Vrije Universiteit Amsterdam

Bachelor Thesis

Benchmarking Software Maintainability in Enterprise-Driven Open Source Projects in Terms of Developer Engagement

---

Author: Ahmet Yasir Uçkun (2666346)

1st supervisor: Sieuwert van Otterloo
daily supervisor: Lodewijk Bergmans (SIG)

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 10, 2025

Contents

Abstract

Software maintainability is a critical quality attribute for long-lived software systems, affecting development costs, agility, and reliability across domains such as healthcare and finance. However, existing dashboards like SIG's Sigrid evaluate only snapshot maintainability without relating it to development-process dynamics. This thesis addresses that gap by benchmarking maintainability against developer engagement metrics in enterprise-driven open-source projects.

From a pool of over 17,000 GitHub repositories, two complementary samples of 40 projects each ("active" and "average") were constructed via controlled-feature and high-activity sampling to balance representativeness and relevance. Developer engagement was quantified monthly using a PCA-based index combining commit churn, frequency, team size, and inter-commit intervals, normalized to $[0, 1]$. Maintainability impact was measured per commit via the Delta Maintainability Model, computing churn-weighted ratios of its indices.

Empirical analysis shows that engagement is bursty and short-lived, while maintainability scores exhibit a right-skewed distribution with occasional deep dips. Cross-correlation reveals only weak, transient links between engagement peaks and subsequent quality gains. These findings suggest that bursts of developer activity alone do not guarantee lasting maintainability improvements as an immediate correlation.

1 Introduction

1.1 Background and Context

In today's digital age, software is critical in infrastructures, from healthcare systems to financial platforms, and its long-term sustainability has become a cornerstone of reliable operations. As projects scale and evolve, the challenge shifts from simply delivering features to ensuring that code remains understandable, modifiable, and resilient. This long-term quality measure, known as maintainability, is central in standards such as ISO-9126/25010 and affects development costs and agility [MM20]. Measurement and improvement of this metric became a goal a long time ago [CM78] and is still an ongoing challenge.

Software Improvement Group (SIG) is a company specialized in evaluating software quality to advise clients on improving software maintainability, security, and overall performance [KA23]. They develop their evaluation model [HKV07] and provide it as a dashboard called Sigrid to their customers. Their dashboard is capable of evaluating the code-bases, providing scores on different categories, including maintainability. They also focus on improving the existing software and possible scenarios concerning the funds it will require to do so. In this way, Sigrid provides a bridge between developing teams and management as a third party to allow the best overview of the software, and they continue to develop their services with more data and additional features.

1.2 Problem Description

The problem to be solved in this project is provided by the SIG. Sigrid evaluates the latest version of a project, but does not provide information on how the characteristics of the development process - such as commit cadence, issue throughput, or team turnover - impact long-term maintainability. The absence of a process-level benchmark inhibits the

elicitation of additional version history data from clients and constrains the road map for advanced analytics.

The lack of data to develop such a benchmark can be alleviated by using similar projects that are representative of the type of project that SIG customers expect. SIG's customers mainly consist of enterprises with closed-source code bases, and their data is not disclosed.

The provided dataset comprises over 17,264 open-source projects from GitHub that are mainly developed by enterprises [Spi+20]. For every repository, 29 features are recorded, including the number of unique committers, total commits, pull requests, watchers, and lines of code, as well as key dates such as creation date and earliest and latest commit dates. The dataset was last updated in April 2020. For some projects, the period between that update and the current date exceeds one third of their total life. Consequently, this data set will only be used as a list for the projects, and the relevant features will be mined later with scripts to be up-to-date.

Since the reason for developing this benchmark is to gather more information about the development steps of a project, rather than the end product, the features to be analyzed should be representative. The quality of a product changes only when there is a change in the code. Commit and churn can be two metrics that can represent these changes. While churn defines the number of lines that changed in the code, commit is the term for defining the change itself on a project. Commit has the data for the developer who made the change, churn, date of the change, etc. Commits can also be grouped per developer to measure the engagement of individual developers to the project and their work depth. With these features, the frequency of change in the code, the depth of the change (churn), and the number of developers actively working on the project can be measured and used for further analysis.

With these representative features, a benchmark can be created as a proof of concept that is reasonably applicable to the expectation of SIG's customers. Since those customers will then have a reason to provide the versioning system data to SIG for use with the newly created benchmark model, it can be further developed towards the customer data, which would resolve the research proposal.

1.3 Research Questions

The primary goal of this thesis is to evaluate how well simple developer engagement metrics can serve as predictors of software maintainability, as measured by the Delta Maintainability Index.

Main Research Question:
How effectively can developer engagement metrics (e.g. commit frequency, contribution duration) be used to benchmark the maintainability of open source projects in terms of the Delta Maintainability Score?

The investigation is refined by the following sub questions:

SQ1. How can a developer engagement metric be generated using commit-level project features?

SQ2. How can maintainability be measured with the DMM (Delta Maintainability Model) as a score, and how does it evolve over time?

SQ3. Is there a relationship between developer engagement and Delta Maintainability Score?

2 Related Work

2.1 Mining GitHub

GitHub offers valuable and publicly available data. It has more than 100 million users and 800 million open data files [Rom+23]. This allows for various and rich possible datasets that can be generated. The PyDriller library allows for easily downloading a GitHub repository and creating an object that can be used later [SAB18]. This library is used through the project to allow further inspection of certain projects features that the original dataset lacks (individual commits, bug reports, etc.).

2.2 Developer Engagement

Spinellis' (who is also the author of the data set used in this project) and other authors' work is concerned with measuring the participation and activity of developers in the face of distributed and incremental development practices [GKS08]. They thoroughly analyze the publicly available data to measure the engagement of a developer. The SPACE framework tries to capture developer productivity. In their research, it is concluded that productivity cannot be reduced to a single dimension or a metric [For+21]. They also discuss how SPACE can guide organizations through disruptions, such as the COVID-19-driven shift to remote work, by ensuring that productivity assessments remain fair, context-aware, and conducive to both engineering outcomes and developer satisfaction. Spinellis's quantitative analysis of the FreeBSD project demonstrated that the average monthly lines of code committed correlates strongly (Pearson's $r = 0.67$) with the share of commits contributed by distinct developers, validating commit-count as a proxy for productive engagement [Spi06].

