the Discussion section.

Merge size and frequency For detecting whether a project's merges are small and frequent, or large and infrequent, the pull request data obtained from the GitHub API is

[1]Issue comments were not included as they were not relevant to our analysis

used. A list of commit SHA hashes is obtained from the pull request's details. This is then correlated against the `git` tree to find the commits that are part of this pull request.

The frequency of merges is simply calculated through the timestamps present in the pull request data obtained from the API. We define 'frequent' merges as occurring at least once per day, following Martin Fowler's recommendation that integrations should happen within "hours rather than days" to support effective CI practices [9].

To assess repository scale and support merge size analysis, we used a tool called *Sloc Cloc and Code* [4] (`scc`). `scc` is a language-aware static analysis tool that reports line counts and approximate cyclomatic complexity[2].

Using this tool, we compute three line-based metrics: `code_loc` (executable lines), `doc_loc` (documentation/comments), and `total_loc` (non-blank lines). While complexity is not analyzed directly, it helps distinguish code from non-code: formats like C or Python yield non-zero values, whereas Markdown or JSON yield zero, allowing us to separate `code_loc` from `doc_loc`.

To classify merge sizes, we adopt a relative threshold approach. Rather than using a fixed number of lines, we define small merges as those involving at most 1% of a repository's total executable code, capped at an upper bound of 400 lines. This limit is grounded in both industry guidelines and empirical research. Atlassian recommends limiting review scope to no more than 200-400 lines per session to maintain reviewer focus and effectiveness [1]. Similarly, empirical studies of review practices show that the number of defects found drops significantly as patch size increases, particularly beyond 200 lines [15]. Thus, we can safely consider anything above 400 lines to be a 'large' PR, regardless of the total size of the codebase.

Using `pyrp`, the line of code metrics are obtained from the `variety.py` script and the merge size and frequency analysis is done by the `merge_size_and_freq.py` script.

Longitudinal analysis of branching and merging trends To extensively analyze the evolution of code management strategies over a repository's lifetime, we selected a random sample of 50 projects from the *extended* dataset. The selection was constrained to repositories with an age between four and five years as of 01/05/2025, ensuring project maturity. For each selected project, we retrieved the full available commit and pull request history via the GitHub API. Due to incomplete data in some repositories (e.g., missing commit timestamps or broken pagination), 3 projects were excluded, resulting in a final sample of 47 repositories for longitudinal analysis. We refer to this as the *longitudinal* dataset and make it available at Zenodo [18].

KPI Definition and Extraction After the projects are grouped together based on the thresholds described (using the `pryp` script `group.py`), we calculate KPIs for each group. For the KPI analysis, we use the same time intervals of one month, and go back 12 months starting from 01/05/2025. The KPIs are broadly categorized into delivery efficiency, issue management, and build quality.

[2] `scc` uses a heuristic definition for complexity which is detailed in its README (github.com)

Delivery Frequency is defined as the average number of releases in a fixed time interval. Releases, along with their timestamps, are obtained from the GitHub API. **Delivery Size** is defined as the average number of changed lines of code per release in a fixed time interval. This data is also obtained from the API. **Change Lead Time (CLT)** is the average time from a commit until its inclusion in the next release. This is calculated as the time from the timestamp of the commit until the timestamp of the first release that occurs after that commit.

For the next two KPIs, in order to identify a 'bug issue', we evaluate issues and use a heuristic. Specifically, we identify an issue as a defect if any of its labels or label descriptions match known keywords (e.g., "bug", "error", "defect") while excluding those that simultaneously indicate resolution (e.g., "fixed", "resolved"). Due to missing bug-label metadata in the *extended* dataset, they are calculated using only data from the *native* dataset[3].

Mean Time to Recovery (MTTR) is the time from the introduction of a failure (e.g., issue or failed CI) to its resolution. We measure this as the time from when a bug issue is created until it is closed. **Defect Count** is the number of bug issues open at any time in a set time interval. For Change Lead Time and Mean Time to Recovery, we measure time in hours.

