Analyzing the Impact of Documentation on Performance Metrics in Different Continuous Integration Open-Source Projects

Is documentation important?

Daniel Rachev[1]

Supervisor(s): Sebastian Proksch[1]**, 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: Daniel Rachev
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang, Marco Zamalloa

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Continuous Integration (CI) has become a standard practice for speeding up software development. However, the effect of comparatively slower artifacts, like documentation, on its performance is still unclear. Although documentation is often regarded as important, there is little data that connects documentation practices to key DevOps metrics. This study examines this relationship by looking at 670 open-source projects that use CI. We measured how documentation completeness, update frequency, and release notes affect delivery frequency, defect counts, and mean time to recovery. Our results show a "tipping point" where high documentation completeness greatly increases delivery frequency. We also found a "sweet spot" for update ratios between 20% and 55%, which relates to the lowest defect counts. On the other hand, we found no proof that long release notes improve recovery time. We conclude that the effectiveness of documentation depends more on quality and rhythm than on volume. This provides developers with practical, data-driven strategies to improve project performance.

1 Introduction

In the software industry, development speed has been increasing more and more over the years. There has been a growing need for Continuous Delivery (CD), paired with a rising trend in the adoption of Agile development. [1] Software is required to be incrementally improved and rolled out in order to meet the constantly evolving needs of customers and business alike.

Around the 2000s, there was a great dysfunction between two camps in the software industry. Software engineers and IT operations teams were not aligned. Despite the fact that the work of both was critical and more importantly, intertwined, these teams were typically in siloes. They could not communicate effectively, had different performance indicators, and sometimes even separate department leadership. This lead to rise of the DevOps movement [2], where the development and IT operations teams had to come together, which resulted in Continuous Integration (CI) gaining popularity. Some of its benefits include reduced manual tasks, improved deployment efficiency, and accelerated delivery cycles.

Software documentation is an often-neglected aspect of software engineering; however, it is intrinsically tied to CI and the development process of any team. As Andrew Forward defines it in his book, it "is an artifact whose purpose is to communicate information about the software system to which it belongs" [3]. Documentation plays a significant role throughout the software development life-cycle, from initial planning and decision-making to system maintenance. Moreover, it has been observed that effective documentation improves the overall quality of a software product and enhances its success by contributing to usability, marketability, and ease of support. [4]

A survey of software engineering experts by Plösch et al. [5] confirmed the high importance of quality documentation. The study revealed that its primary positive impact is on the maintainability of a software product, particularly its analysability. Furthermore, the surveyed experts identified accuracy, clarity, consistency, readability, structuredness, and understandability as the most crucial quality attributes for effective documentation. This research establishes the general value of documentation from a practitioner's standpoint, especially for maintenance.

Scientific literature on software documentation has established its impact on software products as a whole, as well as how practitioners perceive it. It is immensely helpful for maintenance, as it is an artifact that remains even when time passes and developers change, and it is the second-best form of communication - crucial when no one else is around. As much as documentation and CI are interconnected and integral to the Agile process, there is a lack of understanding of how documentation affects DevOps performance. What should developers document? How often should the information be changed? What about delivery documentation? In our research, we aim to answer these questions by looking into industry-based performance indicators like delivery frequency, mean time to recovery (MTTR), and defect count [6] [7]. The main goal is to observe the presence or lack thereof of patterns and relations between the performance indicators and documentation practices. The definitions of the terms used in the research questions can be found in Section 3.3.

Main Research Topic: How do documentation practices impact key performance indicators (KPIs)?

RQ1: Is documentation completeness correlated with delivery frequency?

RQ2: Does documentation update frequency correlate with defect count?

RQ3: Are documentation changes in release cycles correlated with mean time to recovery for reported issues?

This research makes the following contributions:

- We identify and measure important patterns that connect documentation to performance. In particular, we show a "tipping point" where high documentation completeness greatly increases delivery frequency. We also find a "sweet spot" for update ratios, between 20-55% of commits, which helps reduce defect counts.
- We offer actionable insights for developers. Our findings showcase that teams can use the documentation update ratio as a project health metric.
- Additionally, by demonstrating that long release notes do not speed up issue recovery, we emphasize the importance of keeping high-quality technical documentation over other, less effective practices.

This paper is organized as follows. Section 2 provides the background and summarizes the existing literature that supports this work. Section 3 describes the methods for data selection, extraction, and analysis. The findings appear in Section 4. Lastly, Section 5 consolidates the discussion of the results, their implications, threats to validity, and the concluding remarks.

