https://doi.org/10.31891/2307-5732-2025-353-42
УДК 004.415

YATSENKO ROMAN
Lviv Polytechnic National University
https://orcid.org/0009-0009-4662-7873
e-mail: roman.o.yatsenko@lpnu.ua
SERDIUK PAVLO
Lviv Polytechnic National University
https://orcid.org/0000-0002-2677-3170
e-mail: pavlo.v.serdiuk@lpnu.ua

EFFECT OF FILE EDITING FREQUENCY ON SOFTWARE QUALITY

Understanding the impact of file editing frequency on software quality is crucial for maintaining and improving software reliability and maintainability. Software quality is a multifaceted concept that involves various aspects such as code maintainability, reliability, and adherence to principles like the Single Responsibility Principle (SRP). The Single Responsibility Principle emphasizes that a class or module should have only one reason to change, suggesting that high-quality software should be modular and focused.

Frequent edits to files in a GIT repository can be indicative of underlying issues such as evolving requirements, poor initial design, or violations of SRP, leading to an increase in technical debt. This debt can manifest in the form of code that is difficult to maintain, prone to bugs, and challenging to extend. While existing research has explored the relationship between file editing frequency and software quality, this area remains under-explored, particularly in the context of the cumulative effect of these edits on long-term software quality.

The objective of this study is to establish a more definitive link between the frequency of file edits and software quality. This will be achieved by introducing a new metric, the Consecutive File Edit Coefficient (CFE), and using it alongside traditional software quality metrics to analyze and compare the impact of frequent file edits. By doing so, this research aims to provide insights that can inform better practices in software development and maintenance.

The comparative analysis across these open-source repositories reveals consistent patterns that highlight the risks associated with frequent file edits. High CFE values are often accompanied by increased complexity, more code smells, higher violations, and a greater number of bugs.

Keywords: Software quality, Consecutive file edit coefficient, Technical debt

ЯЦЕНКО РОМАН
СЕРДЮК ПАВЛО
Національний університет «Львівська політехніка»

ВПЛИВ ЧАСТОТИ РЕДАГУВАННЯ ФАЙЛУ НА ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розуміння впливу частоти редагування файлів на якість програмного забезпечення є надзвичайно важливим для підтримки та покращення надійності й супровідності коду. Якість програмного забезпечення — це багатогранне поняття, що охоплює аспекти супровідності, надійності та дотримання принципів проєктування, зокрема Принципу єдиної відповідальності (SRP). Згідно з SRP, кожен клас або модуль має мати лише одну причину для внесення змін, що свідчить про важливість модульності та чіткого фокусу у високоякісному програмному забезпеченні.

Часте редагування файлів у GIT-репозиторії може вказувати на приховані проблеми, такі як швидко змінювані вимоги, недосконалий початковий дизайн чи порушення SRP, що зумовлює зростання технічного боргу. Такий борг може проявлятися у складності супроводу коду, схильності до помилок і труднощах із подальшим розширенням. Хоча попередні дослідження вже розглядали зв'язок між частотою редагування файлів і якістю ПЗ, ця тема досі недостатньо висвітлена, особливо щодо кумулятивного впливу повторних змін на довгострокову якість.

Метою цього дослідження є більш чітке встановлення взаємозв'язку між частотою редагування файлів і якістю програмного забезпечення. З цією метою запропоновано нову метрику — Коефіцієнт послідовного редагування файлу (CFE). Поєднуючи її з традиційними метриками якості, у дослідженні буде проведено аналіз і порівняння впливу частих змін файлів на показники якості ПЗ. Очікується, що результати нададуть розробникам цінні рекомендації щодо кращих практик проєктування та супроводу, спрямованих на підвищення стабільності та надійності програмного забезпечення.

Ключові слова: якість програмного забезпечення, коефіцієнт послідовного редагування, технічний борг

Стаття надійшла до редакції / Received 09.04.2025
Прийнята до друку / Accepted 05.05.2025

Problem Statement

In today's context, ensuring high software quality is critically important for its stability and maintainability. Frequent file modifications in version control systems such as GIT often indicate issues in the initial design and the accumulation of technical debt.