2.3 Maintainability

Software maintainability has traditionally been assessed via static-analysis metrics - cyclomatic complexity, coupling, lines of code — combined into composite indices such as the Maintainability Index. The systematic review by Malhotra and Chug [MC16] documents the rise of machine learning–based predictors since 2005, while Borg et al. benchmark four state-of-the-art approaches (ML models, SonarQube ratings, CodeScene code health, and the Microsoft maintainability index) against expert judgments [BET24]. More recent delta-driven methods, typified by the Delta Maintainability Model developed by SIG and TU Delft and integrated into the PyDriller library, track quality evolution across repository snapshots and compute a score that captures both instantaneous maintainability and its temporal change [Di +19]. Empirical studies have established statistical associations between these maintainability metrics and developer engagement indicators [Bir+09] (commit frequency, review iterations, contributor churn), but most remain static or aggregate, offering limited insight into how process-level engagement dynamics drive code health trajectories. Code churn, the total number of code lines added and removed, has been shown to carry real information about both developer activity and software health.

In a large empirical study of the Windows Vista code base, Bird et al. found that modules with higher churn rates experienced modestly more post-release faults even after accounting for differences in team size. This shows that churn is not just noise; it reflects the risk of maintainability in the code itself [Bir+09]. Spinellis' work on the Free-BSD project reinforces churn as a proxy of effort: When more code is churned, more developers tend to be active in the project. At the same time, he showed that churn by itself doesn't fully explain variations in code-style violations or defect reports, unless you also consider how many distinct contributors were involved [Spi06].

3 Methods

3.1 Data Filtering

Every procedure in this chapter is placed in a public repository that will be available to anyone who wishes to replicate the work [Uck25].

To see which of the 17264 projects is still relevant, performing an activity check on the entire list was a necessity. To do this, checking the amount of commits done within a certain time period in the past was the most straightforward solution. The number of projects with at least one commit in the last three months (`last90` as a column) was around 1200. This means that almost 95% of the projects in the original data set are not actively maintained, which is proof of the requirement for this filtering step. Even with non-active projects removed from the data set, remaining projects are still beyond what the resources at hand can handle.

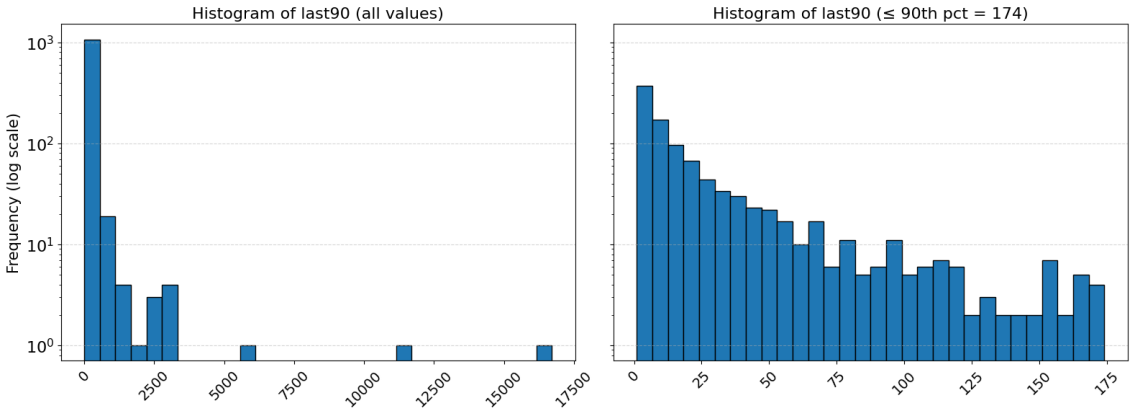


Figure 1: Number of commits in the last 90 days

Creating a smaller sample set from the 1200 active projects to begin working on was the next step to be taken. If the process can be proven on a smaller scale, it could later be adapted to a larger set, if not the whole set, to allow for a more concrete outcome. Two different approaches were taken into account when creating a sample set. The first is to find the median of the existing, outdated features, and aim for a set with similar feature values. This way would allow for an "average" data set that represents the most occurring project values. The other approach was to ignore the outdated values from 2020 and get the most active projects, calculated by how many commits were made in the last three

months (Jan. 2025 - Mar. 2025). This active projects set will not be as representative of the existing data as the previous one, but these projects are a better fit for SIG's expectancy, since SIG's clients are working actively. These methods and sample sets will be addressed as an "average set" and an "active set" throughout the investigation.

3.1.1 Controlled-Feature Sampling

To form a sample set of n projects that (1) stay close to the core volume metrics (commits, lines in the code, number of commits in the last three months) and (2) remain as diverse as possible on the other dimensions, the following two-phase procedure is applied:

1. **Primary filtering.**
    - From the full pool, drop any projects with missing values in the selected features.
    - Standardize (z-score) the *primary* features {`commit_count`, `last90`, `lines`}.
    - Compute each project's Euclidean distance to the coordinate-wise median in this 3-D space.
    - Retain the top $K = n \times f$ projects with smallest distance (e.g. $f = 5$).
2. **Diversity selection.**
    - On the K survivors, take the remaining features {`committer_count`, ...} and z-score them.
    - Initialize the chosen set with the single project closest in the primary 3-D space.
    - Greedily add projects one by one: at each step, pick the candidate whose *minimum* Euclidean distance (in the "diversity" feature space) to the already-selected set is *maximal*.
    - Stop when n projects have been chosen.

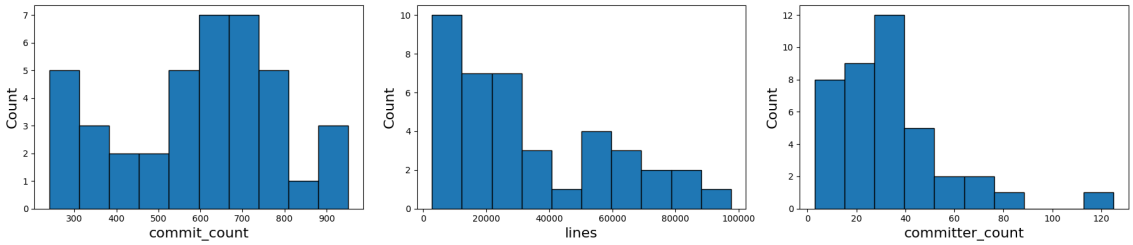


Figure 2: Important features of the average sample set after first sampling process

The resulting sample set in this procedure has 40 projects with similar volumetric feature values {`commit_count`, `lines`, `last90`}. While having a diverse value space for the remaining features regarding the team size {`committer_count`, `author_count`, ...}.

