KFUPM
Journal of Undergraduate Research International

Diamond Open Access 75

Article Computer Science

# Static Analysis of Code Quality in Open-Source Python Projects

Bashir Adam Ahmed Ali

Cite https://doi.org/10.64589/juri/215034

Submitted: September 15, 2025 Revised: November 09, 2025 Accepted: December 02, 2025

ABSTRACT

Ensuring high code quality is essential for the reliability and sustainability of widely used open-source software, yet empirical, multi-metric assessments of popular Python libraries remain scarce. This study presents a systematic static analysis of code quality in two prominent open-source Python projects, Requests and BeautifulSoup4. We employed a suite of static analysis tools—Radon, Pylint, and Flake8—to measure metrics including lines of code (LOC), cyclomatic complexity, maintainability index, Pylint scores, and PEP8 violations across all project modules. Automated Python scripts were developed to ensure a reproducible and comprehensive data collection process. Our analysis reveals a nuanced picture of code quality: while both projects exhibit good overall maintainability, we identified specific modules with high cyclomatic complexity and a significant number of style violations. For instance, dammit.py in BeautifulSoup4 recorded 318 PEP8 violations and a maintainability index of 40.08. The results underscore the practical value of static analysis for pinpointing refactoring candidates and enforcing coding standards.

We recommend that open-source projects integrate these tools into continuous integration pipelines to proactively manage technical debt and enhance long-term maintainability. All code and data for replicating this study are publicly available. This work provides a novel, reproducible, and multi-tool assessment that delivers actionable, module-level insights for maintainers, addressing a gap in empirical Python code quality research.

Keywords: code quality, static analysis, open-source, Python, Python Enhancement Proposal-8, maintainability

## 1. INTRODUCTION

Software quality is a critical determinant of a software system's reliability, maintainability, and long-term sustainability. High-quality code reduces maintenance costs, minimizes defects, and improves developer productivity. Python, as one of the most popular programming languages, powers a vast ecosystem in web development, automation, data science, and machine learning. Consequently, the code quality of foundational open-source Python projects is of paramount practical importance, as they underpin countless academic and commercial applications [1].

Despite the existence of coding standards like PEP8, code quality in open-source projects can vary significantly due to factors such as diverse contributor expertise and evolving project requirements. Poor quality often manifests as high cyclomatic complexity, low maintainability, and style inconsistencies, which collectively increase maintenance effort and risk[2]. Static analysis tools offer an automated, non-execution-based method to evaluate code quality. Tools like Radon, Pylint, and Flake8 can measure key metrics related to complexity, maintainability, and style compliance [3–5].

While previous research has extensively evaluated static analysis for languages like Java and C++ [6,7], and some studies have focused on the capabilities of individual Python tools [8], there is a scarcity of empirical, multi-metric studies that systematically assess code quality across popular, real-world Python libraries. This gap limits our understanding of the actual state of code quality in the Python open-source ecosystem. Recent surveys highlight the growing need for such empirical validations in modern software engineering contexts [9].

This study aims to address this gap by conducting a reproducible static analysis of two widely used Python libraries: BeautifulSoup4 and Requests. To guide our investigation, we formulate the following research questions:

- **RQ1:** How do code quality metrics, such as cyclomatic complexity and maintainability index, vary across different modules within BeautifulSoup4 and Requests?
- **RQ2:** What are the differences in overall code quality and adherence to PEP8 standards between these two projects?
- **RQ3:** Which specific modules are the most critical candidates for refactoring based on the aggregated metrics?

The novelty and contribution of this work are threefold. First, it provides a detailed, empirical snapshot of code quality in two critical Python libraries, moving beyond tool capability studies to deliver actionable, module-level insights for maintainers. Second, it employs a transparent and fully reproducible methodology, with all scripts and data publicly available, addressing a key shortcoming in many existing studies. Third, it synthesizes findings from multiple complementary tools (Radon, Pylint, Flake8) to present a holistic view of quality through a fully

reproducible methodology—addressing a key gap in replicable Python research where prior work often focuses on a single tool or metric.

## 2. RELATED WORK

The critical role of software quality and the utility of static analysis are well-established in software engineering literature. Recent surveys, such as that by Iftikhar et.al[9], provide a comprehensive overview of code quality metrics and their impact on software maintainability, underscoring their continued relevance.

In the domain of open-source software, many foundational studies focused on languages like Java and C++. For example, research has analyzed trends in code smells and defect prediction across multiple Java projects[6]. However, the findings from these studies are not directly transferable to Python due to its dynamic typing and different idiomatic patterns.

