ASSESSING SOFTWARE EVOLUTION WITH THE STICKINESS SCORE: EVALUATING CODE PERSISTENCE ACROSS FILES, FOLDERS, AND DEVELOPERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Selen UYSAL
April 2025

Assessing Software Evolution with the Stickiness Score: Evaluating Code Persistence Across Files, Folders, and Developers
By Selen UYSAL
April 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray TÜZÜN (Advisor)

Uğur DOĞRUSÖZ

İsmail Sengör ALTINGÖVDE

Approved for the Graduate School of Engineering and Science:

Orhan ARIKAN
Director of the Graduate School

ABSTRACT

ASSESSING SOFTWARE EVOLUTION WITH THE STICKINESS SCORE: EVALUATING CODE PERSISTENCE ACROSS FILES, FOLDERS, AND DEVELOPERS

Selen UYSAL
M.S. in Computer Engineering
Advisor: Eray TÜZÜN
April 2025

Software evolution involves continuous code changes, making it essential to understand factors that influence code stability and persistence. This study introduces a metric called the "Stickiness Score," measuring the longevity of lines of code (LOC) within a project. It reflects how much of the LOC written by developers or belonging to a specific file or folder has persisted over time. The goal is to examine its correlation with various software metrics: contributor count, developer Stickiness Scores (average, commit-weighted average, and LOC-weighted average), cyclomatic complexity, bug-fix count, and static code analysis metrics, including bug and code smell counts. Stickiness Scores for developers, files, and folders are calculated using the tool developed for this study, Devotion, across five open-source projects. Spearman correlation tests were used to analyze the relationship between file Stickiness Scores and the specified software metrics. Contributor count exhibited a strong negative correlation with file Stickiness Scores. Commit- and LOC-weighted developer Stickiness Scores showed positive correlations, while unweighted averages produced mixed results. Cyclomatic complexity, bug-fix count, and code smell count showed inconsistent correlations. The bug counts in files showed no significant correlation. In conclusion, files with more contributors or frequent bug-related changes tend to be less sticky. In contrast, files modified by high-commit or high-volume contributions from developers with higher stickiness tend to persist longer. The Stickiness Score provides valuable insight into how contributor activity, code complexity, bugginess, and smells relate to code longevity.

Keywords: Software Development, Code Stickiness, Code Survival, Code Churn, Correlation Analysis, Survival Analysis.

ÖZET

YAZILIM EVRİMİNİN YAPIŞKANLIK SKORUYLA DEĞERLENDİRİLMESİ: DOSYALAR, KLASÖRLER VE YAZILIMCILAR ARASINDA KOD KALICILIĞININ İNCELENMESİ

Selen UYSAL
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray TÜZÜN
Nisan 2025

Yazılım evrimi, sürekli kod değişikliklerini içerir ve bu durum, kodun kararlılığını ve kalıcılığını etkileyen faktörlerin anlaşılmasını önemli kılar. Bu çalışma, bir projedeki kod satırlarının ne kadar süre boyunca varlığını sürdürdüğünü ölçen "Yapışkanlık Skoru" adlı bir metriği tanıtmaktadır. Bu skor, yazılımcılar tarafından yazılan ya da belirli bir dosya veya klasöre ait kodların zaman içinde ne ölçüde korunduğunu yansıtır. Çalışmanın amacı, bu skorun çeşitli yazılım metrikleriyle olan korelasyonunu incelemektir: dosyaya katkıda bulunan yazılımcı sayısı, yazılımcı yapışkanlık skorları (ortalama, commit-ağırlıklı ve LOC-ağırlıklı olarak), döngüsel karmaşıklık, hata düzeltme sayısı ve statik kod analizi metrikleri olan hata sayısı ve kod kokusu sayısı. Yazılımcı, dosya ve klasör düzeyindeki yapışkanlık skorları, bu çalışma kapsamında geliştirilen Devotion adlı araç kullanılarak beş açık kaynaklı proje üzerinde hesaplanmıştır. Dosya yapışkanlık skorları ile belirtilen yazılım metrikleri arasındaki ilişkiyi analiz etmek için Spearman korelasyon testi uygulanmıştır. Analiz sonucunda, dosyaya katkıda bulunan yazılımcı sayısının dosya yapışkanlık skorlarıyla güçlü bir negatif korelasyon gösterdiği bulunmuştur. Commit- ve LOC-ağırlıklı yazılımcı yapışkanlık skorları pozitif korelasyon göstermiştir; öte yandan, ağırlıksız yazılımcı yapışkanlık skoru ortalamaları ile ilişkisi karışık sonuçlar vermiştir. Döngüsel karmaşıklık, hata düzeltme sayısı ve kod kokusu sayısı gibi metriklerin korelasyonları tutarsız bulunmuştur. Dosyadaki hata sayıları ise anlamlı bir korelasyon göstermemiştir. Sonuç olarak, katkıda bulunan yazılımcı sayısının fazla olduğu ya da sık sık hata sebebiyle düzeltmeler yapılan dosyaların kodları genellikle daha az yapışkandır. Buna karşın, kod kalıcılığı yüksek olan yazılımcılar tarafından yüksek sayıda

commit ya da kod hacmi ile düzenlenen dosyaların kodları, daha uzun süre varlıklarını sürdürme eğilimindedir. Yapışkanlık Skoru, yazılımcıların katkıları, kod karmaşıklığı, hata durumu ve kod kokusu gibi faktörlerin kodun ömrüyle olan ilişkisini anlamada değerli içgörüler sunar.

Anahtar sözcükler: Yazılım Geliştirme, Kod Yapışkanlığı, Kod Ömrü, Kod Değişim Oranı, Korelasyon Analizi, Sağkalım Analizi.

Acknowledgement

I want to start by thanking my advisor, Eray TÜZÜN, for all his support, guidance, and encouragement during my studies. His advice has been crucial to helping me grow academically and professionally. I also want to thank Işıl ÖZGÜ for working with me as a co-author on the paper version of my thesis. Her contributions have been nothing short of transformative, profoundly enhancing the depth and impact of this work.

I am thankful to BILSEN members for their feedback and ideas, which were very helpful during my research.

I am also very grateful to my jury members, Prof. Dr. Uğur DOĞRUSÖZ and Prof. Dr. İsmail Sengör ALTINGÖVDE, for taking the time to review my thesis and giving me their feedback and insights.

Lastly, I owe a big thanks to my family and friends, whose love, constant encouragement, and support kept me motivated throughout this journey.

Selen UYSAL

April 2025, Ankara

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software projects continuously evolve as developers iteratively refine, expand, and enhance their codebases to adapt to shifting requirements and tackle emerging challenges [1], [2], [3].

This evolutionary process encompasses various improvements, such as adding features to address stakeholders' changing needs, fixing bugs to ensure software stability, optimizing performance for enhanced user experiences, and implementing architectural enhancements to boost maintainability, and code re-usability [4], [5], [6], [3]. Additionally, integrating newly emerged technologies often brings opportunities for improved performance and easier maintenance [6], [5], such as updating libraries or adopting microservices. These factors collectively contribute to the dynamic nature of software development, where code bases are constantly in flux to meet evolving demands and leverage the latest advancements [4], [6], [3].

Code metrics can help capture the factors driving software changes, the underlying reasons for software evolution, and the extent of these changes [6], [5], [7]. There have also been efforts to extract this evolutionary information from software repositories using various software metrics [6], [5], [7]. Two prominent categories of code metrics are code turnover and code persistence metrics, which

represent complementary opposites. Code turnover metrics, such as code churn and code volatility, capture the extent of change in software projects, while code persistence metrics, including code survival and code stability, focus on how persistent the source code is or how little it changes over time.

1.1 Research Problem

While these metrics provide a solid foundation for understanding software evolution, there remain unexplored areas that require further investigation to fully capture the factors influencing code longevity and persistence [8], [6], [5], [4]. In prior studies, software evolution is evaluated in terms of complexity [8], [9], lines of code (LOC) [5], [7], [10], repository activities, and project size [3], [6]. These metrics serve as key indicators of the evolution process, capturing the growth, maintenance, and refactoring phases of a software system. However, to the best of our knowledge, at the time of writing this paper, there is a lack of empirical study that systematically investigates the possible correlation between the persistence of source code in files and other software metrics, such as developers' tendency to write persistent code, the cyclomatic complexity of the file, and code quality in terms of bugs and code smells. Moreover, existing research does not fully address how current metrics capture the nuances of code persistence, such as accounting for code movements, duplications, and revivals (e.g., re-writing previously deleted code) while determining code persistence or churn. By building upon the existing metrics and studies, we try to capture these additional cases and explore how various types of code modifications can influence code persistence and the longevity of source code in files. This line of inquiry may provide insights for improving development practices.

To explore the correlation between code persistence and other software metrics on an extended scope, we introduce a code persistence metric called *Stickiness Score*, which can be calculated in various contexts, such as developers, files, and folders. Moreover, it captures different types of code modifications such as code movements, duplication, and revival. It can be formulated as the ratio of net code

persistence, calculated as the difference between additions and deletions, to the total number of additions. Moreover, we introduce our tool *Devotion* to calculate the Stickiness Score in the aforementioned granularities. We believe the *Stickiness Score* can serve as an indicator of problematic areas in source code. For example, files with a higher Stickiness Score may represent code that is rarely modified, which could correlate with neglect or the accumulation of undetected issues such as bugs or code smells. Conversely, files or developers with a lower Stickiness Score may reflect frequent code overwriting, potentially associated with unstable development practices, and may be associated with more volatile parts of the codebase, where bugs or code smells are more commonly observed. Our correlation analysis investigates how the Stickiness Score relates to various software metrics to better understand its relationship with software quality. To demonstrate our proposed metric, we performed analyses on five open-source software (OSS) projects to derive correlations between software metrics and code stickiness. In particular, we address the following research questions:

- **RQ1:** Is the number of contributors to the file correlated with the Stickiness Score of the file?
- **RQ2:** Is the Stickiness Score of the contributors correlated with the Stickiness Score of the file?

  RQ2.1: Is the average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file?

  RQ2.2: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of commits contributed to the file?

  RQ2.3: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of lines of code contributed to the file?
- **RQ3:** Is the file's cyclomatic complexity correlated with its Stickiness Score?

- **RQ4:** Is the number of bug-fixing pull requests that modify a file correlated with its Stickiness Score?
- **RQ5:** Is the file's number of issues collected from static code analysis correlated with its Stickiness Score?

  RQ5.1: Is the file's number of bugs collected from static code analysis correlated with its Stickiness Score?

  RQ5.2: Is the file's number of code smells collected from static code analysis correlated with its Stickiness Score?

1.2 Contributions of the Thesis

The main contributions of this thesis are as follows:

- A novel metric, Stickiness Score, is proposed to measure code longevity within a project. This metric evaluates the stability and persistence of code at the granularity of files and folders, as well as the stability of the code authored by individual developers.
- A software tool, Devotion, is developed to compute Stickiness Scores for files, folders, and developers, providing actionable insights into code evolution and developer contributions.
- A comprehensive correlation analysis is conducted to explore the relationship between various software metrics in the project and the files' respective Stickiness Scores.

Thesis Organization. The rest of the paper is organized as follows: Chapter 2 reviews related works, providing an overview of the existing literature and research relevant to this study. Chapter 3 focuses on the tool, Devotion, detailing the processes of data fetching, preprocessing, and the calculation of Stickiness Scores at both developer and file/folder levels, along with its resulting outputs.

Chapter 4 provides the theoretical background for correlation analysis, detailing various correlation tests and a range of normality tests used to inform that choice. Chapter 5 describes the methodology employed in the study, including a practical example illustrating the overall approach, the data preprocessing, and hypothesis testing steps aligned with the research questions. Chapter 6 presents the results of the study, including the criteria for project selection and the outcomes of hypothesis testing conducted for the research questions. Chapter 7 explores the correlations between software metrics and Stickiness Scores on the projects. Chapter 8 discusses potential threats to the study's validity. Finally, Chapter 9 summarizes the findings and conclusions.

Chapter 2

Related Work

In literature, efforts to quantify code change can be categorized into two complementary topics: one focusing on code turnover (i.e., quantifying the number of modifications) [11], [12], [13], [10], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] and the other emphasizing the persistence of code [24], [25], [26], [27], [28], [29], [30], [27], [31], [32], [33] .

Furthermore, these categories contain metrics that have similar objectives but are named differently, for example, code churn and code volatility for turnover metrics and code stability and survival for persistence metrics.

The summarized version of the comparison between our work and the related works can be found in Table 2.1.

2.1 Code Turnover Metrics and Their Utility

Code churn, in the most general sense, is the sum of the number of added and deleted lines of code [34]. The simple nature of code churn is extended by many studies to capture various aspects of the code changes, such as cumulating the code churn [35], normalizing it [15], or relating it to other aspects such as software

faults [15], [14].

Indeed, both churn and volatility metrics are similar regarding their objectives, while their computation and explicit definition can differ slightly depending on the context. Nonetheless, they can provide valuable insights into the patterns and extent of software evolution, such as identifying areas with frequent or significant changes that may require closer attention.

2.1.1 Code Churn

In this section, the studies that used code churn metrics for their studies are investigated. These studies' aim of using code churn is bug/fault prediction, code vulnerability prediction, effort estimation, and organizational efforts.

Giger et al. [11] explored bug prediction using fine-grained source code changes (SCC) and lines modified (LM). While LM showed importance in predicting bugs, SCC, which categorizes changes such as method updates or added if-statements, was more strongly correlated with bugs. Karus et al. [35] analyzed object-oriented, XML, and organizational metrics to predict yearly cumulative code churn in software projects, concluding that XML and organizational metrics are significant predictors.

Kocaguneli et al. [12] investigated software effort estimation in six projects using static code metrics and code churn. They found that 20% of developers cause 90% of code churn, highlighting inefficiencies in resource allocation.

In addition to prediction, code churn is also used to detect issues in source code. Shin et al. [13] used code churn, complexity, and developer metrics to predict vulnerable code locations. They found that code churn predicts vulnerable code with 59% accuracy and identified that files worked on by highly distributed developers are more prone to vulnerabilities, while files owned by central developers are less likely to be vulnerable.

In another perspective, Meneely et al. [10] studied the human factors of churn, introducing metrics such as self-churn (developers modifying their own code) and interactive churn (developers modifying others' code). They found that interactive churn correlates with post-release vulnerabilities but not strongly with churn or lines of code. Another study [14] on Apache HTTP Server linked vulnerability-contributing commits with higher churn.

Another study that utilizes code churn for detecting abnormalities is from Nagappan and Ball [15]. They introduced normalized code churn metrics (relative churn), including total churn divided by total lines, and applied them to Windows Server 2003 SP1. Their metrics detected fault-prone binaries with 89% accuracy, outperforming absolute churn metrics. Unlike their focus on defect density, our proposed metric considers unchanged lines relative to lines added by developers.

Apart from using code churn on line granularity, Faragó et al. [16] examined cumulative code churn on file granularity. Using maintainability scores from SVN and ColombusQM, they found that lower cumulative churn improves maintainability.

In addition to academic studies regarding code survival, there are tools and discussions regarding the usage of code churn for gaining insights into software projects. For example, Code Maat [17] used churn metrics to analyze developer, file, and project-level changes, including age analysis. Nick Gerner [18] investigated the link between code churn and reviews, finding that reviewed code is 3.5 times more likely to survive a year, with OSS projects averaging 4% churn per month.

2.1.2 Code Volatility

Code volatility is defined as the proneness to change or the total number of changes occurring [23]. It can be used on different granularities such as modules [20], files [21], or LOC [19].

Lopez and van der Hoek [19] developed Code Orb, an Eclipse plug-in that warns developers about potential volatility in code lines. Using commit logs and diffs, it measures volatility based on three dimensions: code churn, past bugs, and the number of authors. The tool calculates a normalized probability within the class to assign a warning level to each line.

Braunschweig et al. [20] expanded volatility to software modules, identifying factors influencing higher volatility through OSS developer interviews. Historical factors include developer interest, changes to untouched parts (sometimes driven by fun or errors), and the cycle of changes triggered by newcomer developers. They noted that modules dependent on volatile modules or during volatile periods are more prone to change.

Bugayenko [21] analyzed 240 Java OSS projects and proposed a volatility metric measuring the relationship between actively modified files and untouched files. Higher volatility values correlate with a larger distance between these file groups, implying potential maintenance anomalies. However, no evidence proves that high volatility directly causes anomalies.

Benestad et al. [22] investigated change request volatility using two software systems and developer interviews. High volatility in change requests indicates uncertainty or inadequacy, leading to high costs. Conversely, code volatility reduces effort in volatile components due to developer familiarity. User interface volatility caused by feedback had a minimal impact on effort.

Costello et al. [23] used a broader volatility metric, addressing both software-level (requirements, design, code, build content) and system-level aspects (system/segment, integrated CI, hardware CI). Requirements volatility was emphasized as critical for predicting design and code volatility, linking requirement changes to consistency across development phases.

2.2 Code Persistence Metrics and Their Utility

Code stability and code survival are used as indicators of software that can persist change. Code survival indicates the number of operations, such as additions, deletions, and modifications that were performed by developers and were never changed by anyone [24]. Code survival can be perceived in the context of developers [24] or LOC [25], [26], [29] or both [27], [28].

Similarly, code stability can be defined as the capacity of code to remain unchanged over time [36], which can then be used as a measure of software product reliability. Code stability can be investigated for modules [32], [31], or be used on LOC granularity [32], while some studies incorporate cyclomatic complexity for its interpretation [33].

Similar to the code turnover metrics previously presented, the calculation and definition of these metrics can vary based on the context of the studies. Still, code persistence metrics can offer valuable insights into the long-term stability of software, helping to identify areas that remain largely unchanged over time or those that consistently show stability, which may warrant closer attention.

2.2.1 Code Survival

This section focuses on studies related to code survival in the context of developers, LOC, as well as the temporal changes in these attributes.

Code survival measures the number of developer operations that remain unaltered over time [24] and can be studied at the developer [24] or LOC level [25], [26], [29], or both [27], [28].

Moura et al. [24] analyzed developers' additions, deletions, and modifications at both line and file levels. The code survival metric for modifications was calculated as the number of lines classified as modifications that were previously touched by another developer. Developer effort was measured by the number

of modifications made. Metrics with higher values identified key developers, as confirmed by project managers.

