Exploring the Impact of Modularity on Software Quality Across Different Python Application Domains

Aasritha Thota, Devi Sree Dornala, Divyasri Harshita Sriram, Keerthana Jukanti, Nikitha Durga Chakka
Lewis University

***Abstract*—This project investigates the relationship between modularity and software quality within various Python applications, utilizing empirical data from widely-used open-source projects—Apache Airflow, SaltStack, and Spyder IDE. Given the importance of modularity in software design, characterized by components or modules that are highly cohesive yet loosely coupled, our research aims to quantify its impact on maintainability and complexity. By employing Radon to measure cyclomatic complexity and maintainability indices, we seek to determine if higher levels of modularity correlate with improved software metrics. Initial findings suggest that projects with better modularity ratings exhibit reduced complexity and enhanced maintainability. This study underscores the potential benefits of modular software architecture, providing significant insights into best practices for Python programming across diverse application domains.**

I. Introduction

Modularity in software design promotes the division of systems into discrete, manageable components, facilitating easier management, scalability, and adaptability to technological changes. Despite Python's extensive use in various applications from web development to machine learning, the impact of modularity on the quality of Python applications is not well-documented. This project aims to fill this gap by empirically analyzing how modularity influences software quality across different Python application domains. We selected three prominent open-source Python projects—Apache Airflow, SaltStack, and Spyder IDE—to serve as our study subjects, each representing a unique domain. By utilizing Radon to measure cyclomatic complexity and maintainability indices, we seek to quantify the relationship between modularity and code quality, hypothesizing that greater modularity correlates with reduced complexity and enhanced maintainability. This research is expected to provide valuable insights into the benefits of modular software architecture, guiding Python developers in best practices for software design and potentially influencing methodologies across various industries. Through this study, we aim to underscore the practical advantages of modularity in improving software quality and developer efficiency

II. Method or Approach

To systematically evaluate the impact of modularity on software quality, our study employs a mixed-methods approach, combining quantitative analysis with a review of software architecture. The core of our empirical analysis is conducted on three distinct Python projects: Apache Airflow, SaltStack, and Spyder IDE. These projects were selected based on their diverse application domains, which allows for a broad assessment of modularity across different types of software systems.

A. Tool Selection and Setup

For our quantitative analysis, we use two tools, Radon and Lizard, which are well-regarded in the software engineering community for their ability to measure code quality metrics effectively. Radon is utilized to calculate cyclomatic complexity and maintainability index, providing insights into the code's structural complexity and its maintainability over time. Lizard offers additional complexity measures and function length analysis, which helps in understanding the granularity of modularity and its execution.

B. Data Collection

Our data collection involves running Radon and Lizard on the latest versions of each selected project. Specific commands used include:

For Radon:

```
radon cc <projectpath> -s -a > <output.txt>
```

This command calculates the cyclomatic complexity for each module, summarizing the data with average values and categorizing them by complexity grade (A-F).

```
radon mi <projectpath> -s > <output.txt>
```

This command provides a maintainability index, also graded from A to F, which assesses the ease of maintaining the code.

For Lizard, we extract metrics related to function length and complexity, which are crucial for evaluating the detailed characteristics of modularity within the code.

C. Analysis

The collected data are then analyzed to observe patterns and correlations between modularity (as reflected in complexity and maintainability scores) and software quality. This includes statistical analysis to confirm or refute our hypothesis that higher modularity (indicated by better grades in complexity and maintainability) is associated with improved software quality. Additionally, we compare the results across the different projects to identify if the impact of modularity varies by application domain.

D. *Interpretation and Reporting*

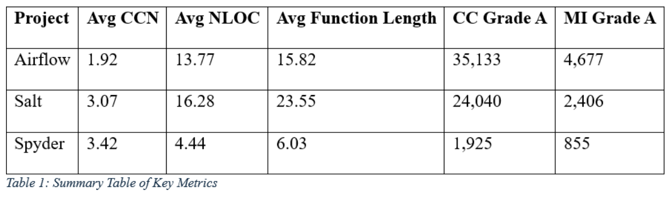
The final step involves interpreting the data in the context of existing literature and the specific characteristics of each project. This will help us draw conclusions about the role of modularity in Python software development and its potential benefits and limitations. These findings will be compiled into a comprehensive report, discussing the implications for software developers and the broader field of software engineering.

III. Results and Discussion

A. *Summary Table of Key Metrics*

The comprehensive analysis of Apache Airflow, SaltStack, and Spyder IDE provided clear insights into how modularity impacts software quality. Below is the summary table combining all relevant metrics from our analysis:

B. *Cyclomatic Complexity Grades Frequency*



Project	Avg CCN	Avg NLOC	Avg Function Length	CC Grade A	MI Grade A
Airflow	1.92	13.77	15.82	35,133	4,677
Salt	3.07	16.28	23.55	24,040	2,406
Spyder	3.42	4.44	6.03	1,925	855

Table 1: Summary Table of Key Metrics

Fig. 1. Correlation Between WMC and Bad Smells

The summary table provides a detailed comparison of key metrics across Apache Airflow, SaltStack, and Spyder IDE, highlighting their respective software complexities and maintainability profiles. Airflow demonstrates the lowest average cyclomatic complexity (CCN) and a moderate function length, coupled with the highest grades in both cyclomatic complexity and maintainability index, indicating a well-modularized and maintainable codebase. In contrast, Salt exhibits higher average CCN and the longest average function length, which correlates with a slightly lower modularity and increased maintenance challenges. Spyder, while having the highest CCN, maintains the shortest function lengths, suggesting compact but complex functions that could benefit from further modular decomposition to enhance readability and maintainability. This comparative analysis underscores the significant impact of modularity on software quality, particularly in how it facilitates easier maintenance and adaptability in varying project scales and complexities.