2 Background and Related Work

In this section, we will begin by discussing the Agile development trend in the software industry. Then, we will go over academic and industry research on CI. Finally, we will highlight research on documentation, how these topics are related, and where there is a knowledge gap.

Agile Adoption One of the catalysts for changes in the software industry was the release of the Agile manifesto [8] in 2001. Around 2007, an exploratory study at Microsoft [9] found that approximately one-third of surveyed respondents were utilizing Agile methodologies to varying degrees, with Scrum being the most popular variant. Almost five years later, a 2011 survey of Finnish software practitioners [10] revealed that 58% of organizational units were employing agile and/or lean methods. By 2017, a survey of software engineers and managers [11] confirmed a clear trend towards agile adoption across organizations, with Scrum remaining the most common process in use. Most recently, the 17th State of Agile Report [1] in 2023 indicated that 71% of survey participants utilize Agile in their software development life-cycle, and Scrum continues to be the dominant team-level methodology.

Academic Research on CI There have been many studies on CI as a practice. Some explored pipeline standardization [12], while others evaluated the causal impact of adopting CI in software engineering (effects on frequency and size of commits, pull request handling, issue tracking, and testing practices) [13]. Due to the growing literature on CI and a lack of an integrated review on approaches, tools, challenges, and practices, Shanin et al. attempted to classify them and provide an evidence-based guide for selecting appropriate techniques depending on the context [14].

More concretely, Vasilescu et al. examined the impact of specific development practices on project outcomes by conducting a large-scale study on GitHub projects to discern the effects of Continuous Integration (CI) adoption. [15] They primarily investigated how CI influences team productivity, which they measured through aspects like the volume and handling of pull requests (e.g., the number of pull requests merged per month), and code quality, measured by the number of bug reports over time. Their main finding was that CI adoption improved the productivity of project teams, allowing them to integrate more contributions, particularly from core developers, and reduced the rejection rate for pull requests from external contributors. Importantly, this increase in productivity did not lead to an observable decrease in user-experienced code quality; while CI helped developers discover more defects internally, it was not associated with an increase in user-reported bugs.

Industry Research on CI To quantify the impact of the evolving practices and to guide organizations in their DevOps journey, industry research has focused on identifying key performance indicators. Notably, Google's DevOps Research and Assessment (DORA) program has been influential in this area. Through years of research, DORA proposed four key metrics to measure software delivery performance, categorizing them into measures of throughput and stability [6]. For throughput, there are deployment frequency and lead time for changes. For stability, the metrics are change failure rate and mean time to recovery (MTTR). Additionally, industry guides like the one provided by JetBrains for their TeamCity CI/CD tool also emphasize the importance of monitoring relevant IT operations indicators to optimize delivery pipelines and ensure operational health [7].

Research on Documentation Well-crafted documentation facilitates quick learning for new users and contributors, simplifies product understanding, and can reduce support costs by making information easily accessible. [4] To move beyond these general principles, however, academic research has looked to capture the practitioner's perspective on the value and challenges of documentation.

Building on the work by Plösch et al. and others, Aghajani et al. conducted a more extensive study with 146 practitioners to delve deeper into specific documentation issues. [16] Their surveys revealed that practitioners are most concerned with problems in the information content itself, such as faulty tutorials, incomplete installation instructions, and documentation that is inconsistent with the code. From a process standpoint, the most significant obstacle reported was the universal lack of time to create and maintain documentation. Additionally, this research established that code comments and contribution guidelines are perceived as more useful for various tasks. It also highlighted that contribution guidelines are often neglected and while that may not have a directly visible negative effect, it can have a strong positive impact when it is well-written.

Bringing it All Together The widespread adoption of Agile development has driven the growth of DevOps practices and, consequently, related academic research. Vasilescu et al., for instance, showed how to quantitatively assess the effects of Continuous Integration (CI) on team productivity. In parallel, industry researchers have sought to develop quantifiable metrics for DevOps performance. While qualitative studies have thoroughly explored the value, importance, and challenges of software documentation, a critical gap exists: the lack of quantitative data linking specific documentation practices directly to software delivery performance.

3 Methodology

For the aims of our research, we want to analyze open-source data. GitHub, being one of the most popular platforms for version control and collaboration, is ideal for this. Additionally, it is important to note that the key performance indicators we observe are typically applied in industry contexts. Therefore, we will need to adapt them for our needs. In section 3.3, we will go over the original definitions and our interpretations more in depth.

3.1 Data Selection

GitHub as a source of data provides immeasurable opportunities for researchers; however, it comes with caveats. [17] For example, many repositories are personal projects. Others have infrequent activity or are completely abandoned. Therefore, it was important to have a set of criteria when selecting projects. As a result, we developed a multi-step filtering process. Additionally, we analyzed data from May 15, 2024,

Table 1: Quartile distribution for project activity metrics, based on the initial dataset of 6715 projects.