Spinellis et al. [25] introduced the concept of code survival as the "lifetime" of LOC and tokens. The "birth" of code is its creation, and its "death" is when it gets replaced or moved. Lifetime is the duration between birth and death. They analyzed 89 OSS projects (3.3 billion LOC) and found the median LOC lifespan to be 2.4 years, with no significant correlation between lifetime and developer characteristics. In earlier work [26], they applied similar methods to the Unix source code, calculating a half-life for LOC between 0.8 and 9.7 years. They used `git log` and `git blame` to track authorship, changes, and code duplication.

Non-academic tools also explore code survival. Git-of-Theseus [27] visualizes code survival through stack plots, highlighting major contributors but becoming cluttered with increasing complexity. Hercules [28], a GitHub Actions tool, tracks file-developer coupling and identifies co-modified files, though it suffers from outdated dependencies. In a separate study, Stainberg [29] analyzed Curl's [30] source code survival using `git log` and `git blame`, finding that 75% of LOCs added in the last five years survived, while only 18% persisted throughout the project's lifetime.

Git-of-Theseus [27] is a GitHub project that visualizes code survival over time using stack plots that highlight key contributors. While effective in showcasing significant contributors and tracking their involvement, the clarity of visualizations decreases as the number of contributors and repository complexity grow.

Hercules [28] is a tool within GitHub Actions that analyzes temporal changes in code ownership and file-developer coupling. It identifies files frequently modified together and developers who often work on the same files. However, it is no longer actively maintained, and outdated dependencies make the setup challenging.

In addition to these tools for visualizing code survival, there are also studies that do not involve explicit tool development. For instance, Stainberg [29] conducted a temporal analysis of Curl's [30] source code survival using `git log` and

`git blame`. The analysis showed that 75% of LOCs added in the last five years survived, while only 18% survived across the entire project timeline.

2.2.2 Code Stability

This section presents studies focused on code stability. The studies mainly focus on the stability of source code lines, while some interpret it in the context of cyclomatic complexity.

One study that interprets stability in the source code line levels is Staron et. al's [31]. They proposed a method to measure and visualize code stability using heat maps. Code stability was calculated as the sum of added, deleted, and changed lines per module weekly. Interviews with three companies revealed that stability measures, such as code churn and change burst, help identify risky components in software, aiding in product quality improvement.

Kelly [32] analyzed stability using four metrics: LOC per Module, Variable Names per Common Block, Fanout per Module, and Modules Referencing each Common Block. Connectivity measures, such as Modules Referencing each Common Block, were the strongest indicators of stability, while LOC per Module was the least related.

Yau and Collofello [33] defined stability as a program's resistance to ripple effects caused by modifications. They introduced the logical stability metric, incorporating cyclomatic complexity to assess module stability. Higher cyclomatic complexity indicated lower stability and increased maintenance effort due to ripple effects.

Table 2.1: Comparison of Related Work and Our Study

Study	Metrics Used					Cases Considered			Scope			Analysis
	LOC	Cyc Compl.	Bug Count	Code Smells	Other	Move	Copy	Revival	Dev.	File	Project	
Giger et al. [11]	✓	×	×	×	AST changes	×	×	×	×	✓	×	LM (code churn) and SCC
Kárus et al. [35]	✓	×	×	×		×	×	×	✓	×	✓	Predicting yearly churn
Kovagunedi et al. [12]	✓	×	×	×		×	×	×	✓	×	×	Effort estimation with churn
Shin et al. [13]	✓	✓	×	×	Developer activity	×	×	×	✓	✓	✓	Vulnerability detection
Meenely et al. [10]	✓	×	×	×	Security vulnerabilities, Churn varieties	×	×	×	✓	✓	×	Vulnerability prediction
Meenely et al. [14]	✓	×	×	×	Interactive churn	×	×	×	✓	×	×	VCC and churn varieties
Nagappan and Ball [15]	✓	×	×	×	Relative churn	×	×	×	×	✓	×	Relative churn metrics and faults in binaries
Paragó et al. [16]	✓	×	×	×	Cumulative churn	×	×	×	×	✓	✓	Churn and maintainability relation for each commit
Code Maat [17]	✓	×	×	×	Absolute churn, churn by entities	×	×	×	✓	✓	✓	Analyses of the project per metric
Nick Gerner [18]	✓	×	×	×	Pull request churn	×	×	×	×	✓	✓	Code review's effect on churn
Lopez and van der Hoek [19]	✓	×	✓	×	Author number, bug count, ownership	×	×	×	×	×	×	Bugginess and ownership influencing volatility of each line
Braunschweig et. al [20]	×	×	×	×	Developer interest, intention	×	×	×	✓	✓	✓	Developer and module characteristics affecting volatility
Bugayenko [21]	×	×	×	×	Consecutive file changes	×	×	×	×	✓	✓	Number of files and file volatility
Benestad et. al [22]	✓	✓	×	×	Change effort	×	×	×	×	×	✓	Change effect and change request volatility
Costello et. al [23]	×	×	×	×	Code, design, requirement volatility	×	×	×	×	×	✓	Requirement volatility
Moura et al. [24]	✓	×	×	×	Effort	×	×	×	✓	✓	×	Metrics to identify developer similarity
Spinellis et al. [25]	×	×	×	×	Code/token lifespan, Developer metrics	×	×	×	✓	×	✓	Correlation between lifespan and other metrics
Spinellis [26]	×	×	×	×	Code lifespan	✓	✓	×	×	×	✓	Lifespan of Unix
Git-of-Theseus [27]	✓	×	×	×	Commit churn	×	×	×	✓	×	✓	Temporal changes
Hercules [28]	✓	×	×	×	Overwrite couples, Efforts	×	×	×	✓	✓	✓	Temporal changes
Stainberg [29]	✓	×	×	×		×	×	×	×	×	✓	Code survival with respect to time
Staron et. al [31]	✓	×	×	×		×	×	×	×	×	×	Temporal changes to the stability of components
Kelly [32]	✓	×	×	×	Fanout and variable names		✓	×	×	×	×	Module characteristics' effect on stability
Yan and Colfolello [33]	×	×	×	×	Normalized logical stability	×	×	✓	✓	✓	✓	Module stability after each change
Our Study	✓	✓	✓ (Both Bug-Fix and Bug Count)	✓		✓	✓	✓	✓	✓	✓	Stickiness Score analysis

Chapter 3

Devotion

We introduce our software tool, Devotion, to gather Stickiness Scores for each developer, file, and folder. The tool uses `git log` [37] and `git blame` [38] commands to traverse each commit in the main branch. We chose these Git commands because they can capture the evolution of code changes through consecutive diffs and identify the initial authors of each LOC, which ensures accurate tracking of code persistence. Moreover, these commands are commonly used for tracking author contributions and code evolution [14], [10], [26]. While these methods can be computationally expensive [25], we prioritized their flexibility, particularly the ability to use `git blame` flags such as `-M -C` to identify code duplication and movement, which enhances the precision of our analysis. Without these flags, the `git blame` command identifies the author who was last to modify the line to bring it to its current state. However, by providing the `-M` (move) and `-C` (copy) flags, we force `git blame` to iterate over the commit history and identify the initial author. The difference between the initial author and the last author to touch a line is that the latter may have only reformatted or reverted the line, whereas the initial author is the one who originally introduced the line content.

The workflow of its methodology is given in Figure 3.1. By traversing each commit in the version control history, Devotion aims to determine the author's written LOCs throughout the history of the project and whether the LOCs written

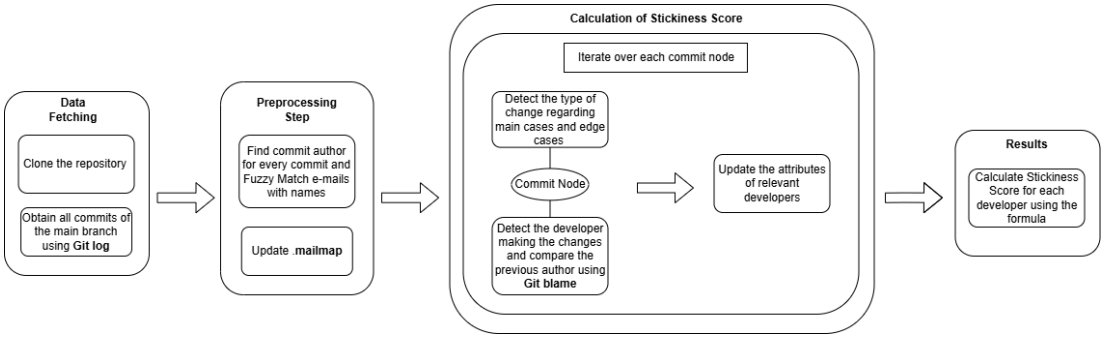


Figure 3.1: Workflow of Devotion

by the developer have survived. It does not simply identify the commit author as the author of the LOC, rather, it traverses the history to find if the LOC actually was written by another developer in possibly another file. Therefore, it can identify if the LOC was copied from another developer's code or if the commit author moved the LOC written by another developer to the current place in the project. If the LOC was copied, moved, or revived, then the initial developer is credited for that LOC.

Since Devotion can correctly find the initial developer of the LOCs in the project, this tool is used to gather the Stickiness Scores of the developers. The tool itself also calculates the Stickiness Scores with the granularity level of the file and folder, providing a detailed view of where the sticky code resides within the project.

3.1 Data Fetching

Devotion clones the repository to be analyzed and uses the `git log` [37] to fetch all the commits on the main branch. Commits from branches merged into the main branch are fetched in topological order. In this context, topological order ensures that no parent commit is shown before all its children have been displayed. This ensures the detection of edge cases, such as code movements, duplications, and revivals to be consistent.

3.2 Data Preprocessing

Devotion uses the data in the `.git` directory and assigns an author to each LOC using `git blame` command [38]. In the case of developers committing with different mail addresses or under different commit author names, Devotion fuzzy-matches the developer's name with either the local part of their committing e-mail or the name of the committing author and connects them in the mailmap file. The fuzzy matches are done with the help of the `FuzzyWuzzy` library [39]. Devotion cross-checks these handles using fuzzy matching and selects the maximum similarity ratio. The current threshold for a successful match is set at 90%, based on the overlap of common alphanumerical characters in the compared strings. This value was calibrated using the five projects analyzed in this study and represents the highest threshold that consistently identified and merged developer identities correctly, as confirmed through manual validation. It is important to note that this threshold may require adjustment when applying the tool to other repositories with different naming conventions or contributor patterns.

3.3 Calculation of Stickiness Scores

Devotion tracks code additions, deletions, and modifications of LOCs, which are considered the main cases. Additionally, edge cases such as code movements, duplications, and revivals are detected and incorporated into the Stickiness Score calculation. These edge cases are handled differently depending on the specific variation of the Stickiness Score, as detailed later in this section.

For each developer, each file and each folder has two attributes: *# of added lines* and *# of deleted lines*, which are then used to calculate their corresponding Stickiness Scores that can range from zero to 100, where zero indicates no code persistence, and 100 represents maximum persistence. These attributes are chosen because Devotion uses git commands to track the evolution of LOC, with git recording code modifications as additions and deletions. In general, the Stickiness

Score is calculated as follows:

$$Stickiness\ Score = \frac{\#\ of\ added\ lines - \#\ of\ deleted\ lines}{\#\ of\ added\ lines} \times 100$$

The following subsections highlight how these attributes are updated in each type of code modification case.

3.3.1 Overall Developer Stickiness Score

For the calculation of developers' Stickiness Scores, the initial author who wrote each LOC is credited as its originator. Even if the LOC is later deleted, moved, revived, or copied by another developer, the credit remains with the original author who introduced the LOC to the project. This ensures that the contribution of the initial author is consistently recognized, regardless of subsequent actions involving the same LOC.

This logic resembles patent principles, where inventors are recognized through attribution rights, safeguarding their connection to the invention even if economic rights are transferred [40]. Similarly, the initial author is credited for introducing the LOC, and any subsequent reuse by others does not alter the attribution, much like a patent holder retains recognition even if the invention is implemented in different contexts. This approach ensures that the originator's role in shaping the project's codebase is preserved, even when the LOC is reused or moved.

Devotion calculates Stickiness Scores by tracking LOC interactions across various scenarios, the complete algorithm can be seen in Algorithm 1. For new additions, the commit author's *# of added lines* attribute increases. When LOC are deleted, the initial author's *# of deleted lines* is incremented, regardless of who committed the deletion. For modifications (tracked as deletion-addition pairs), *# of added lines* increases for the commit author, while *# of deleted lines* increases for the initial author. In cases of duplication (e.g., copy-pasting), the initial author's *# of added lines* is incremented instead of the commit author who added the LOC.

Algorithm 1 Devotion Pseudocode

Require: `repoUrl` ▷ URL of the repository to analyze
Ensure: `authors.json, folder_structure.json` ▷ Computed outputs after analysis
Clone *repo* ← CloneRepository(`repoUrl`) ▷ Data fetching
commits ← GetMainBranchCommits(*repo*)
FuzzyMatchEmails(repo) ▷ Mailmap preprocessing
for each commit **in** commits **do**
    changes ← DetectTypeOfChanges(*commit*)
    ▷ Add/Delete/Modify/Move/Copy/Revive
    for each line **in** changes **do**
        commitAuthor, *initialAuthor* ← IdentifyRelevantAuthor(*line*)
        ▷ via `git blame` with flags -M -C
        if line is Added **then**
            IncrementAddedLines(commitAuthor, file, folder)
        else if line is Deleted **then**
            IncrementDeletedLines(initialAuthor, file, folder)
        else if line is Modified **then**
            IncrementAddedLines(commitAuthor)
            IncrementDeletedLines(initialAuthor)
        else if line is Copied **then**
            IncrementAddedLines(initialAuthor, destinationFile)
        else if line is Moved or Revived **then**
            UpdateAttributesAsNeutral(initialAuthor, commitAuthor)
        end if
    end for
end for
CalculateStickinessScores(allDevelopers, allFiles, allFolders)
WriteOutput(`authors.json`, `folder_structure.json`)
return

Code movements (relocation within the project) and revivals (re-added after deletion) are treated as neutral events, leaving Stickiness Scores unchanged, as these are considered refactoring or temporal extensions of movement. Moreover, modifications made by developers to their own previously written code do not update the scores, allowing room for them to refine or correct their own work. These approaches ensure accurate attribution while handling complex LOC dynamics.

Crediting the initial author for each LOC, as explained above, may result in some common LOC being attributed to a single author. However, while designing Stickiness Score, we aimed for it to complement code churn, capturing code

persistence while also accounting for more edge cases under the assumption that developers write unique code lines and follow the DRY (Don't Repeat Yourself) principle. With this approach, code movements, duplications, and revivals are meaningfully captured. However, we acknowledge that some common coding blocks in source code may be negatively affected by this assumption, impacting the calculation of the Stickiness Score. For example, widely used patterns such as for loops or standard logging statements (e.g., `logger.info("Starting process...")`) might appear in multiple files but be attributed to a single author. Nevertheless, we believe the Stickiness Score remains a useful indicator of potential problem areas in source code, as extremely high or low values may highlight maintainability issues related to the extent of code changes.

An example is depicted in Figure 3.2. A file in the project initially contains one LOC written by developer 1, dev1 for short, whose contribution is represented with pink. Assume the initial Stickiness Score of dev1 is $\frac{10-1}{10} \times 100 = 90$, which means dev1 had originally written 10 LOC to the project, but one LOC was then removed from the project. The dev2, who is represented with green, has $\frac{5-0}{5} \times 100 = 100$ as their Stickiness Score, indicating that they had written five LOCs in the project, and these LOCs still remain on the project. Each case for the Stickiness Score calculation will be given as modifications that occur to the scope of the file, which initially has one LOC from dev1. Assume that this file is modified independently in each given case, i.e., the given cases do not occur sequentially, and each modification is done by dev2. This is done to simplify the explanation of each case and highlight how the authors of the LOCs are credited in each case.

- In the line addition case, the leftmost one, an LOC that has unique content regarding the existing LOC in the project scope, is added to the file by dev2. In such a case, the *# of added lines* attribute of dev2 is incremented, and their Stickiness Score becomes $\frac{6-0}{6} \times 100 = 100$, while dev1's score remains the same.

- In the line deletion case, an LOC originally written by dev1 is removed, thus dev1's *# of deleted lines* attribute gets incremented, and their Stickiness

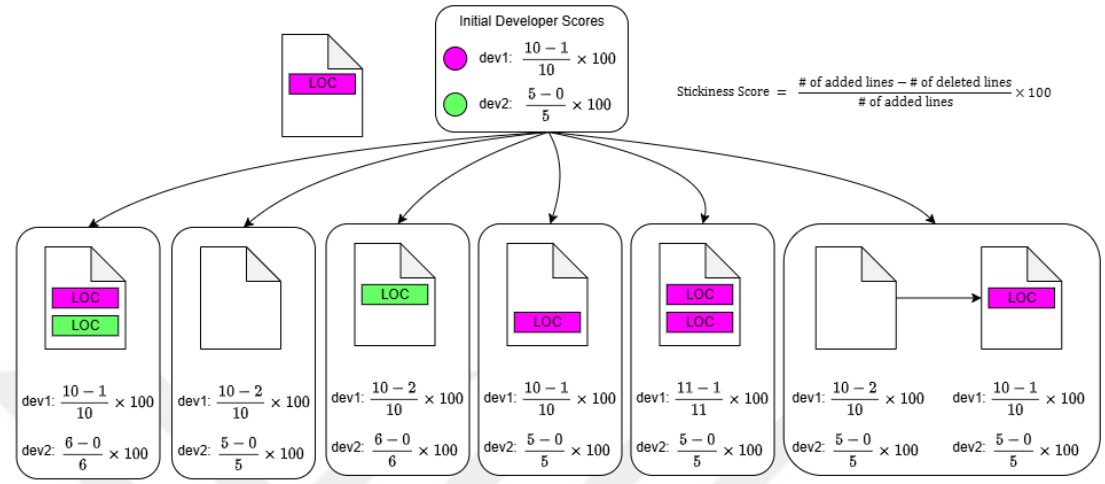


Figure 3.2: An example of Overall Developer Stickiness Score calculation. The initial scores of the developer are given at the top, and the resulting scores for each scenario are given separately.

Score decreases to $\frac{10-2}{10} \times 100 = 80$. Notice that dev2 is not affected by this as their number of LOCs stays the same, even though they are the one making the modification.

- In line modification, the LOC that was written by dev1 is changed regarding its contents, and since dev2 is the one making the unique modification, dev2 is credited with an extra line and their *# of added lines* is incremented. The dev2's Stickiness Score becomes $\frac{6-0}{6} \times 100 = 100$, while the dev1's *# of deleted lines* attribute gets incremented as one of their lines is removed from the project as the content of the line changes, which makes dev1's score $\frac{10-2}{10} \times 100 = 80$.

