An Exploratory Study on the Relationship Between Code Complexity and Bug-Fixing Activity in Open-Source Projects

Samira Santos da Silva
Universidade Federal de Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
samirasilva@ufla.br

Bento Rafael Siqueira
Universidade Federal de Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
bento.siqueira@ufla.br

Cristiane Aparecida Lana
Universidade Federal de Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
cristiane.lana@ufla.br

Sincler Peixoto de Meireles
Universidade de São Paulo
São Paulo, São Paulo, Brazil
sincler@usp.br

Abstract

Code complexity has long been considered a key factor influencing software quality, particularly in relation to fault proneness. However, existing empirical studies report mixed results and often focus on large-scale or industrial systems. In this paper, we present an exploratory empirical study investigating the relationship between code complexity and bug-fixing activity in small-scale open-source projects. We analyze six GitHub repositories, where code complexity is measured using cyclomatic complexity and bug-fixing activity is identified through commit messages containing bug-related keywords. The results show weak positive bivariate correlations between complexity and bug-fixing activity, while process-related metrics present stronger associations. Unexpectedly, the complexity coefficients become negative after controlling for size and development activity. We interpret this sign reversal not as a protective effect of complexity, but as evidence that its marginal relationship with bug-fixing activity is entangled with file size and change exposure. These findings reinforce that complexity should not be used as a standalone maintenance-prioritization criterion and motivate further studies combining structural, evolutionary, and finer-grained information.

Keywords

Software Complexity, Cyclomatic Complexity, Bug Fixing, Empirical Study, Mining Software Repositories

1 Introduction

Code complexity is widely recognized as an important factor in software quality, particularly in relation to fault proneness [4]. One of the most commonly used measures of complexity is cyclomatic complexity, originally proposed by McCabe [10], which captures the number of independent paths through a program. Higher complexity values are often associated with increased cognitive load and reduced maintainability, which may increase maintenance difficulty and potentially contribute to fault occurrence.

The relationship between software metrics and fault proneness has been extensively investigated since early empirical studies in software engineering. For instance, Basili et al. [1] demonstrated that certain code metrics, including complexity, can be used as indicators of fault-prone modules. Since then, numerous studies have explored whether more complex code tends to exhibit higher fault proneness and how such metrics can support quality assurance activities.

Despite this body of work, empirical evidence on the relationship between code complexity and faults remains inconclusive [2]. While several studies report a positive correlation between complexity and fault proneness [1, 13, 20], others observe weaker or context-dependent effects [6, 15, 18]. These inconsistencies may be explained by differences in project characteristics, development processes, programming languages, and measurement approaches. Factors such as team size, code ownership, and evolution patterns can influence both complexity and fault occurrence, making it difficult to generalize findings across contexts.

Process-related factors, such as code churn, number of commits, and developer activity, may also play an important role in explaining faults [8, 14, 15]. They capture aspects of code evolution and instability that are not directly reflected by static complexity measures. A recent large-scale analysis found process metrics to be stronger predictors than product metrics and warned that metric importance can change substantially between small and large samples [9]. Recent work also reports that no individual source-code metric performs consistently across projects [16].

Many empirical studies investigating software metrics and faults focus on large-scale or industrial systems [13, 25], often relying on proprietary datasets that limit reproducibility. Such findings may not fully generalize to small-scale open-source projects, where development dynamics, contribution models, and maintenance practices can differ. Consequently, investigations using publicly available repositories remain necessary to understand how structural and process-related factors interact across contexts.

Revisiting this classic debate remains relevant because cyclomatic complexity remains a widely investigated structural metric, while recent evidence indicates that source-code metrics do not perform consistently across projects and that process metrics can outperform product metrics in fault prediction [9, 16]. In our results, the positive marginal association between complexity and bug-fixing activity reverses after controlling for file size and development activity. Therefore, complexity thresholds alone may direct attention to the wrong hotspots.