3.1.2 High Activity Sampling

The later approach was to create another sample set consisting of highly active projects that are closer to the profile of the SIG's customers. In high activity projects, the outdated data is expected to change more. Hence, choosing based on these features would not be as useful. This is why instead of the previous steps taken in creating the average set, a static threshold was chosen for `last90` to act as an activity metric. Unlike the previous approach where the volume of the project is tried to kept similar, no other feature related to volume was included in this approach. This is because of the assumption when the activity is higher, the existing values on the list are tend to increase more.

- From the full pool, drop any projects with missing values in our selected features.
- Sample n number of project with `last90` value being at max k.

The goal of having a value k to limit the number of commits is to keep the number of total commits in a reasonable amount. Even within a limit, the collection of the data could take tens of hours. This is a possible limitation that will be explained later. However, in this case k is chosen to be 200. This way, the values will be around 90th percentile for `last90`.

This straightforward method would allow for a set of projects with high and similar activity. Their old features are ignored, considering the possible change with high activity volume. The new features are also collected for these projects.

3.2 Data Gathering

3.2.1 Developer Engagement

Commit frequency, considered in isolation, does not distinguish between trivial and substantial contributions. A repository may record numerous commits that represent only whitespace adjustments, comment tweaks, or minor typo fixes, thereby inflating the apparent activity level without introducing meaningful code changes. In contrast, a single commit can encompass extensive feature development or large-scale refactoring, contributing much more to the project's evolution than multiple smaller commits combined. That is why, code-churn is able to represent the magnitude of the change over time. Therefore, it was added as a target value in this process alongside commits.

As it is seen in the Figure 3, an increase in the number of commits does not always mean an increase in the number of lines changed in the project. This proves the need to gather churn data alongside commits to better understand the depth of the development.

3.2.2 Delta Maintainability Metrics

Per-commit maintainability impact was quantified using the Delta–Maintainability Model (DMM). For each change, three ratios were calculated in the interval $[0, 1]$:

- **Unit size:** the fraction of modified units whose size decreased,
- **Unit complexity:** the fraction of modified units whose cyclomatic complexity decreased,

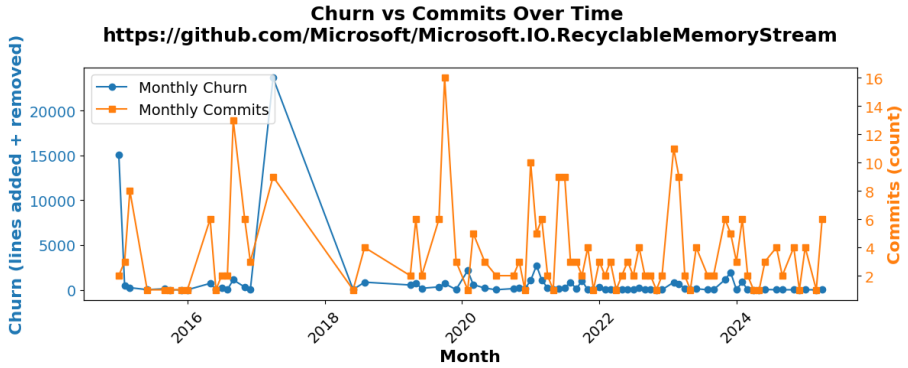


Figure 3: Comparison of commit against churn data on an example repository.

- **Unit interfacing:** the fraction of modified units whose coupling decreased.

A DMM score of 0.8 for unit size, for example, indicates that 80% of the affected functions or classes in that commit became smaller, implying a likely improvement in readability and maintainability. This index can also be considered as *good change / all change*. By combining these "good-change" ratios with the total churn (lines added plus removed) per commit, a time series of net maintainability impact can be constructed and directly compared to measures of developer activity.

3.2.3 Automated Data Extraction

Commit level data were harvested in an automated fashion using the PyDriller library. Each repository was cloned and only commits newer than a specified cutoff date were traversed. For every commit, the following attributes were recorded:

- repository identifier,
- developer identifier,
- commit timestamp,
- total churn (lines added + removed),
- DMM ratios (unit size, unit complexity, unit interfacing).

The data is then saved for each project in both sample sets. The gathered data in this case was chosen to be limited to 30 months. This is a choice made to keep the data gathering process relatively easier. The amount of data is enough to see repeating monthly trends. In the active set, there are more than 75000 commits collected for 40 projects in this time period. In average set, the case is around 6600 commits, less than a 10th of the active dataset. After collecting the needed data, the analysis can continue.

3.3 Data Post-Processing

The goal of this section is to turn the raw engagement data—changed lines, commit counts, and who is on the team—into a single, easy-to-interpret number that measures

'how engaged' project developers were in any given moment. And calculating a cumulative maintainability score based on the DMM metrics to validate the applicability of the suggested approach.

3.3.1 Developer Engagement

A monthly engagement index $E_{p,m} \in [0, 1]$ is assigned to each project p and calendar month m. Its computation proceeds in five stages, balancing simplicity and comparability among projects of different sizes. The method preferred for creating the developer engagement metric, using the collected data is as follows:

1. **Raw-metric extraction.**
   For every commit c in project p, the following values are recorded:

$$\Delta_c \;=\; \text{lines added} + \text{lines removed}, \quad d_c \;=\; \text{developer identifier}$$

$$t_c \;=\; \text{commit timestamp.}$$

   Commits are sorted by (url, d_c, t_c) and for each commit the interval

$$\tau_c = \frac{t_c - t_{\text{prev}(c)}}{1\,\text{hour}}$$

   is computed (zero for a developer's first commit). Each commit is then assigned to its calendar month YYYY-MM(t_c).