Metric	Q1	Q2	Q3	Q4
Total Releases	≤ 1	≤ 21	≤ 69	>69
Commits (1-year)	≤ 1	≤ 35	≤ 263	>263

to May 15, 2025, for all three research questions. This time frame was chosen for its recency, relevance, and the availability of crucial workflow data used in our selection process.

Preliminary Filtering Based on the methodology conducted by Vasilescu et al. [15], we selected projects in seven of the most popular languages on GitHub: Python, JavaScript, TypeScript, Java, C#, C++, and PHP [18], and filtered out projects with less than 50 stars [17]. Additionally, we excluded forks and archived repositories. Using these criteria, we were able to select 6715 projects using the GitHub API.[19]

Secondary Filtering It was also necessary to identify the presence of CI in a project. Due to the various ways to configure pipelines, there is no standardized method of identifying them. Unfortunately, it is insufficient to check for the presence of *.yml* files, as sometimes they are used for automated bots or even issue templates. To overcome this obstacle, we instead relied on the GitHub Actions feature, which brings CI/CD to the platform. The GitHub API can provide data related to Actions usage for a given repository [20], but the data is limited to roughly one year in the past for all projects. Therefore, we considered a project to utilize CI if there was at least one workflow created before May 15, 2024. Following this step, we were left with 4097 repositories.

Tertiary Filtering To examine projects with meaningful data for our research questions, we adapted methods by Ray et al. [21] to measure activity. We established quantifiable thresholds by collecting data for the first 6715 projects. We observed the number of commits in the selected time window of one year and the total number of releases for a project during its lifetime. Based on these two metrics, we calculated quartiles to exclude projects with low measurable activity. An overview can be seen in Table 1 This method led to us including projects with at least 50 commits and at least 20 releases. As a result, 1921 repositories remained.

Final Filtering As we required GitHub issues data for two of our KPIs, we followed another technique by Vasilescu et al. [15] Projects were selected if they had a minimum of 100 issues and at least 75% of these issues were labeled. These metrics indicated adequate usage of the GitHub project management functionality and resulted in our final dataset of 670 projects.

3.2 Data Extraction

From our list of 670 projects, we methodically gathered data from two main sources: the Git repository history and the GitHub API.

Figure 1: The `cloc` command used to measure code and comment volume.

```
cloc --quiet
    --exclude-dir=doc,docs,test,tests
    --exclude-lang=JSON,Markdown,Text .
```

Repository Content Data For each project, we regularly sampled the main branch to gather information on in-repository documentation. At each sampling point, we collected:

- **Standard Documentation Files:** This includes the presence and line count of important files suggested by GitHub for community health. These files are `README`, `CODE_OF_CONDUCT`, `LICENSE`, and `CONTRIBUTING`. It also covers issue and pull request templates. [22] [23] To ensure our measurements matched the platform's behavior, our data extraction script looked for these files in the order stated by GitHub: first in the ".github/" directory, then in the project's root, and finally in the "docs/" directory. [24]
- **Code and Comment Volume:** This is the total number of lines of code and lines of comments. We extracted it by running the `cloc` command-line tool on the contents of the repository, using the specific command shown in Listing 1 to exclude documentation and test directories, as well as JSON, markdown, and text files.

GitHub API Data We used the GitHub API to collect historical data on project activity:

- **Commits:** The commit history, which tracks changes to documentation files over time.
- **Issues:** Data for all project issues, including creation and closing timestamps, labels, titles, and body text. This data is important for identifying bugs and measuring how long it takes to resolve them.
- **Releases:** The complete history of project releases, including publication timestamps and the body text of the release notes.

3.3 Concepts and Metrics

To answer our research questions, we transformed the extracted raw data into quantifiable metrics. This section explains the independent variables, which are documentation practices, and the dependent variables, which are key performance indicators, used for our analysis. We calculated all KPIs over 12 consecutive 30-day intervals to observe trends over time.

Documentation Completeness (RQ1) We utilize this metric to assess the quality and detail of a project's documentation over a 30-day time period. It is calculated using a step-by-step process that allows for comparison across different projects:

1. For each commit, we calculate three basic metrics: (a) the number of standard documentation files present, (b) the total lines of code (LoC) in these files, and (c) the

percentage of comments in the source code, which is calculated as:

$$\text{code comment percentage} = \frac{\text{num of code comment lines}}{\text{num of code lines}} \quad (1)$$

2. To address large differences between projects, we standardize each of these three metrics (convert them to a z-score) on a per-project basis. This shows whether a commit's documentation is above or below that project's own average.
3. We create a final documentation completeness score for each commit by adding these three standardized scores.
4. The value used for each 30-day analysis period is the average of this composite score across all commits made during that time.