In this paper, we present an exploratory empirical study investigating the relationship between code complexity and bug-fixing activity in small-scale open-source projects. Code complexity is measured using cyclomatic complexity, while bug-fixing activity is

identified through commit messages containing bug-related keywords (e.g., "fix" and "bug"). More specifically, we address the following research question:

> *RQ: Is there a relationship between code complexity and bug-fixing activity at the file level in small-scale open-source projects?*

To answer this question, we analyze six Python-based open-source repositories and compute, for each file, structural metrics related to code complexity. We also collect process-related metrics from repository history, including bug-fixing activity, code churn, number of commits, and number of authors, allowing a comparative analysis between structural and process-related factors.

The contributions of this paper are threefold: (i) an empirical analysis of the relationship between code complexity and bug-fixing activity in six small-scale open-source projects; (ii) an explicit analysis of the contrast between weak positive marginal correlations and negative conditional complexity coefficients; and (iii) a research agenda for interpreting complexity together with change exposure, file age, refactoring activity, architectural role, and function-level evidence. Rather than proposing a new fault-prediction technique, the paper offers a thought-provoking reflection on when a familiar metric may become misleading.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the methodology adopted in this study. Section 4 presents and discusses the results. Section 5 discusses threats to validity. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Related Work

The relationship between code complexity and software faults has been widely investigated in Software Engineering, particularly in fault prediction and software quality assessment. Early studies established that software metrics can serve as indicators of fault-proneness. Basili et al. [1], for instance, demonstrated that structural code metrics can be used to identify fault-prone modules. Similarly, McCabe's cyclomatic complexity [10] has long been adopted as a proxy for structural complexity by capturing the number of independent execution paths in a program.

Subsequent research explored the extent to which complexity metrics are correlated with faults. Several studies report a positive relationship, suggesting that more complex modules tend to exhibit higher fault density. However, this relationship is not always consistent, and its strength may vary depending on the metric and the characteristics of the analyzed projects [2]. These findings indicate that complexity alone may not fully explain fault-proneness across software contexts.

More recent studies reinforce the need for contextual interpretation. Wan et al. [23] emphasize that fault-prediction difficulty also depends on dataset characteristics, while Esposito et al. [7] report that cyclomatic complexity captures only part of developers' perceived code understandability. Majumder et al. [9], analyzing 722,471 commits from 700 GitHub projects, found that process metrics generally outperform product metrics and that conclusions about metric importance may change with sample scale. Rebro et al. [16] found contrasting predictive contributions among individual source-code metrics across 275 releases of 39 projects. In a different context, Dalla Palma et al. [3] found product metrics more effective than process metrics for Infrastructure-as-Code faults, showing that their relative value depends on the software context.

Research on software evolution has also shown that process-related factors may play a more significant role than structural complexity alone. Metrics such as code churn, number of commits, and number of contributors are often strong predictors of faults and maintenance activity. Nagappan and Ball [14] demonstrated that relative code churn can predict fault density, while Tornhill [21] argues that highly evolving and unstable codebases tend to require more maintenance effort than stable ones. Controlling for process information is therefore established practice rather than the methodological novelty of this paper.

Granularity is another important concern. Mo et al. [11] showed that only a minority of methods within bug-prone files may actually be involved in fixes and that combining code and history measures improves method-level analysis. This result directly motivates a finer-grained extension of our design because the current study aggregates function-level complexity to files and associates a bug-fixing change with the entire modified file.

Despite the extensive literature, there is still no consensus on the strength and generalizability of the relationship between complexity and faults. Observed correlations are context-dependent and influenced by project size, development practices, and team structure. Moreover, many studies rely on industrial datasets that are not publicly available, limiting reproducibility.

Our work complements existing studies through a lightweight and replicable analysis of six small-scale Python projects. Its contribution is not the introduction of process controls, but the examination of a counterintuitively result: complexity is positively associated with bug-fixing activity when considered alone, yet its coefficient becomes negative once size and process variables are included. We use this sign reversal to reflect on why complexity should not be interpreted as a context-free maintenance-risk score.

3 Study Design

We conducted an exploratory empirical study of the relationship between code complexity and bug-fixing activity at the file level. The data collection, preprocessing, and analysis were performed using automated scripts. The dataset and scripts are available in an online repository[1].