Research specifically targeting Python code quality is growing. Siddik and Bezemer et al.[10] explored the correlation between code style and code quality, finding that consistent style often accompanies higher quality, which justifies the inclusion of PEP8 analysis in our study. Other works have evaluated the effectiveness of specific tools. Scalabrino et al.[11] conducted a comprehensive study on automatically assessing code understandability, a key aspect of maintainability. Recent empirical studies have applied these tools in specific contexts, such as comparing code quality in Python Jupyter notebooks and scripts for data science[12] or detecting code smells in Python software[13].

Furthermore, the detection of complex issues in Python has been a focus of recent research. Studies have adapted traditional metrics and definitions for Python's unique characteristics[14,15]. Our work directly builds on this by applying established complexity and maintainability metrics (via Radon) and code quality scoring (via Pylint) to large-scale, real-world projects, providing a validation of these concepts in practice.

This study synthesizes these strands of research by conducting a systematic, multi-faceted assessment of two large, community-driven Python projects using a suite of complementary tools. Unlike studies that focus on a single aspect of quality or a specific domain, we provide a holistic view of complexity, maintainability, and style in general-purpose libraries, identifying specific, actionable refactoring targets.

## 3. METHODOLOGY

This study employs a systematic and reproducible methodology to evaluate the code quality of the selected open-source Python projects. The process encompasses project selection, tool selection and configuration, metric extraction, data aggregation, visualization, and analysis.

**3.1. Project Selection.** We selected two high-profile Python projects based on their popularity, active maintenance, and significance within the Python ecosystem:

- **Requests**[16] — A simple, yet powerful HTTP library for Python.
- **BeautifulSoup4**[17] — A library for pulling data out of HTML and XML files.

These projects were chosen because their widespread use makes their code quality a concern for a large community, and their manageable size allows for a comprehensive module-by-module analysis.

**3.2. Tool Selection and Configuration.** To ensure a comprehensive assessment, we used a suite of established static analysis tools. The exact commands and configurations are provided in the supplementary materials to ensure reproducibility.

- **Radon (v6.0.1):** Used to compute raw metrics. We used the radon cc -a command for the average cyclomatic complexity per module and radon mi -s for the maintainability index.
- **Pylint (v3.1.0):** Used for general code quality analysis and scoring. We used a default configuration.
- **Flake8 (v7.0.0):** Used to detect PEP8 style violations. We used the default rule set without modifications.

***3.2.1. Justification for tool selection.*** Radon, Pylint, and Flake8 were selected for their complementary strengths, widespread adoption in the Python community, and ability to measure distinct aspects of code quality. Radon provides raw complexity and maintainability metrics, Pylint offers a holistic quality score, and Flake8 is the de facto standard for PEP8 compliance. Using their default configurations ensures reproducibility and aligns with common practitioner usage, allowing our results to be directly comparable to typical developer workflows.

**3.3. Metric Selection.** We focused on the following metrics to provide a multi-dimensional view of code quality:

- **Lines of Code (LOC):** Measures module size.
- **Cyclomatic Complexity (CC):** Measures the structural complexity of functions and methods.
- **Maintainability Index (MI):** A composite metric (scale of 0-100) where higher values indicate better maintainability.
- **Pylint Score:** A normalized score (typically 0-10) for overall code quality.
- **PEP8 Violations:** A count of style guide deviations.

These metrics directly address our RQs: CC and MI (RQ1), PEP8 violations (RQ2), and their aggregation (RQ3).

**3.4. Data Collection and Reproduction.** The entire data collection process was automated with Python scripts to ensure accuracy and reproducibility. The steps were:

- **Repository Cloning:** The latest code was cloned from the official GitHub repositories.
- **Static Analysis Execution:** Scripts sequentially executed Radon, Pylint, and Flake8 on every Py file.
- **Metric Extraction:** Outputs were parsed to extract numerical values for each metric.
- **Data Aggregation:** Results were compiled into structured CSV files.

**3.5. Visualization.** We used Matplotlib and Seaborn to generate visualizations for intuitive analysis. All figures are placed after their first textual reference, have clear axis labels and captions, and exclude the "SUMMARY" row from module-level charts.

**Table 1.** *Static analysis metrics for BeautifulSoup4*

| Module/File | LOC | CC (avg) | Maintainability Index | Pylint Score | PEP8 Violations |
|---|---|---|---|---|---|
| dammit.py | 829 | 3.85 | 40.08 | 7.39 | 318 |
| test_tree.py | 1829 | 3.59 | 6.87 | 7.13 | 69 |
| testing.py | 592 | 3.72 | 45.19 | 0.0 | 39 |
| *__init__*.py | 406 | 3.17 | 100.0 | N/A | 34 |
| element.py | 1611 | 0 | 0 | N/A | 26 |
| SUMMARY | 7755 | 3.40 | 74.78 | 3.98 | 656 |

*Note: A Maintainability Index (MI) of 0.0, as seen in element.py, occurs when the calculated index is non-positive due to very high complexity and/or size overwhelming the effect of comments, and is reported as 0 by the Radon tool.*

- **Bar charts** of average cyclomatic complexity per module.
- **Histograms** of Pylint score distribution.
- **Scatter plots** of LOC vs. Cyclomatic Complexity.