This method lets us track improvements in documentation within a project over time, instead of comparing unrelated values between projects.

Documentation Update Ratio (RQ2) This independent variable measures how much of the development effort goes into keeping documentation up to date. To find this, we first looked at every commit within a 30-day period that changed a standard documentation file (for example, README, CONTRIBUTING). We then calculated the ratio for each project and each interval using the following formula:

$$\text{Doc Update Ratio} = \frac{\text{Num of commits altering documentation}}{\text{Total num of commits}} \quad (2)$$

This metric highlights the frequency of documentation updates relative to the overall project activity.

Release Documentation (RQ3) This metric measures the effort spent on communicating changes through release notes. For each 30-day interval, we first calculate the total number of lines of text across the bodies of all releases published within that period. To control for project-specific norms where some projects naturally have longer release notes than others, this total line count is then standardized (converted to a z-score) on a per-project basis. The final metric used in our model represents whether the release documentation effort in a given interval was higher or lower than that project's historical average.

Delivery Frequency (RQ1)

- **Original Definition:** JetBrains [7] defines deployment frequency as the number of times a CI/CD pipeline deploys to a production environment.
- **Adapted Metric:** In open-source projects, we cannot easily identify production deployments. Therefore, we adapted this metric by looking at official project releases.
- **Calculation:** We measure delivery frequency as the total number of a project's releases during each 30-day period.

Defect Count (RQ2)

- **Original Definition:** This metric refers to the "number of open tickets in your backlog classified as bugs" [7].
- **Adapted Metric:** We use GitHub's issue tracking system to estimate this. An issue is labeled as a "bug" if its title or body includes keywords like *"defect", "error", "bug", "issue", "mistake", "incorrect", "fault,"* or *"flaw,"* and does not include resolution terms such as *"fix"* or *"resolve."* This method comes from Vasilescu et al. [15].
- **Calculation:** We calculate the defect count by counting the number of "bug" issues that were open during each 30-day period.

Mean Time to Recovery (MTTR) (RQ3)

- **Original Definition:** MTTR is the average time it takes to restore a service after a production failure [7].
- **Adapted Metric:** As it is hard to identify production failures in open-source projects, we adapted MTTR to measure the average time to fix "bug" issues.
- **Calculation:** We first calculate the time taken to solve each individual "bug" issue (as identified in RQ2) that was closed during a specific period. MTTR for that period is the average of these solve times.

$$\text{solve time} = \text{time issue is closed} - \text{time issue is created} \quad (3)$$

3.4 Data Analysis

To examine the relation between documentation practices and key performance indicators, we used a statistical modeling approach designed for longitudinal panel data with possibly non-linear relationships. The main method for all three research questions was the Generalized Additive Mixed Model (GAMM). We chose GAMMs for several important reasons.

Non-Linearity The effect of documentation on performance metrics is not always linear. For instance, the benefit of extending documentation may reduce after a certain point. GAMMs can represent these complex, non-linear patterns with spline functions, offering a more flexible and realistic model of the data.

Project-Specific Effects Every open-source project has unique baseline characteristics. For example, one project may regularly update their documentation and have an update ratio of around 35% consistently, with peaks around 60%. For another repository, it might be more typical to have an update ratio of 20% with spikes at 40%. If we do not consider these differences, it could lead to inaccurate results. By adding the project's identity as a fixed effect term (`f(project_id)`) in our models, we can account for these project-specific differences. This enables us to focus on the impact of changes in documentation within each project over time.

Different Data Distributions Our dependent variables (KPIs) have different statistical properties. For count data like delivery frequency (RQ1), we used a Poisson GAMM. For the other KPIs, which were highly skewed, such as defect count and MTTR, we applied a logarithmic transformation

Table 2: Summary of the Poisson GAMM for RQ1. The dependent variable is delivery frequency.

Feature	EDoF	P-value
s(Doc. Completeness Score)	8.2	<0.001
s(Time Interval)	8.7	<0.001
f(Project ID)	629.0	<0.001

(`log(x+1)`). This was done to stabilize the variance and normalize the distribution. We then analyzed this transformed data using a Linear GAMM, which assumes a Gaussian distribution.