3.1 Subject Systems

We purposively selected six widely used Python-based open-source projects hosted on GitHub: *Requests*, *Flask*, *Click*, *Scrapy*, *Celery*, and *Pylint*. The selection used four practical criteria: Python as the primary language, public availability of version-control history, active development during the analysis period, and variation in application domain and repository size. Using a single language enables consistent extraction with the same static-analysis tool, while the domain and size variation supports an initial cross-context comparison. This is not a probabilistic sample of the Python ecosystem; it is a small-scale set of recognizable cases chosen to test whether the observed pattern recurs across projects.

[1] https://github.com/samirasilva/SBES_Complexity_Bug_Fix

Only Python source files were considered. Files located in directories typically associated with tests, documentation, or examples (e.g., *tests/*, *docs/*, and *examples/*) were excluded to reduce bias from non-production code. The initial dataset comprises 798 Python files across the six repositories.

3.2 Data Collection

Data collection combined static code analysis with repository mining. All metrics were extracted at the file level to connect source-code characteristics, development activity, and bug-fixing behavior.

3.2.1 Complexity Metrics. We used the *Radon* library [5] to compute static complexity metrics. For each Python file, we extracted the cyclomatic complexity of every function following McCabe's definition [10]. We then derived these file-level metrics:

- **Total Complexity:** sum of cyclomatic complexity values across all functions in the file;
- **Average Complexity:** mean per-function complexity;
- **Maximum Complexity:** largest per-function complexity; and
- **Lines of Code (LOC):** total number of lines in the source file.

These metrics capture complementary aspects of structural complexity: overall logical complexity, concentration of decision paths, per-function complexity, and source-code size.

Although complexity is initially measured for functions, the current analysis aggregates it to files so that it can be matched with file changes from Git history. This aggregation is scalable, but it cannot determine which particular function was affected by a bug-fixing commit.

3.2.2 Process-Related Metrics. We mined repository histories with *GitPython* [22], considering commits from January 1, 2022 onwards. For each file, we extracted:

- **Bug-fixing Activity:** number of commits classified as bug fixes;
- **Code Churn:** total number of inserted and deleted lines;
- **Number of Commits:** total number of commits affecting the file; and
- **Number of Authors:** number of distinct contributors who modified the file.

A commit was classified as bug-fixing when its message contained at least one of the following terms: *"fix bug"*, *"bug fix"*, *"fixes"*, *"bug"*, *"error"*, *"defect"*, or *"fault"*. Only messages containing between 4 and 49 words were considered. This scalable heuristic may introduce false positives and false negatives, which we discuss in Section 5.

Code churn was computed as the cumulative number of inserted and deleted lines associated with each file throughout its commit history. It captures modification intensity and evolutionary instability. The number of commits represents modification frequency, while the number of authors captures the degree of collaborative development associated with the file. Together, these process metrics provide complementary perspectives on software evolution and maintenance activity.

3.3 Data Preprocessing

Files containing zero lines of code were removed. Files with no measurable cyclomatic complexity were also excluded because they contain no executable function-level logic relevant to the study. After filtering, structural and process metrics were aggregated into a unified file-level dataset.

Software-engineering metrics commonly present skewed distributions and extreme values. We therefore applied the $\log(1 + x)$ transformation to bug-fixing activity, code churn, LOC, and number of commits. This transformation preserves zero-valued observations while reducing the influence of outliers and stabilizing variance. No additional normalization or standardization was applied. Table 1 summarizes the repositories after preprocessing.

Table 1: Summary of the analyzed repositories after preprocessing.

	Initial	Final				Repository URL
Repo.	**#Files**	**#Files**	**#Commits**	**#Contr.**	**LOC**	**https://github.com**
Click	17	17	1245	43	890	/pallets/click
Requests	19	15	217	30	1046	/psf/requests
Flask	24	20	9120	131	1625	/pallets/flask
Scrapy	181	163	6532	210	8740	/scrapy/scrapy
Pylint	186	170	5876	180	7980	/pylint-dev/pylint
Celery	371	319	8921	315	12300	/celery/celery
Total	**798**	**704**	–	–	–	–

We describe these repositories as *small-scale* because their reported sizes range from 890 to 12,300 LOC. Our conclusions are restricted to this exploratory context rather than generalized to medium or large systems.