2. **Per-month aggregation.**
   For each project p and month m, the following aggregates are computed:

$$churn_{p,m} = \sum_{c\,:\,\text{month}(c)=m} \Delta_c, \qquad commits_{p,m} = \big|\{c : \text{month}(c) = m\}\big|,$$

$$devs_{p,m} = \big|\{\, d_c : \text{month}(c) = m\}\big|, \qquad \overline{\tau}_{p,m} = \frac{commits_{p,m}}{\sum_{c:\text{month}(c)=m} \tau_c \;+\varepsilon} \quad (\varepsilon \ll 1)$$

   and the per-developer average churn,

$$churn\_per\_dev_{p,m} = \frac{churn_{p,m}}{devs_{p,m}}.$$

3. **Feature scaling and dimensionality reduction.**
   First, each raw feature is made robust to outliers via a median-and-IQR transform (RobustScaler). Second, within each project p each feature is min–max normalized to $[0, 1]$. Finally, the five resulting axes

$$\big\{\, churn_{p,m}, commits_{p,m}, devs_{p,m}, \overline{\tau}_{p,m}, churn\_per\_dev_{p,m} \big\}$$

   are reduced via PCA to a single scalar

$$e^{\text{raw}}_{p,m} = \text{PCA}_1\big(\text{robust \& min–max scaled}\big).$$

   This collapsed dimension captures the dominant mode of joint variation in activity, size, and participation.

4. **Normalization to** $[0, 1]$**.**
   After PCA, each project's raw engagement series $\{e^{\text{raw}}_{p,m}\}$ is linearly rescaled so that its minimum maps to 0 and its maximum maps to 1:

$$E_{p,m} \;=\; \frac{e^{\text{raw}}_{p,m} \;-\; \min_m e^{\text{raw}}_{p,m}}{\max_m e^{\text{raw}}_{p,m} \;-\; \min_m e^{\text{raw}}_{p,m}}.$$

   This final step guarantees $E_{p,m} \in [0, 1]$, making the engagement curve for each project span the full unit interval and thus directly comparable across projects and months. This step allows for cross-project comparison and easier visualization.

Principal-component analysis (PCA) is particularly well-suited for distilling multiple, correlated indicators of developer activity into a single "engagement" metric. By construction, PCA finds the one linear combination of the five standardized features that maximizes explained variance:

$$\mathbf{w}^\top \mathbf{x}_{p,m} \quad \text{with} \quad \|\mathbf{w}\| = 1,$$

so that months exhibiting simultaneously high churn, commit counts, developer headcount, and commit rate receive the highest scores. This procedure automatically balances redundant inputs (e.g., churn vs. commits) and ignores extreme outliers, yielding data-driven feature weights rather than hand-tuned coefficients.

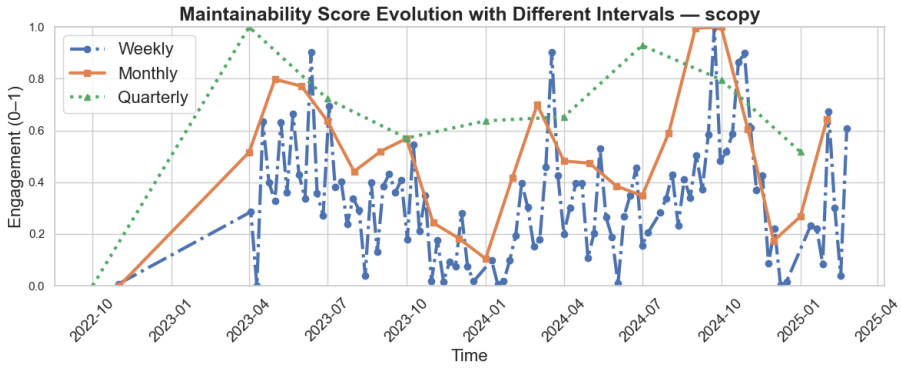


Figure 4: Comparison of engagement data with different analysis periods.

The reason for choosing monthly visualization and calculation over weekly or quarterly is to clearly see the trends and behavior of the project development. In a monthly representation, the data become too noisy. However, the values in the quarterly analysis are relatively the same. The monthly values manage to capture the fluctuation in the data while still being non-noisy. It also shows 2-3 month length trends among several projects, where it is later going to be explained.

3.3.2 Delta Maintainability Score

The DMM score per commit was first calculated as the arithmetic mean of the three unit-level ratios. There was no priority on any index, so this method was chosen. The formula

for creating this metric is as follows:

$$dmm_c = \frac{dmm\_u\_size_c + dmm\_u\_complexity_c + dmm\_u\_interfacing_c}{3} \quad \text{for each commit } c.$$

Then each commit was assigned to its calendar month. Two monthly aggregates were formed for each project p and month m:

1. **Total churn:** $churn_{p,m} = \sum_{c\,:\,\mathrm{month}(c)=m} \Delta_c$, where Δ_c denotes the lines added plus the lines removed in commit c.
2. **Weighted DMM sum:** $W_{p,m} = \sum_{c\,:\,\mathrm{month}(c)=m} \big(dmm_c \times \Delta_c\big)$.

From these two quantities, *monthly weighted DMM* was defined by

$$DMM_{p,m}^{\mathrm{month}} = \frac{W_{p,m}}{churn_{p,m}},$$

which represents the average fraction of the 'good' change (measured by DMM) per line of code modified. In addition, cumulative series were maintained to observe longer-term trends.

$$C_{p,m}^{\mathrm{churn}} = \sum_{k \le m} churn_{p,k}, \quad C_{p,m}^{\mathrm{DMM}} = \sum_{k \le m} W_{p,k},$$

and *cumulative weighted DMM* was given by

$$DMM_{p,m}^{\mathrm{cum}} = \frac{C_{p,m}^{\mathrm{DMM}}}{C_{p,m}^{\mathrm{churn}}}.$$

The monthly and cumulative weighted indices lie in the interval $[0, 1]$, with higher values meaning better maintainability in the project.

Even though the resulting score for each project cannot be considered as the absolute maintainability score, it is still representative in terms of good and bad changes in the code. Figure 5 shows an example evolution of the maintainability score.