4 Findings

We present our findings in three parts, corresponding to the research questions introduced earlier. Each subsection summarizes the methods used, presents the results, and ends with a brief interpretation.

RQ1: *Does feature branch development show a clear improvement in CI KPIs compared to trunk-based development?*

This subsection explores how branching models relate to CI performance. We begin by examining how consistently projects apply a given branching strategy, then compare the KPI outcomes for projects classified as either feature-based or trunk-based.

Branching consistency and influencing factors We begin the analysis by examining the consistency over time of branching strategies within projects. Using the methodology described, we calculated the ratio of commits directly to the default branch in time intervals of one month. To determine consistency, we computed the variance of this ratio over time for each repository.

Correlation analysis showed that the only moderately significant factor observed was with the average number of commits per month ($r = -0.197$). Other factors taken into account showed minimal to no correlation: total number of lines ($r = -0.107$), ratio of executable lines ($r = -0.075$), and project age ($r = 0.010$) appear to have little to no effect on model consistency.

Thus, one observation is that **projects with more frequent commits tend to exhibit greater consistency in their**

[3] See Section 6 for details

Table 1: Average KPI values per group. Time is measured in hours, Delivery Size in number of lines of code (LOC).

KPI	Feature	Trunk
Delivery Frequency	2.152	2.042
Delivery Size (LOC)	7824.42	12795.63
Change Lead Time (h)	157.229	169.581
Defect Count	20.268	26.152
Mean Time to Recovery (h)	336.938	284.167

branching model, and that other factors do not directly affect this.

KPI comparison by branching model Each project is first classified as either feature-based or trunk-based by averaging its default branch commit ratio over time. Based on this classification, the dataset is split into two groups: 522 feature-based and 18 trunk-based projects. We then compute average KPI values for each group and compare them to evaluate differences in CI performance between the two development models.

Contrary to expectations, trunk-based development performs better on several key indicators. It is associated with a lower **Mean Time to Recovery** (284.17h vs. 336.94h) and a shorter **Change Lead Time** (157.23h vs. 169.58h), both of which indicate faster delivery and resolution of changes. Additionally, **Delivery Size** is higher under trunk-based workflows (12795 vs. 7824 LOC per interval), suggesting more substantial changes per release.

However, feature-based development shows advantages in other areas. It is associated with a higher **Delivery Frequency** (2.152 vs. 2.042 releases per interval), reflecting more frequent releases, and a lower **Defect Count** (20.27 vs. 26.15 bug issues per interval), indicating fewer identified failures between releases.

Limitations of trunk-based analysis These results challenge the common assumption that feature-based workflows are categorically superior. While prior work has advocated for feature branching due to its support for code review and parallel development [23], empirical evidence comparing its effectiveness to trunk-based development remains limited. Our findings suggest that trunk-based development, though less common, may deliver comparable or even superior performance under certain conditions.

However, we caution against overinterpreting these results. The number of trunk-based projects is small relative to the size of the entire dataset, and some of these repositories exhibit characteristics indicative of mirrors, archives, or downstream copies. These traits may confound the KPI results, as data points we are trying to track may be present somewhere other than the repositories we are attempting to track.

This aligns with prior observations: the work of Kalliamvakou et al. states that there are large amounts of mirrors on GitHub, and that large amounts of projects use external (i.e., non-GitHub) tools [14].

Summary The primary contribution of RQ1 is not a conclusive performance comparison between workflows, but rather a reframing of assumptions. Trunk-based workflows are both rare and underexamined, yet appear to perform competitively, at least within the limit of our dataset and classification. Future work should investigate whether these outcomes reflect advantages intrinsic to a trunk-based workflow, or are results of other (possibly external) factors.