Traditional analysis methods do not always account for the cumulative impact of repeated changes, creating a need for a new metric — the Consecutive File Edit coefficient (CFE)**.** This metric will enable quantitative assessment of the impact of repeated modifications on the structural and cognitive complexity of code, thus contributing to the timely identification of problematic areas and optimization of development processes.

Analysis of research and publications

Modern research in the field of software quality measurement demonstrates a variety of approaches to defining and aggregating metrics that reflect both the internal structure of code and its external interactions.

According to Dalla Palma et al. (2020) [5], a catalog of 46 metrics specifically adapted for Infrastructure-as-Code (IaC) was proposed. The authors emphasize that traditional metrics developed for general-purpose programming languages are not always applicable to domain-specific languages used in modern DevOps practices.

Early studies, such as those by Kafura and Henry (1981) [2] and Gaffney Jr. (1981) [8], laid the foundation for the objective measurement of quality through information flow analysis and other indicators.

These studies highlight the importance of automating metric collection at the design stage, enabling the identification of structural issues long before implementation.

In the work by Jiang et al. (2008) [10], a comparative analysis of models based on design-level and code-level metrics for predicting defective modules was conducted. The results show that models based on code metrics generally outperform those using only design metrics, and their combination yields even higher predictive accuracy.

The study by Rosenberg and Hyatt (1997) [1] focuses on the development of specific metrics for object-oriented systems, analyzing both the internal structure of classes (e.g., cyclomatic complexity) and external interactions between objects. This allows for effective assessment of maintainability and reusability.

The research by Oliveira et al. (2008) [9] analyzes the impact of accumulated complexity on the quality of embedded software, highlighting the importance of using aggregated indicators to identify potential defects.

Furthermore, Mordal et al. (2013) [4] propose methods for aggregating individual metrics to obtain a holistic quality assessment at the system level, which is particularly relevant for large industrial projects.

The study by Rawat et al. (2012) [6] emphasizes that the use of various types of metrics not only helps predict defects but also contributes to the overall improvement of development and quality assurance processes.

Finally, the work by Lee (2014) [12] integrates quality factors with corresponding metrics using quality models (e.g., McCall, Boehm, FURPS, Dromey, ISO/IEC 25000). This enables the development of a coherent methodology that ensures metric application across all stages of the software lifecycle for timely defect detection.

Thus, the literature points to the necessity of a comprehensive approach to software quality measurement—one that incorporates domain specificity, development phase, and the aggregation of various metrics to ensure effective product quality control.

Formulation of the article goals

The aim of this article is to establish and quantitatively assess the relationship between file editing frequency and software quality indicators. To achieve this goal, the following objectives are proposed:

1. To develop and justify a new metric — the Consecutive File Edit coefficient (CFE), which allows tracking the cumulative effect of repeated changes.
2. To integrate the CFE calculation with traditional metrics provided by static code analysis tools (e.g., SonarQube) for comprehensive comparison and analysis.
3. To investigate, using several open-source GIT repositories, the impact of a high CFE on structural complexity, technical debt, defect risk, and code maintainability indicators.

Thus, the article aims to demonstrate the practical significance of the new CFE metric in identifying code areas that may require additional developer attention and to provide recommendations for improving software development and maintenance processes.

Summary of the main material

Existing Methods for GIT Repository Analysis

Overview of GIT Analysis Techniques

Analyzing GIT repository history is a widely used approach for understanding the dynamics of software development and its impact on quality. Previous studies have utilized a variety of methodologies to explore the relationship between file editing frequency and software quality. For example, studies like [1] have empirically examined how file editing patterns affect software quality, concluding that frequently edited files tend to have more critical bugs. This suggests that frequent changes may indicate evolving requirements or poor initial code quality.

Other significant contributions include the development of tools like Codebook, as discussed in [2], which helps in discovering and exploiting relationships in software repositories. Codebook analyzes historical data from repositories to identify patterns in file changes and their impact on the overall quality of the software. Such tools provide valuable insights into areas of the codebase that require more attention due to frequent modifications.