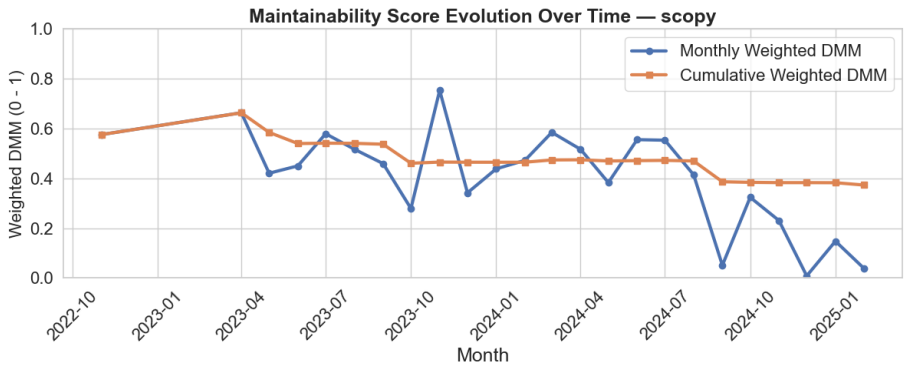


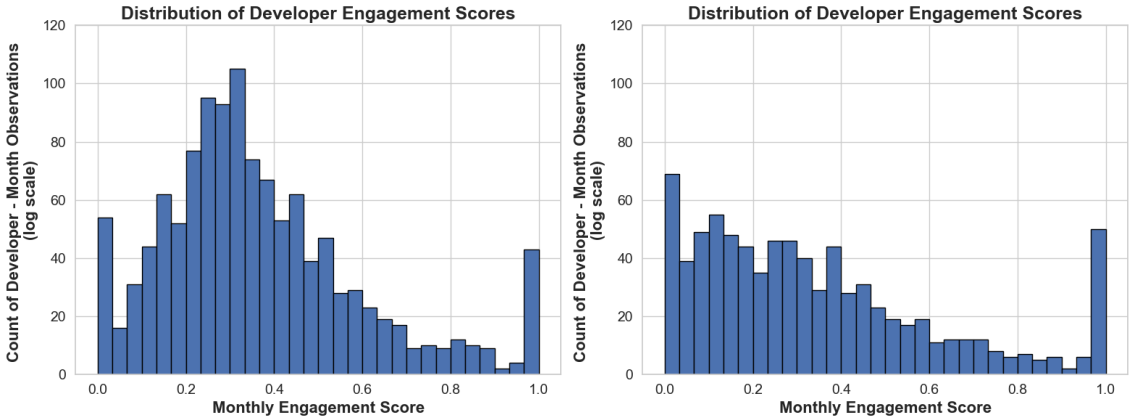
Figure 5: Monthly maintainability score of an example project changing over time.

4 Results

The following section presents the descriptive findings for the developer engagement metric in the 40 selected projects in two sets. This section does not test any hypothesis, but is purely to explain the outcome of the methods.

RQ1. How can a developer engagement metric be generated using the commit-level project features?

4.1.1 Key Observations



(a) Active Set (b) Average Set

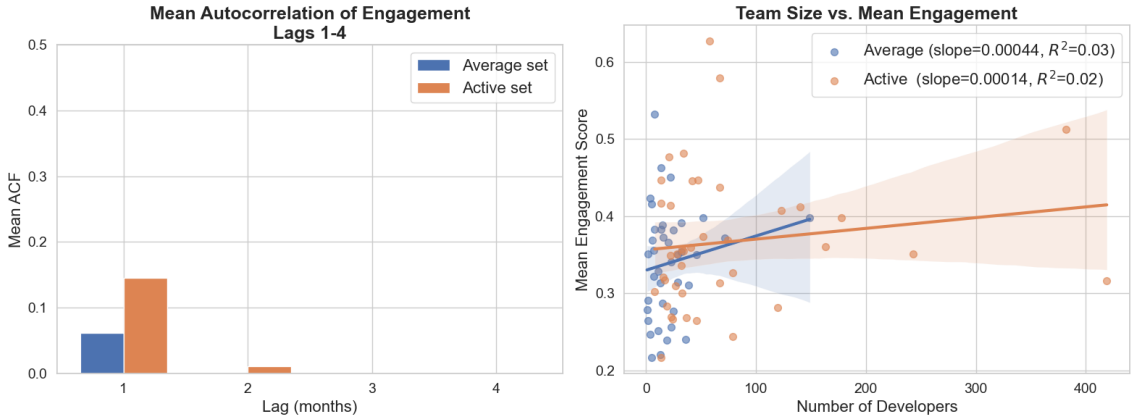
Figure 6: Histogram of the engagement values in both sets

Table 1: Descriptive summary of developer–month engagement scores

	Active Set	**Average Set**
Total observations	1195	819
Median engagement	0.32	0.28
Mean engagement	0.37	0.34
Std. deviation	0.23	0.27
Proportion near-zero months	$> 4.9\%$	$> 11.5\%$
High-intensity tail (> 0.8)	$\approx 7\%$	$\approx 9\%$

A total of 1,195 developer-month observations were obtained for the active set and 819 for the average set. Almost a third of the observations are missing for the Average Set, which means that there were no engagement at all to do the calculation for. For the Active set, the distribution median is easier to observe in Figure 6. The Average set has a more equally distributed engagement score over months, with the median value interval at zero. The higher standard deviation in the average set also supports this result in Table 1. The peaks on the leftmost and rightmost side of the graphs are caused by per-project normalization.

High-engagement months (engagement ≥ 0.5) appear as isolated spikes rather than clustered. The mean autocorrelation function (ACF) calculated in all projects for lags 1–4 demonstrates this persistence. At lag 1, the ACF is 0.14 for the Active set and 0.06 for the Average set, confirming that elevated (or suppressed) engagement in one month has a slight tendency to continue the workload. By lag 2, even though there are observations of high engagement in the active set, the value drops below 0.01 for the average set. And by lag 3 both of them are effectively zero, indicating that any momentum in developer involvement dissipates in two to three months. Consequently, although intense development periods tend to carry over briefly, the engagement series resets quickly, and sustained high activity beyond a quarter is rare.