3.4 Data Analysis

The dependent variable is the log-transformed number of bug-fixing commits per file. The independent variables comprise structural, size, and process-related metrics: cyclomatic complexity, LOC, code churn, number of commits, and number of authors. We used correlation analysis to evaluate bivariate associations and regression analysis to examine conditional relationships while controlling for size and development activity.

3.4.1 Correlation Analysis. We used Spearman's rank correlation coefficient [19] because the analyzed variables are skewed, non-normal, and contain extreme values. Spearman correlation evaluates monotonic associations based on rank ordering and is less sensitive to these characteristics than parametric alternatives.

3.4.2 Regression Analysis. We used two exploratory Ordinary Least Squares (OLS) regression models [12]. Model 1 uses average cyclomatic complexity, whereas Model 2 uses maximum cyclomatic complexity. Both models control for log-transformed LOC, code churn, number of commits, and number of authors. The analysis was performed with *Statsmodels* [17]. Variance Inflation Factor (VIF) values were computed to assess multicollinearity.

4 Results and Discussion

This section presents the results of the empirical analysis conducted to answer RQ: *Is there a relationship between code complexity and bug-fixing activity at the file level in small-scale open-source projects?*

4.1 Descriptive Results

The initial dataset contained 798 Python files distributed across six open-source repositories. After preprocessing and filtering files with zero LOC or zero complexity, the final dataset used in the analysis contained 704 files. Table 2 presents the descriptive statistics of the analyzed metrics for the final dataset. The results indicate that most files present relatively low to moderate cyclomatic complexity. The average file-level complexity is 2.94, while the maximum observed value reaches 55. The distributions of complexity, churn, commits, and bug-fixing activity are notably skewed, which is consistent with observations commonly reported in software engineering datasets, where a relatively small subset of modules concentrates higher development and maintenance activity.

Table 2: Descriptive statistics of the analyzed metrics after preprocessing.

Metric	Mean	Std	Min	Median	Max
Average Complexity	2.94	1.73	1.00	2.50	17.00
Maximum Complexity	7.92	7.00	1.00	6.00	55.00
Total Complexity	48.27	85.20	1.00	21.00	956.00
Lines of Code (LOC)	287.20	422.58	6.00	139.50	3815.00
Bug-Fixing Activity	1.70	2.86	0.00	1.00	45.00
Code Churn	588.56	1166.31	0.00	141.00	9926.00
Number of Commits	21.57	28.89	0.00	10.00	284.00
Number of Authors	5.17	5.00	0.00	4.00	43.00

4.2 Correlation Analysis

We first investigated the bivariate relationship between code complexity and bug-fixing activity using Spearman correlation. Spearman correlation was adopted due to the non-normal and skewed nature of the analyzed variables.

The analysis revealed a positive but weak correlation between average cyclomatic complexity and bug-fixing activity ($\rho = 0.1698$, $p < 0.001$), suggesting that files with higher complexity tend to exhibit slightly higher bug-fixing activity. Maximum complexity also presented a positive correlation with bug-fixing activity ($\rho = 0.3567$, $p < 0.001$), slightly stronger than the association observed for average complexity.

In contrast, process-related metrics presented substantially stronger associations with bug-fixing activity. In particular, code churn showed a strong positive correlation ($\rho = 0.6633$, $p < 0.001$), indicating that files undergoing more intense modification activity are considerably more likely to require bug-fixing changes than complexity metrics alone.

Figure 1 illustrates the relationship between the complexity metrics and bug-fixing activity. Although both metrics exhibit positive trends, substantial dispersion is observed, suggesting that complexity alone is not a strong indicator of bug-fixing activity. The relationship appears slightly more pronounced for maximum complexity than for average complexity.