- In line move, only the placement of dev1's LOC is changed in the project, thus no modification of Stickiness Scores is done to any developers.

- In line duplication, the LOC written by dev1 is copied and pasted into the project by dev2. In this case, dev1 is attributed as they are the originator of this code and their Stickiness Score becomes $\frac{11-1}{11} \times 100 \approx 91$, while no modification is done to dev2's score.

- In line revival, first, the LOC written by dev1 is deleted by another developer, but after a while, it is brought back into the project by dev2. In this

case, Devotion behaves as if the LOC of dev1 was never deleted in the first place and preserves the existing Stickiness Scores of the developers, which was $\frac{10-1}{10} \times 100 = 90$ for dev1, and $\frac{5-0}{5} \times 100 = 100$ for dev2.

3.3.2 File and Folder Stickiness Score

The Stickiness Score calculation for files and folders is similar only for the main cases: code addition, deletion, and modification.

Devotion calculates Stickiness Scores for files and folders by tracking LOC changes. New lines increment the *# of added lines*, while deletions increment the *# of deleted lines* for both the file and its folders. Modifications update both metrics. Copying a line updates the *# of added lines* for the destination file, leaving the source unchanged. Moving a line between files adjusts the *# of deleted lines* for the source and *# of added lines* for the destination, with folder Stickiness Scores updated accordingly. Line revivals increment *# of deleted lines* for the original file and *# of added lines* for the revival file. File moves increment *# of deleted lines* for the source folder (i.e., the folder that contained the old file that moved away) and *# of added lines* for the destination (i.e., the folder that contains the new file that is moved in), with common ancestor folders updated such as code movements. Moreover, the file renaming does not affect the scores.

Tracking the "initial author" is not applicable in the file/folder context, as Devotion credits the original file for LOCs, but when LOCs move, the original file's total is decremented. Crediting the "initial file" for copied code is inconsistent since its content remains unchanged. Thus, the Stickiness Score in this context reflects the absence of code churn.

An example to solidify the way Devotion calculates File Stickiness Score as given in Figure 3.3. The initial project contains two files, where File1 contains nine LOCs, which are represented with turquoise, and File2, which has five LOCs given with orange. The initial Stickiness Score of File1 is $\frac{10-1}{10} \times 100 = 90$, which means File1 had originally contained 10 LOC, but one LOC was then removed.

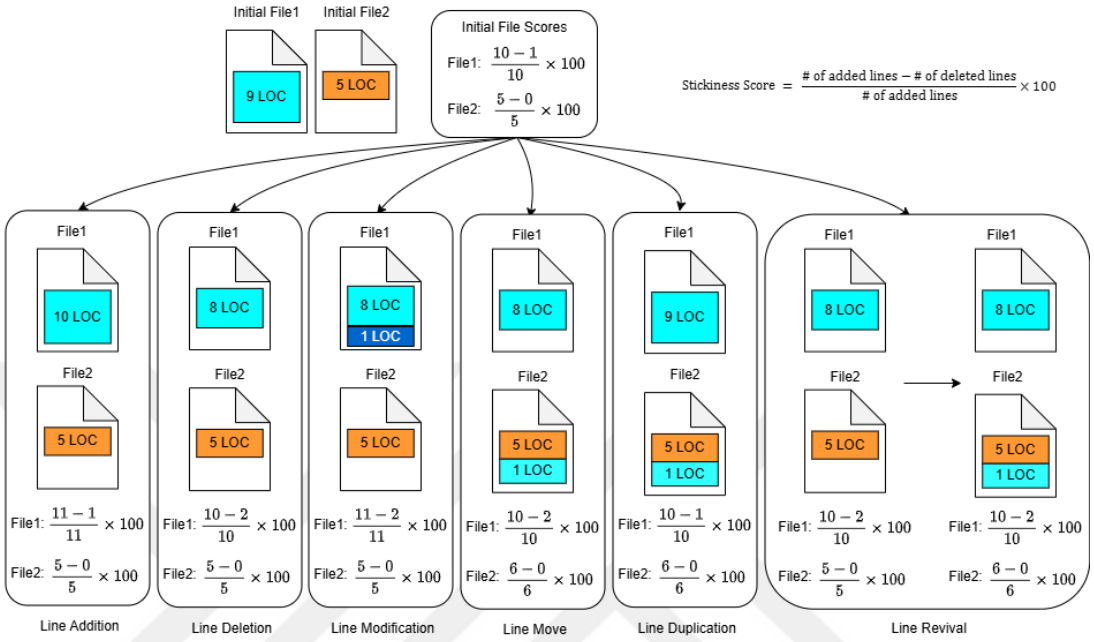


Figure 3.3: An example of File Stickiness Score calculation. The initial scores of the files are given at the top, and the resulting scores of each scenario are given separately.

The File2 has $\frac{5-0}{5} \times 100 = 100$ as its Stickiness Score, indicating that it had five LOC initially, and they still remain on it. Assume that each case occurs independently, i.e., the given cases do not occur sequentially. This is done to simplify the explanation of each case.

- In the line addition case, one LOC is added to File1; hence, the *# added lines* attribute of File1 is incremented, and its Stickiness Score becomes $\frac{11-1}{11} \times 100 \approx 91$, while File2's score remains the same.
- In line deletion case, an LOC from File1 is removed; thus, its *# of deleted lines* attribute gets incremented and its Stickiness Score decreases to $\frac{10-2}{10} \times 100 = 80$.
- In line modification, when an LOC is changed regarding its contents, both the *# of added lines* and *# of deleted lines* attributes of the file are incremented. This is similar to code churn, as we credit the change as a deletion and an addition. In the end, File1's score becomes $\frac{(10+1)-(1+1)}{(10+1)} \times 100 \approx 82$.

- In line move, the placement of one LOC in File1 is changed to be at File2, thus, the resulting Stickiness Score of the files will be as if one line is deleted from File1 and one line is added to File2. File1 will have Stickiness Score $\frac{10-2}{10} \times 100 = 80$ while File2 $\frac{6-0}{6} \times 100 = 100$.
- The line duplication case is considered as another line addition. In the depicted example, one LOC is copied from File1 and pasted into File2. Since File1 has not undergone any changes in its scope, its score stays the same, while File2 is credited with a line addition. File2's Stickiness Score becomes $\frac{6-0}{6} \times 100 = 100$.
- In line revival, first the LOC in File1 is deleted, but after a while it is brought back into the project on File2. In this case, since Devotion does not track the edge cases for Files, it tracks them as a deletion from File1 and an addition to File2, in the same manner as a line move. Thus the Stickiness Scores of files become $\frac{10-2}{10} \times 100 = 80$ for File1, and $\frac{6-0}{6} \times 100 = 100$ for File2.

3.4 Outputs

The results of the calculations are presented in the output files, which provide detailed information about developers and files in the project. For each developer, it includes their name, Stickiness Score, rank within the project, LOC they added, modified, copied, moved, or revived, and their total commit count. For each file, it details the Stickiness Score, total LOC added and deleted, contributor count, and contributor metrics (LOC added/deleted and commit counts).

Devotion also presents these results visually with a web interface that has two tabs, illustrated in Figures 3.4 and 3.5. The first tab ranks developers by Stickiness Scores alongside their LOC metrics and commits, as illustrated in Figure 3.4, where the results of the **core** project are shown with developer names anonymized. Figure 3.5 displays the second tab, which presents the project's overall Stickiness Score, file and folder Stickiness Scores, as well as the LOC added and modified

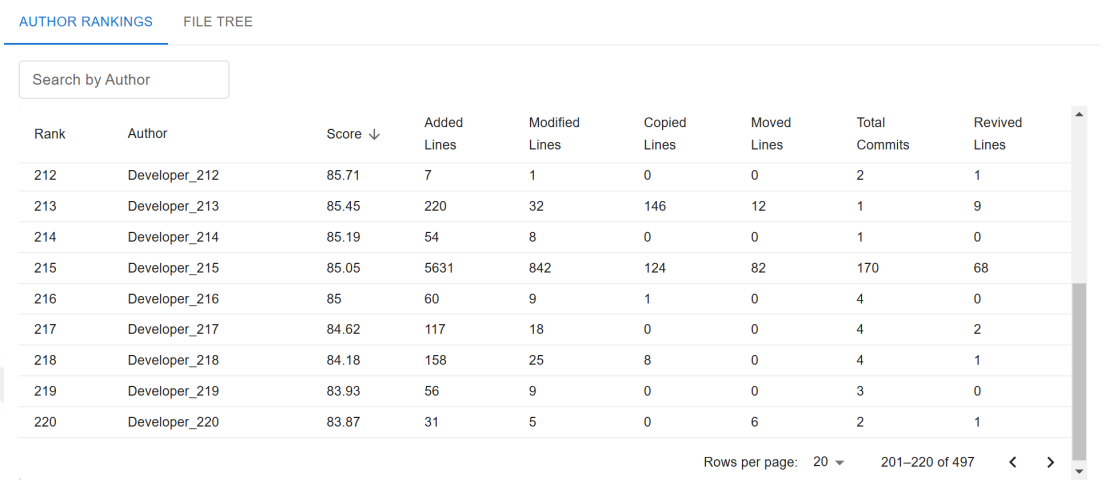


Figure 3.4: Devotion User Interface: Developers (Developer Names are Anonymized)

(i.e., deleted or changed) for each.

Figure 3.5: Devotion User Interface: Folders and Files

Chapter 4

Background for Correlation Analysis

To provide context for the research methodology employed in this study, a brief background on correlation analysis is presented. This background sets the stage for interpreting the statistical tests and results that will be conducted later in the analysis.

Correlation analysis measures the statistical relationship between two or more quantitative variables, indicating how changes in one variable are associated with changes in another. It is a foundational method widely used in fields such as biology, medical sciences, engineering, robotics, finance, bioinformatics, and sports, among others, to uncover patterns and relationships in data [41].

Correlation coefficients range between -1 and 1, where values close to -1 indicate a strong negative correlation, values close to 1 indicate a strong positive correlation, and values near 0 suggest no correlation [42].

However, the significant correlation coefficient does not always indicate a strong association. The significance of the correlation coefficient is influenced by the size of the sample. In smaller samples, the correlation coefficient must be much closer to -1 or 1 to indicate a significant association. On the other hand,

in larger samples, even a correlation coefficient near zero can appear significant, despite reflecting only a weak association. To properly evaluate the strength of a relationship, it is essential to consider the results of the significance test, the value of the correlation coefficient, and the density plot of the variables together [43].

4.1 Types of Correlation Tests

Two widely used correlation tests are **Pearson's correlation coefficient** and **Spearman's rank correlation coefficient**. These tests are chosen based on the underlying assumptions of the data, particularly the normality of the distribution.

4.1.1 Pearson's Correlation Coefficient

Pearson's correlation measures the strength and direction of the linear relationship between two continuous variables. It assumes [44]:

- Both variables are jointly normally distributed and continuous random variables.
- A linear relationship exists between the variables.
- There are no outliers that can influence the coefficient.
- Each pair of X-Y values must be measured independently of all other pairs. Collecting multiple observations from the same subjects would breach this assumption.

The formula for Pearson's correlation is [45]:

$$r = \frac{\sum(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum(X_i - \overline{X})^2 \cdot \sum(Y_i - \overline{Y})^2}} \tag{4.1}$$

where:

- X_i and Y_i are the individual data points.
- $\overline{X}$ and $\overline{Y}$ are the means of the variables.

While highly efficient for normal data, Pearson's correlation may yield misleading results when assumptions are violated.

4.1.2 Spearman's Rank Correlation Coefficient

Spearman's correlation assesses the strength and direction of the monotonic relationship between two variables. It is a non-parametric method that does not assume normality or linearity in the data. Spearman's method computes correlation based on rank orders of values instead of their actual values, making it robust for ordinal data or data with non-linear relationships [44].

Spearman's formula is [46]:

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{4.2}$$

where:

- r_s: Spearman's rank correlation coefficient.
- d: The difference between the ranks of two variables.
- n: The number of observations (ordered pairs).

4.2 The Role of Normality in Choosing Correlation Tests

Before selecting a correlation test, it is crucial to determine whether the data follows a normal distribution. This is because Pearson's correlation relies heavily on the normality assumption, while Spearman's correlation does not. Normality tests provide a statistical framework to assess whether the data deviates significantly from a normal distribution, guiding the selection of the appropriate correlation method.

4.3 Normality Tests

Several statistical tests evaluate whether a dataset is normally distributed. Each test applies different mathematical criteria, and their choice depends on factors such as sample size and sensitivity to deviations from normality.

4.3.1 Shapiro-Wilk Test

The Shapiro-Wilk test is one of the most commonly used methods for assessing normality. It compares the sample data to a normal distribution based on the sample's quantiles. A p-value below a chosen significance level (e.g., 0.05) indicates that the data significantly deviates from normality. Its formula is as follows [47]:

$$W = \frac{\left(\sum_{i=1}^{n} a_i y_{(i)}\right)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.3}$$

where

- n: The sample size.
- y_i: The i-th ordered value in the sample ($y_1 < y_2 < \cdots < y_n$).

- $\bar{y}$: The sample mean, calculated as $\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$.
- a_i: Weights based on the expected values of order statistics from a normal distribution.

The test is highly effective for small to moderate sample sizes.

4.3.2 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test compares the empirical cumulative distribution and the hypothetical cumulative distribution of the data. The test statistic quantifies the maximum distance between these distributions [48]. It is not highly sensitive to the accuracy of the fit in the tails of the tested distribution. This is particularly problematic when tail events are the primary interest. [49].

4.3.3 Jarque-Bera Test

The Jarque-Bera test evaluates normality based on skewness, which is a measure of the asymmetry of the probability distribution of a dataset about its mean, and kurtosis. It is particularly suitable for large datasets. The test statistic is [50]:

$$JB = n\left(\frac{S^2}{6} + \frac{(K-3)^2}{24}\right) \tag{4.4}$$

Where S is skewness, K is kurtosis, and n is the sample size.

4.3.4 Anderson-Darling Test

The Anderson-Darling test is an enhancement of the Kolmogorov-Smirnov test, giving more weight to the tails of the distribution. This sensitivity makes it suitable for identifying deviations in extreme values. A lower test statistic suggests a better fit to the normal distribution [47].

The Anderson-Darling test statistic is defined as:

$$A = -n - \frac{1}{n} \sum_{i=1}^{n} [(2i - 1) \ln(p_i) + \ln(1 - p_{n+1-i})]$$

where:

- p_i is the cumulative probability of the standard normal distribution for the standardized value $z_i = \frac{x_i - \bar{x}}{s}$, where $\bar{x}$ is the sample mean and s is the sample standard deviation.
- n is the sample size.

The modified statistic M for small sample sizes, ensuring the asymptotic critical values are applicable, is given by [51]:

$$M = A \left(1 + \frac{0.75}{n} + \frac{2.25}{n^2}\right)$$

4.3.5 D'Agostino's K^2 Test

D'Agostino's K^2 test is a comprehensive method for assessing normality, designed to detect deviations caused by either skewness (b_1) or kurtosis (b_2). The K^2 statistic combines the squared standardized skewness ($Z(\sqrt{b_1})$) and kurtosis ($Z(b_2)$) measures:

$$K^2 = Z^2(\sqrt{b_1}) + Z^2(b_2)$$

Under the null hypothesis of normality, K^2 follows a chi-squared distribution with two degrees of freedom. This makes the test particularly effective for identifying non-normality arising from asymmetry or tail heaviness in the data [52].

4.4 Integrating Normality Tests with Correlation Analysis

The choice between Pearson's and Spearman's correlation is contingent upon the results of normality tests. If the data passes normality checks, Pearson's correlation is preferred due to its sensitivity to linear relationships. Conversely, if normality is violated, Spearman's correlation is more appropriate, as it is robust against non-normal distributions and non-linear patterns.

By combining normality tests with correlation methods, researchers can ensure that statistical analyses are both valid and reliable, enhancing the interpretability of their results.

Chapter 5

Methodology

This section outlines our methodology to investigate the relationships between Stickiness Scores and various software metrics across multiple research questions as depicted in Figure 5.1. An illustrative example is presented to demonstrate the formulations and calculations used for each research question and to show how data is prepared for hypothesis testing. The subsequent subsections describe the processes of data collection and preprocessing. Finally, the statistical hypothesis testing methodology is explained.

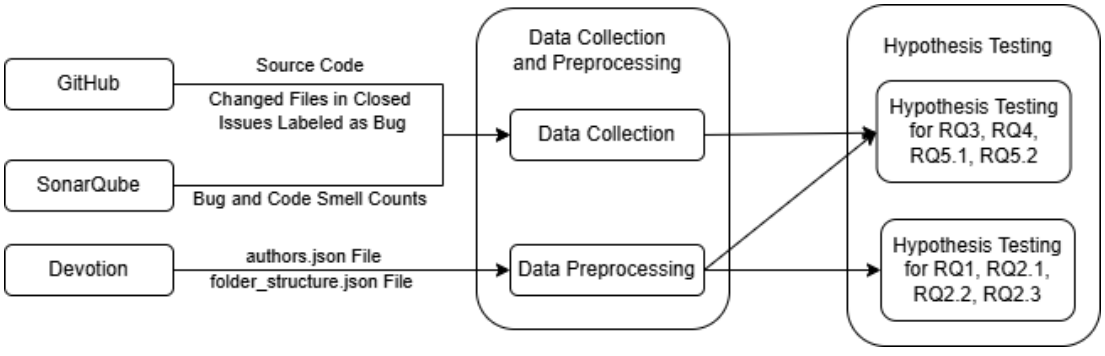


Figure 5.1: Workflow of Study

5.1 An Example Demonstrating the Study's Approach

Figure 5.2 shows two files within the same project. Commits and authors for the corresponding LOCs are marked in red, with "dev" representing a developer. File1 originally had 10 lines; then, the deletion of Line7 and Line10 resulted in a Stickiness Score of 80. File2, also with 10 lines, had Lines 7-10 deleted, leading to a Stickiness Score of 60. It is important to note that Stickiness Scores are measured on a scale from 0 to 100.

In RQ1, we examine the relationship between a file's Stickiness Score and its number of contributors. For instance, File1 has a Stickiness Score of 80, with contributions from three developers throughout its history, while File2 has a Stickiness Score of 60, with contributions from four developers. The data points for hypothesis testing are then (80, 3) and (60, 4).