RQ2: *Do frequent, small merges correlate with better CI KPIs compared to infrequent, large merges?*

This subsection investigates whether merge characteristics influence CI outcomes. Projects are grouped by merge size and frequency, and their performance is compared across a set of CI-relevant KPIs.

Merge group classification Each project is first categorized as having either small or large merges, and as merging either frequently or infrequently. Applying the thresholds defined in our methodology, we obtain the split shown in Table 2, with a total of four groups: small-frequent (SF), small-infrequent (SI), large-frequent (LF), and large-infrequent (LI). We then compute average KPI values for each of the four groups, shown in Table 3.

Table 2: Merge size and frequency splits for projects

Type	Count
Small-Frequent	21
Small-Infrequent	134
Large-Frequent	35
Large-Infrequent	341

Table 3: Average KPI values per group (S = small, L = large, F = frequent, I = infrequent). Abbreviations: DF = Delivery Frequency, DS = Delivery Size, DC = Defect Count, CLT = Change Lead Time, MTTR = Mean Time To Recovery.

KPI	SF	SI	LF	LI
DF	3.509	2.385	4.736	1.709
DS	10526	3523	30950	6491
DC	20.052	17.312	41.729	12.667
CLT	137.909	153.282	123.977	164.763
MTTR	409.573	347.660	326.010	312.267

KPI comparison across merge groups Projects with **frequent merges** (both SF and LF) generally show improved CI performance. They exhibit notably higher **Delivery Frequency** (3.509 for SF and 3.476 for LF) compared to their infrequent counterparts (2.385 for SI and 1.709 for LI), suggesting a more continuous and stable integration process.

In terms of **Change Lead Time**, frequent merging is again associated with better outcomes, particularly in the large merge group: LF projects show the shortest CLT at 123.98 hours, compared to 164.76 in LI projects. SF projects also outperform SI projects (137.91h vs. 153.28h). These results suggest that regardless of size, higher merge frequency is correlated with quicker integration of changes.

Interpreting defect count However, when it comes to **Defect Count**, the trend is less consistent. While LF projects show significantly higher defect counts (41.73), SF and SI projects are similar (20.05 vs. 17.31), and LI projects show the lowest average (16.29). This suggests that more frequent merges may lead to more bug reports, but this does not necessarily indicate lower quality in frequently merging projects. Higher defect counts could indicate a faster development cycle, where issued are identified and surfaced more rapidly. Conversely, the lower defect counts observed in infrequently merging projects may suggest more rigorous review processes, potentially filtering out issues before integration.

These results could also be influenced by the fact we are only evaluating *reported* bugs. It is also possible that projects that merge infrequently fail to detect issues more often, which could link to a potential weakness in their review process. Thus, a lower defect count may imply reduced observability, rather than improved performance. Future work could clarify this with deeper analysis of review processes present in such projects.

Patterns in recovery time No clear relation can be observed between repositories' merge sizes and frequency, and **Mean Time to Recovery (MTTR)**. The lowest MTTR is observed in LI projects (312.27h), while SF projects have the highest (409.57h), suggesting that merge characteristics alone do not have a consistent effect on recovery time.

Summary These results lend partial support to the notion that small, frequent merges are beneficial, but they also indicate that **merge frequency may matter more than merge size**. Frequent merges correlate with faster delivery and lower lead times, aligning with best practices advocated in industry recommendations [9; 1], yet largely overlooked in empirical research to date. However, the defect and recovery metrics complicate this narrative, as they suggest that higher frequency does not always reduce failures, and that recovery may depend on other structural factors.

These findings show the importance of frequent integration as a central factor in CI performance and suggest that code management strategies aimed at reducing batch size and increasing merge frequency may yield better delivery outcomes.

RQ3: *How do code management strategies evolve over the lifetime of a project?*

In this subsection, we analyze how branching and merging behaviors change over time. The *longitudinal* dataset of 47 mature projects is used to assess long-term trends in development workflows.