SonarQube and SonarScanner

SonarQube and SonarScanner are popular tools for analyzing software quality. These tools compile a range of static software metrics, such as lines of code (LOC), cyclomatic complexity, and coupling between

objects (CBO), which are linked to software quality characteristics like reliability and maintainability. For instance, [3] focuses on how these metrics can be aggregated from GitHub repositories to provide a comprehensive view of a project's health.

SonarQube's approach to code quality analysis is particularly robust, as it includes the detection of code smells, technical debt, and other maintainability issues. This makes it a valuable tool for assessing the long-term quality of software projects, especially when combined with additional metrics like the CFE, which this study introduces.

The effectiveness of SonarQube in analyzing software quality can be compared with other GIT analysis tools and methodologies. Studies, such as [4], have explored methods for detecting similar repositories on GitHub, providing insights into common quality issues and best practices in managing repository histories. Additionally, resources like the Public Git Archive, as discussed in [9], enable large-scale analysis of code changes and their impact on software quality, offering a valuable dataset for researchers. By comparing these tools and datasets, we can establish the strengths and limitations of various approaches to GIT repository analysis. SonarQube's comprehensive metric-based approach, combined with the new Consecutive File Edit (CFE) metric introduced in this research, presents a promising avenue for investigating the link between file editing frequency and software quality.

The CFE metric, which tracks the frequency of edits to the same files across consecutive commits, has significant implications for various software complexity metrics. For example, *Cyclomatic Complexity* often increases with higher CFE, as repeated edits introduce new decision points, loops, and conditionals without refactoring the existing structure. This increase in complexity makes the codebase more challenging to test comprehensively and raises the likelihood of introducing bugs. *Cyclomatic Complexity* can be calculated using the formula:

$$\text{Cyclomatic Complexity} = E - N + 2P \quad (1)$$

where E is the number of edges in the control flow graph, N is the number of nodes, and P is the number of connected components.

Similarly, *Cognitive Complexity* tends to rise with high CFE because frequent modifications, especially by different developers, can create convoluted logic that is difficult to follow. This makes the code harder to understand and maintain over time, leading to a decline in overall code quality. *Cognitive Complexity* does not have a straightforward mathematical formula but is calculated by SonarQube based on the nesting and flow of the code, penalizing deep nesting and complex flow structures more heavily.

Frequent consecutive edits without proper refactoring can also lead to the accumulation of *Code Smells* and an increase in *Technical Debt*. Code smells, such as long methods or large classes, contribute to technical debt, raising the future cost of maintaining the codebase and complicating the development process. *Technical Debt* is often expressed in terms of the time and resources required to refactor the codebase to eliminate these issues.

Furthermore, CFE generally results in an increase in *Lines of Code (LOC)*, as more lines are added with each edit, contributing to a bloated codebase. This is often accompanied by a rise in *Halstead Complexity*, which measures the difficulty of understanding the code based on the number of operators and operands. Halstead Complexity can be calculated using the following formulas:

$$Halstead\ Length = n_1 + n_2 \quad (2)$$

$$Halstead\ Volume = N * \log_2(n_1 + n_2) \quad (3)$$

where n_1 is the number of distinct operators, and n_2 is the number of distinct operands.

Finally, CFE can negatively impact *Object-Oriented Metrics* such as *Weighted Methods per Class (WMC)* and *Coupling Between Objects (CBO)*. Frequent additions of new methods or tighter coupling between objects reduce the modularity and reusability of the code, making it harder to maintain and extend. *WMC* is typically calculated by summing the cyclomatic complexity of all methods in a class:

$$WMC = \sum_{i=1}^{n} C_i \quad (4)$$

where C_i is the cyclomatic complexity of method i.

By understanding these relationships, we can better appreciate the impact of frequent file edits on software quality and the importance of maintaining a balance between necessary changes and code stability.