4.3 Regression Analysis

To better understand the relationship between code complexity and bug-fixing activity while accounting for potential confounding factors, we fitted two OLS regression models. OLS regression was adopted as an exploratory modeling approach to analyze the

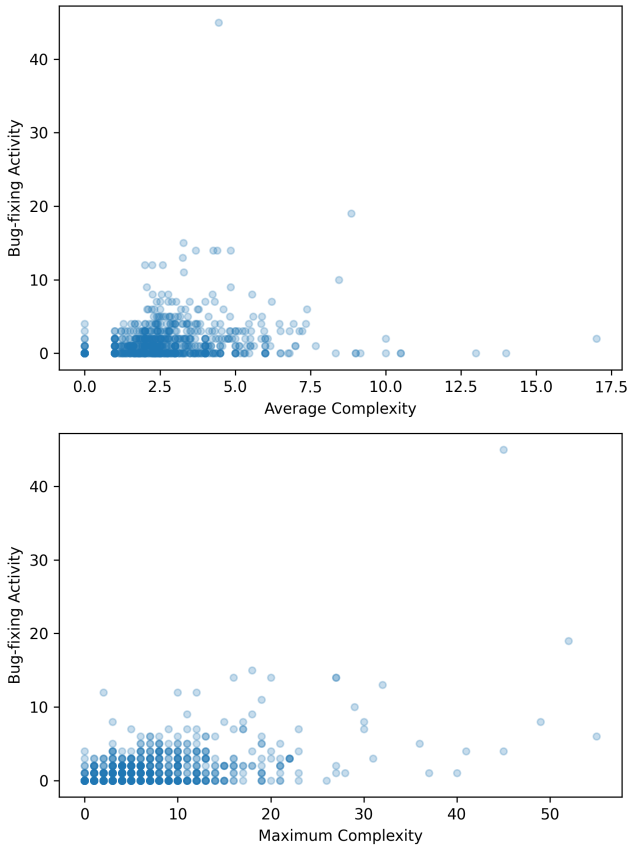

Figure 1: Relationship between complexity metrics and bug-fixing activity.

joint effects of structural and process-related metrics. The models differ only in the complexity metric adopted: Model 1 uses average complexity, whereas Model 2 uses maximum complexity. Both models additionally include LOC, code churn, number of commits, and number of authors as control variables. Log transformations were applied to selected variables to reduce skewness and stabilize variance. Table 3 summarizes the regression results.

Table 3: OLS regression results for bug-fixing activity.

Metric	Model 1 (Avg. Complexity)	Model 2 (Max. Complexity)
Average Complexity	-0.0486***	–
Maximum Complexity	–	-0.0098**
Lines of Code (LOC)	0.1182***	0.1398***
Code Churn	-0.0568**	-0.0563**
Number of Commits	0.2996***	0.2863***
Number of Authors	0.0514***	0.0543***
R^2	0.656	0.648
Adj. R^2	0.654	0.645

*** p<0.001, ** p<0.01

Since regression coefficients are estimated considering all variables simultaneously, small variations in the coefficients of the control variables between the two models are expected. Both models present similar explanatory power, with R^2 and adjusted (Adj.)

R^2 values above 0.64, indicating that the selected variables explain a substantial portion of the variance in bug-fixing activity.

Interestingly, although the correlation analysis revealed a positive association between complexity and bug-fixing activity, both regression models produced negative coefficients for the complexity metrics. This suggests that, when controlling for process-related variables, structural complexity alone does not independently explain increased bug-fixing activity.

In contrast, process-related metrics, particularly number of commits and number of authors, present stronger positive associations with bug-fixing activity. LOC also presents positive coefficients, indicating that larger files tend to require more bug-fixing changes. Conversely, churn presents negative coefficients when the remaining variables are simultaneously considered. The negative churn coefficients contrast with the positive bivariate correlation results, suggesting shared variance among correlated process-related variables when analyzed simultaneously in the regression models.

To further assess multicollinearity among the predictor variables, we computed Variance Inflation Factor (VIF) values for both regression models, as shown in Table 4. The complexity metrics presented low VIF values in both models, indicating limited collinearity with the remaining predictors. Moderate multicollinearity was observed for code churn and number of commits, although all VIF values remained below commonly adopted thresholds for severe multicollinearity. Residual diagnostics indicate mild deviations from normality, although skewness values remain relatively small.