## 4. RESULTS

The results of the static analysis for BeautifulSoup4 and Requests are presented below.

**4.1. BeautifulSoup4.** The analysis of BeautifulSoup4 covered 19 modules with a total of 7,755 LOC. The project-level averages are a cyclomatic complexity of 3.4, a maintainability index of 74.78, a Pylint score of 3.98, and 656 total PEP8 violations (Table 1). Figure 1 illustrates the average cyclomatic complexity per module, highlighting dammit.py and test_tree.py as particularly complex. The distribution of Pylint scores (Figure 2) shows a bimodal pattern, indicating variability in code quality across modules. A scatter plot of LOC versus cyclomatic complexity (Figure 3) reveals a moderate positive correlation, suggesting that larger modules tend to be more complex.

**4.2. Requests.** The Requests project is larger, with 34 modules and 11,248 LOC. It shows better average metrics: a lower cyclomatic complexity (2.87), a similar MI (73.26), a higher Pylint score (5.96), and fewer total PEP8 violations (533) (Table 2). Figure 4 displays the average cyclomatic complexity per module, with test-related modules showing the highest values. The distribution of Pylint scores (Figure 5) is skewed toward higher values, reflecting better overall code quality. Figure 6 shows the relationship between LOC and cyclomatic complexity, with a moderate positive correlation ($r = 0.41$).

## 5. DISCUSSION

This study provides empirical insights into the code quality of two foundational Python libraries, directly addressing our research questions.

**5.1. RQ1: Variation of Metrics within Projects.** The analysis reveals significant internal variation, identifying clear refactoring candidates. In BeautifulSoup4, dammit.py is a clear outlier with high complexity (3.85), low maintainability (40.08), and a high density of PEP8 violations. Similarly, test_tree.py is very large and complex with a critically low MI of 6.87. In Requests, test_requests.py is the primary concern, being the largest and one of the most complex modules, with an MI floored

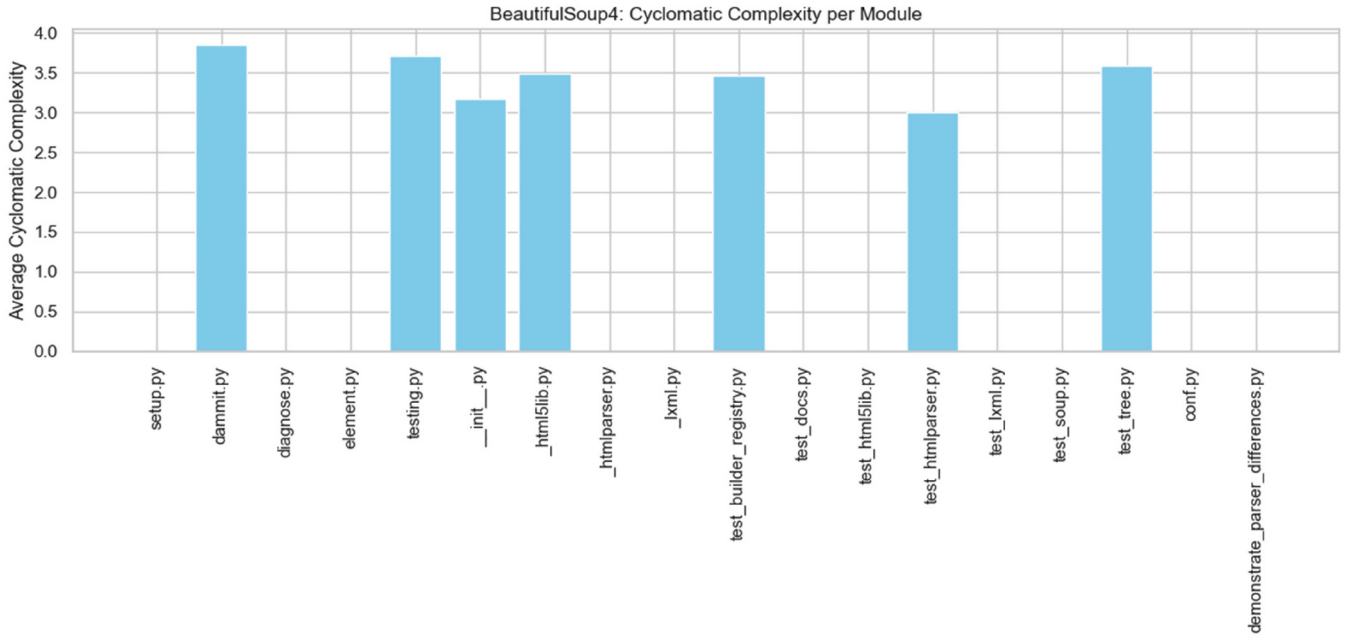


**Figure 1.** Average cyclomatic complexity per module in BeautifulSoup4; higher bars indicate greater structural complexity