For each research question, the model took the general form:

$$KPI \sim s(documentation\_metric) + s(time\_interval) + f(project\_id)$$

where `s()` denotes a smoothing spline function. This model lets us evaluate the partial effect of the documentation metric on the KPI, while also controlling for overall time trends and project-specific baselines. For time periods where we could not calculate an independent variable because there was no activity, such as no commits or releases, we replaced the value with 0. We carried out the analysis in Python using the `pygam` library. The results will be presented in the next section.

4 Results

In this section, we will present the findings from our statistical analysis. For each question, we give a summary table of the fitted Generalized Additive Mixed Model (GAMM). This table shows the statistical significance and the type of relationships found. Next, we include a partial dependence plot. This plot showcases how the documentation metric impacts the corresponding key performance indicator, after accounting for the other variables in the model.

4.1 RQ1: Documentation Completeness and Delivery Frequency

To answer our first research question, we aimed to observe how our documentation completeness score affects delivery frequency by using a Poisson GAMM. The model fit well and explained a significant portion of the variance in the data (Pseudo R-Squared = 0.84).

The summary of the GAMM results can be seen in Table 2. All terms in the model were statistically significant predictors of delivery frequency ($p < 0.001$). The documentation completeness score had an effective degrees of freedom (EDoF) of 8.2. This value is much greater than 1. It shows that the relationship between documentation completeness and delivery frequency is complex and non-linear. A simple linear correlation would not be able to capture completely.

The nature of this non-linear relation is visualized in the partial dependence plot in Figure 2. The plot showcases the predicted delivery frequency as a function of the standardized documentation completeness score, after accounting for all other variables in the model.

For most projects, the documentation score falls within a range of standardized values between -3.0 and +3.0. Here,

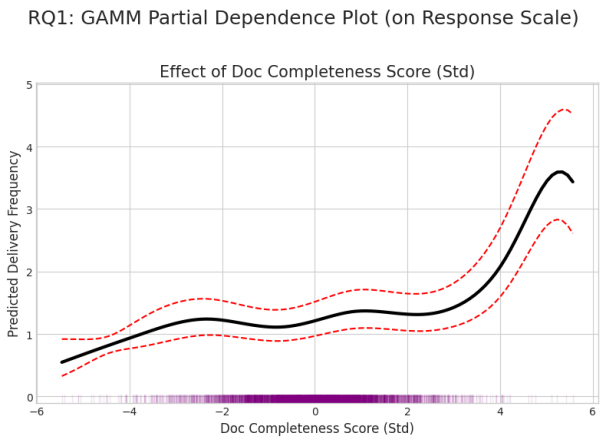

Figure 2: Partial dependence plot for the documentation completeness score. The y-axis shows the predicted effect on delivery frequency. The solid black line is the predicted relationship, the dashed red lines represent the 95% confidence interval, and the purple rug plot at the bottom shows the distribution of data points.

there is a positive but modest link to delivery frequency. This indicates that keeping a basic level of documentation, such as having a README, contributing guidelines, and some code comments, is helpful. However, it may not be enough by itself to significantly speed up a project's development cycle.

On the other hand, a clear tipping point occurs with scores greater than +3.0. At this level, the investment in documentation starts to pay off, resulting in a noticeable increase in delivery frequency. This indicates that the documentation has reached a "critical mass", changing how the team operates.

In summary, our analysis addresses RQ1 by showing a clear, non-linear relationship. Delivery frequency increases slightly with basic documentation. However, it speeds up significantly once documentation completeness exceeds a 'tipping point' of +3.0. The main takeaway is that excellent documentation serves as a powerful tool for development. It likely goes beyond simple descriptions and actively reduces problems for developers by speeding up onboarding, strengthening their understanding of the code base, and allowing them to make changes more quickly and confidently. Therefore, while basic documentation is good practice, teams aiming to increase their delivery speed should focus on achieving a high standard of documentation quality, as this is where the real benefits come from.

4.2 RQ2: Documentation Update Ratio and Defect Count

For our second research question, we looked at the relation between the documentation update ratio and the number of open defects. Because the defect count data was skewed, we used a logarithmic transformation (`log(x+1)`) and modeled the outcome with a Linear GAMM. The overall model fit was very high, achieving a Pseudo R-Squared of 0.96.

Table 3 shows the summary of the model's terms. All terms included were highly significant predictors ($p < 0.001$). The documentation update ratio has an effective degrees of freedom (EDoF) of 7.1. This value is much greater than 1, which

Table 3: Summary of the Linear GAMM for RQ2. The dependent variable is log(defect count + 1).