Table 4: Variance Inflation Factor (VIF) values for the regression models.

Metric	Model 1 (Avg. Complexity)	Model 2 (Max. Complexity)
Average Complexity	1.10	–
Maximum Complexity	–	1.98
Lines of Code	1.94	2.32
Code Churn	5.45	5.48
Number of Commits	6.50	6.45
Number of Authors	3.12	3.46

4.4 Cross-Project Variability

To better understand whether the relationship between complexity and bug-fixing activity is consistent across projects, we performed a repository-level correlation analysis. Table 5 reports Spearman correlations between bug-fixing activity and three metrics: average complexity, maximum complexity, and code churn.

Table 5: Spearman correlation by repository.

Repository	#Files	Average Complexity	Maximum Complexity	Code Churn
Requests	15	0.172	0.241	0.420
Click	17	0.428	0.639**	0.813***
Flask	20	0.272	0.238	0.499*
Scrapy	163	0.289***	0.510***	0.665***
Pylint	170	0.259***	0.403***	0.647***
Celery	319	0.133*	0.412***	0.511***

* p<0.05, ** p<0.01, *** p<0.001

The results show that the relationship between complexity and bug-fixing activity varies across repositories. Average complexity presents weak to moderate positive correlations, whereas maximum complexity generally presents stronger correlations than average complexity. Across repositories, maximum complexity generally exhibited stronger associations with bug-fixing activity than average complexity, suggesting that isolated highly complex functions may provide more informative signals than average file-level complexity.

Nevertheless, code churn consistently exhibits stronger associations with bug-fixing activity than complexity metrics across repositories. These findings reinforce the overall result that development activity is more strongly associated with bug-fixing behavior than structural complexity alone.

Finally, repositories with smaller numbers of files, such as *Requests*, *Flask*, and *Click*, should be interpreted with caution due to limited sample sizes.

4.5 Discussion

The results partially support the assumption that more complex files are associated with increased bug-fixing activity. The correlation analysis revealed statistically significant but relatively weak positive relationships between complexity metrics and bug-fixing activity. However, when controlling for process-related variables through regression analysis, the complexity coefficients became negative.

The positive correlations and negative regression coefficients answer different questions. The correlations ask whether files with greater complexity also tend to accumulate more bug-fixing activity. The regressions ask whether complexity contributes additional information among files with comparable size and development activity. Complex files are often also large, frequently changed, and touched by more developers. Once these dimensions are held constant, the remaining complexity variation is no longer positively associated with bug-fixing activity.

This sign reversal should not be interpreted as evidence that complexity prevents bugs. The coefficients are conditional on correlated process measures, the outcome is based on commit-message keywords rather than confirmed faults, and repository-level results show substantial heterogeneity. The safer interpretation is that the marginal complexity signal is coupled with change exposure and project context, and that complexity alone has limited independent explanatory value in this sample.

A targeted inspection of contrasting files illustrates the point. Pylint's *recommendation_checker.py* has maximum complexity 26 and 37 commits, but no keyword-labeled bug-fixing commit in the analysis window. Its branching encodes specialized static-analysis rules, so high structural complexity can coexist with relative corrective stability. Conversely, Scrapy's *default_settings.py* has maximum complexity 2, yet it has 81 commits and 12 keyword-labeled bug-fixing commits. As a central configuration module, it can attract corrective changes despite little control-flow complexity. These examples do not establish causality, but they show why architectural role and modification exposure can dominate a structural metric.

Overall, process-related metrics provide stronger signals of bug-fixing activity than cyclomatic complexity alone. Prioritizing maintenance solely through complexity thresholds may overlook simple but central files and over-prioritize complex but stable ones. Complexity remains relevant, but maintenance-risk assessment should

combine it with process history, project-specific baselines, and architectural context.

5 Threats to Validity

As with any empirical study, our results are subject to several threats to validity. Following standard classifications in empirical software engineering [24], we discuss construct, internal, external, and conclusion validity.