The impact of consecutive file edits on software complexity metrics underscores the need for effective code management and regular refactoring. As consecutive edits increase, they tend to elevate both the structural and cognitive complexity of the codebase, leading to a higher accumulation of code smells and technical debt. These changes make the software harder to maintain, more error-prone, and more expensive to modify over time. Understanding these impacts allows developers and project managers to identify potential risks early and take proactive steps to ensure the long-term health and maintainability of the software.

Table 1

Impact of Consecutive File Edits on Software Quality MetricsConclusion

Metric	Definition	Potential Impact of High CFE	Analysis
Cyclomatic Complexity	Measures the number of linearly independent paths through the code.	Increase: High CFE may introduce more decision points (e.g., loops, conditionals) as developers add new logic.	Frequent edits may complicate the control flow, making the code harder to test and increasing the risk of introducing bugs((1) The Research on Sof…).
Cognitive Complexity	Assesses the mental effort required to understand the code.	Increase: High CFE can lead to more convoluted logic and deeper nesting, making the code harder to understand.	Repeated changes by different developers can result in less readable code, with complex logic that is difficult to follow ((1) The Research on Sof…).
Code Smells	Indicates potential issues in the code that may not be bugs but could lead to problems later.	Increase: High CFE can accumulate code smells such as long methods or large classes.	Frequent edits without refactoring can result in an accumulation of technical debt and code smells, reducing maintainability((1) The Research on Sof…).
Technical Debt	Represents the cost of rework caused by choosing a suboptimal solution in the short term.	Increase: High CFE suggests that quick fixes may have been applied repeatedly, increasing technical debt.	Frequent file edits may indicate that developers are applying patches or workarounds, leading to a growing need for refactoring ((1) The Research on Sof…).
Lines of Code (LOC)	Counts the number of lines in the codebase.	Increase: High CFE can lead to increased LOC as new features or fixes are added without optimizing existing code.	Repeated edits often increase the size of the codebase, but without necessarily improving its quality or maintainability((6) The Correlation amo…).
Test Coverage	Measures the percentage of code covered by automated tests.	Potential Decrease: High CFE might result in untested code if changes are not accompanied by updated tests.	If frequent edits are not accompanied by corresponding updates in test cases, test coverage may decline, increasing the risk of undetected bugs((6) The Correlation amo…).
Halstead Complexity	Quantifies complexity based on the number of operators and operands in the code.	Increase: High CFE can lead to more complex expressions as additional logic is added.	Frequent edits may introduce new operations and operands, increasing the overall complexity and making the code harder to understand((6) The Correlation amo…).
C&K Metrics	Set of metrics specific to object-oriented design, such as Weighted Methods per Class (WMC) and Coupling Between Objects (CBO).	Increase: High CFE may lead to higher WMC and CBO if new methods are added or existing methods become more interconnected.	Consecutive edits might introduce more methods in a class or increase dependencies between classes, complicating the design((1) The Research on Sof…).

Methodology for Calculating Consecutive File Edits (CFE) and Integrating with SonarQube Metrics

Calculating the Consecutive File Edit (CFE) Metric

The Consecutive File Edit (CFE) metric is designed to quantify the frequency with which specific files are modified within a software project. The rationale behind this metric is that files undergoing frequent changes may be more susceptible to issues related to code stability, maintainability, or design flaws. Unlike traditional metrics that consider the impact of changes, the CFE metric focuses solely on the frequency of edits, providing a distinct perspective on how often the same parts of the codebase are revisited.

CFE Calculation Algorithm

The CFE value for a given file increases each time the file is modified in a commit, regardless of the magnitude or nature of the change. This method allows us to track how often a file is edited without considering the specific changes made. The cumulative CFE for a commit is calculated by summing the CFE values of all the files that were edited in that commit.

The algorithm works as follows:

1) Initialization: We maintain a dictionary to track the CFE value for each file. Initially, all files have a CFE value of zero.

2) Commit Iteration: We iterate through all commits in the repository, processing them in chronological order from the oldest to the most recent.