In RQ2.1, we analyze the relationship between a file's Stickiness Score and the average Stickiness Score of the developers who contributed to it. A developer's Stickiness Score, calculated across the entire project, represents their tendency to write sticky code, and the overall developer Stickiness Scores for this example are given in the table of Figure 5.2. For File1, the Stickiness Score is 80, while the average Stickiness Score of its contributing developers is

$$(90 + 70 + 50)/3 = 70$$

Similarly, for File2, the Stickiness Score is 60, and the average Stickiness Score of its contributing developers is

$$(90 + 70 + 50 + 10)/4 = 55$$

The data points for the hypothesis testing in this case are (80, 70) and (60, 55).

In RQ2.2, we investigate the correlation between a file's Stickiness Score and the weighted average of developers' Stickiness Scores, adjusted by their commit count. The weighted average is calculated by multiplying each developer's

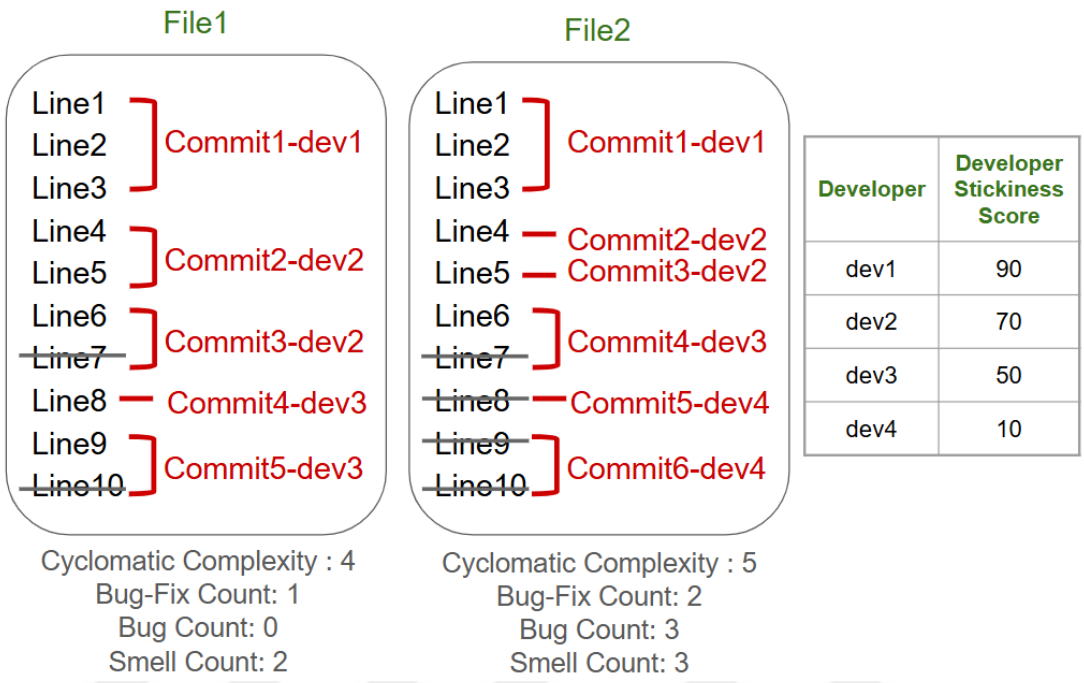


Figure 5.2: An Example to Demonstrate the Study's Approach

Stickiness Score by their commit count, summing the weighted scores across all developers (n), and dividing by the total number of commits (C). This ensures the average reflects each developer's contribution. The calculation of the average commit-weighted developer Stickiness Score is as follows:

$$\frac{\sum_{i=1}^{n}(\text{Stickiness Score}_i \times \text{Commit Count}_i)}{C}$$

For File1, which has a Stickiness Score of 80, the number of commits weighted average contributor Stickiness Score is calculated as:

$$(90 \times 1 + 70 \times 2 + 50 \times 2)/5 = 66$$

For File2, which has a Stickiness Score of 60, the number of commits weighted average contributor Stickiness Score is:

$$(90 \times 1 + 70 \times 2 + 50 \times 1 + 10 \times 2)/6 = 50$$

The data points for the hypothesis testing, in this case, are (80, 66) and (60, 50).

In RQ2.3, we explore the relationship between a file's Stickiness Score and the weighted average of developers' Stickiness Scores, adjusted by the LOC they

contributed. The weighted average is calculated by multiplying each developer's Stickiness Score by their contributed LOC, summing the results across all developers (n), and dividing by the total LOC (L) of the file. This ensures that developers who contributed more LOC have a greater influence on the final average. The calculation of the LOC-weighted average developer Stickiness Score is as follows:

$$\frac{\sum_{i=1}^{n}(\text{Stickiness Score}_i \times \text{LOC}_i)}{L}$$

For File1, which has a Stickiness Score of 80%, the weighted average developer Stickiness Score is now calculated as:

$$(90 \times 3 + 70 \times 4 + 50 \times 3)/10 = 70$$

For File2, which has a Stickiness Score of 60, the weighted average contributor Stickiness Score is:

$$(90 \times 3 + 70 \times 2 + 50 \times 2 + 10 \times 3)/10 = 54$$

The data points for the hypothesis testing, in this case, are (80, 70) and (60, 54).

In RQ3, we analyze the relationship between a file's cyclomatic complexity and its Stickiness Score. File1 has a Stickiness Score of 80 and a cyclomatic complexity of four, while File2 has a Stickiness Score of 60 and a cyclomatic complexity of five. The data points for the hypothesis testing in this case are (80, 4) and (60, 5).

In RQ4, we examine the correlation between a file's Stickiness Score and the number of GitHub-reported bugs resolved through changes to that file. The number of such changes in a file is referred to as the "bug-fix count of the file" throughout the remainder of this paper. For instance, File1 has a Stickiness Score of 80 and one associated bug fix, while File2 has a Stickiness Score of 60 and two associated bug fixes. The corresponding data points for hypothesis testing are (80, 1) and (60, 2).

In RQ5.1, we investigate the relationship between the number of bugs detected

by the SonarQube static code analyzer and the file's Stickiness Score. File1 reports a Stickiness Score of 80 with zero bugs detected, whereas File2 has a Stickiness Score of 60 and three detected bugs. The data points used for hypothesis testing in this case are (80, 0) and (60, 3).

In RQ5.2, we assess the correlation between the number of code smells detected by SonarQube and the file's Stickiness Score. File1 has a Stickiness Score of 80 with two detected smells, while File2 has a Stickiness Score of 60 and three smells. The corresponding data points are (80, 2) and (60, 3).

5.2 Data Collection and Preprocessing for Research Questions

This section outlines the data collection process from GitHub and SonarQube, as well as the data preprocessing performed using Python scripts, which process the Devotion tool's output files: "author.json" and "folder_structure.json".

RQ1: Is the number of contributors to the file correlated with the Stickiness Score of the file?

For this question, the Stickiness Scores and the number of contributors are crucial. The "folder_structure.json" file provides file Stickiness Scores, while the number of contributors is determined by counting the developers listed as contributors of each file.

RQ2.1: Is the average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file?

This question requires the Stickiness Scores of files and the average Stickiness Score of file contributors. The "author.json" file lists developers and their Stickiness Scores, while the "folder_structure.json" file maps contributors to files. The average Stickiness Score is calculated from this data.

RQ2.2: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of commits contributed to the file?

This question builds on RQ2.1, incorporating commit counts as weights. The necessary data includes the Stickiness Scores of files, developers' Stickiness Scores, and the number of commits per developer for each file, all of which are available in the "folder_structure.json" file.

RQ2.3: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of lines of code contributed to the file?

This question requires the Stickiness Scores of files, the developers' average Stickiness Scores, and the LOC contributed by each developer for each file. The preprocessing is similar to RQ2.1 but includes the LOC data from the "folder_structure.json" file, calculated as:

$$\textit{Total Contributed LOC to the File} = \textit{Total Added LOC to the File} + \textit{Total Deleted LOC to the File}$$

RQ3: Is the file's cyclomatic complexity correlated with its Stickiness Score?

This question requires Stickiness Scores and cyclomatic complexity. The "folder_structure.json" file contains Stickiness Scores and cyclomatic complexity is calculated using Python's Lizard library [53], analyzing source code obtained from GitHub API [54] for files with extensions such as ".java", ".js", ".cs", ".ts", ".py", and ".go".

RQ4: Is the number of bug-fixing pull requests that modify a file correlated with its Stickiness Score?

To analyze the correlation between a file's Stickiness Score and the number of bugs resolved through pull requests that change this file, data was collected by identifying bug-related issues in GitHub and mapping them to the files modified in corresponding pull requests. A Python script was developed to automate data collection using the GitHub API. Bug-related issues were extracted using the GitHub search filter **is:issue state:closed linked:pr label:"bug"**, which retrieves closed issues labeled as bugs and linked to pull requests. Since different repositories use varying labels to categorize bugs, the script dynamically filters issues based on the relevant bug labels for each project. It should be noted that labeling practices vary across repositories, and not all bug-related issues may be explicitly labeled, which introduces a potential threat to the completeness of bug identification. To mitigate this, we tailored the search filter regarding the labeling conventions used in the five OSS projects we analyzed. Each pull request contains a list of modified files, and for every file changed within a pull request linked to a bug-related issue, its bug-fix count was incremented by one. The file Stickiness Scores are obtained from "folder_structure.json".

RQ5.1: Is the file's number of bugs collected from static code analysis correlated with its Stickiness Score?

SonarQube [55] is an open-source platform for continuous code quality inspection, enabling the detection of bugs, vulnerabilities, and code smells across multiple programming languages. It integrates seamlessly with CI/CD pipelines, performing static analysis to provide insights into code maintainability and security. By generating detailed reports, SonarQube helps developers identify problematic code patterns and improve software quality.

To collect bug counts from OSS projects, a local SonarQube server was set up, and project-specific properties files were created based on the programming languages used. These configuration files define key parameters such as project identifiers, source directories, and language settings, ensuring accurate analysis. A Python script was developed to interact with SonarQube's API, automating the extraction of bug counts for each file. The file Stickiness Scores are again gathered from "folder_structure.json".

RQ5.2: Is the file's number of code smells collected from static code analysis correlated with its Stickiness Score?

SonarQube also identifies code smells, which are indicators of maintainability issues that do not necessarily affect functionality but can lead to technical debt over time. These code smells are detected through static analysis, highlighting areas where best coding practices are not followed. By analyzing code smells at the file level, potential refactoring opportunities can be identified to improve code structure and readability.

Similar to bug counts, a Python script was used to extract smell counts for each file by querying SonarQube's API. The Stickiness Scores are once again retrieved from "folder_structure.json".

5.3 Hypothesis Testing

Hypothesis testing explores the relationship between various metrics and the Stickiness Score of a file. Spearman or Pearson correlation tests are used based on data distribution. Spearman's rho assumes a monotonic relationship, while Pearson's correlation requires linearity, normality, and the absence of significant outliers. Another correlation test, Kendall's tau, is also a nonparametric measure akin to Spearman's rho but is computationally more expensive for large datasets. While Kendall's tau has better statistical properties and a probability-based interpretation, it was historically harder to compute, making Spearman more widely adopted. Since both measures yield similar results and Spearman's correlation is treated as the rank-based equivalent of Pearson's correlation, it remains the preferred choice for statistical analysis [56]. Moreover, the normality of the data to choose between Pearson and Spearman correlation tests is calculated using the Shapiro-Wilk, Kolmogorov-Smirnov, Jarque-Bera, Anderson-Darling, and D'Agostino's K^2 tests. The correlation strength is classified according to the correlation coefficient (r) as weak ($0.1 \leq |r| < 0.3$), moderate ($0.3 \leq |r| < 0.5$), or strong ($0.5 \leq |r| \leq 1$) [57].

Each research question is tested with a null hypothesis (H_0), assuming no correlation, and an alternative hypothesis (H_a), suggesting a correlation exists. For each research question, the hypothesis testing includes:

Null Hypothesis (H_0): There is no correlation between the software metric of interest and the Stickiness Score of the file ($r = 0$).

Alternative Hypothesis (H_a): A correlation exists between the software metric of interest and the Stickiness Score of the file ($r \neq 0$).

Test Selection: Spearman or Pearson correlation is applied based on the normality tests of the data.

Chapter 6

Results

This section outlines the project selection criteria and the results of the correlation analyses for the research questions.

6.1 Project Selection

We used OSS projects from GitHub to ensure access to real-world software development histories since they provide rich metadata and enable the systematic analysis of code evolution. Projects were chosen to balance popularity, collaboration dynamics, and practicality. To ensure manageable execution times that do not exceed five days, we tested projects with <10K commits and confirmed that they were completed within a reasonable duration, as shown in Table 6.1. In contrast, larger projects took significantly longer to process. Additionally, for statistical significance, projects with at least 10K stars and 100 contributors were preferred for Stickiness Score calculation. Table 6.1 lists the selected projects, including their stars, contributors, commits, and execution times. The number of files in each project is also provided in the table to indicate the repository size in terms of file count.

The selected repositories cover web frameworks, state management, microservices, GraphQL, and data visualization. **core** and **fastapi** are popular for frontend and backend development, while **go-micro** supports micro-services. **redux** manages state in JavaScript apps, and **apollo-server** simplifies GraphQL APIs. These projects were chosen to test our methodology on a diverse set of front-end and back-end projects, ensuring a comprehensive evaluation.

The study was conducted on a server running Oracle Linux Server 8.10 with dual Intel Xeon E5-2603 v4 1.7 GHz CPUs and 64 GB of RAM. The data collection and test runs were completed on January 6th, 2025, by which time all commits on the main branch had been analyzed.

Table 6.1: The Selected Projects

Project	Stars	Contributors	Commits	Execution Time	File Count
core	48.3k	524	6504	38.22 hours	574
fastapi	79.1k	794	5359	12.95 hours	1353
go-micro	22k	191	3816	14.99 hours	254
redux	61k	996	4054	1.89 hours	220
apollo-server	13.8k	569	8426	76.85 hours	131

6.2 Hypothesis Testing Results for Research Questions

This section presents the correlation test results for each research question, addressing the correlations between Stickiness Scores and software metrics. Consequently, all five projects exhibited non-normal distributions based on the results for each research question. Consequently, the Spearman correlation test was used for the correlation analysis throughout this study.

RQ1: Is the number of contributors to the file correlated with the Stickiness Score of the file?

All five projects exhibited non-normal distributions based on the Shapiro-Wilk, Kolmogorov-Smirnov, Jarque-Bera, Anderson-Darling, and D'Agostino's K^2 tests. Consequently, the Spearman correlation test was used to assess the relationship between the Stickiness Score of a file and its number of contributors.

The Spearman correlation test revealed statistically significant negative correlations between Stickiness Scores and contributor count in all projects. **redux** showed the strongest negative correlation ($r_s = -0.72348$), followed by **apollo-server** ($r_s = -0.62989$), **core** ($r_s = -0.50131$), and **go-micro** ($r_s = -0.49685$). **fastapi** showed a moderate negative correlation with a coefficient of $r_s = -0.41483$.

The density plots that illustrate this correlation are given in Appendix A, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.2: Hypothesis Testing Results for RQ1

Project	**Correlation Coefficient**	**p-Value**
core	-0.50131	$7.42e-38$
fastapi	-0.41483	$2.07e-57$
go-micro	-0.49685	$3.06e-17$
redux	-0.72348	$6.14e-37$
apollo-server	-0.62989	$7.65e-16$

RQ2.1: Is the average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file?

For most projects, including **core**, **fastapi**, **go-micro**, and **apollo-server**, all normality tests concluded that at least one of the variables, either the data of the file Stickiness Scores or their average of the developer Stickiness Scores, was non-normally distributed. This consistent finding led to the use of Spearman

correlation as the appropriate method for these projects, given its robustness to deviations from normality.

In the case of the **redux** project, the Kolmogorov-Smirnov test suggested normality for both the file Stickiness Scores and the averages of the developer Stickiness Scores, which could have supported Pearson correlation. However, the Kolmogorov-Smirnov test is less sensitive to subtle deviations from normality, especially in the tails. Other tests, such as Shapiro-Wilk, Anderson-Darling, and D'Agostino's K^2, identified significant non-normalities in the data of Stickiness Scores, highlighting deviations that Kolmogorov-Smirnov missed. These tests are more sensitive to skewness and kurtosis, indicating that the data violated the assumptions required for Pearson correlation. To ensure a consistent and assumption-robust analysis across all projects, Spearman correlation was selected for **redux** as well. Given the presence of non-normality and potential outliers, this decision provides a more reliable and uniform basis for interpreting monotonic relationships in the data.

Spearman correlation results revealed varying correlations between file Stickiness Scores and the average developer Stickiness Scores. Three projects demonstrated statistically significant positive correlations ($p < 0.05$), and the other two projects showed no significant relationship. **redux** showed the strongest significant positive correlation ($r_s = 0.50763$), followed by **apollo-server** ($r_s = 0.39791$), and **fastapi** ($r_s = 0.11338$). **core** ($r_s = 0.06854$), and **go-micro** ($r_s = 0.06797$) did not show significant correlations.

The density plots that illustrate this correlation are given in Appendix B, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.3: Hypothesis Testing Results for RQ2.1

Project	Correlation Coefficient	p-Value
core	0.06854	0.101
fastapi	0.11338	$2.91e-5$
go-micro	0.06797	0.281
redux	0.50763	$8.23e-16$
apollo-server	0.39791	$2.52e-6$

RQ2.2: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of commits contributed to the file?

For the analysis of the correlation between the Stickiness Score of the file and the commit-weighted average developer Stickiness Score, all normality tests indicated non-normal distributions. Consequently, the Spearman correlation test was chosen to assess the relationship between these two variables.

Spearman correlation results showed a positive correlation for all the projects. **redux** exhibited the strongest positive correlation ($r_s = 0.55599$), followed by **core** ($r_s = 0.39653$), **fastapi** ($r_s = 0.12347$), **go-micro** ($r_s = 0.20459$), and **apollo-server** ($r_s = 0.30583$).

The density plots that illustrate this correlation are given in Appendix C, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.4: Hypothesis Testing Results for RQ2.2

Project	Correlation Coefficient	p-Value
core	0.39653	$4.71e-23$
fastapi	0.12347	$5.24e-6$
go-micro	0.20459	0.00104
redux	0.55599	$3.00e-19$
apollo-server	0.30583	$3.82e-4$

RQ2.3: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of lines of code contributed to the file?

Based on the normality tests, which showed that at least one variable in each project did not follow a normal distribution, the Spearman correlation test was selected to evaluate the relationship between the Stickiness Score of the files and the LOC-weighted average developer Stickiness Score of the files for each project.