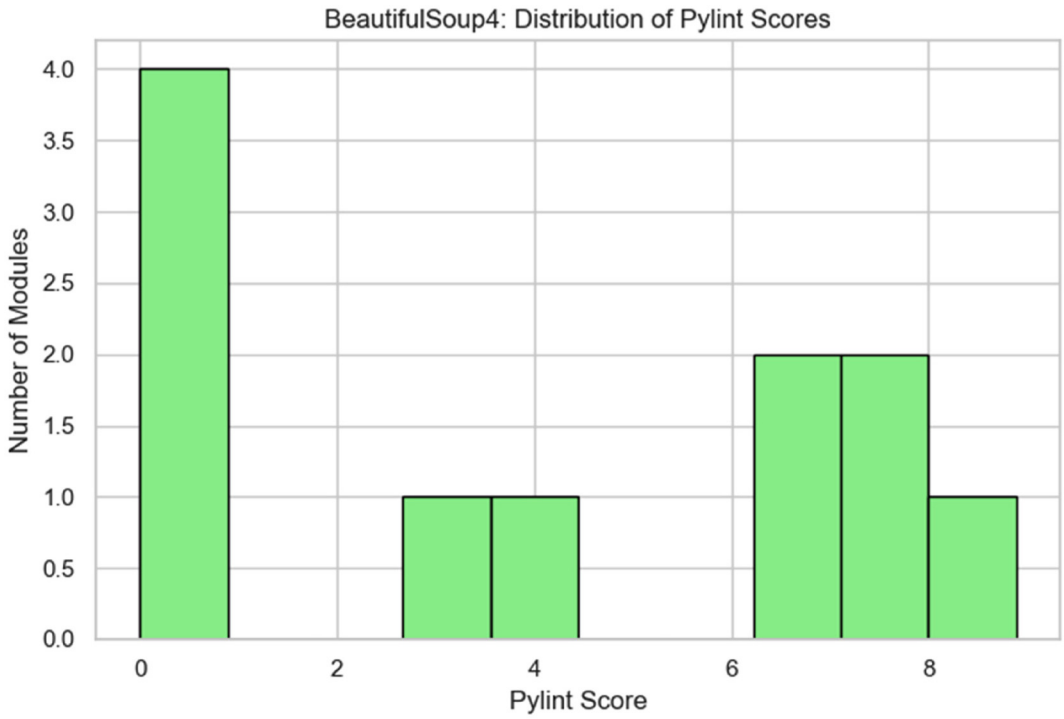


**Figure 2.** Distribution of Pylint scores, showing a bimodal spread across modules

at 0. This answers RQ1 by pinpointing the most complex and least maintainable modules within each project.

**5.2. RQ2: Differences between Projects.** Comparing the two projects, Requests demonstrates higher average code quality. It has a lower overall cyclomatic complexity (2.87 vs. 3.40) and a higher average Pylint score (5.96 vs. 3.98). While both projects have a similar number of total PEP8 violations, the distribution is different; BeautifulSoup4's violations are concentrated in a few key modules, whereas Requests' are more spread out. This suggests that Requests may have more consistent coding practices and a more mature quality assurance process, likely due to its larger and more structured contributor base.

**5.3. RQ3: Refactoring Candidates.** Based on the aggregated metrics, we can clearly identify the most critical refactoring candidates:

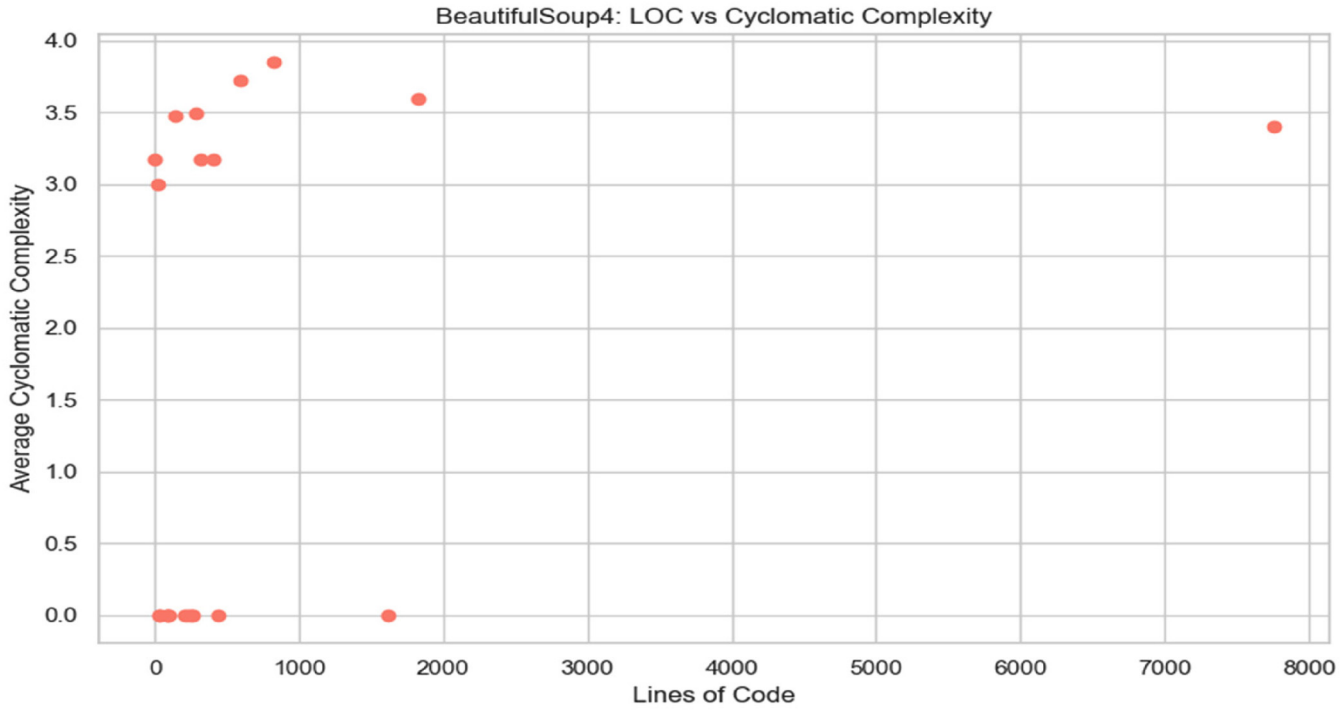


**Figure 3.** Relationship between lines of code (LOC) and cyclomatic complexity in BeautifulSoup4

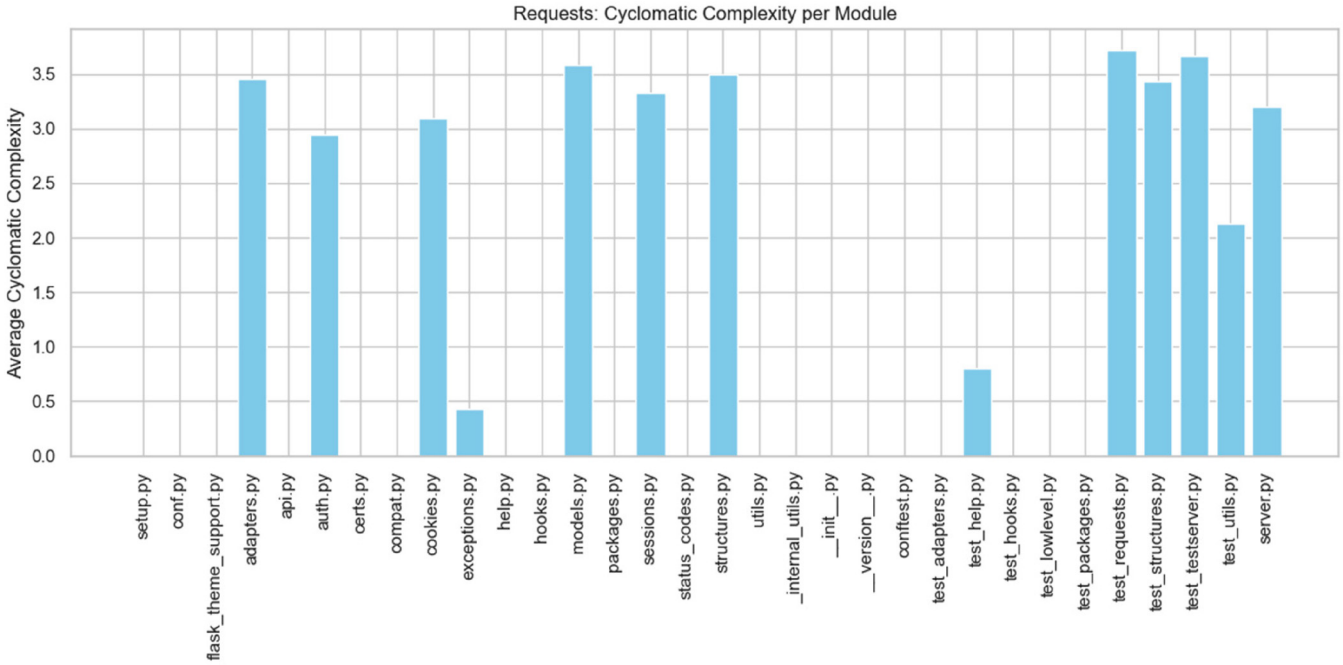


**Figure 4.** Average cyclomatic complexity per module in Requests; test-related modules (test_requests.py, test_testserver.py) show the highest values