Construct Validity. Construct validity refers to whether the chosen metrics accurately capture the intended concepts. In this study, code complexity is measured using cyclomatic complexity. Although commonly adopted, this metric captures only control-flow complexity and does not account for other relevant aspects such as data complexity, coupling, or cognitive complexity. As a result, it may not fully represent the overall difficulty of understanding or maintaining a file. Bug-fixing activity was identified using a heuristic keyword-based matching strategy applied to commit messages (e.g., *fix*, *bug*). While this approach is commonly used, it may introduce noise due to false positives (e.g., commits mentioning "fix" in a non-bug context) or false negatives (e.g., bug fixes without explicit keywords). To mitigate this issue, we applied basic filtering on commit messages, but some inaccuracies may remain. Additionally, aggregating complexity at the file level may obscure finer-grained relationships that occur at the function or class level. Because a bug-fixing commit is attributed to every modified file, this granularity mismatch can hide whether the most complex function was actually changed.

Internal Validity. Internal validity concerns the extent to which observed relationships are affected by confounding factors. Although we included control variables such as lines of code, churn, number of commits, and number of authors, other factors may still influence bug-fixing activity. For instance, developer expertise, code ownership, and review practices are not explicitly captured in our analysis. Furthermore, the use of observational data limits our ability to establish causal relationships. The identified associations should therefore be interpreted as correlations rather than causal effects. We also assessed multicollinearity among the independent variables using Variance Inflation Factor (VIF) analysis. Most variables presented low VIF values, while churn and number of commits exhibited moderate multicollinearity. However, all VIF values remained below commonly adopted thresholds for severe multicollinearity, suggesting limited impact on coefficient stability. File age is not controlled: older files have had more time to accumulate commits and bug fixes. Refactoring may also inflate churn and commit counts without representing fault correction, and the current heuristic does not distinguish refactorings systematically.

External Validity. External validity refers to the generalizability of the results. Our study focuses on a purposive sample of six small-scale Python open-source projects, including three repositories with at most 20 analyzed files. While these projects vary in size and domain, the results may not generalize to other programming languages, industrial systems, or domains with different development practices. Additionally, open-source development environments differ from industrial settings in terms of team structure, contribution models, and quality assurance processes. The findings should therefore be read as initial evidence, and a broader, systematically sampled replication is necessary.

Conclusion Validity. Conclusion validity concerns the reliability of the statistical analysis. We used Spearman correlation and OLS regression, which are commonly adopted techniques in empirical software engineering. Log transformations were applied to reduce skewness and stabilize variance. However, software engineering data is often highly skewed and noisy, and the small per-project samples reduce statistical power. Although statistically significant results were obtained, the effect sizes for complexity are relatively small. OLS was adopted as an exploratory approach, but bug-fixing activity represents count-based data that can also be modeled using Poisson or Negative Binomial regression. Future analyses should compare file- and function-level results and manually validate refactoring and architectural explanations for selected cases.

6 Conclusion

In this paper, we presented an exploratory empirical study investigating the relationship between code complexity and bug-fixing activity at the file level in small-scale open-source Python projects. Cyclomatic complexity exhibits statistically significant but relatively weak bivariate associations with bug-fixing activity. In contrast, process-related metrics, particularly code churn, number of commits, and number of authors, present substantially stronger associations with maintenance-related changes.

The analysis also revealed that maximum complexity tends to exhibit stronger relationships with bug-fixing activity than average complexity. The central result, however, is the contrast between these positive correlations and the negative complexity coefficients obtained after controlling for size and development activity. We do not interpret this result as a protective effect of complexity. Instead, it suggests that much of the apparent complexity–bug relationship is interconnected with how large, central, and frequently modified a file is. Cross-project variation and contrasting file examples reinforce that complexity is not a context-free maintenance-risk score.

From a practical perspective, relying solely on cyclomatic complexity to identify problematic files may lead to incomplete or misleading conclusions. Complexity thresholds should complement, rather than replace, change history and architectural knowledge when prioritizing inspection. The lightweight and replicable methodology adopted in this work facilitates further empirical investigations and replications.