Spearman correlation revealed a positive correlation for four projects and no correlation for one project. **redux** showed the strongest significant positive correlation ($r_s = 0.51890$), followed by **apollo-server** ($r_s = 0.39201$), **core** ($r_s = 0.32886$), and **fastapi** ($r_s = 0.27257$). On the other hand, **go-micro** ($r_s = -0.04256$) showed no significant correlations.

The density plots that illustrate this correlation are given in Appendix D, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.5: Hypothesis Testing Results for RQ2.3

Project	**Correlation Coefficient**	**p-Value**
core	0.32886	$6.07e-16$
fastapi	0.27257	$1.774e-24$
go-micro	-0.04256	0.499
redux	0.5189	$1.45e-16$
apollo-server	0.39201	$3.65e-6$

RQ3: Is the file's cyclomatic complexity correlated with its Stickiness Score?

Normality tests indicated that at least one of the variables, either the Stickiness Score or the cyclomatic complexity score of the file, was non-normally distributed.

This led to the use of Spearman correlation for assessing the relationship between these variables across all these projects.

Spearman correlation showed mixed results between Stickiness Score and cyclomatic complexity. **core** showed a weak negative correlation ($r_s = -0.09688$), and **fastapi** showed a weak positive correlation ($r_s = 0.08952$). **go-micro** ($r_s = -0.05236$), **redux** ($r_s = 0.08019$), and **apollo-server** ($r_s = -0.11412$) showed no significant correlations.

The density plots that illustrate this correlation are given in Appendix E, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.6: Hypothesis Testing Results for RQ3

Project	**Correlation Coefficient**	**p-Value**
core	-0.09688	0.0288
fastapi	0.08952	0.00131
go-micro	-0.05236	0.409
redux	0.08019	0.2714
apollo-server	-0.11412	0.225

RQ4: Is the number of bug-fixing pull requests that modify a file correlated with its Stickiness Score?

Normality tests revealed non-normal distribution in at least one variable, so Spearman correlation was used to assess their relationship across projects.

Spearman correlation revealed a negative correlation between the file's number of bugs in GitHub resolved through pull requests and file Stickiness Scores for two projects, no correlation for one project, and insufficient data for two projects. **core** showed the strongest significant negative correlation ($r_s = -0.32$), followed by **fastapi** ($r_s = -0.09506$). **go-micro** ($r_s = 0.01374$) exhibited no significant correlation. For the **redux** and **apollo-server** projects, no files that were changed due to bug-fix pull requests were identified, so correlation results were

not computed.

The density plots that illustrate this correlation are given in Appendix F, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.7: Hypothesis Testing Results for RQ4

Project	**Correlation Coefficient**	**p-Value**
core	-0.32	$4.82e-15$
fastapi	-0.09506	0.00046
go-micro	0.0135	0.8305
redux	-	-
apollo-server	-	-

RQ5.1: Is the file's number of bugs collected from static code analysis correlated with its Stickiness Score?

Due to non-normality in at least one variable, the Spearman correlation was applied to analyze their relationship across projects.

Spearman correlation analysis for the number of bugs collected from static code analysis and the file Stickiness Scores revealed no significant correlations across the analyzed projects. **core** ($r_s = 0.04347$), **fastapi** ($r_s = 0.01867$), and **redux** ($r_s = 0.06421$) showed weak positive correlations, but none were statistically significant. Similarly, **apollo-server** ($r_s = 0.01557$) exhibited an extremely weak and non-significant correlation. Due to a lack of available data, correlation results were not computed for **go-micro**.

The density plots that illustrate this correlation are given in Appendix G, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.8: Hypothesis Testing Results for RQ5.1

Project	Correlation Coefficient	p-Value
core	0.04347	0.3277
fastapi	0.01867	0.50359
go-micro	-	-
redux	0.06421	0.37875
apollo-server	0.01557	0.86884

RQ5.2: Is the file's number of code smells collected from static code analysis correlated with its Stickiness Score?

Spearman's correlation was used to examine the relationship, as at least one variable was not normally distributed.

Spearman correlation analysis for this research question revealed both significant and non-significant correlations between the number of smells in the file and the file's Stickiness Score across projects. **fastapi** ($r_s = -0.08487$) showed statistically significant negative correlation. On the other hand, **core** ($r_s = -0.21937$), **go-micro** ($r_s = 0.04816$), **redux** ($r_s = 0.07101$), and **apollo-server** ($r_s = -0.17292$) did not show statistically significant correlations, despite weak to moderate positive or negative coefficients.

The density plots that illustrate this correlation are given in Appendix H, highlighting regions of high data concentration and overall trends using a smoothed LOWESS curve.

Table 6.9: Hypothesis Testing Results for RQ5.2

Project	Correlation Coefficient	p-Value
core	-0.21937	5.7917
fastapi	-0.08487	0.00232
go-micro	0.04816	0.44749
redux	0.07101	0.33026
apollo-server	-0.17292	0.06459

Chapter 7

Discussion

This section examines the interpretations of the results of the study, some examples from the projects, and discusses their implications for software development researchers and practitioners.

7.1 Interpretation of Results

This section discusses the results and presents examples from the selected five OSS projects that support our hypotheses, along with the cases that may explain why the hypotheses are rejected in certain instances. It is important to note that we are only examining the correlation between two variables—the file Stickiness Score and the corresponding software metric. This analysis does not imply causation; it simply identifies whether a correlation exists between these variables without making any claims about one causing the other.

RQ1: Is the number of contributors to the file correlated with the Stickiness Score of the file?

The negative correlation between the number of contributors and the Stickiness Score across all projects indicates that higher contributor counts are associated with lower Stickiness Scores. Projects with more contributors may see a greater diversity of coding styles and approaches, which could introduce inconsistency and reduce the perceived stability of the codebase. In OSS projects, especially with many contributors, there may be less coordination or communication regarding coding standards, architecture decisions, or general design principles. This lack of alignment can lead to fragmented or inconsistent code, further reducing the Stickiness Score.

An example from the **core** project, comparing two files and their corresponding Stickiness Scores and software metrics, is provided in Table 7.1. To analyze the correlation between a file's Stickiness Score and its contributor count, other software metrics were intentionally chosen to be similar. Furthermore, the selected files are from the same folder, ensuring a fair comparison by avoiding the influence of differing file purposes. For instance, files in folders designated for exporting libraries often experience more frequent changes, which could skew the comparison. Moreover, we adopt this constraint across all our analyses to increase internal validity, ensuring that the observed relationships between variables and Stickiness Score are not driven by variations in file roles or architectural layers.

The file "index.ts" serves as the entry point of the server-renderer package, primarily aggregating and re-exporting modules. Due to this centralized role, it acts as a natural integration point for many components, making it more likely to be touched during development or when new features are added. This leads to a higher contributor count (seven contributors), as various developers interact with it through feature integration or API exposure. However, this breadth of interaction often leads to fragmented knowledge among contributors, as most are familiar with only a small portion of the file's purpose. This limited understanding contributes to its relatively low Stickiness Score of 7.2.

In contrast, the "internal.ts" file contains specific internal logic related to stream rendering and teleports within the server-rendering process. It is more specialized and has fewer interdependencies, making it a less frequently modified file. With only two contributors and a much higher Stickiness Score of 92.31, it exemplifies the benefits of concentrated ownership and domain-specific responsibility. The limited scope and focused functionality of this file likely support better developer familiarity and code retention.

These observations suggest that the negative correlation between contributor count and file Stickiness Score might be attributed to the challenges of coordinating among many contributors, the dilution of ownership, and the disruption of knowledge retention, all of which are amplified in files with higher contributor counts.

Table 7.1: RQ1- **core** Example

File	Stickiness Score	Contributor Count	Average Developer Stickiness Score	Cyclomatic Complexity	Bug-Fix Count	Bug Count	Code Smell Count
packages/server-renderer/src/internal.ts	92.31	2	71.42	0	0	0	0
packages/server-renderer /src/index.ts	7.2	7	73.18	0	0	0	0

A negative correlation between contributor count and file Stickiness Score was observed across all projects, suggesting that files edited by more developers tend to be less stable. This may stem from fragmented ownership, inconsistent coding practices, and reduced knowledge retention.

RQ2.1: Is the average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file?

The correlation between the average Stickiness Scores of contributors and file Stickiness Score was positive in three projects but not significant in the other two.

Table 7.2 presents an example from the **fastapi** project, highlighting two files

that demonstrate a positive correlation between the file's Stickiness Score and the average Stickiness Scores of its contributors. To analyze this relationship, the contributor count, cyclomatic complexity, bug-fix count, and bug and smell counts gathered from static code analysis, as well as the folder paths and file extensions, are intentionally kept constant.

The file "test_forms_from_non_typing_sequences.py" has a high Stickiness Score of 86.79, accompanied by a correspondingly high average developer Stickiness Score of 89.18. This suggests that contributors who are generally writing more stable code for the overall project tend to produce more stable and cohesive changes to the file, contributing positively to the stickiness of individual files. In contrast, the file "test_serialize_response.py" shows a lower Stickiness Score of 69.51 and an average developer Stickiness Score of 62.95. The lower overall contributor familiarity likely results in less consistent ownership and a more fragmented understanding of the file's purpose and structure, which in turn may reduce the file's stickiness.

Table 7.2: RQ2.1- **fastapi** Example

File	Stickiness Score	Average Developer Stickiness Score	Contributor Count	Cyclomatic Complexity	Bug-Fix Count	Bug Count	Code Smell Count
tests/test_serialize_response.py	69.51	62.95	3	6	0	0	0
tests/test_forms_from_non_typing_sequences.py	86.79	89.18	3	6	0	0	0

However, in some projects, this correlation was found to be not statistically significant. The possible reason this hypothesis resulted differently is not using weights such as the number of LOC or the number of commits that the said developer has contributed. When we calculate the average of the developers' Stickiness Scores, regardless of their level of involvement, the correlation analysis can lead to inaccurate results as we treat all contributors equally. For instance, a developer who introduces a single LOC or commit would be given the same weight as someone who contributes 100 LOC or multiple commits, even though the latter possibly has a deeper engagement with the file. This uniform treatment overemphasizes minor contributions, dilutes the impact of core contributors, and

misrepresents developer knowledge and familiarity. Incorporating weights ensures that significant contributions are proportionally reflected, providing a more accurate representation of both file and developer stickiness. An example from the **core** project is presented in Table 7.3. Due to space constraints, the file "packages/server-renderer/src/helpers/ssrRenderComponent.ts" is referred to as A, and "packages/server-renderer/src/helpers/ssrRenderSuspense.ts" as B in the table. Both files have the same average developer Stickiness Scores and similar other software metrics, yet their file Stickiness Scores differ. This suggests the influence of additional factors on file stickiness. Upon calculating the commit-weighted and LOC-weighted average developer Stickiness Scores, we observe a positive correlation between these weighted scores and the file Stickiness Scores, offering a potential explanation for the observed difference.

Table 7.3: RQ2.1- **core** Example

File	Stickiness Score	Average Developer Stickiness Score	Commit-Weighted Average Developer Stickiness Score	LOC-Weighted Average Developer Stickiness Score	Cont. Count	Cyc. Comp.	Bug-Fix Count	Bug Count	Code Smell Count
A	74.07	70.02	70.02	72.68	3	1	1	0	0
B	25.00	70.02	66.39	61.30	3	1	0	0	0

A positive correlation between average developer Stickiness Scores and file stickiness was observed in three projects, but not in two others. This inconsistency may be due to treating all contributors equally, regardless of their level of involvement; using weighted metrics such as LOC or commit count offers a more accurate reflection of developer impact on file stability.

RQ2.2: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of commits contributed to the file?

The commit-weighted Stickiness Scores of contributors demonstrated a positive correlation with file stickiness across all projects. This suggests that when developers with high Stickiness Scores contribute more frequently to a file (i.e., make more commits), the file itself tends to be stickier. This indicates that the stability of a file is influenced not just by who contributes to it, but also by how often stable contributors are involved in its development.

For instance, in the **go-micro** project, as shown in Table 7.4, the file "memory_test.go" has a lower Stickiness Score than "profile.go", which aligns with its lower commit-weighted average developer Stickiness Score. The other software metrics, such as the folder path of these files, file extensions, contributor counts, cyclomatic complexities, bug-fix, bug, and smell counts, remain consistent between these two files.

Table 7.4: RQ2.2- **go-micro** Example

File	Stickiness Score	Commit-Weighted Average Developer Stickiness Score	Contributor Count	Cyclomatic Complexity	Bug-Fix Count	Bug Count	Code Smell Count
debug/log/memory /memory_test.go	50.79	33.04	3	4	0	0	0
debug/profile /profile.go	97.73	65.08	3	4	0	0	0

A positive correlation across all projects suggests that files receiving more commits from stable developers tend to be stickier.

RQ2.3: Is the weighted average of the contributors' Stickiness Scores correlated with the Stickiness Score of the file, where the weight is the number of lines of code contributed to the file?

The LOC-weighted Stickiness Scores of contributors demonstrated a positive correlation with file stickiness across four projects, with the exception being the **go-micro** project.

In comparing the two files, "test_custom_schema_fields.py" and "test_additional_response_extra.py" presented in Table 7.5, we observe that several factors—such as contributor count, cyclomatic complexity, and the bug-fix, bug, and code smell counts remain constant. However, a clear distinction emerges when examining the LOC-weighted average developer Stickiness Score and the file Stickiness Score. "test_custom_schema_fields.py", which achieves a perfect file Stickiness Score of 100, is primarily composed of contributions from developers with high individual Stickiness Scores, and the majority of the code was written by the stickier contributor. In contrast, "test_additional_response_extra.py" has a lower file stickiness score of 77.78, and while it also involves developers with relatively high Stickiness Scores, most of the changes were made by the contributor with the lower of the two. This example illustrates how a higher LOC-weighted developer Stickiness Score correlates with a higher file Stickiness Score, suggesting that files maintained by developers whose code tends to remain unchanged across the project are themselves more stable and less frequently modified.

The lack of correlation in the **go-micro** project may be due to the even distribution of developer contributions across many files, which limited the ability of the LOC-weighted metric to meaningfully differentiate contributor impact at the file level.

Table 7.5: RQ2.3- **fastapi** Example

File	Stickiness Score	LOC-Weighted Average Developer Stickiness Score	Contributor Count	Cyclomatic Complexity	Bug-Fix Count	Bug Count	Code Smell Count
tests/test_additional_response_extra.py	77.78	81.99	2	3	0	0	0
tests/test_custom_schema_fields.py	100	92.01	2	3	0	0	0

A positive correlation in four projects suggests that when more stable contributors contribute with higher volume (LOC) to a file, that file tends to be stickier. In contrast, go-micro showed no such trend, likely due to evenly distributed contributions that diluted the influence of individual contributor stability.

RQ3: Is the file's cyclomatic complexity correlated with its Stickiness Score?

The mixed results for cyclomatic complexity indicate that its impact on the Stickiness Score is highly project-dependent. While a negative correlation was observed in one project, suggesting that higher complexity decreases code stability, a positive correlation in another project implies that complexity can sometimes reflect well-maintained and modular code. Moreover, the other three projects found no significant correlation between the cyclomatic complexity of the file and its Stickiness Score. This finding suggests that complexity metrics should be interpreted with caution and in conjunction with other factors.

In the **core** project example, a comparison between two test files, "ssrSuspense.spec.ts" and "ssrVShow.spec.ts", reveals a negative correlation between cyclomatic complexity and file Stickiness Score. Despite having the same software metrics, "ssrVShow.spec.ts" has a cyclomatic complexity of 12 and a lower file Stickiness Score of 44.19, while "ssrSuspense.spec.ts" has a complexity of 3 and a higher stickiness score of 60. Reviewing the file contents confirms this trend: "ssrVShow.spec.ts" tests a variety of nuanced v-show behaviors, involving multiple test cases and branching logic that likely increase the chance of future modifications. In contrast, "ssrSuspense.spec.ts" includes fewer, more focused test cases with linear structure and limited variation. This supports the idea

that files with higher complexity are more prone to change, leading to lower stickiness over time. Thus, complexity may serve as a signal for the potential low stickiness of the file.

Table 7.6: RQ3- **core** Project Example

File	Stickiness Score	Cyclomatic Complexity	Contributor Count	Average Developer Stickiness Score	Bug-Fix Count	Bug Count	Code Smell Count
packages/compiler-ssr/__tests__/ssrSuspense.spec.ts	60	3	2	71.42	0	0	0
packages/compiler-ssr/__tests__/ssrVShow.spec.ts	44.19	12	2	71.42	0	0	0

In contrast to the **core** project, the **fastapi** project presents an example of a positive correlation between cyclomatic complexity and file Stickiness Score in Table 7.7. Comparing "test_security_http_basic_realm.py" and "test_generate_unique_id_function.py", we observe that the file with greater cyclomatic complexity (33) also has a significantly higher Stickiness Score (98.97) compared to the file with lower complexity (6) and a lower Stickiness Score (67.89). An examination of the file contents substantiates this observation: "test_generate_unique_id_function.py" contains numerous test cases covering edge behaviors, unique ID generation patterns, and complex assertion logic. Despite its complexity, the file has remained largely stable, suggesting it was well-designed early and required minimal revision. In contrast, "test_security_http_basic_realm.py" includes a smaller number of tests with simpler authentication checks, yet has experienced more frequent changes over time. This might also be due to its lower complexity, making the code easier to understand and, consequently, more prone to developer-driven modifications or refinements. This example illustrates that in some contexts, higher complexity may be associated with greater long-term stability.

Table 7.7: RQ3- **fastapi** Project Example

File	Stickiness Score	Cyclomatic Complexity	Contributor Count	Average Developer Stickiness Score	Bug-Fix Count	Bug Count	Code Smell Count
tests/test_security_http_basic_realm.py	67.89	6	2	87.94	0	0	0
tests/test_generate_unique_id_function.py	98.97	33	2	90.7	0	0	0

The relationship between cyclomatic complexity and Stickiness Score varied across projects, with one showing a negative correlation and another a positive one. This suggests that complexity alone does not consistently predict file stability and should be interpreted alongside other contextual factors.

RQ4: Is the number of bug-fixing pull requests that modify a file correlated with its Stickiness Score?

The number of bug-fix count of the file and its Stickiness Score are found to be negatively correlated in the **core** and **fastapi** projects. An example from the **core** project to showcase this correlation is given in Table 7.8. The file "packages/runtime-core/src/renderer.ts" underwent 55 changes due to a GitHub-associated bug, resulting in a relatively low Stickiness Score of 26.81. Given its central role in the rendering process, frequent modifications are expected, as even minor issues can significantly impact the framework's functionality. In contrast, "packages/runtime-core/src/customFormatter.ts", which has remained unchanged with zero modifications, has a high Stickiness Score of 88.09. The key difference between these two files is their susceptibility to bug-related changes—while "renderer.ts" is deeply involved in **core** project's rendering logic and frequently updated to address issues, "customFormatter.ts" remains stable with minimal need for bug-related modifications. This indicates that files frequently modified due to bug fixes tend to retain less of their original structure over time, reinforcing the observed negative correlation in this project.