Feature	EDoF	P-value
s(Doc. Update Ratio)	7.1	<0.001
s(Time Interval)	7.9	<0.001
f(Project ID)	636.6	<0.001

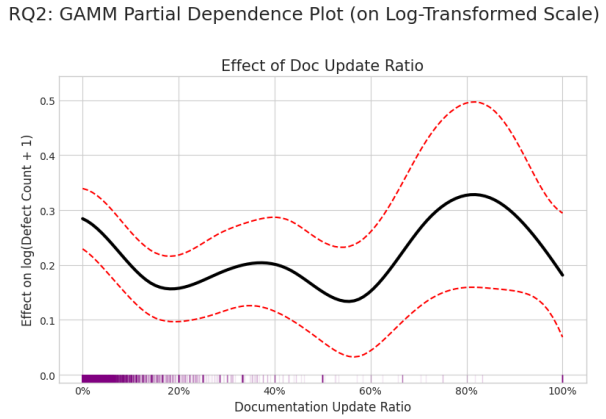

Figure 3: Partial dependence plot for the documentation update ratio. The y-axis shows the predicted effect on log(defect count + 1). The solid black line is the predicted relationship, the dashed red lines represent the 95% confidence interval, and the purple rug plot shows the data distribution.

confirms that the relationship between how often documentation updates occur and the defect count is complex and non-linear.

This non-linearity appears in the partial dependence plot in Figure 3. The plot displays the predicted effect on the log-transformed defect count. It shows a complex, wave-like pattern that indicates the relationship is very contextual.

The analysis shows an interesting and not immediately clear link between how often documentation is updated and the quality of software. The results indicate that the goal is not just to increase the documentation update ratio, but to keep it within a "healthy" range.

The projects with the lowest defect counts tend to have 20% to 55% of their commits focused on documentation changes. This "sweet spot" probably reflects a well-developed and cohesive process where documentation is regularly updated alongside code. The rug plot at the bottom of the chart supports this view, showing that many project intervals fall within this effective range, especially between 20% and 40%. Additionally, the narrow confidence intervals suggest that the model is statistically confident about this positive impact.

Additionally, the model shows that going to either extreme is linked to a higher defect count. A low update ratio (<20%), which the rug plot indicates as quite common, is associated with a higher number of defects. The model is confident in this finding, suggesting that allowing documentation to become outdated actively contributes to lower software quality. On the other hand, a very high update ratio (>55%) is related

Table 4: Summary of the Linear GAMM for RQ3. The dependent variable is log(MTTR + 1).

Feature	EDoF	P-value
s(Release Doc. Size)	7.9	0.685
s(Time Interval)	7.9	<0.001
f(Project ID)	636.8	<0.001

to the highest number of defects. This seemingly surprising result is explained by the rug plot and confidence intervals. The rug reveals that such high ratios are uncommon. Instead of a standard practice it might rather be a sign of underlying project issues, such as a major refactoring. While the confidence intervals widen in this upper range due to limited data, the upward trend is statistically significant ($p < 0.001$) and clearly shows that this situation is problematic.

Therefore, in response to RQ2, our findings indicate that documentation update frequency has a complex, non-linear correlation with defect count. There is a 'sweet spot' between a 20% and 55% update ratio that corresponds to the lowest number of defects, suggesting this range represents a healthy project rhythm.

4.3 RQ3: Release Documentation and Mean Time to Recovery

For our final research question, we observed how the size of release documentation affects mean time to recovery (MTTR). As the MTTR data was skewed (similarly to defect count), we used a logarithmic transformation (`log(x+1)`) and applied a Linear GAMM. The overall model fit was moderate, with a Pseudo R-Squared of 0.50.

The results of the GAMM, summarized in Table 4, show a clear and important finding. The control variables for time and project identity were highly significant. However, the independent variable of interest, release documentation size, had no statistically significant relationship with MTTR ($p = 0.685$).

The partial dependence plot in Figure 4 visually confirms this lack of a significant effect. The plot showcases the predicted effect of release documentation size on the log-transformed MTTR.

The estimated effect, shown by the black line, remains close to zero throughout the data range. A zero effect on the y-axis indicates no impact on the log-transformed MTTR. Additionally, the 95% confidence interval, represented by the red dashed lines, is wide and always includes the zero line. This visual pattern suggests a non-significant result. It likely stems from random noise in the data, rather than a true relationship.

The rug plot demonstrates that the model had enough data, particularly for projects with average to above-average release documentation sizes, indicated by scores from -1.0 to +2.0. The lack of a significant effect, despite this data, reinforces the conclusion.