(a) ACF Results

(b) Team Size vs Mean Engagement Score

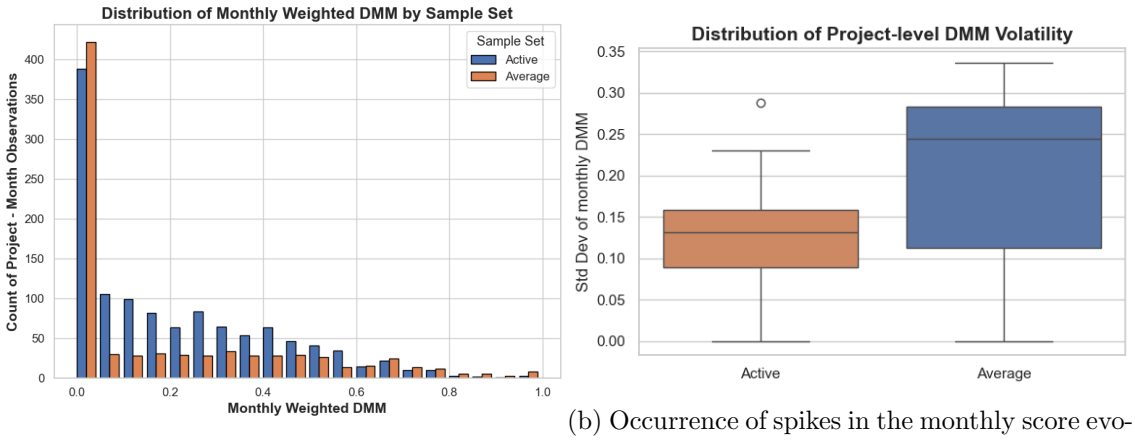
Figure 7: Per month engagement score analysis

The scatter-and-regression analysis of roster size against each project's median engagement score for each project shows essentially positive trends. In other words, in both sample sets with different activity characteristics, the larger team size results in higher mean engagement. However, the slopes of the trend lines are close to being flat. This means that even though there is a correlation, it is not too significant. Between the two sets, the average sample set with lower activity and smaller team sizes throughout the timeline has a higher correlation between the two values.

RQ2. How can maintainability be measured with DMM (Delta Maintainability Model) as a score, and how does it evolve over time?

4.2.1 Overall Maintainability Scores

The histogram of monthly weighted DMM scores shows that most of the project-month observations are concentrated at the lower end of the scale of 0 to 1, with a strong right skew. In both active and average sets, almost half of all months register DMM below 0.1, indicating that the proportion of good code changes tends to dominate from month to month. However, the active set has a wider tail: approximately 25% of its monthly score between 0.1 and 0.4, compared to approximately 15% in the average set. This suggests that projects in the active sample occasionally introduce bad changes more frequently



(a) Distribution of Monthly DMM Values

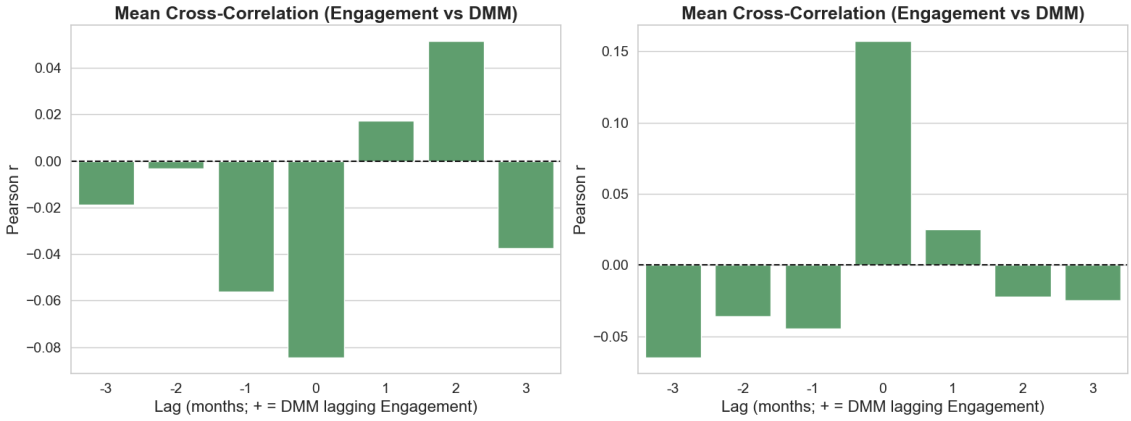
(b) Occurrence of spikes in the monthly score evolution

Figure 8: Per month maintainability score analysis

than in the broader average cohort.

The box plots of per-project standard deviation in monthly DMM reveal clear differences in volatility between the two samples. The median volatility of the DMM for the active set is near 0.13, with an inter-quartile range of approximately 0.09 to 0.16, indicating relatively stable maintenance over time. In contrast, the average set shows substantially greater variability: its median volatility is around 0.25, with the middle 50% of projects ranging from 0.12 to 0.34. The higher spread and more frequent upper-whisker outliers in the average group reflect more pronounced spikes and dips in its maintainability scores, suggesting that less active projects are subject to more extreme swings in code-change risk from month to month.

RQ3. Is there a relationship between developer engagement and Delta Maintainability Score?