- **BeautifulSoup4:** dammit.py is the highest priority, requiring immediate refactoring to simplify logic and fix style issues. test_tree.py and testing.py are also important targets.
- **Requests:** test_requests.py should be broken down into smaller, less complex test suites. models.py also shows high complexity and low maintainability, warranting investigation.This concentration of violations in complex modules aligns with literature linking style and quality[10].

**5.4. Analytical Depth and Broader Implications.** A correlational analysis reveals a moderate negative relationship between cyclomatic complexity and maintainability index across all modules ($r = -0.58$, $p < 0.001$). This statistically significant correlation underscores that as code becomes more complex, its maintainability decreases, validating the interconnected nature of these metrics.

The findings have several practical implications for software sustainability:

- **Readability and Collaboration:** A high density of PEP8 violations, as seen in dammit.py, directly impairs readability. This creates a barrier to entry for new contributors and slows down collaborative development, ultimately threatening the long-term health of open-source projects.
- **Technical Debt Management:** The identified high-complexity, low-MI modules represent significant technical debt. Prioritizing their refactoring is an investment in long-term sustainability, reducing the future cost of changes and bug fixes.
- **Quality Assurance Automation:** Integrating static analysis tools into CI/CD pipelines can prevent the accumulation of technical debt by enforcing quality gates for new contributions, ensuring consistent quality across distributed teams.

## 6. LIMITATIONS AND ETHICAL CONSIDERATIONS

This study has several limitations. The analysis is restricted to two Python projects, which, while influential, limit the generalizability of the findings. Future work should expand to a larger and more diverse set of projects. Furthermore, static analysis provides a snapshot of code quality and does not capture runtime behavior or developer intent.

**Table 2.** *Static Analysis Metrics for Requests*

| Module/File | LOC | CC (avg) | Maintainability Index | Pylint Score | PEP8 Violations |
|---|---|---|---|---|---|
| test_requests.py | 3040 | 3.72 | 0.0 | 7.66 | 61 |
| models.py | 1039 | 3.58 | 21.69 | 7.83 | 28 |
| adapters.py | 696 | 3.45 | 47.45 | 6.49 | 36 |
| test_testserver.py | 165 | 3.67 | 49.21 | 8.58 | 5 |
| internal_utils.py | 50 | 0.0 | 100.0 | 10.0 | 2 |
| SUMMARY | 11248 | 2.87 | 73.26 | 5.96 | 533 |

*Note: A Maintainability Index (MI) of 0.0, as seen in test_requests.py, occurs when the calculated index is non-positive due to very high complexity and size overwhelming the effect of comments, and is reported as 0 by the Radon tool.*