As future work, we plan to incorporate additional complexity metrics, refine bug identification using issue-tracking linkage or machine learning, and expand the analysis to other languages and larger datasets. Larger and systematically sampled studies should control file age, distinguish refactoring from corrective activity, and connect bug-fixing commits to the specific functions they modify. Such function-level analysis can test whether localized complex code is associated with corrective effort even when the surrounding file is not.

Artifact Availability

The dataset, scripts, and analysis materials are available at https://github.com/samirasilva/SBES_Complexity_Bug_Fix.

References

[1] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22, 10 (1996), 751–761. doi:10.1109/32.544352

[2] Hao Chen, Ying Zhang, and Sunghun Kim. 2019. On the Relationship Between Code Complexity and Software Defects: An Empirical Study. *Empirical Software Engineering* 24, 6 (2019), 1–32. doi:10.1007/s10664-019-09701-4

[3] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. 2022. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering* 48, 6 (2022), 2086–2104. doi:10.1109/TSE.2021.3051492

[4] D. I. De Silva, R. D. New, B. L. O. Sachethana, S. M. D. T. H. Dias, P. Y. C. Perera, M. E. Katipearachchi, and T. D. D. H. Jayasuriya. 2023. The Relationship between Code Complexity and Software Quality: An Empirical Study. *Journal of Software Engineering Research and Development* 11, 1 (2023), 1–.

[5] Michele Di Pierro. 2024. Radon: Code Metrics in Python. https://radon.readthedocs.io/. Accessed: 2026-05-22.

[6] Karim O. Elish and Mahmoud O. Elish. 2008. Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems and Software* 81, 5 (2008), 649–660.

[7] Matteo Esposito, Andrea Janes, Terhi Kilamo, and Valentina Lenarduzzi. 2023. Early Career Developers' Perceptions of Code Understandability: A Study of Complexity Metrics. *arXiv preprint arXiv:2303.07722* (2023). https://arxiv.org/abs/2303.07722

[8] Ahmed E. Hassan. 2009. Predicting Faults Using the Complexity of Code Changes. *Proceedings of the 31st International Conference on Software Engineering* (2009), 78–88.

[9] Suvodeep Majumder, Pranav Mody, and Tim Menzies. 2022. Revisiting process versus product metrics: a large scale analysis. *Empirical Software Engineering* 27, 3 (2022), 60. doi:10.1007/s10664-021-10068-4

[10] Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[11] Ran Mo, Shaozhi Wei, Qiong Feng, and Zengyang Li. 2022. An Exploratory Study of Bug Prediction at the Method Level. *Information and Software Technology* 144 (2022), 106794. doi:10.1016/j.infsof.2021.106794

[12] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. 2021. *Introduction to Linear Regression Analysis* (6 ed.). Wiley.

[13] Nachiappan Nagappan and Thomas Ball. 2005. Static Analysis Tools as Early Indicators of Pre-Release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 580–586.

[14] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 284–292.

[15] Foyzur Rahman and Premkumar Devanbu. 2013. How, and Why, Process Metrics Are Better. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 432–441.

[16] Dominik Arne Rebro, Bruno Rossi, and Stanislav Chren. 2023. Source Code Metrics for Software Defects Prediction. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing* (Tallinn, Estonia) *(SAC '23)*. Association for Computing Machinery, New York, NY, USA, 1469–1472. doi:10.1145/3555776.3577809

[17] Skipper Seabold and Josef Perktold. 2010. Statsmodels: Econometric and Statistical Modeling with Python. In *Proceedings of the 9th Python in Science Conference*. 92–96.

[18] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[19] Charles Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101.

[20] Raghuvar D. Subramanyam and M. S. Krishnan. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering* 29, 4 (2003), 297–310.

[21] Adam Tornhill. 2022. *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*. Pragmatic Bookshelf.

[22] Michael Trier and contributors. 2024. GitPython. https://gitpython.readthedocs.io/. Accessed: 2026-05-22.

[23] Xiaohui Wan, Zheng Zheng, Fangyun Qin, and Xuhui Lu. 2023. Data Complexity: A New Perspective for Analyzing the Difficulty of Defect Prediction Tasks. *arXiv preprint arXiv:2305.03615* (2023). https://arxiv.org/abs/2305.03615

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

[25] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE, 9.