However, the **go-micro** project did not yield a statistically significant correlation. This is primarily because only 15 out of 254 files had a single change due to a bug fix, while all others remained unchanged. This finding reflects the development practices within the project. Despite having 986 closed issues, only 19 were either formally labeled as bugs or contained "[BUG]" in the title. While some issues were clearly categorized as bugs, features, refactors, or questions, others lacked any such classification in their labels or titles. The inconsistent use of a formal issue labeling process introduced variability in how bugs were identified

and tracked, and impacted the correlation analysis.

Similarly, the **redux** and **apollo-server** projects all had zero changes in their files due to not classifying any issues as bugs, making it impossible to conduct a correlation analysis for this research question in these projects.

Table 7.8: RQ4- **core** Project Example

File	Stickiness Score	Bug-Fix Count
packages/runtime-core/src/renderer.ts	26.81	55
packages/runtime-core/src/customFormatter.ts	88.09	0

A negative correlation between bug-related changes and stickiness was found in core and fastapi, where frequently fixed files due to bugs were less stable. In other projects, inconsistent or missing bug labeling limited data quality, preventing meaningful correlation analysis.

RQ5.1: Is the file's number of bugs collected from static code analysis correlated with its Stickiness Score?

The correlation between a file's number of bug counts collected from SonarQube and its Stickiness Score did not yield statistically significant results across any of the analyzed projects, largely due to the sparse distribution of detected bugs. In **apollo-server**, only 7 out of 115 files contained SonarQube-defined bugs, while in **core**, just 54 out of 509 files had nonzero bug counts. Similarly, in **fastapi**, only 6 out of 1,286 files were flagged, and in **redux**, just 7 out of 190 files showed any SonarQube-detected issues. Moreover, the **go-micro** project had no SonarQube-detected bugs across its 251 files. The low overall detection rates of SonarQube-identified bugs across projects suggest that the dataset lacked sufficient variance to establish a meaningful correlation with the Stickiness Score, ultimately affecting the statistical significance of the analysis.

Due to the low number of SonarQube-detected bugs across all projects, no statistically significant correlation with Stickiness Score was found. The lack of variance in bug counts limited the ability to draw meaningful conclusions from the data.

RQ5.2: Is the file's number of code smells collected from static code analysis correlated with its Stickiness Score?

For this research question, **core** and **fastapi** yielded statistically significant negative correlations between code smell count in the file and its Stickiness Score.

In the **fastapi** project, a negative correlation can be observed between code smell count and file Stickiness Score, as presented in Table 7.9. The two files, "test_security_oauth2_optional.py" and "test_starlette_exception.py", share the same contributor count, have no reported bugs or bug-fix activity, and possess identical cyclomatic complexity, along with similar average developer stickiness scores. Despite these similarities, the "test_starlette_exception.py" has a higher stickiness score than "test_security_oauth2_optional.py". The key differentiating factor is the number of code smells. An examination of the file contents reveals that all five smells in "test_security_oauth2_optional.py" stem from unresolved "TODO" comments, indicating incomplete or deferred implementation work. These placeholders suggest the code is still evolving or awaiting further refinement, which can attract more developer attention and result in more frequent modifications. In contrast, "test_starlette_exception.py" contains no such markers and is more concise and finalized in structure. This example suggests that the presence of code smells can serve as a signal of code instability, potentially lowering file stickiness as the codebase is subject to ongoing revisions.

However, it is worth noting that in three other projects, no statistically significant correlation was observed, indicating that this relationship is not universal. In those cases, architectural style, developer discipline, or consistent refactoring efforts may have decoupled code quality from change frequency, allowing some frequently updated files to maintain high stickiness despite minor smells or vice

versa.

Table 7.9: RQ5.2- **fastapi** Project Example

File	Stickiness Score	Code Smell Count	Contributor Count	Average Developer Stickiness Score	Cyclomatic Complexity	Bug-Fix Count	Bug Count
tests/test_security _oauth2_optional.py	71.05	5	7	71.17	13	0	0
tests/test_starlette _exception.py	82.3	0	7	74.47	13	0	0

Code smells can indicate lower stickiness in some projects, as demonstrated by core and fastapi projects, where frequent changes lead to technical debt. However, this is not universal, as other projects show no clear correlation, which might be related to disciplined development practices during code changes.

7.2 Implications to Practitioners

Stickiness Scores metric and Devotion, which is developed to calculate the scores, offers a range of practical benefits for software practitioners, providing actionable insights into code stability, developer contributions, and project management. Below are its implications explained in detail:

- **Assessment of Code Stability:** One of the primary functions of the Devotion tool is to provide Stickiness Scores for each file and folder in a software project. These scores represent the long-term stability of the code, allowing practitioners to evaluate how consistently certain parts of the codebase remain unchanged. This feature is particularly valuable in large projects where maintaining stability is critical for scalability and maintainability.
- **Identification of Unstable Areas in the Codebase:** The tool helps identify specific files and folders with low Stickiness Scores, which indicate they are frequently modified or prone to instability. These unstable areas are often sources of bugs, performance issues, or architectural problems.

Recognizing these parts of the code early allows developers to focus their attention on reducing volatility and improving code quality.

- **Prioritization of Stabilization Efforts:** By highlighting files and folders that are high-risk or unstable, the Devotion tool enables teams to prioritize their stabilization efforts effectively. Instead of addressing every issue uniformly, teams can target the areas that have the greatest impact on the overall stability and reliability of the software, thereby optimizing time and resources.

- **Evaluation of Developer Contributions:** Beyond assessing code stability, the tool provides insights into the contributions of individual developers. It evaluates not just the volume of contributions but also their quality by analyzing how "sticky" the code written by a developer is. High stickiness in a developer's contributions might indicate robust and enduring code, which is a key indicator of their technical expertise and understanding of best practices.

- **Identification of Key Developers and Core Contributors**: The Stickiness Scores, when paired with data on the quantity and type of contributions, allow teams to identify developers who consistently create stable code. These developers may be the core contributors who play a critical role in the success of the project. Recognizing such individuals helps organizations ensure that these key developers remain engaged and rewarded for their contributions.

- **Ownership of Critical Files:** Stickiness Scores can also inform file ownership. For instance, developers with a history of creating stable, high-quality code can be assigned ownership of critical files or modules. Additionally, these insights can be used for performance reviews, helping managers evaluate developer impact objectively and fairly. This information may also guide reorganization efforts to ensure the right team members are working on the most critical or complex parts of the project.

- **Support for Long-Term Project Health and Sustainability:** By systematically analyzing Stickiness Scores and using the data to guide decisions, teams can improve the long-term health and sustainability of their software. Regularly addressing unstable areas and recognizing the contributions of key developers reduces the likelihood of technical debt and ensures the project remains manageable and scalable as it grows.

By incorporating these insights into their workflows, software practitioners can address challenges related to code stability and optimize resource allocation.

7.3 Implications to Researchers

This study provides several meaningful contributions and opens new avenues for researchers to explore. The implications are outlined below:

- **Introduction of Stickiness Score as a Metric:** This study introduces the Stickiness Score as a novel and quantifiable metric for assessing code stability. It creates opportunities for researchers to further explore its applications in various contexts, refine its calculation methodology, and validate its effectiveness across different types of software projects. This metric offers a fresh perspective for analyzing code evolution and stability over time.
- **Developer Contribution Patterns:** The findings of this study emphasize the significant role of developer contribution patterns in shaping the longevity and stability of code. By analyzing how these factors influence Stickiness Scores, researchers can investigate developer behaviors and their effects on software quality and sustainability.
- **Exploration of Related Factors:** Stickiness Scores may also interact with other factors, such as code reviews and test coverage. Future research could investigate how these elements interplay with Stickiness Scores to

provide a more holistic view of software health. Understanding these relationships could lead to the development of more comprehensive metrics for software evaluation.

- **Broadening the Application of Stickiness Scores:** The Devotion tool and its Stickiness Score metric serve as a foundation for further research. Researchers can apply these concepts to a broader range of software projects, encompassing various domains and development methodologies. Additionally, Stickiness Scores can be explored as predictive indicators of long-term project outcomes, such as maintainability, scalability, and developer productivity, offering valuable insights into project lifecycle management.
- **Extending Stickiness Scores to AI-Generated Code:** The findings of this study can be broadened for detecting Stickiness Scores of AI-generated code, which is, to the best of our knowledge, an area that remains relatively unexplored. By adapting the Stickiness Score framework to account for AI-written code, researchers can gain deeper insights into the evolution and longevity of such contributions, as well as the extent of human intervention required. By analyzing this score, one can infer the extent to which developers trust AI-generated code, either because they perceive it to be of high quality and requiring minimal modification or because they are reluctant to alter it due to its perceived correctness or authority.

These implications encourage researchers to not only adopt the Stickiness Score as a tool for studying software stability but also to build on this foundation to investigate its broader potential.

Chapter 8

Threats to Validity

This section outlines the possible limitations that could affect the accuracy of the study's findings.

8.1 Internal Threats

Since Devotion uses Git commands to identify authorship, it inherits Git's limitations. For instance, in preprocessing, Devotion fuzzy matches developer names with email handles to track authorship, recognizing them as the same person if the match is at least 90%. If the same author does not match the fuzzy criteria, they are tracked as different developers, which may impact our calculations.

Tracking edge cases such as code duplications, movements, and revivals may lead to overly detailed tracking, especially for common code blocks such as loop declarations. However, assuming that developers follow the DRY principle, these scenarios become meaningful. Extreme Stickiness Scores can signal low maintenance or excessive edits. While not a standalone maintainability measure, it serves as a useful guideline for identifying potential issues.

Devotion tracks edge cases for LOCs with at least 20 alphanumeric characters.

This limitation may affect calculations for lines with less alphanumeric content. However, for common code blocks with less than 20 characters, tracking edge cases such as code duplications, movements, and revivals can lead to excessive detail and thus, computational expense.

Coding styles, such as splitting lines for readability, can affect Stickiness Scores, however, Devotion captures these whitespace changes without influencing the score. Smaller changes, such as adding parentheses, impact the score. Despite these, the Stickiness Score remains a valuable indicator of code persistence and maintainability, offering insights into the codebase's evolution.

Devotion traverses the commit history topologically, processing sub-branches individually. Without topological ordering, LOCs moved to one sub-branch but were deleted in another might be misinterpreted as code revivals. This approach ensures a coherent analysis of each subbranch, leading to a more accurate understanding of code evolution [25]. Despite these efforts, if sub-branches delete the same LOC, Devotion tracks them as separate deletions, potentially affecting Stickiness Scores. To the best of our knowledge, distinguishing such cases using Git commands is not possible due to Git's branch-focused development model. This corner case caused some data points in the Stickiness Score to fall outside the valid range (i.e., below zero or above 100) due to the same action being counted multiple times. For the correlation analysis, we manually corrected these invalid points to zero and 100, respectively. These cases represent a small fraction of the overall dataset, as summarized in Table 8.1. The table presents the total number of developers and files for each project, and the counts of those with out-of-bounds Stickiness Scores. It is worth noting that the number of developers differs from Table 6.1 because some contributors have been merged based on Devotion's 90% similarity threshold. As part of our future work, we plan to identify and address these out-of-bound Stickiness Scores for both developers and files.

In addition, the choice between the Spearman and Pearson correlation tests depends on assessing the normality of the data. Multiple statistical tests were used for this purpose, including the Shapiro-Wilk, Kolmogorov-Smirnov, Jarque-Bera,

Project	Developers		Files	
	Total	Out-of-Bounds	Total	Out-of-Bounds
core	497	1	574	0
fastapi	694	0	1353	0
go-micro	183	0	254	1
redux	859	4	316	1
apollo-server	492	16	131	1

Table 8.1: Developers and Files with Out-of-Bound Scores Across Projects

Anderson-Darling, and D'Agostino's K^2. While these tests comprehensively assess normality, their inherent limitations are acknowledged. Additionally, Spearman assumes monotonicity and Pearson assumes linearity, which may not fully capture real-world complexity.

Moreover, the correlation analyses include auto-generated files, which are not authored by developers. As a result, any associated bug counts, code smells, or similar metrics originating from these files do not accurately reflect developer activity or intent and ideally should have been excluded. However, since our study focuses on OSS projects where it is difficult to reliably identify and filter out autogenerated files, they remain part of the dataset. This inclusion may have introduced noise into the correlation results and potentially affected their accuracy.

8.2 External Threats

We aimed to choose open-source GitHub projects with domain variation to ensure generalizability. However, these repositories differ in terms of software lifecycle maintenance approaches, which may affect the interpretation of the results. Additionally, some repositories had a limited data size for some of the research questions. These assumptions and potential limitations are acknowledged as part of the study's scope and may have implications for the validity of the results. While our study analyzes popular OSS projects from GitHub across diverse domains, such as frontend, backend, and microservices, it may not fully represent the broader OSS ecosystem, potentially limiting the generalizability of the findings. Furthermore, the findings may not fully generalize to proprietary projects,

where different development practices and maintenance strategies could lead to different outcomes.

Chapter 9

Conclusion

This study introduces the *Stickiness Score* as a novel metric for evaluating the persistence and stability of code over time and presents *Devotion*, a tool that calculates Stickiness Scores for developers, files, and folders within a project. By analyzing five diverse open-source repositories, we investigated the relationship between Stickiness Scores and a variety of software metrics to gain a deeper understanding of how developers and code quality indicators affect code stickiness.

We addressed eight research questions grouped into five main categories:

- **RQ1:** We examined whether the number of contributors to a file correlates with its Stickiness Score. Across all projects, a strong negative correlation was observed, indicating that files edited by more developers tend to be less stable.
- **RQ2:** We explored how contributor behavior influences file stickiness.
  - **RQ2.1:** A positive correlation was found in three projects between a file's Stickiness Score and the unweighted average Stickiness Score of its contributors. Two projects, however, showed no significant relationship.

  - **RQ2.2:** When contributor Stickiness Scores were weighted by the number of commits, a stronger and consistent positive correlation was observed across all projects.
  - **RQ2.3:** Similarly, weighting Stickiness Scores by the number of lines of code contributed revealed a positive correlation in four out of five projects.

- **RQ3:** We investigated whether cyclomatic complexity correlates with file Stickiness Scores. The results were mixed and project-dependent, with both weak positive and negative correlations observed, while some projects showed no significant relationship.

- **RQ4:** We evaluated whether the number of bug-related changes through GitHub pull requests impacts stickiness. Two projects demonstrated a negative correlation, suggesting that files with frequent bug-fix activity tend to make files less stable. Other projects lacked sufficient or consistent labeling to produce reliable results.

- **RQ5:** We focused on static code analysis metrics:
  - **RQ5.1:** No significant correlation was found between SonarQube-detected bug counts and Stickiness Scores, primarily due to low variance in detected bugs across projects.
  - **RQ5.2:** A negative correlation between code smell count and Stickiness Score was observed in two projects, implying that files with more code smells may be more volatile. However, this trend was not consistent across all datasets.

These findings suggest that code stickiness is influenced by both *social* factors, such as contributor collaboration patterns, and *technical* aspects, including code quality and complexity. Specifically, high contributor turnover and frequent bug-related changes are associated with lower stickiness, while contributions from stable developers and low-smell code are more likely to endure.

The Stickiness Score provides a complementary lens to traditional churn-based metrics, capturing persistence and code survival while handling edge cases such as duplication, movement, and revival. Future work may extend this metric to AI-generated code, explore causation, and refine the model using additional metrics such as code review history and test coverage.

By bridging code evolution research with actionable insights, this work empowers both researchers and practitioners to help identify sticky components, recognize key contributors, and foster sustainable development practices grounded in empirical evidence.

Bibliography

[1] M. Lehman and J. C. Fernáandez-Ramil, "Software evolution," *Software evolution and feedback: Theory and practice*, pp. 7–40, 2006.

[2] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *2008 Frontiers of Software Maintenance*, pp. 129–138, IEEE, 2008.

[3] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[4] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Proceedings Fourth International Software Metrics Symposium*, pp. 20–32, 1997.

[5] N. E. Fenton and M. Neil, "Software metrics: roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 357–370, 2000.

[6] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, *Analysing Software Repositories to Understand Software Evolution*, pp. 37–67. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[7] E. E. Mills, *Software metrics*. Software Engineering Institute, 1988.

[8] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, vol. 54, no. 2, pp. 111–118, 2000. Special Issue on Software Maintenance.

[9] J. Munson and S. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 24–31, 1998.

[10] A. Meneely and O. Williams, "Interactive churn metrics: socio-technical variants of code churn," *SIGSOFT Softw. Eng. Notes*, vol. 37, p. 1–6, nov 2012.

[11] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, (New York, NY, USA), p. 83–92, Association for Computing Machinery, 2011.

[12] E. Kocaguneli, A. T. Misirli, B. Caglayan, and A. Bener, "Experiences on developer participation and effort estimation," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 419–422, 2011.

[13] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[14] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 65–74, 2013.

[15] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 284–292, 2005.

[16] C. Faragó, P. Hegedűs, and R. Ferenc, "Cumulative code churn: Impact on maintainability," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 141–150, IEEE, 2015.

[17] A. Tornhill, "Code maat." `https://github.com/adamtornhill/code-maat`. Accessed: January 6, 2025.

[18] N. Gerner, "Code review and churn." `https://blogautodevtech.wordpress.com/2021/07/27/code-review-and-churn/`, 2021. Accessed: January 6, 2025.

[19] N. Lopez and A. van der Hoek, "The code orb: supporting contextualized coding via at-a-glance views (nier track)," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 824–827, 2011.

[20] B. Braunschweig, N. Dhage, M. J. Viera, C. Seaman, S. Sampath, and G. A. Koru, "Studying volatility predictors in open source software," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, (New York, NY, USA), p. 181–190, Association for Computing Machinery, 2012.

[21] Y. Bugayenko, "Volatility metric to detect anomalies in source code repositories," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code*, BCNC 2021, (New York, NY, USA), p. 1–4, Association for Computing Machinery, 2021.

[22] H. C. Benestad, B. Anda, and E. Arisholm, "Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems," *Empirical Software Engineering*, vol. 15, pp. 166–203, 2010.