Answering RQ3, our analysis found no statistically significant relationship between the volume of release documentation and the mean time to recovery. Practically, increasing the amount of documentation in a release does not seem to

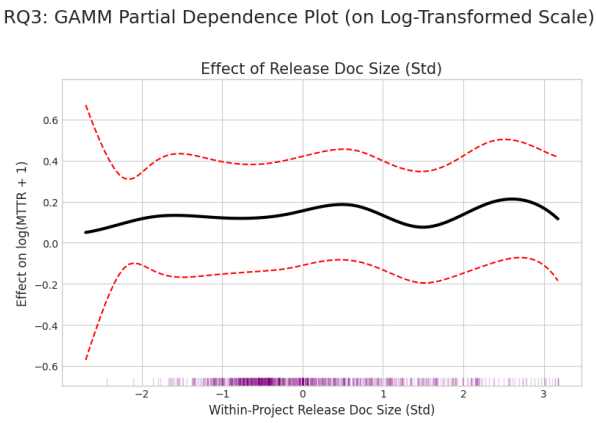

Figure 4: Partial dependence plot for release documentation size. The y-axis shows the predicted effect on log(MTTR + 1). The solid black line is the predicted relationship, the dashed red lines represent the 95% confidence interval, and the purple rug plot shows the data distribution.

help teams restore service faster. This implies that for incident response, the overall size of release notes is not the key factor. Other types of documentation or the actual content might serve as better indicators. In general, teams should be careful about prioritizing documentation volume as a strategy for enhancing system stability and recovery speed.

5 Discussion

Our research aimed to measure the effect of documentation on key performance indicators, going beyond the belief that it is simply "important." The findings show that the relationship is complex, dependent on context, and not necessarily straightforward. This suggests that the "how" and "when" of documentation are more important than just the amount.

Quality, Rhythm, and Purpose over Volume Our analysis shows a "tipping point" where detailed, high-quality documentation changes from helping with maintenance to directly increasing delivery frequency. This supports the views of practitioners who believe that well-made documentation reduces cognitive load and simplifies development [5] [16]. Having basic files is good practice. However, reaching a high standard in documentation seems to significantly boost team development speed.

This is supported by our finding of a "sweet spot" for keeping documentation up to date. A documentation update ratio of 20-55% relates to the lowest defect counts. This shows that regularly updating documentation with code is important for quality. Going outside this range signals issues. A low ratio shows outdated documentation that can confuse developers. On the other hand, a very high ratio may be a symptom of project instability, like a major refactoring, rather than being a direct cause of defects.

Finally, our finding about release note size and mean time to recovery (MTTR) shows the importance of purpose. Lengthy release notes do not help developers solve problems. Instead, the content should be clearer, or other documents might work better.

Implications and Future Directions These findings offer actionable insights for both practitioners and researchers.

For Practitioners: First, they should focus on good, thorough documentation to cross the "tipping point" and speed up delivery, instead of just meeting a minimum standard. Second, they should track the documentation update ratio as a measure of project health. A steady, moderate ratio shows healthy practices, while extremes could suggest technical debt or project issues. Lastly, they should not depend on release notes; instead, they should turn to technical documentation for quick recovery.

For Researchers: There are new opportunities for diving deeper into the topic. Future studies could use NLP to look at the content and quality of documentation, or apply causal inference models to go beyond correlation. Additionally, expanding the scope to include documentation on external platforms, like wikis and ReadTheDocs, presents both a major challenge and an opportunity.

Threats to Validity This study, like any empirical research, faces threats to its validity. We outline the most important threats and our strategies to address them below. These are organized by internal, construct, and external validity.

Threats to Internal and Construct Validity A major threat to construct validity is how we defined important concepts. For example, our metrics for defect count and MTTR rely on finding "bugs" through keyword analysis of GitHub issues. This method can yield unreliable results because the classification of issues may vary. A study by Herzig et al. [25] revealed that approximately one-third (33.8%) of issues marked as 'bugs' are frequently misclassified. Although we used standard keyword filtering techniques to minimize this problem, some mislabeling is unavoidable, which adds noise to our dataset. Additionally, our measurement of documentation only includes items found in the Git repository, such as README files, code comments, and release notes. This method does not take into account documentation stored on external platforms like wikis or ReadTheDocs. Because of this, we may underestimate the total documentation effort of a project. We decided to focus on standardized, repository artifacts to establish a consistent and comparable baseline across all the projects in our study. Lastly, a minor threat to internal validity is data stability. Project artifacts such as release notes and commits can be changed even after they are created. We believe such changes are infrequent and doa not systematically bias the results within our analysis period.

Threats to External Validity We conducted our study on a specific group of open-source projects from GitHub. These projects had to meet several criteria: CI through GitHub Actions, minimum level of activity, and developed in one of seven popular programming languages. This sampling was necessary to ensure data quality and relevance to our research questions, but it limits the scope of our conclusions. As a result, our findings may not apply to all software projects. We should be careful when applying these results to different settings, such as proprietary or closed-source projects, projects in different fields or less common programming languages, or projects that do not use a formal CI process.