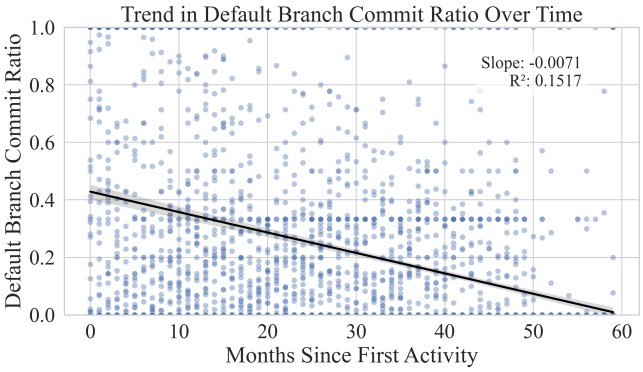

Figure 3: Trend in the ratio of commits made directly to the default branch over time.

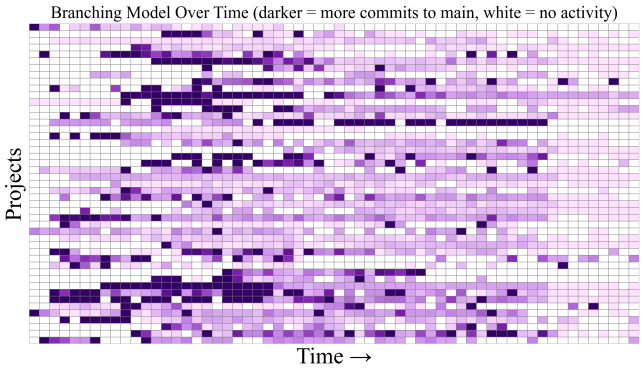

Figure 4: Branching model evolution over time per project. Darker cells indicate higher ratios of main commits; white indicates no activity.

Branching behavior trends Figure 3 shows a clear downward trend in the ratio of direct commits to the default branch over a project's lifetime. This suggests that as projects mature, direct trunk-based development becomes less common, likely replaced by more feature-based workflows. The heatmap in Figure 4 supports this observation, with many projects transitioning from high main activity (darker shades) to lower main ratios over time.

Comparing results across the *extended* dataset used for RQ1 and RQ2 (which contains data only going back one year), and the *longitudinal* dataset used here, reveals that the mild negative correlation between development activity and workflow variance weakens slightly over time ($r = -0.197$ *extended* vs. $r = -0.149$ *longitudinal*). This suggests that while active projects are somewhat more likely to maintain consistent workflows in the short term, branching models may shift over longer periods regardless of activity level. Other evaluated factors, including total lines of code ($r = -0.107$ vs. 0.073), project age ($r = 0.010$ vs. 0.064), and executable code ratio ($r = -0.075$ vs. 0.039), show negligible correlation.

Merge size and frequency trends We analyzed the evolution of merging habits over time by examining two indicators: average merge size (in lines changed) and average time be-