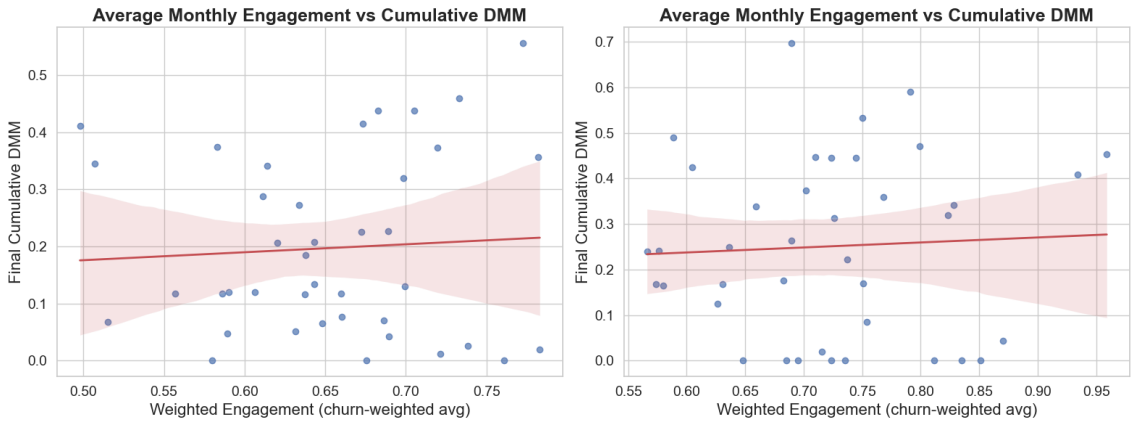
(a) Active Set

(b) Average Set

Figure 9: Lag analysis of the Engagement / Maintainability Relation

When decomposed by sample set, the month-to-month 'echo' between engagement and maintainability differs markedly in Figure 9. In active projects, the Pearson correlation for zero delay is slightly negative ($r \approx -0.08$), indicating that exceptionally busy months tend to coincide with marginally lower DMM scores in the same month. A small positive bump then appears at the lags +1 ($r \approx 0.02$) and +2 ($r \approx 0.05$), suggesting that intense developer activity may yield modest improvements in maintainability one to two months later. However, all correlations remain small ($|r| < 0.10$), and by lag 3 the relationship has effectively reversed.

In contrast, the average project set exhibits its strongest positive correlation right at lag 0 ($r \approx 0.16$), implying that for these code bases, the high-activity months are more immediately aligned with better maintainability. A weaker positive tail follows at lag +1 ($r \approx 0.03$), but thereafter the effect dissipates. In both samples, the rapid drop off beyond two months indicates only the fleeting momentum between developer engagement and the DMM-based quality metric.



(a) Active Set (b) Average Set

Figure 10: Scatterplot of the average engagement score against final cumulative maintainability score ($p \approx 0.06$)

A scatterplot of each repository's long-term engagement score against its final cumulative DMM reveals almost no trend in Figure 10. For both sets, with no obvious outliers, the slope of the maintainability score against the mean engagement of the projects is almost 0. This means that there is almost no relation between the engagement of the developers and the quality of the code they write.

5 Discussion

5.1 Findings

The bursty, short-lived pattern of developer engagement closely matches the tempo observed by Bird et al. in large-scale open-source development, where intense collaboration phases rarely last more than a few weeks [Bir+09]. In the Average sample, most of all developer-month observations register engagement scores on the lower end, almost a third of the months being completely quiet, reflecting extended periods of quiet maintenance

punctuated by brief sprints of activity. The Active sample set results in a higher overall engagement score, but still drops below 0.5 for most of the projects.

The autocorrelation for the first lag (0.14 for Active, 0.07 for Average), decaying below 0.05 by lag 2, confirms that these sprints exert only transient momentum, rarely lasting more than one or two months. Such rapid decay underlines the importance of capturing engagement at a fine temporal granularity rather than relying on coarse aggregates at the project level.

Spinellis' FreeBSD study demonstrated that neither churn nor headcount alone suffices to predict long-term quality without considering commitment frequency and team breadth [Spi06]. The composite engagement metric developed here, standardized by churn, commit count, roster size, and intervals between commits, addresses this gap by balancing intensity and participation. Its distribution also reveals an extreme right-hand tail: roughly 10% of the months exhibit engagement above 0.8, corresponding to major feature pushes or release preparations. These high-intensity bursts are rare in both sample sets, but the overall maintainability is higher in the Active one.

Maintainability, measured via the Delta Maintainability Model (DMM), also exhibits a pronounced right skew in both samples. Most months yield low DMM values, indicating that the majority of code changes tend to decrease the code quality metrics. However, there is a pattern where after a huge drop in DMM, followed by a slight bump in the score, probably caused by refactoring and fixing the issues from a bulk merge. The Active set shows lower volatility (median per project $\sigma \approx 0.13$) compared to the Average set ($\sigma \approx 0.25$), echoing the findings that infrequently updated code bases undergo more extreme episodic swings in risk [Di +19]. This contrast suggests that a steady cadence of incremental improvements can dampen the impact of large disruptive changes.

The relationship between engagement and maintainability was investigated using cross-correlation and per-project regressions. In the Active sample, high engagement months slightly coincide with poorer maintainability of the same month ($r \approx -0.09$), with a marginal positive effect emerging one month later before disappearing due to lag 2. In contrast, the Average sample shows a modest positive alignment of the same month ($r \approx 0.17$) but a similarly rapid decay. The per-project analysis of these two metrics also shows a relation with a low value (≈ 0.06) which shows that there is almost no project-wide relation of engagement and code quality.

These weak and fleeting associations reinforce the emphasis of the SPACE framework on capturing activity rhythms rather than inferring quality from static engagement aggregates [For+21]. They also imply that other factors, such as code review practices, automated testing coverage, and architectural stability, likely play a decisive role in determining whether bursts of development lead to maintainability gains or regressions.

5.2 Threats to Validity

While the findings provide a valuable starting point for understanding the relationship between developer engagement and maintainability, there are several limitations worth noting.

One notable limitation is the instability of developer identifiers, particularly changes in email address over time. The reason is that the way that the current implementation calculates engagement in project is based on identifying individual developers, and it is

done by the email addresses. Any possible change in a persons email causes it to be counted as a different developer. This issue complicates the computation of consistent developer metrics and may understate the actual activity of long-term contributors. In cases where the only difference is that the letters are capital or lower, or there is only a domain difference between two emails, it can be solved relatively easily. However, in situations where the email changes completely, there is not an easy way to understand and fix this issue.

One example to this identifier problem can be from the ConnectSDK/Connect-SDK-Android-Core repository, where there are developers with identifiers jonghen.han@lge.com and jonghen.han@jonghenhanui-MacbookAir.local. These two emails are easy to detect as humans, and it can also be split into a name and domain part to be considered as the same developer by the code as easily. However, in a case like the one in Microsoft/Vipr repository, where two identifiers are c.bales@outlook.com and caitbal@microsoft.com, these may or may not be different persons. There is no easy way to identify these two developers based only on emails.