Responsible Research Our study was conducted using publicly available data from GitHub, and our analysis was performed at the project level. To ensure transparency and enable replication, all scripts and data are available.

- The data collection tool is available at: https://doi.org/10.5281/zenodo.15711349
- The final dataset is available at: https://doi.org/10.5281/zenodo.15713021

Concluding remarks This study aimed to examine the impact of documentation practices on key performance indicators in Continuous Integration open-source projects. We looked at 670 repositories to go beyond the common belief that documentation is important and offer specific, data-driven evidence of its effects.

Our analysis revealed three important findings. First, we found a "tipping point" where having complete documentation goes from being a helpful practice to a key factor in increasing delivery frequency. Second, we discovered a "sweet spot" for maintaining documentation. Projects that spent between 20% and 55% of their commits on updating documentation had the lowest defect counts. This suggests that this range indicates a healthy development rhythm. Finally, our research showed no clear link between the length of release notes and faster mean time to recovery. This indicates that having more content is not a replacement for clear and usable technical documentation when fixing problems.

Collectively, these results support a central theme. The effectiveness of documentation depends on its quality, rhythm, and purpose, rather than its sheer volume. This research offers useful metrics that give developers and teams a clearer understanding of how to use documentation. It is not just an artifact for maintenance, but rather it should be utilized as a way to improve software delivery and quality.

References

[1] Digital.ai. 17th annual state of agile report. Technical report, Digital.ai, 2023. Based on a survey of 788 respondents, providing insights into Agile adoption, usage, and challenges in 2023.

[2] Bill Phifer. Next-generation process integration: Cmmi and itil do devops. *Cutter IT Journal*, 24(8):28, 2011.

[3] Andrew Forward. *Software documentation: Building and maintaining artefacts of communication.* University of Ottawa (Canada), 2002.

[4] Noela Jemutai Kipyegen and William PK Korir. Importance of software documentation. *International Journal of Computer Science Issues (IJCSI)*, 10(5):223, 2013.

[5] Reinhold Plösch, Andreas Dautovic, and Matthias Saft. The value of software documentation quality. In *2014 14th International Conference on Quality Software*, pages 333–342. IEEE, 2014.

[6] DORA Research Team. 2024 accelerate state of devops report. Technical report, Google Cloud, 2024. A decade with DORA.

[7] JetBrains. Measure ci/cd performance with devops metrics. https://www.jetbrains.com/teamcity/ci-cd-guide/devops-ci-cd-metrics/. Accessed: 2025-05-12. Undated.

[8] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001. Accessed: 2025-05-21.

[9] Andrew Begel and Nachiappan Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 255–264. IEEE, 2007.

[10] Pilar Rodríguez, Jouni Markkula, Markku Oivo, and Kimmo Turula. Survey on agile and lean usage in finnish software industry. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 139–148, 2012.

[11] Mohammad S Raunak and David Binkley. Agile and other trends in software engineering. In *2017 IEEE 28th Annual Software Technology Conference (STC)*, pages 1–7. IEEE, 2017.

[12] Ionut-Catalin Donca, Ovidiu Petru Stan, Marius Misaros, Dan Gota, and Liviu Miclea. Method for continuous integration and deployment using a pipeline generator for agile software projects. *Sensors*, 22(12):4637, 2022. Publisher: MDPI.

[13] 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 *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017.

[14] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017. Publisher: IEEE.

[15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir 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, 2015.

[16] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 590–601, New York, NY, USA, 2020. Association for Computing Machinery.

[17] 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.

[18] GitHub Staff. Octoverse: Ai leads python to top language as the number of global developers surges. GitHub Blog, October 2024. Accessed: 2025-06-18.

[19] GitHub, Inc. *REST API endpoints for repositories*. GitHub Docs, June 2025. API version: 2022-11-28.

[20] GitHub, Inc. *REST API endpoints for workflows*. GitHub Docs, June 2025. Version: 2022-11-28 (latest as of access).

[21] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014.

[22] GitHub. About Community Profiles for Public Repositories. https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/about-community-profiles-for-public-repositories, 2024. Accessed: 2024-06-02.

[23] Open Source Guides. Best Practices for Maintainers. https://opensource.guide/best-practices/, 2024. Accessed: 2024-06-02.

[24] GitHub. Creating a Default Community Health File. https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/creating-a-default-community-health-file, 2024. Accessed: 2024-06-02.

[25] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*, pages 392–401. IEEE, 2013.