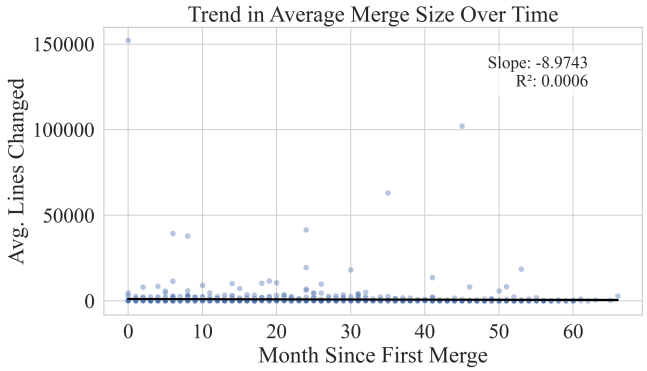

Figure 5: Merge size trend over projects' lifetimes

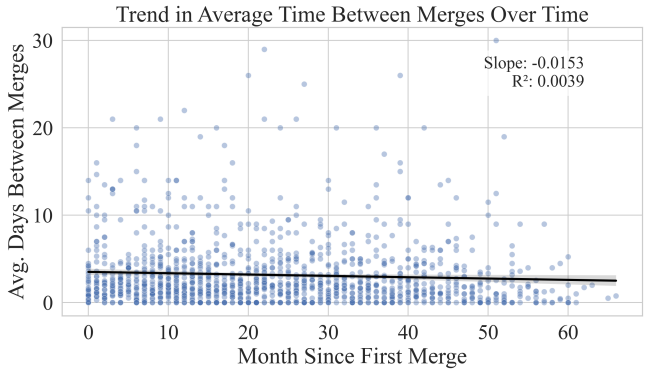

Figure 6: Merge frequency trend over projects' lifetimes

tween merges (in days), plotted against the number of months since a repository's first recorded merge.

In both cases, we observe only weak downward trends. The average merge size exhibits a near-flat slope of -8.97 lines per month, with an R^2 value of just 0.0006, indicating no trend (Figure 5). Similarly, the average number of days between merges decreases by approximately 0.015 per month ($R^2 = 0.0039$), which is not significant enough to highlight a trend (Figure 6).

These findings suggest that, at scale, repositories do not significantly shift toward smaller or more frequent merges over time. Any such behavioral shifts are either minimal or highly context-dependent, rather than being a consistent trend across projects.

Summary Taken together, these results suggest that branching strategies tend to evolve toward more structured, feature-based workflows as projects mature. The observed trend away from default branch commits reflects this. Notably, we find little evidence that codebase size or project age meaningfully influences this evolution, suggesting that projects are consistent in their merge size and frequency.

5 Discussion

This section interprets the results of our empirical analysis, reflecting on the research questions, practical implications, and directions for future work.

Reassessing branching model assumptions The results of RQ1 indicate that trunk-based workflows, though significantly less common, exhibit comparable or even superior performance on several CI metrics. Trunk-based projects report lower Change Lead Time and faster recovery (MTTR), suggesting quicker turnaround for changes and failures. However, they also show larger delivery sizes and higher defect counts. In contrast, feature-based workflows show higher delivery frequency and fewer reported defects, reflecting the benefits of modular, isolated changes.

These findings challenge prevailing assumptions that feature-based workflows are inherently more effective. Our results suggest that trunk-based development, under the right conditions, can achieve performance levels similar to or better than those of feature-driven processes. However, the validity of this comparison is undermined by structural limitations. Trunk-based projects are underrepresented in the dataset (only 18 out of 565), and many exhibit traits indicative of archival mirrors or downstream forks. This aligns with prior concerns about GitHub data quality [14], and suggests that the performance advantages observed may stem from external workflows not visible in GitHub's metadata.

As such, the findings for RQ1 do not present a settled comparison of workflows, but rather a reframing: trunk-based strategies are rare, potentially misclassified, and underexplored. Their perceived efficiency may stem from low-collaboration, low-friction update patterns. Future research should more carefully identify true trunk-based projects and investigate the environments in which this strategy performs well.

Merge habits The analysis in RQ2 highlights the role of merge frequency in shaping CI performance. Frequent merges, regardless of size, are consistently associated with faster delivery and lower lead times. This observation affirms long-standing industry advice advocating for continuous, small-batch integration [9; 1], but which was not quantified in empirical studies.

Merge size, on the other hand, has a more ambiguous impact. While large-frequent projects exhibited the highest throughput, they also reported the highest defect counts. The interpretation of this metric is nuanced: higher defect reports may indicate lower quality, but may also reflect improved observability or shorter feedback loops. Infrequently merging projects show lower defect counts, which may stem from more rigorous review or, conversely, from latent defects going unreported.

Mean Time to Recovery (MTTR) showed no consistent trend across merge groups. This suggests that merge characteristics alone are insufficient to predict recovery efficiency and that other factors may be more determinative.