[23] R. J. Costello and D.-B. Liu, "Metrics for requirements engineering," *Journal of Systems and Software*, vol. 29, no. 1, pp. 39–63, 1995. Oregon Metric Workshop.

[24] M. H. de Moura, H. A. do Nascimento, and T. C. Rosa, "Extracting new metrics from version control system for the comparison of software developers," in *2014 Brazilian Symposium on Software Engineering*, pp. 41–50, 2014.

[25] D. Spinellis, P. Louridas, and M. Kechagia, "Software evolution: the lifetime of fine-grained elements," *PeerJ Computer Science*, vol. 7, p. e372, 2021.

[26] D. Spinellis, "A repository of unix history and evolution," *Empirical Software Engineering*, vol. 22, pp. 1372–1404, 2017.

[27] E. Bern, "Git-of-theseus." `https://github.com/erikbern/git-of-theseus`. Accessed: January 6, 2025.

[28] V. Markovtsev, "Hercules insights." `https://github.com/marketplace/actions/hercules-insights`. Accessed: January 6, 2025.

[29] D. Stenberg, "Source code survival rate." `https://daniel.haxx.se/blog/2013/11/05/source-code-survival-rate/`, 2013. Accessed: January 6, 2025.

[30] D. Stenberg and Contributors, "curl: Command-line tool for data transfers." `https://curl.se/`. Accessed: January 6, 2025.

[31] M. Staron, J. Hansson, R. Feldt, A. Henriksson, W. Meding, S. Nilsson, and C. Höglund, "Measuring and visualizing code stability – a case study at three companies," in *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*, pp. 191–200, 2013.

[32] D. Kelly, "A study of design characteristics in evolving software using stability as a criterion," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 315–329, 2006.

[33] S. Yau and J. Collofello, "Some stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545–552, 1980.

[34] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 11–19, IEEE, 2017.

[35] S. Karus and M. Dumas, "Code churn estimation using organisational and code metrics: An experimental comparison," *Information and Software Technology*, vol. 54, no. 2, pp. 203–211, 2012.

[36] M. Salama, R. Bahsoon, and P. Lago, "Stability in software engineering: Survey of the state-of-the-art and research directions," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1468–1510, 2021.

[37] "Git log documentation." `https://git-scm.com/docs/git-log`. Accessed: 2025-01-06.

[38] "Git blame documentation." `https://git-scm.com/docs/git-blame`. Accessed: 2025-01-06.

[39] "Fuzzywuzzy, fuzzy string matching in python." `https://pypi.org/project/fuzzywuzzy/`. Accessed: January 6, 2025.

[40] N. Lee, "Inventor's moral right and morality of patents," in *Research Handbook on Intellectual Property and Moral Rights*, pp. 96–112, Edward Elgar Publishing, 2023.

[41] R. Chattamvelli, *Correlation in engineering and the applied sciences: Applications in R*. Springer Nature, 2024.

[42] N. J. Gogtay and U. M. Thatte, "Principles of correlation analysis," *Journal of the Association of Physicians of India*, vol. 65, no. 3, pp. 78–81, 2017.

[43] P. Sedgwick, "Pearson's correlation coefficient," *Bmj*, vol. 345, 2012.

[44] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & analgesia*, vol. 126, no. 5, pp. 1763–1768, 2018.

[45] J. Lee Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, 1988.

[46] K. Ali Abd Al-Hameed, "Spearman's correlation coefficient in statistical analysis," *International Journal of Nonlinear Analysis and Applications*, vol. 13, no. 1, pp. 3249–3255, 2022.

[47] N. M. Razali, Y. B. Wah, *et al.*, "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[48] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[49] R. Chicheportiche and J.-P. Bouchaud, "Weighted kolmogorov-smirnov test: Accounting for the tails," *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, vol. 86, no. 4, p. 041115, 2012.

[50] C. M. Jarque and A. K. Bera, "A test for normality of observations and regression residuals," *International Statistical Review/Revue Internationale de Statistique*, pp. 163–172, 1987.

[51] L. S. Nelson, "The anderson-darling test for normality," *Journal of Quality Technology*, vol. 30, no. 3, pp. 298–299, 1998.

[52] R. B. D'agostino, A. Belanger, and R. B. D'Agostino Jr, "A suggestion for using powerful and informative tests of normality," *The American Statistician*, vol. 44, no. 4, pp. 316–321, 1990.

[53] "Lizard, an extensible cyclomatic complexity analyzer." `https://pypi.org/project/lizard/`.

[54] GitHub, Inc., "Github rest api v3." `https://docs.github.com/en/rest`, 2024.

[55] SonarSource, "Sonarqube." `https://www.sonarsource.com/products/sonarqube`, 2024.

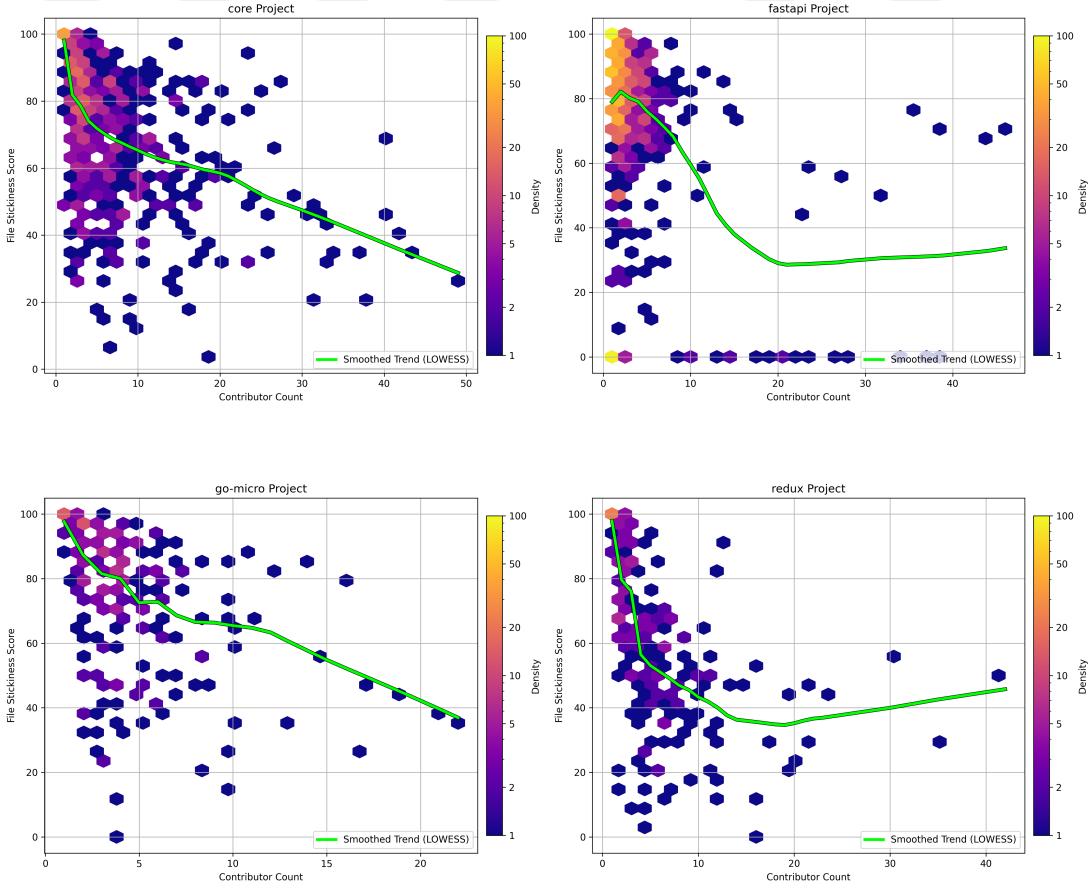
[56] C. Pearson's, "Comparison of values of pearson's and spearman's correlation coefficients," *Comparison Of Values Of Pearson's And Spearman's Correlation Coefficients*, 2011.

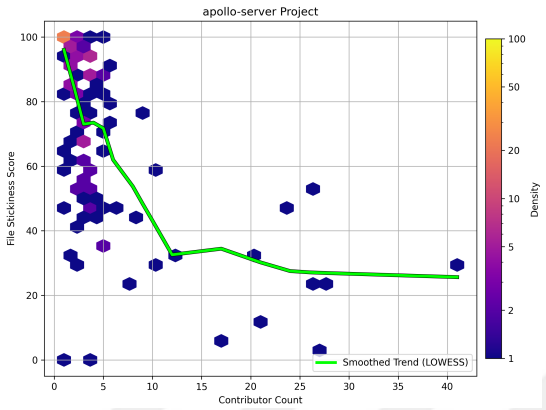
[57] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

Appendix A

RQ1

Figure A.1: Density Plots of Contributor Count vs File Stickiness Score

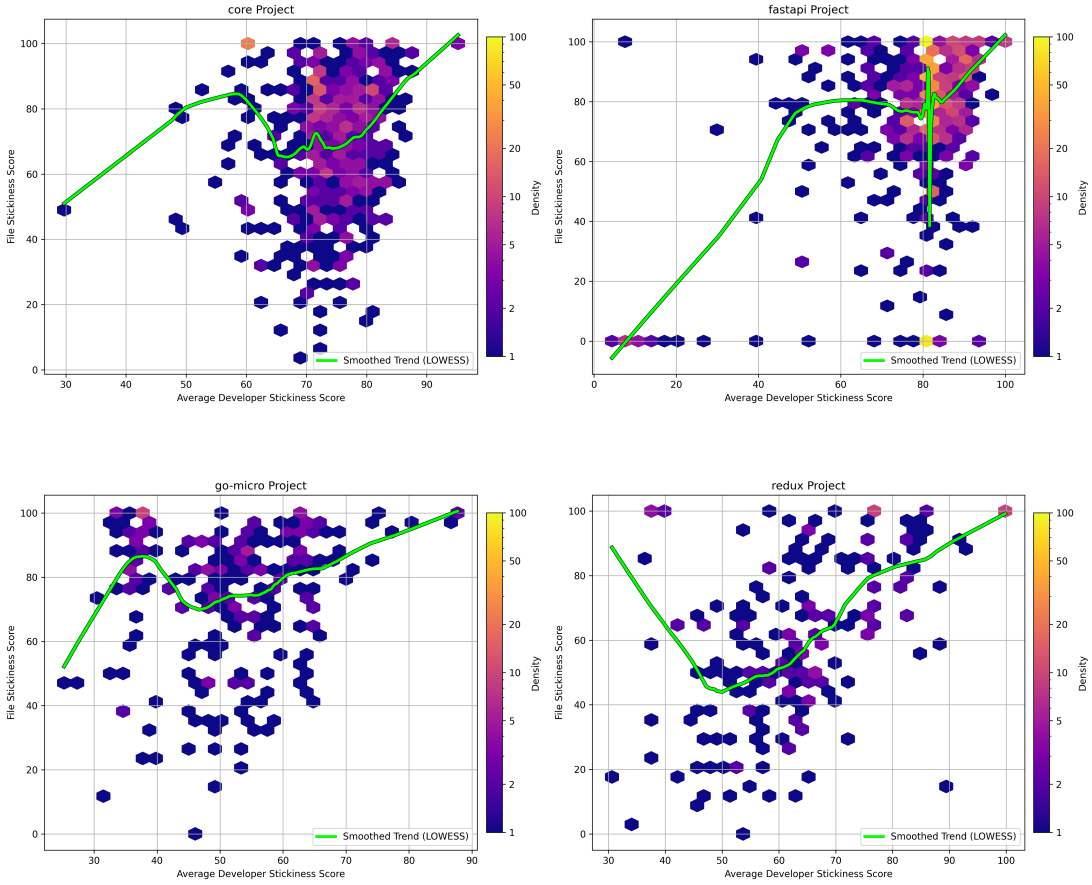


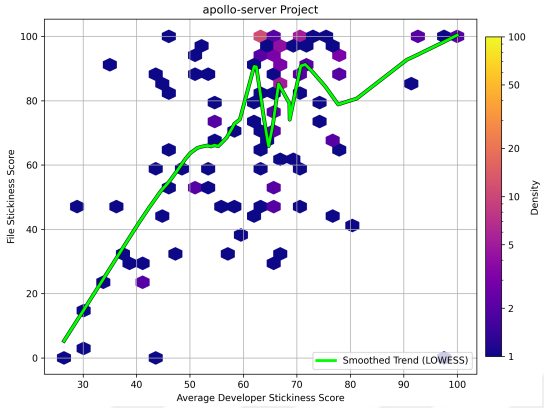

apollo-server Project
File Stickiness Score
Contributor Count
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40
1
2
5
10
20
50
100

Appendix B

RQ2.1

Figure B.1: Density Plots of Average Developer Stickiness Score vs File Stickiness Score

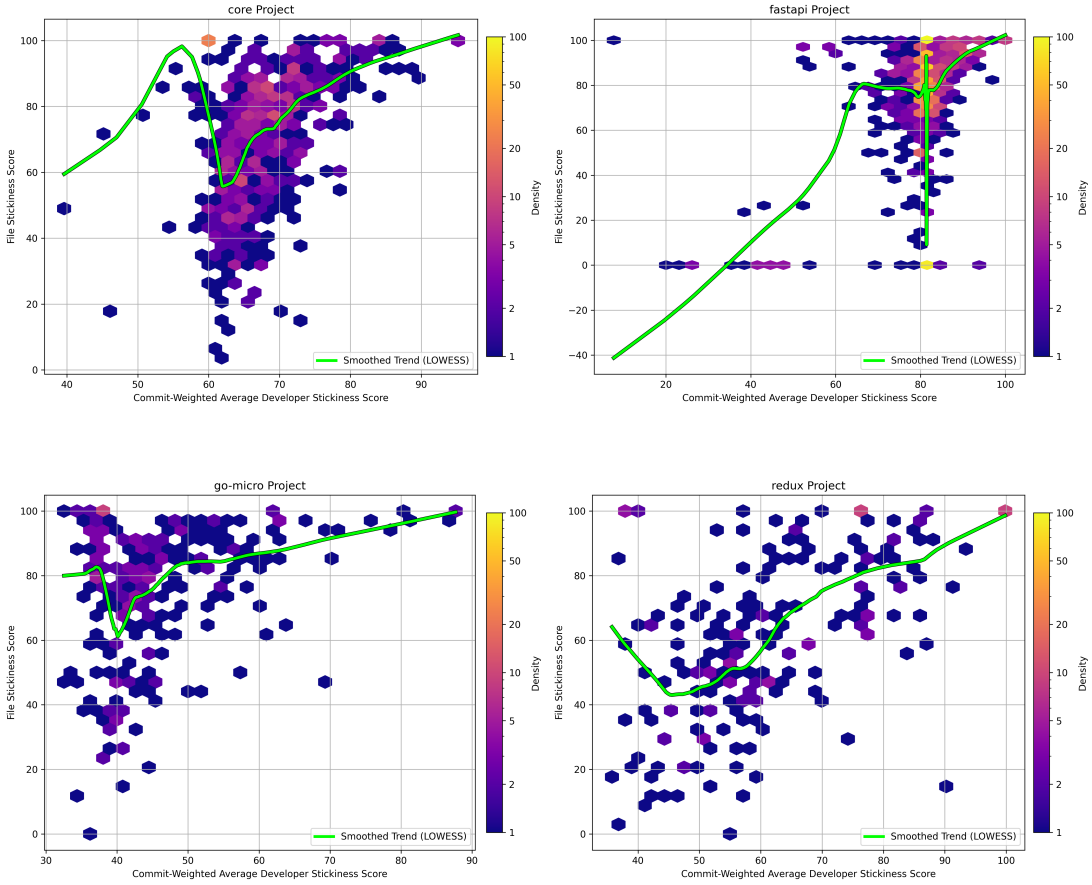


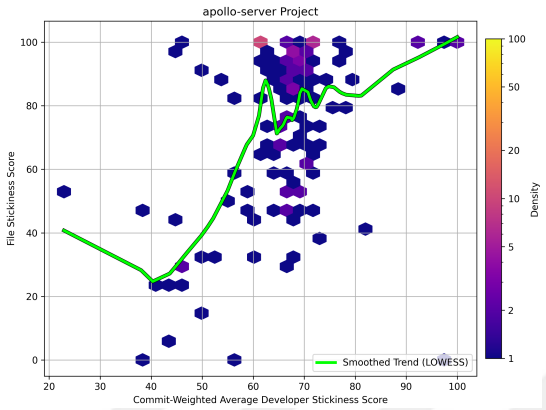

apollo-server Project
File Stickiness Score
Average Developer Stickiness Score
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
30
40
50
60
70
80
90
100
1
2
5
10
20
50
100

Appendix C

RQ2.2

Figure C.1: Density Plots of Commit-Weighted Average Developer Stickiness Score vs File Stickiness Score