3) File Processing: For each commit, we identify the files that were modified. If a file has been modified in previous commits, its CFE value is incremented by 1. If the file is being edited for the first time, its CFE value is set to 1.

4) Cumulative CFE Calculation: The CFE for the current commit is calculated by summing the CFE values of all files modified in that commit. This cumulative value provides an overall measure of how frequently the files in that commit have been edited over the project's history.

5) Result Storage: The calculated CFE values for each commit are stored for further analysis and comparison with other metrics.

Pseudocode for CFE Calculation

Below is the pseudocode that describes the algorithm for calculating the CFE metric:

Explanation of the Algorithm

- File Edit Tracking: The dictionary file_edit_count tracks the CFE value for each file in the repository. This value increments each time the file is modified in a new commit, regardless of the extent or type of modification.
- Commit Processing: By iterating through all commits in reverse chronological order, the algorithm ensures that each file's edit history is considered accurately. This approach allows us to track how the CFE value evolves as the project progresses.
- Cumulative CFE: The sum of all file CFE values in a commit provides a cumulative measure for that commit. A higher cumulative CFE indicates that the commit involves files that have been frequently edited, potentially flagging areas of the codebase that require closer scrutiny.

This algorithm provides a straightforward yet effective method for calculating the CFE metric, allowing for subsequent comparison with traditional software complexity metrics. The next section will discuss how this CFE metric is integrated with SonarQube to analyze and compare its correlation with other code quality indicators.

Integrating CFE Calculation with SonarQube Metrics

In this section, we detail the process of integrating the Consecutive File Edit (CFE) metric calculation with the static code analysis capabilities of SonarQube. This integration allows us to compare the CFE metric directly with traditional code quality metrics such as cyclomatic complexity, cognitive complexity, and code smells, providing a comprehensive analysis of how frequently edited files correlate with these metrics.

Overview of the Integration Process

The integration process involves the following key components:

1) Program: The custom-built program is responsible for traversing the commits in a GIT branch, extracting the relevant code, and calculating the CFE for each commit.

2) SonarScanner: This tool acts as a bridge between the program and SonarQube. The program sends the code from each commit to SonarScanner, which then forwards it to SonarQube for analysis.

3) SonarQube: SonarQube performs static code analysis on the submitted code, calculating various metrics including complexity, code smells, and others. After the analysis, the program retrieves these metrics for comparison with the CFE values.

Workflow of the program

The workflow of the *program* can be described in the following steps:

1) Traversing GIT Commits:

The process begins with the program traversing through each commit in the specified GIT branch. For each commit, the program retrieves the state of the code as it existed at that point in time.

2) Sending Code to SonarScanner:

Once the code for a commit is retrieved, the program sends it to SonarScanner. SonarScanner is configured to analyze the code using SonarQube's static analysis capabilities.

3) Static Code Analysis by SonarQube:

SonarScanner passes the code to SonarQube, where it undergoes static analysis. SonarQube calculates various code quality metrics, such as cyclomatic complexity, cognitive complexity, code smells, and others. These metrics provide insight into the maintainability, reliability, and overall quality of the code.

4) Retrieving Metrics**:**

After SonarQube completes the analysis, the program sends a request to SonarQube's API to retrieve the calculated metrics for the commit. These metrics are then stored alongside the CFE value calculated by the program.

5) Storing CFE and Metrics:

The final step involves aggregating and storing the retrieved metrics with the CFE values. This allows for detailed analysis and comparison of how frequent file edits relate to various aspects of code quality.