Overall, the results support the use of frequent integration as a foundation for CI success, while complicating the assumption that small merges always yield better outcomes.

Strategic shifts over time In addressing RQ3, we find that most projects trend away from trunk-based development over time. The ratio of direct commits to the default branch decreases consistently across project lifespans, and heatmap analysis (Figures 3, 4) confirms a shift toward feature-based

workflows as the norm. This aligns with practical expectations: as projects grow and contributors increase, coordination overhead rises, necessitating more structured workflows such as branching and code review.

However, the evolution of merge size and frequency does not show similar consistency. Both average merge size and the interval between merges show only weak downward trends, which are not statistically significant. This implies that while branching model transitions are common, merge practices may be more resistant to change, driven more by team culture or domain-specific needs than by project age.

Interestingly, structural properties such as codebase size, age, and executable code ratio do not meaningfully predict branching model variance. This pattern holds in both the short-term and longitudinal datasets. Only commit activity exhibits a mild negative correlation with workflow variability, suggesting that more active projects are somewhat more consistent in their approach, but this trend also weakens over longer timescales.

Implications for developers These findings suggest that development teams should match their workflow strategy to the scale and needs of their project. Feature-based workflows offer modularity, parallelism, and review infrastructure, but may introduce delays unless tightly managed. Trunk-based development, while rare, may remain viable in smaller teams or low-collaboration environments, where the benefits of process overhead do not outweigh the costs.

Merge frequency emerges as a more consistent predictor of CI success than merge size, reaffirming continuous integration as a best practice. As such, teams should prioritize frequent, testable merges where possible. Practices such as stacked pull requests may offer a middle ground between feature isolation and rapid integration, though more research is needed to evaluate their effects at scale. We elaborate on this point in Paragraph 5 (**Richer developer-centric data**).

Longitudinal repository tracking A key direction for future research lies in real-time, longitudinal tracking of repositories. Unlike retrospective mining, which relies on static snapshots of the current state, continuous tracking would enable more precise activity tracking. This approach directly addresses limitations in our current study, such as the inability to observe rebased commits if branches are deleted.

Tracking repositories over time would also make it possible to study the lifespan of branches, something also not available through our methodology. This would enable a shift from binary classifications (main vs. feature) toward a more nuanced understanding of development patterns, such as distinguishing between short-lived and long-lived feature branches, or between batch-style integrations and continuous merges. These distinctions could be analyzed in a similar fashion, while providing a much richer picture than static classification allows.

Additionally, GitHub's API offers only limited, present-state information: contributor lists are not temporally scoped, team membership changes are not exposed, and collaboration patterns are difficult to observe. With a longitudinal dataset, it would be possible to track when contributors join or leave, how they interact with changing workflows, and whether specific practices correlate with retention or disengagement.

Beyond structural analysis, longitudinal data would lay the groundwork for connecting technical behaviors to developer experiences, which points to a second promising direction.

Richer developer-centric data To alleviate issues with data unreliability, these insights could be complemented with surveys or interviews. In addition to capturing attitudes toward workflows and tooling, surveys could fill in missing context about the development environment itself. Many projects use infrastructure that is not visible through GitHub, including mailing lists, self-hosted archives, mirrors, and external code review systems [14]. As a result, our current data suffers from blind spots, especially in projects that do not rely heavily on GitHub pull requests or issue tracking. Collecting this information directly from developers would help mitigate sparsity and provide a more complete view of how real-world teams coordinate, communicate, and maintain quality in their CI practices.

Future work could also investigate whether industry domain (e.g., web development, systems programming, or machine learning) and programming language influence CI practices and development outcomes. Different ecosystems often rely on distinct toolchains, testing strategies, and collaboration models. Similarly, language-specific conventions and tooling (such as Rust's `cargo`, Python's `tox`, or Java's Maven/Gradle-based pipelines) can shape how developers structure their workflows and coordinate across teams. A more granular analysis along these dimensions may uncover patterns obscured by aggregated metrics, and could also correlate to metrics like developer satisfaction, or the aforementioned attraction of outside collaborators.