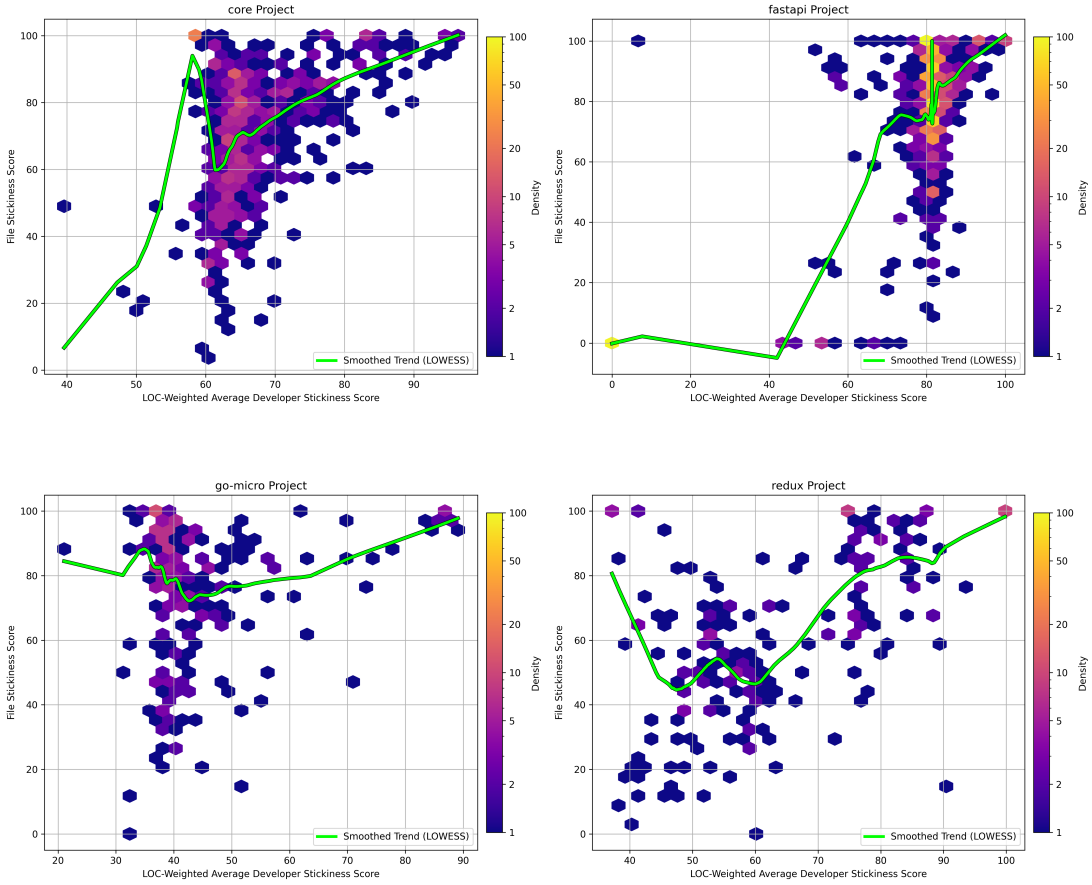


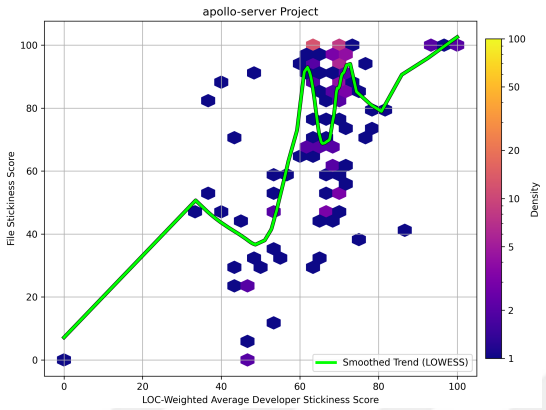

apollo-server Project
File Stickiness Score
Commit-Weighted Average Developer Stickiness Score
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
20
30
40
50
60
70
80
90
100
1
2
5
10
20
50
100

Appendix D

RQ2.3

Figure D.1: Density Plots of LOC-Weighted Average Developer Stickiness Score vs File Stickiness Score

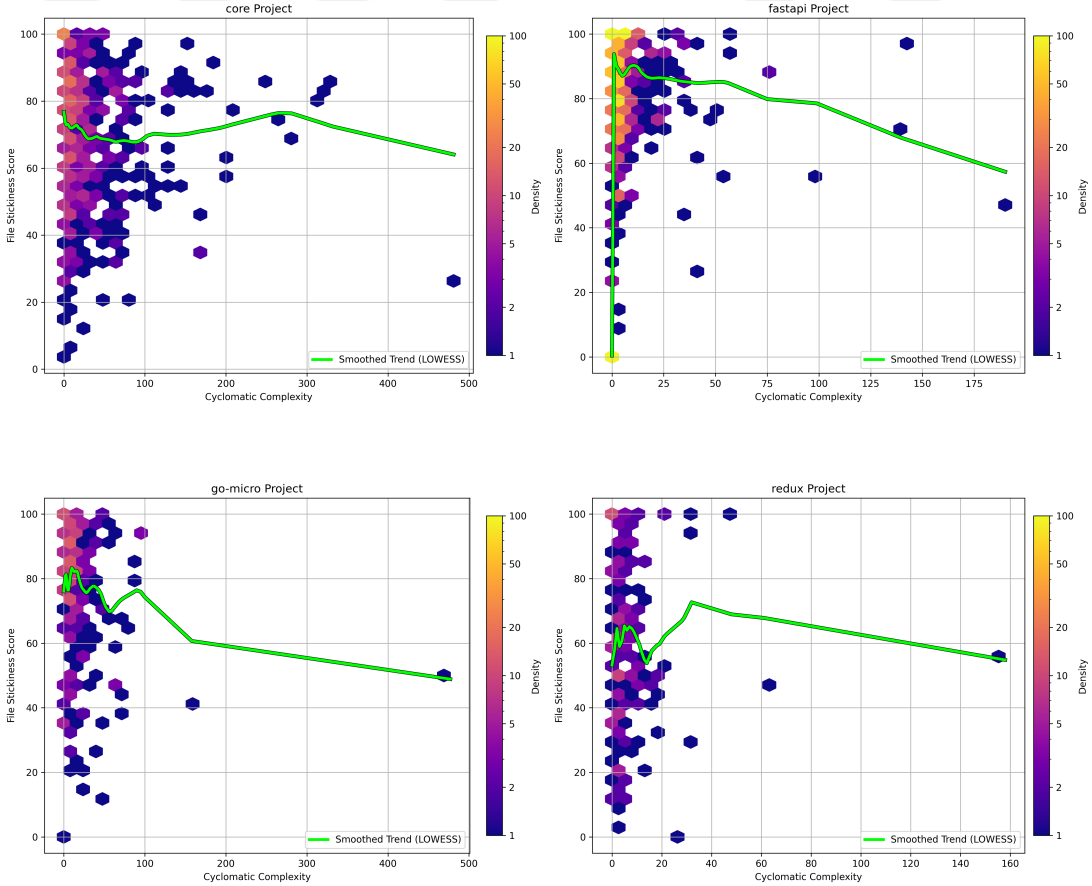


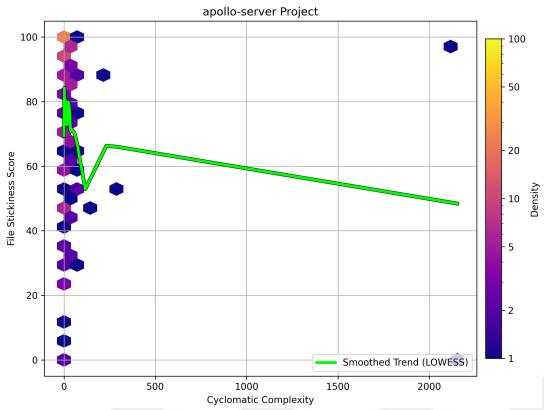

apollo-server Project
File Stickiness Score
LOC-Weighted Average Developer Stickiness Score
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
1
2
5
10
20
50
100

Appendix E

RQ3

Figure E.1: Density Plots of Cyclomatic Complexity vs File Stickiness Score

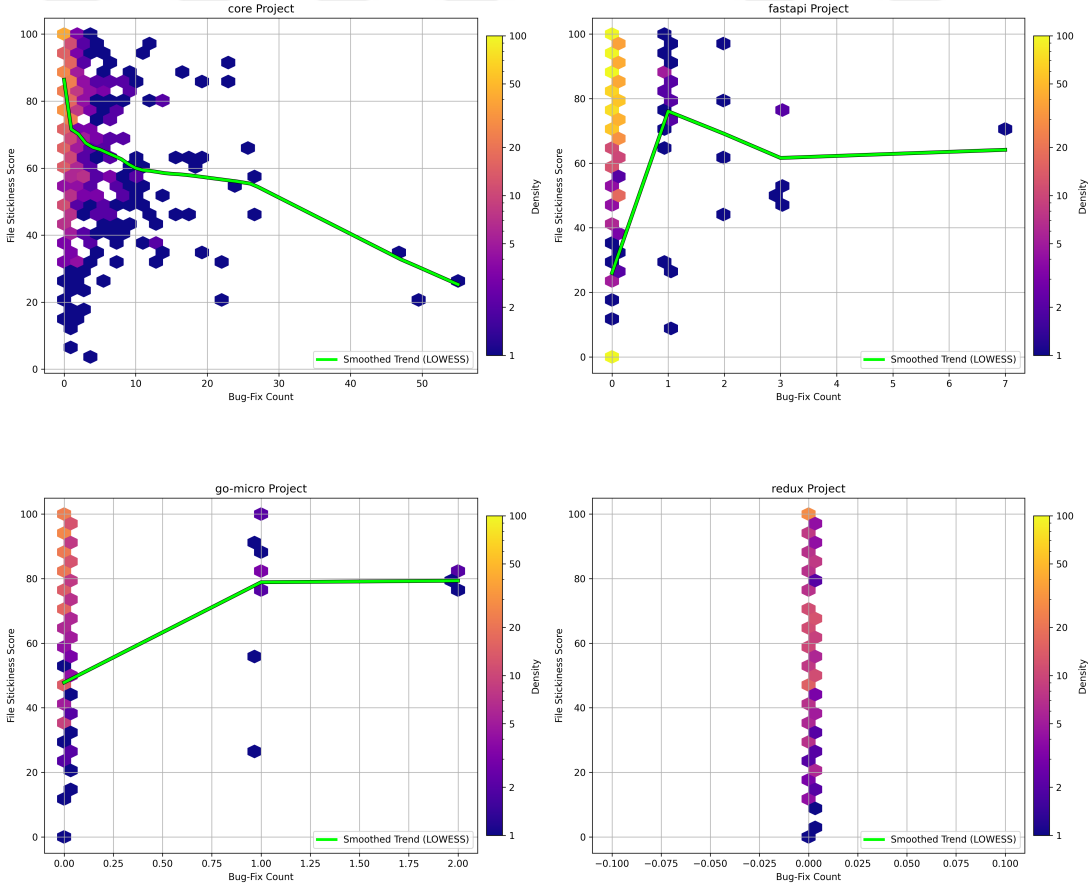


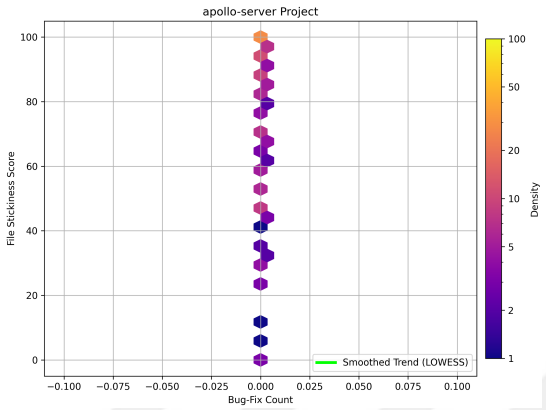

apollo-server Project
File Stickiness Score
Cyclomatic Complexity
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
0
500
1000
1500
2000
1
2
5
10
20
50
100

Appendix F

RQ4

Figure F.1: Density Plots of Bug-Fix Count vs File Stickiness Score

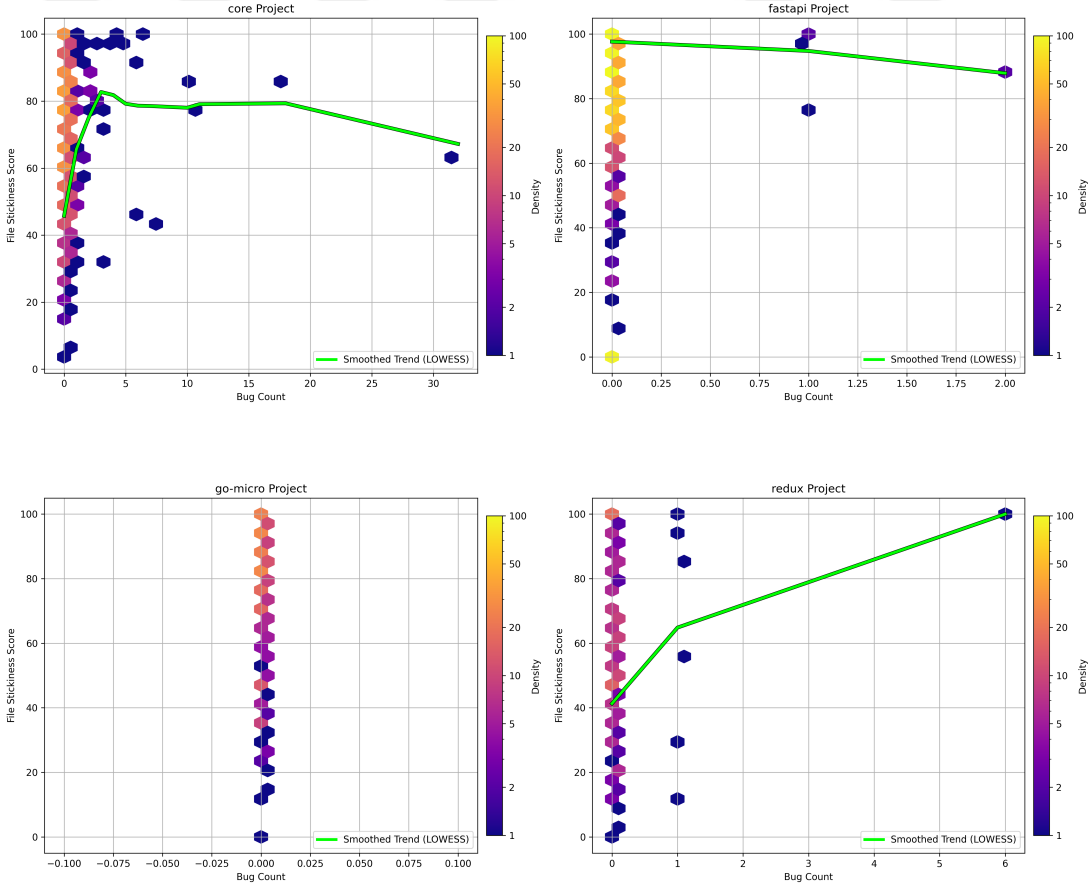


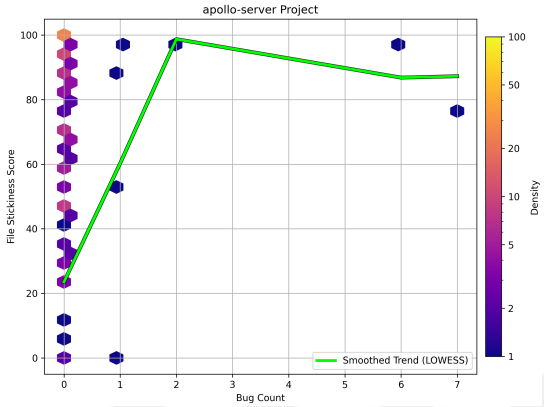

apollo-server Project
File Stickiness Score
Bug-Fix Count
Density
Smoothed Trend (LOWESS)
100
80
60
40
20
0
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100
100
50
20
10
5
2
1

Appendix G

RQ5.1

Figure G.1: Density Plots of Bug Count vs Stickiness Score

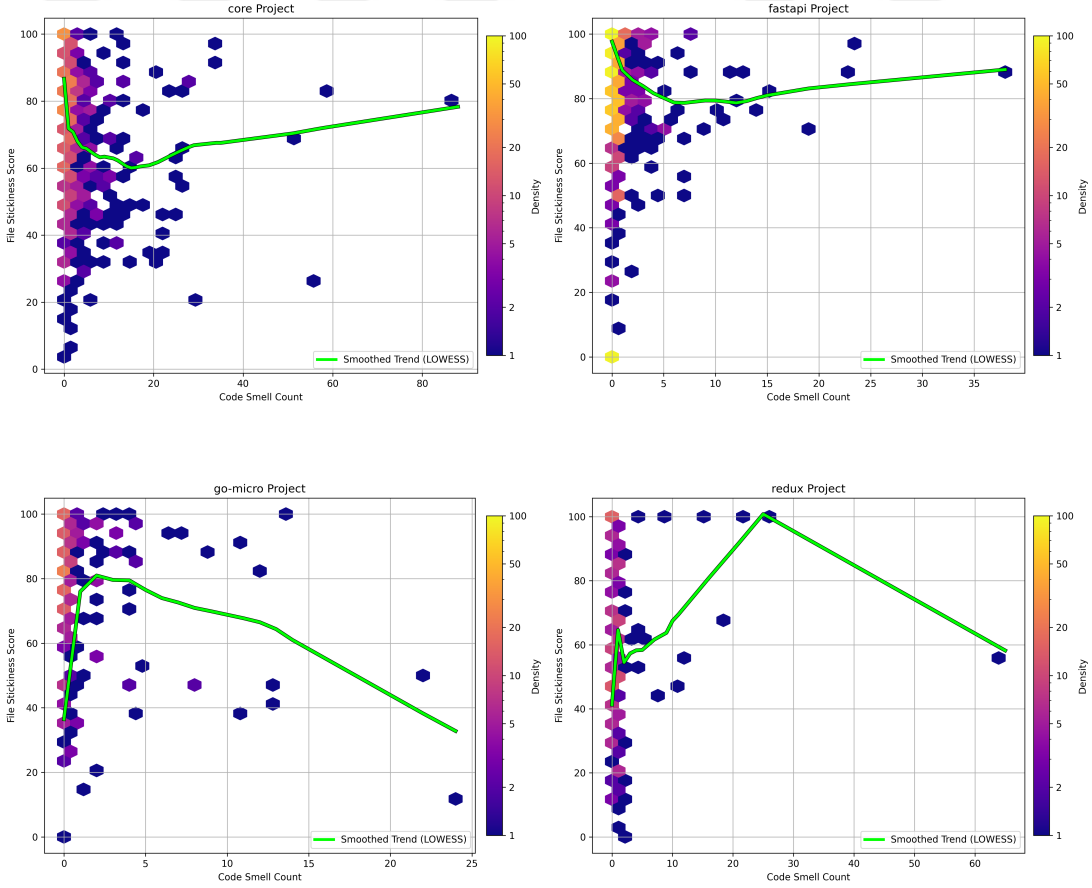


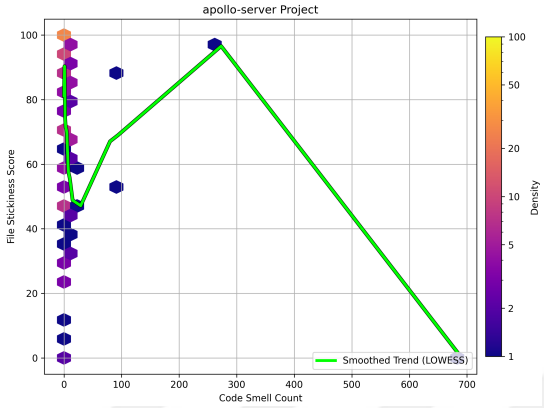

apollo-server Project
File Stickiness Score
Bug Count
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
0
1
2
3
4
5
6
7
1
2
5
10
20
50
100

Appendix H

RQ5.2

Figure H.1: Density Plots of Code Smell Count vs Stickiness Score




apollo-server Project
File Stickiness Score
Code Smell Count
Density
Smoothed Trend (LOWESS)
0
20
40
60
80
100
0
100
200
300
400
500
600
700
1
2
5
10
20
50
100