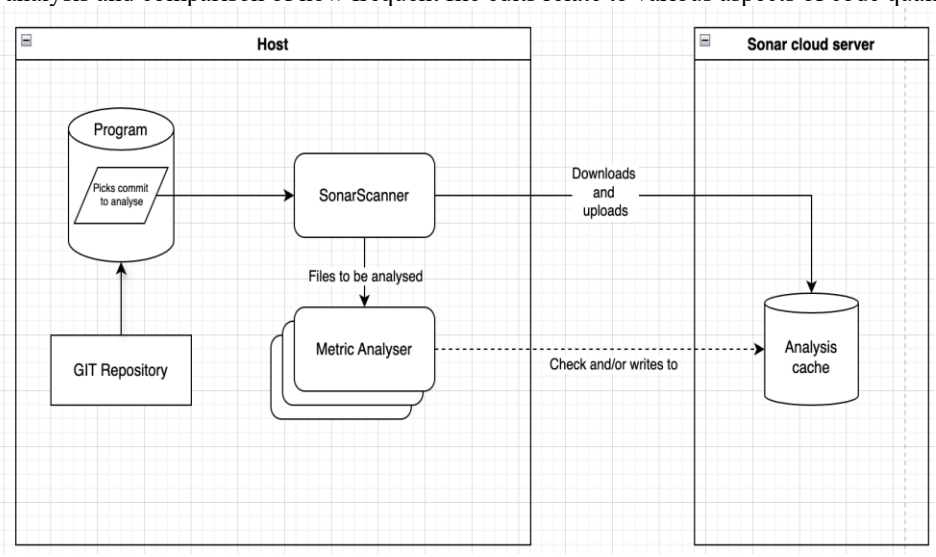

Рис. 1. Integration Scheme for Calculating CFE with SonarQube

Comparative Analysis of CFE Results Across Various GIT Repositories

In this chapter, we analyze the Consecutive File Edit (CFE) metric and its correlation with various software quality metrics, including complexity, code smells, violations, and bugs, across multiple open-source GIT repositories. These repositories, characterized by contributions from multiple developers, provide a rich dataset to understand how frequent file edits impact code quality.

CFE Distribution and Correlation with Complexity

The first step in our analysis is to examine the distribution of CFE values across different commits and investigate their correlation with the complexity metrics provided by SonarQube.

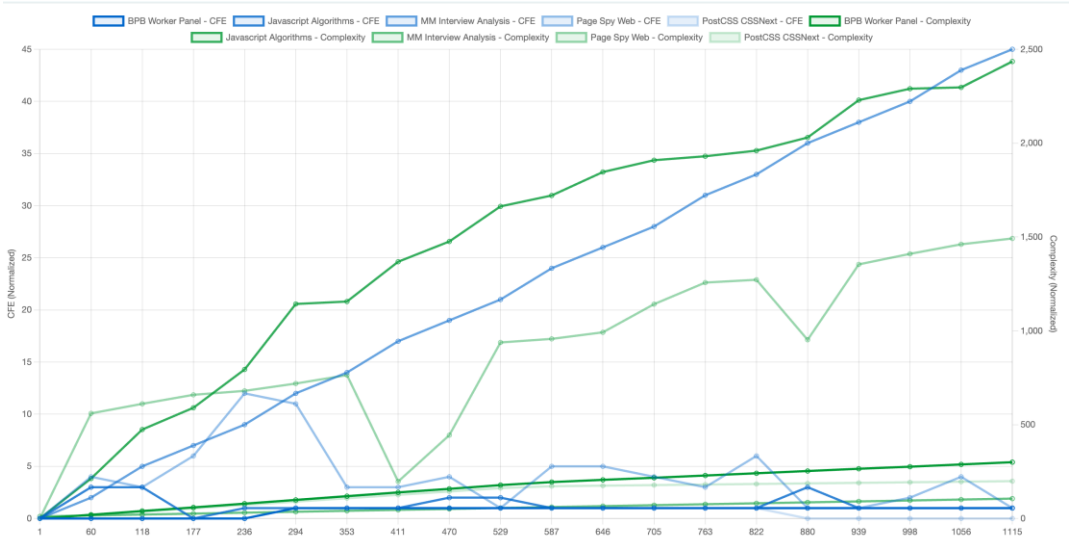

Рис. 2 Distribution of CFE Values and Correlation with Complexity Metric

Observation: Higher CFE values tend to correlate with increased complexity in many of the repositories. This suggests that files that are frequently edited tend to become more complex over time, potentially making the codebase harder to understand and maintain.