This could also provide insights into alternative workflows, such as stacked diffs[4]. Stacked diffs are rare, particularly in the GitHub ecosystem, where they are not directly supported, leading to the creation of external tools that extend Git. As such practices gain adoption, future work should explore their impact on KPIs and developer experience.

The measurability gap in trunk-based development Contrary to the patterns suggested by our dataset, trunk-based development is still extremely common by a wide margin. As noted by Kalliamvakou et al. [14]:

> *Of the 2.6 million GitHub projects that represent actual collaborative projects (at least 2 committers), only 268,853 (10 %) used the pull request model at least once [...] The median number of pull requests per project is 2.*

However, nearly all of the repositories in our dataset use feature-based development. This is not merely a filtering artifact, but a consequence of our selection criteria: repositories were intentionally chosen to exclude small, inactive, or personal projects, in order to ensure measurable development activity and valid KPI computation. As a result, the dataset disproportionately reflects collaborative, CI-enabled projects, where trunk-based workflows are far less common. Including smaller or inactive repositories would have increased the number of projects with trunk-based development, but at the

[4] https://newsletter.pragmaticengineer.com/p/stacked-diffs

cost of rendering KPI analysis ineffective. Metrics such as Change Lead Time or MTTR are not meaningful for personal archives or abandoned projects.

This highlights a broader gap in the literature: **while trunk-based development is prevalent in the long tail of GitHub repositories, it remains largely unmeasurable in the ways that CI research currently operates**. Future work could address this gap through survey-based or qualitative methods, gathering self-reported insights on workflows that lie outside the bounds of what can be evaluated with tools. However, such data would not yield the same performance metrics and would require a shift in evaluation criteria to reflect developer perception and intent, rather than delivery outcomes.

6 Threats to Validity

This study is subject to both internal and external limitations that may affect the reproducibility and precision of its findings.

Internal threats Several KPIs rely on the presence of GitHub Releases. Since not all projects use this mechanism consistently, we excluded repositories without formal releases from those parts of the analysis. This may have biased results toward projects with more structured or mature workflows. Additionally, the method used to detect rebasing relies on developers not deleting old feature branches. Thus, it cannot fully resolve classification ambiguities, especially in repositories that frequently delete branches. Finally, for the computation of Defect Count and Mean Time to Recovery, the *extended* dataset (n = 565) does not contain all the required data: namely, the data used for identifying an issue as a bug (as explained in the Methodology) is missing. Due to time constraints in obtaining the data, we resorted to using the *native* dataset (n = 88) for these KPIs.

External threats Some limitations arise from factors beyond our control. Despite efforts to exclude non-primary repositories, the dataset may still include mirrors, forks, or downstream copies. This is especially relevant for trunk-based projects, where upstream activity may distort local KPI measurements. Additionally, the GitHub API provides only partial visibility into project workflows. External tools, mailing lists, or alternative CI systems may play a significant role in development but are not captured in our analysis. This restricts our ability to fully observe or evaluate certain practices.

Summary Despite these limitations, the study offers an empirical foundation for evaluating development strategies in CI-capable projects and identifies structural gaps that future work could address through longitudinal tracking and developer-centric data collection.

7 Responsible Research

This project involved the analysis of publicly available data from open-source software repositories. The following considerations were made to ensure responsible and ethical research conduct.

Privacy and anonymity All data was collected from GitHub's public API, and no attempt was made to identify or profile individual developers. Analysis was performed on aggregate data, with no focus on personal activity or behavior.

Data use Repository data was used solely for academic purposes. While individual project licenses were not explicitly reviewed, all repositories analyzed were public and not used in any commercial context.