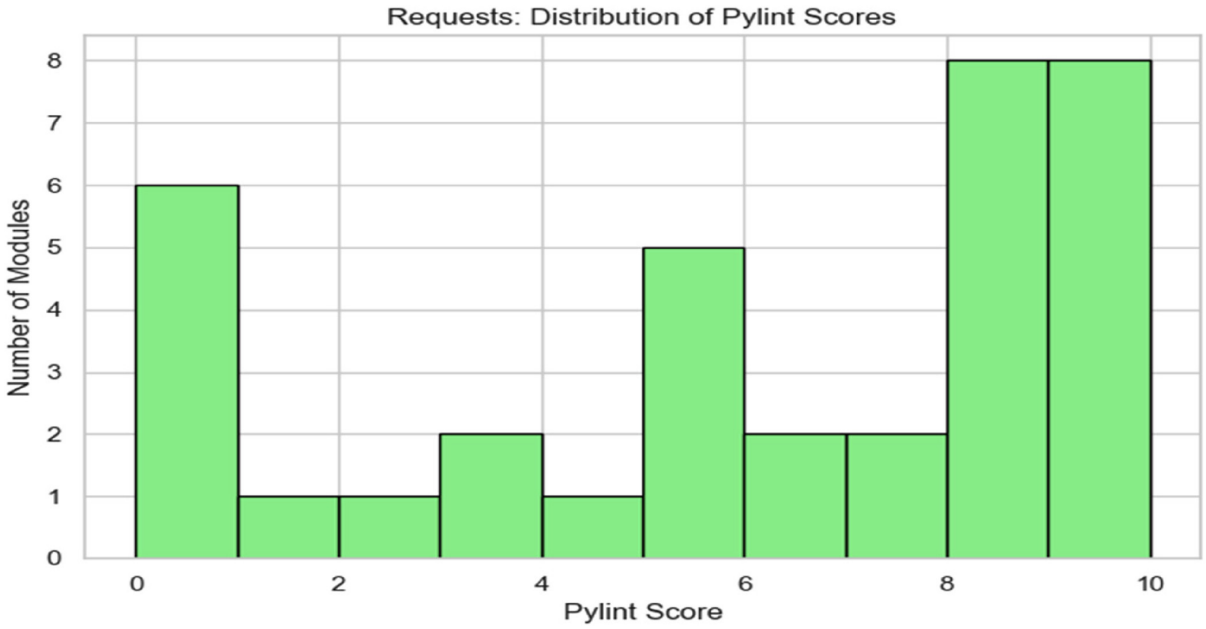


**Figure 5.** Distribution of Pylint scores across Requests modules; higher scores reflect better overall code quality

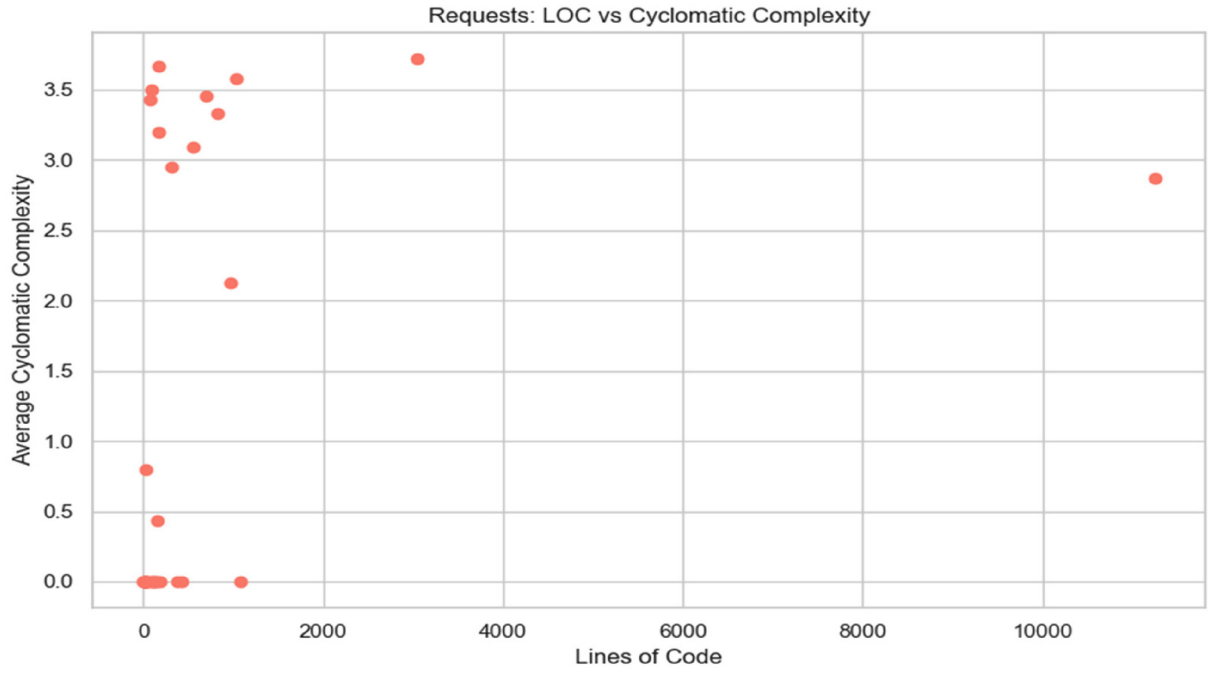


**Figure 6.** Relationship between lines of code (LOC) and cyclomatic complexity in Requests; moderate positive correlation ($r = 0.41$)

Regarding ethical considerations, this study exclusively used publicly available source code from open-source repositories. No private or proprietary software was analyzed. The research respects the licenses of the projects studied and aims to contribute back to the open-source community by providing insights that can help improve their code quality.

## 7. CONCLUSION

This study presented a reproducible static analysis of code quality in the Requests and BeautifulSoup4 projects. While generally maintainable, certain modules exhibited high complexity and numerous style violations, identifying them as prime refactoring candidates. Requests demonstrated more consistent code quality overall compared to BeautifulSoup4.

The primary contribution of this work is the detailed, empirical, and reproducible snapshot of code quality in these essential libraries, providing actionable insights for their maintainers. The significant negative correlation between complexity and maintainability reinforces the importance of monitoring these metrics. Furthermore, we contribute a transparent methodology and a full replication package for the community.