CFE and Code Smells

Next, we investigate how the CFE metric relates to the occurrence of code smells, which are indicative of potential design flaws or areas in need of refactoring.

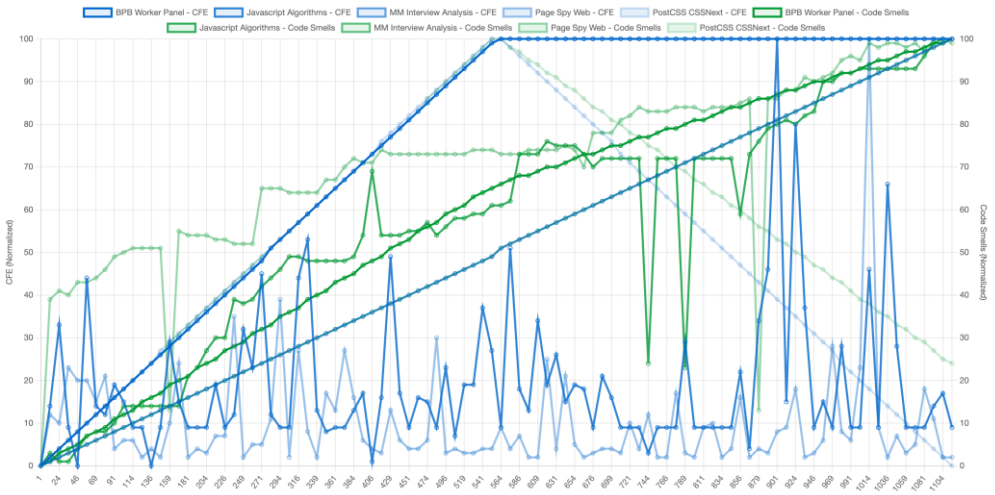

Рис. 3 Relationship Between CFE and the Number of Code Smells

Observation: Repositories with high CFE values show a higher number of code smells, indicating that frequent modifications without adequate refactoring can lead to the accumulation of technical debt.

CFE and Violations

We also examine the relationship between CFE values and the number of violations detected by SonarQube. These violations represent rule breaches that could affect the maintainability and reliability of the software.

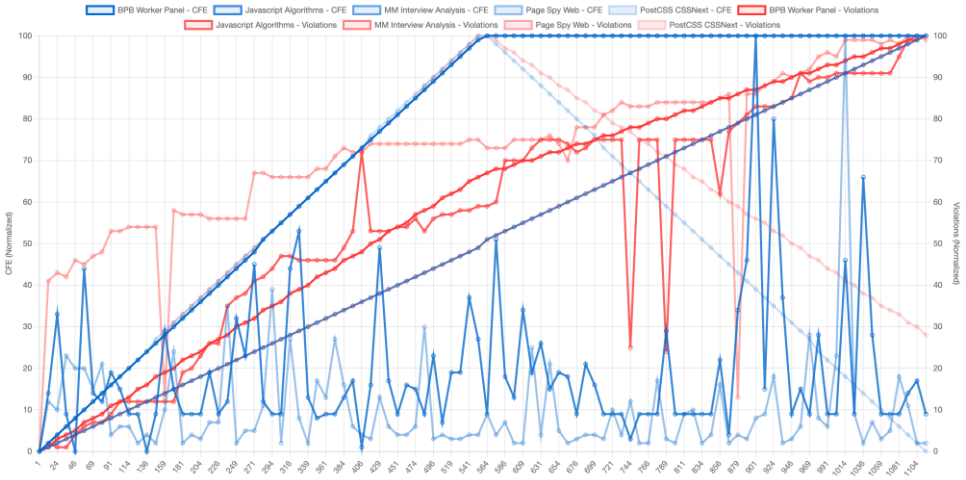

Рис. 4 Correlation Between CFE and the Number of Violations

Observation: A positive correlation between CFE and violations suggests that frequently edited files are more prone to rule violations, potentially due to rushed or inconsistent changes by different contributors.

CFE and Bugs

Finally, we analyze how frequently edited files relate to the number of bugs detected, which directly impacts the reliability of the software.