8 Conclusion

This study examined how code management strategies influence Continuous Integration (CI) performance across open-source projects on GitHub. By analyzing 565 repositories and a longitudinal subset of 47, we evaluated how branching models, merge practices, and project evolution relate to key CI metrics such as **Delivery Frequency**, **Change Lead Time**, **Defect Count**, **Mean Time to Recovery**, and **Delivery Size**.

Our findings challenge the assumption that feature-based workflows are universally superior. While they are more common and offer advantages in delivery frequency and defect rates, trunk-based workflows (though rare) can perform comparably in lead time and recovery under certain conditions. Frequent merging correlates consistently with faster delivery, supporting established CI recommendations, while merge size showed a weaker and less stable relationship with performance.

Longitudinal analysis revealed that projects tend to adopt more structured, feature-heavy workflows as they mature, though merge behavior evolves less predictably. Structural properties such as codebase size or project age showed little influence on workflow consistency, while development activity had a mild stabilizing effect.

These insights contribute to a more nuanced understanding of CI success factors and highlight the limitations of snapshot-based mining. Future research should prioritize longitudinal tracking and developer-centered data to fill visibility gaps in modern software workflows.

References

[1] Atlassian. Code review best practices. https://www.atlassian.com/blog/add-ons/code-review-best-practices, 2022. Accessed: 2025-06-09.

[2] Sebastian Baltes, Jascha Knack, Daniel Anastasiou, Ralf Tymann, and Stephan Diehl. (no) influence of continuous integration on the commit activity in github projects. In *SWAN 2018: Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, pages 1–7, 11 2018.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 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)*, pages 447–450, 2017.

[4] Ben Boyter. scc - sloc, cloc and code. https://github.com/boyter/scc, 2024. Accessed: 2025-06-09.

[5] Atanas Buntov. Dataset used in the thesis "evaluating the impact of collaboration modes on software delivery efficiency in open-source projects". https://doi.org/10.5281/zenodo.15681547, June 2025.

[6] Nathan Cassee, Bogdan Vasilescu, and Alexander 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)*, pages 423–434, 2020.

[7] Google Cloud. Dora metrics: The four keys. https://dora.dev/guides/dora-metrics-four-keys/. Accessed: 2025-05-02.

[8] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[9] Martin Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, 2006. Accessed: 2025-06-03.

[10] Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane 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)*, pages 491–494, 2017.

[11] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny 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*, ASE '16, pages 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. The value of open source software. *SSRN Electronic Journal*, jan. 2024.

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

[14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*, 21:2035–2071, 2016. Publisher: Springer.

[15] Chris Kemerer and Mark Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35:534–550, 07 2009.

[16] Kiril Panayotov, Daniel Rachev, atns bntv, and userban3000. Zakrok09/ci-tool-du: v1.0.0. https://doi.org/10.5281/zenodo.15711350, June 2025.

[17] Serban Ungureanu and Atanas Buntov. Extended dataset used in the thesis "the impact of branching and merging strategies on kpis in open-source software". https://doi.org/10.5281/zenodo.15707347, June 2025.

[18] Serban Alexandru Ungureanu. Longitudinal dataset used in the thesis "the impact of branching and merging strategies on kpis in open-source software". https://doi.org/10.5281/zenodo.15710413, June 2025.

[19] Serban Alexandru Ungureanu. Native dataset used in the thesis "the impact of branching and merging strategies on kpis in open-source software". https://doi.org/10.5281/zenodo.15707333, June 2025.

[20] userban3000. userban3000/pyrp: Initial release. https://doi.org/10.5281/zenodo.15707311, June 2025.

[21] Gustavo Vale, Angelika Schmid, Alcemir Rodrigues Santos, Eduardo Santana de Almeida, and Sven Apel. On the relation between github communication activity and merge conflicts. *Empirical Software Engineering*, 25(1):402–433, Jan 2020.

[22] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816, 8 2015.

[23] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344, 05 2017.

[24] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 60–71. IEEE Press, 2017.