Other limitations considering the data is the limited sample size (n = 40). The dependence on a single engagement metric and the fact that the dataset used was last updated in 2020, leading to a possible mismatch between historical and current project behaviors. This was tried to be avoided in the data collection part in this project. But during the sampling where the outdated data is used, this issue persists. The 90% of the projects from the original data set are no longer maintained. These may be caused by several reasons but the list needs to be updated.

Another possible limitation is caused by the missing DMM indices in commits. The DMM calculation has requirements for the changes to be calculated. If they are not provided, or the changes are in a different language where PyDriller can not do this procedure on, the values are missing. In the commit data, the missing values consist of 70% for Active and 75% for Average sets.

The final and most important limitation was the time and resources needed to collect the data. In cases where only surface-level metrics are collected, such as commit count, number of developers, the requirement for resource is not a major issue. But in cases where DMM is calculated per commit, or a further calculation is done on the commit object, such as churn, the active projects take tens of hours each to process completely. This is why an upper limit was used for the activity, and only a certain time period was used during this project. With parallel mining and faster and better processors, this issue can be minimized. The original set mentioned in this thesis can be updated using the same methods that they used. Since they did it before the worldwide pandemic COVID-19, it would be a great way to see how open source enterprise projects changed their characteristics during that time period.

6 Conclusion

A per-month engagement index and a churn-weighted maintainability score were implemented and evaluated across diverse GitHub projects. Engagement was found to be bursty and short-lived, while maintainability improvements appeared in a right-skewed pattern with occasional deep dips. The two measures exhibited only weak, transient correlations,

indicating that spikes in developer activity do not reliably translate into immediate or lasting quality gains.

Several practical implications arise. First, engagement metrics should be interpreted alongside quality indicators rather than in isolation. Second, the maintainability measurement needs improvement. The current state misses most of the data and the cumulative calculation using DMM is not the best practice in general. If the final code is submitted as a single commit, the DMM indices of that commit would not necessarily have to relate to the outcome of this calculation method.

Finally, the focus of researchers and practitioners should expand to include code review dynamics, automated testing coverage, and issue tracking activity to form a more comprehensive view of software health.

The resulting commit data, the generated graphs and the code that was used in this project were submitted to a public library on GitHub to be delivered to the SIG [Uck25]. The final version of the scripts can be considered a single feature benchmark on their own. However, it is best to use it as a support feature or model alongside an existing system. The same procedures can be used for the entire data set provided by Spinellis with more resources. It can also be used on software projects of SIG's customers. And if the findings are aligned with SIG's dashboard outcome, it can even help them develop further features.

References

[BET24] Markus Borg, Marwa Ezzouhri, and Adam Tornhill. "Ghost echoes revealed: Benchmarking maintainability metrics and machine learning predictions against human assessments". In: *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Flagstaff, AZ, USA: IEEE, Oct. 2024, pp. 678–688. URL: https://doi.org/10.1109/ICSME58944.2024.00072.

[Bir+09] Christian Bird et al. "Does distributed development affect software quality?" en. In: *Commun. ACM* 52.8 (Aug. 2009), pp. 85–93. URL: https://doi.org/10.1109/ICSE.2009.5070550.

[CM78] Joseph P Cavano and James A McCall. "A framework for the measurement of software quality". In: *Proceedings of the software quality assurance workshop on Functional and performance issues*. 1978, pp. 133–139.

[Di +19] Marco Di Biase et al. "The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). Montreal, QC, Canada: IEEE, May 2019, pp. 113–122. ISBN: 978-1-7281-3371-3. URL: https://doi.org/10.1109/TechDebt.2019.00030.

[For+21] Nicole Forsgren et al. "The SPACE of Developer Productivity: There's More to It than You Think." In: *Queue* 19.1 (Feb. 2021), pp. 20–48. ISSN: 1542-7730, 1542-7749. URL: https://doi.org/10.1145/3454122.3454124.

[GKS08] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. "Measuring developer contribution from software repository data". en. In: *Proceedings of the 2008 international working conference on Mining software repositories*. Leipzig Germany: ACM, May 2008, pp. 129–132. ISBN: 978-1-60558-024-1. URL: https://dl.acm.org/doi/10.1145/1370750.1370781.

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability". In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. Sept. 2007, pp. 30–39. URL: https://doi.org/10.1109/QUATIC.2007.8.

[KA23] Shahbaa I Khaleel and Ghassan Khaleel Al-Khatouni. "A literature review for measuring maintainability of code clone". In: *Indones. J. Electr. Eng. Comput. Sci.* 31.2 (Aug. 2023), p. 1118. URL: https://doi.org/10.11591/ijeecs.v31.i2.pp1118-1127.

[MC16] Ruchika Malhotra and Anuradha Chug. "Software Maintainability: Systematic Literature Review and Current Trends". en. In: *International Journal of Software Engineering and Knowledge Engineering* 26.08 (Oct. 2016), pp. 1221–1253. URL: https://doi.org/10.1142/S0218194016500431.

[MM20] Arthur-Jozsef Molnar and Simona Motogna. "A study of maintainability in evolving open-source software". In: (2020). eprint: 2009.00959 (cs.SE). URL: https://doi.org/10.48550/arXiv.2009.00959.

[Rom+23] Anthony Cintron Roman et al. “Open data on GitHub: Unlocking the potential of AI”. In: (2023). eprint: 2306.06191 (cs.LG). URL: https://doi.org/10.48550/arXiv.2306.06191.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. “PyDriller: Python Framework for Mining Software Repositories”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, Oct. 26, 2018, pp. 908–911. ISBN: 978-1-4503-5573-5. URL: https://dl.acm.org/doi/10.1145/3236024.3264598.

[Spi+20] Diomidis Spinellis et al. “A dataset of enterprise-driven open source software”. In: *arXiv [cs.SE]* (2020). URL: https://doi.org/10.48550/arXiv.2002.03927.

[Spi06] Diomidis Spinellis. “Global software development in the freeBSD project”. In: *Proceedings of the 2006 international workshop on Global software development for the practitioner*. Shanghai China: ACM, May 2006. URL: https://doi.url/10.1145/1138506.1138524.

[Uck25] Ahmet Yasir Uckun. *Benchmarking Software Maintainability in Enterprise-Driven Open Source Projects in Terms of Developer Engagement*. Version 1.0.0. July 2025. URL: https://github.com/ayuckun/developer-engagement.