Beyond immediate maintainer benefits, this work has broader societal and educational implications. By improving code quality in foundational open-source libraries, we contribute to more reliable and sustainable software ecosystems that support

education, research, and industry. The reproducible methodology and public dataset also serve as a valuable resource for students and educators teaching software engineering best practices, promoting transparency and rigor in empirical software research.

For future work, we plan to expand this analysis to a larger set of projects from different domains, including historical analysis to track quality evolution, and incorporate machine learning techniques to predict defect-prone modules based on these static metrics.

## 8. SUPPLEMENTARY INFORMATION

All code, raw data, and detailed instructions for reproducing the results, including the exact shell commands and environment setup, are available in our supplementary repository: https://github.com/BashirAdam/Static-Analysis-of-Code-Quality-in-Open-Source-Python- Projects. A single script (run_analysis.py) is provided to regenerate all tables and figures.

## AFFILIATIONS AND AUTHOR DETAILS

### Undergraduate Author and Corresponding Author

**Bashir Adam Ahmed Ali** – *Department of Software Engineering, Ostim Technical University, Ostim, 06374 Yenimahalle/Ankara, Turkiye;* 0009-0009-0267-2904
Email: 210208999@ostimteknik.edu.tr

## ACKNOWLEDGEMENTS

The author would like to acknowledge the Department of Software Engineering at Ostim Technical University for their support. Special thanks also to the open-source developer communities behind BeautifulSoup4 and Requests, whose projects formed the basis of this analysis.

## REFERENCES

(1) Python Software Foundation. (2024). Python. https://www.python.org.

(2) Liu, H., Jin, J., Xu, Z., Zou, Y., & Bu, Y. (2021). Deep learning-based code smell detection. IEEE Transactions on Software Engineering, 47(9), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

(3) Radon. (2024). Radon Documentation. Radon Project. https://radon.readthedocs.io.

(4) Pylint. (2024). Pylint Documentation. Python Code Quality Authority. https://pylint.org.

(5) Flake8. (2024). Flake8 Documentation. Python Code Quality Authority. https://flake8.pycqa.org.

(6) Fontana, F. A., Mantyla, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine-learning techniques for code-smell detection. Empirical Software Engineering, 21(4), 1143-1191.https://doi.org/10.1007/s10664-015-9376-6

(7) Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A. & Gall, H. C. (2018). Context is king: The developer perspective on the usage of static analysis tools. Proceedings of SANER 2018 25th International Conference on Software Analysis, Evolution and Re-engineering, 38–49 (2018). https://doi.org/10.1109/SANER.2018.8330195

(8) McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837

(9) Iftikhar, U., Ali, N. B., Börstler, J., & Usman, M. (2024). A tertiary study on links between source code metrics and external quality attributes. Information and Software Technology, 165, 107348. https://doi.org/10.1016/j.infsof.2023.107348

(10) Siddik, M. S., & Bezemer, C.-P. (2023). Do code quality and style issues differ across (non-)machine learning notebooks? Yes! In 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 72–83). IEEE. https://doi.org/10.1109/SCAM59687.2023.00018

(11) Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., & Oliveto, R. (2017). Automatically assessing code understandability: How far are we? *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 417-427. https://doi.org/10.1109/ASE.2017.8115654

(12) Grotov, K., Tiufanov, S., Serebriakova, Y., Golubev, Y., Kovalchuk, N., & Bryksin, T. (2022). A large-scale comparison of Python code in Jupyter notebooks and scripts. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22) (pp. 653664). ACM. https://doi.org/10.1145/3524842.3528447

(13) Chen, Z., Chen, L., Ma, W., Zhou, X., Zhou, Y., & Xu, B. (2018). Understanding metric-based detectable smells in Python software: A comparative study. Information and Software Technology, 94, 1429. https://doi.org/10.1016/j.infsof.2017.09.011

(14) AlOmar, E. A., Mkaouer, M. W., & Ouni, A. (2021). Can refactoring be self-affirmed? An exploratory study on how developers document refactoring activities. IEEE Access, 9, 21449-21467. https://doi.org/10.1109/ACCESS.2021.3055175

(15) Liu, H., Jin, J., Xu, Z., Zou, Y., & Bu, Y. (2021). Deep learning-based code smell detection. IEEE Transactions on Software Engineering, 47(9), 1811-1837. https://doi.org/10.1109/TSE.2019.2961345

(16) Reitz, K. (2024). Requests: HTTP for Humans. GitHub repository. Python Software Foundation. https://github.com/psf/requests.

(17) Richardson, L. (2024). Beautiful Soup Documentation. Crummy.com. https://www.crummy.com/software/BeautifulSoup/.