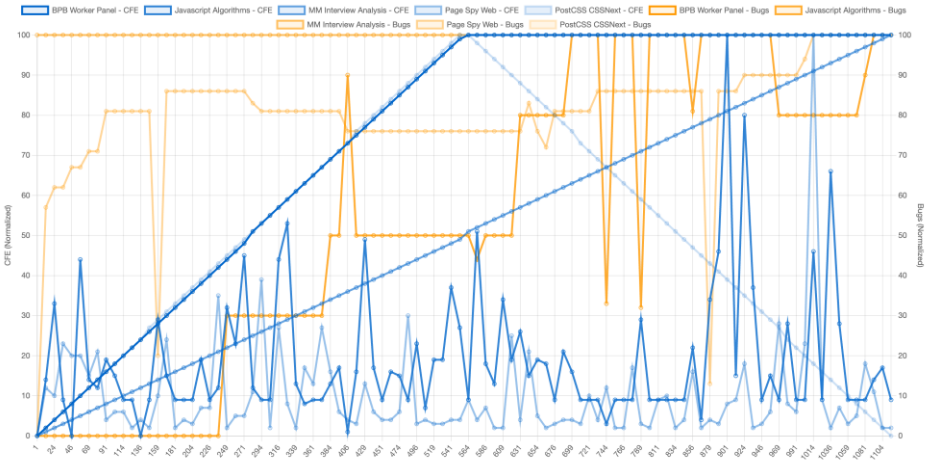

Рис. 5 Impact of CFE on the Number of Bugs

Observation: The data shows that higher CFE values often coincide with an increased number of bugs, underscoring the risk of introducing defects when files are frequently modified without sufficient testing or review.

Comparative Summary

The comparative analysis across these open-source repositories reveals consistent patterns that highlight the risks associated with frequent file edits. High CFE values are often accompanied by increased complexity, more code smells, higher violations, and a greater number of bugs. These findings underscore the importance of monitoring and managing CFE as part of a broader software quality assurance strategy. By identifying and addressing frequently edited files early, development teams can mitigate the accumulation of technical debt and maintain a more robust and maintainable codebase.

References

1. Rosenberg, L. H., & Hyatt, L. E. (1997). Software quality metrics for object-oriented environments. *Crosstalk Journal*, *10*(4), 1–6.

2. Kafura, D., & Henry, S. (1981). Software quality metrics based on interconnectivity. *Journal of Systems and Software*, *2*(2), 121–131.

3. Pargaonkar, S. (2021). Quality and metrics in software quality engineering. *Journal of Science & Technology*, *2*(1), 62–69.

4. Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., & Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, *25*(10), 1117–1135. https://doi.org/10.1002/smr.1562

5. Dalla Palma, S., Di Nucci, D., Palomba, F., & Tamburri, D. A. (2020). Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, *170*, 110726. https://doi.org/10.1016/j.jss.2020.110726

6. Rawat, M. S., Mittal, A., & Dubey, S. K. (2012). Survey on impact of software metrics on software quality. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *3*(1). https://doi.org/10.14569/IJACSA.2012.030111

7. Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, *20*(3), 199–206. https://doi.org/10.1109/32.295895

8. Gaffney, J. E., Jr. (1981). Metrics in software quality assurance. In *Proceedings of the ACM '81 Conference* (pp. 126–130). ACM.

9. Oliveira, M. F., Redin, R. M., Carro, L., da Cunha Lamb, L., & Wagner, F. R. (2008). Software quality metrics and their impact on embedded software. In *5th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software* (pp. 68–77). IEEE. https://doi.org/10.1109/MOMPES.2008.4479903

10. Jiang, Y., Cuki, B., Menzies, T., & Bartlow, N. (2008). Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering* (pp. 11–18). https://doi.org/10.1145/1370788.1370807

11. Schneidewind, N. F. (1997). Software metrics model for quality control. In *Proceedings of the Fourth International Software Metrics Symposium* (pp. 127–136). IEEE. https://doi.org/10.1109/METRIC.1997.637177