C. *Average Cyclomatic Complexity per Project*

The bar chart presents a clear visualization of the average cyclomatic complexity (CCN) across three Python projects: Apache Airflow, SaltStack, and Spyder IDE. It reveals that Spyder has the highest average CCN, followed closely by Salt, and then Airflow with the lowest. This ordering indicates that Spyder's functions are generally the most complex, which may reflect the intricate operations typical of an IDE.

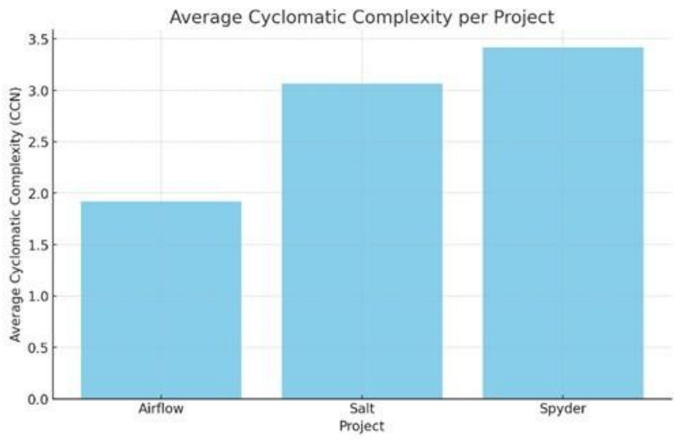


Fig. 2. Average Cyclomatic Complexity per Project

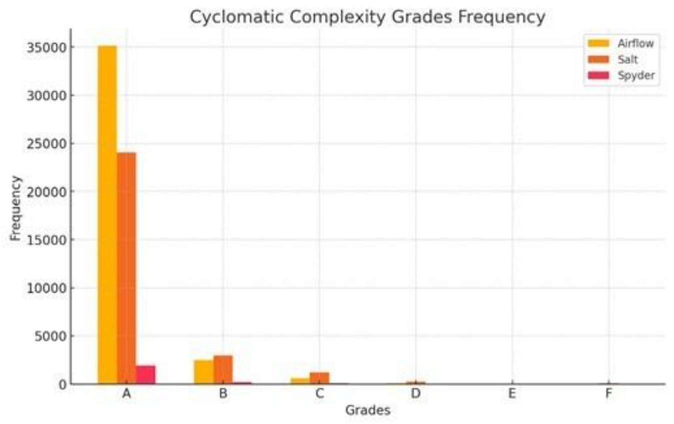


Fig. 3. Cyclomatic Complexity Grades Frequency

D. *Cyclomatic Complexity Grades Frequency*

The bar chart compares the frequency distribution of cyclomatic complexity grades across Apache Airflow, SaltStack, and Spyder IDE. Airflow displays a dominant number of Grade A ratings, indicating a high prevalence of simpler, more maintainable functions. In contrast, Salt and Spyder exhibit a greater presence in higher complexity grades (B and C), with Spyder also showing minor entries in the most complex grades (D, E, F). This visualization suggests that Airflow's codebase benefits from lower complexity, enhancing its maintainability, while Salt and Spyder contain more complex functions, potentially increasing the maintenance burden.

E. *Maintainability Index Grades Frequency*

The bar chart compares the distribution of maintainability index grades across Apache Airflow, SaltStack, and Spyder IDE. Airflow shows a significantly high frequency of Grade A ratings, indicating that a large portion of its code is easy to maintain and adapt. Salt follows with a moderate presence of Grade A but also shows entries in Grades B and C, suggesting areas for potential improvement. Spyder has the lowest frequencies overall, particularly in Grade A, highlighting a need for enhanced maintainability in its codebase.

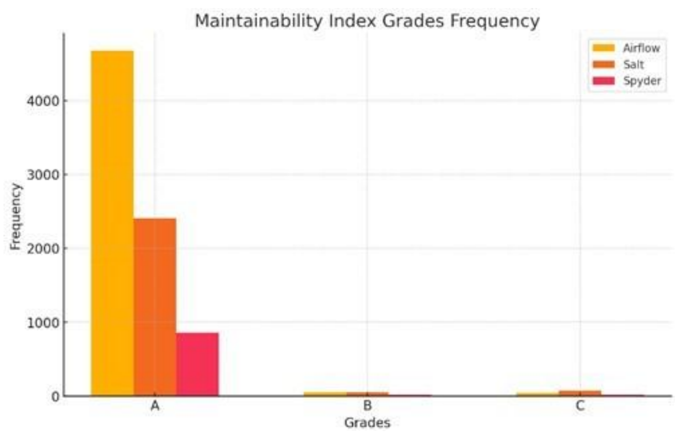


Fig. 4. Maintainability Index Grades Frequency

IV. Observation on Software Architecture and Design

In analyzing the software architecture and design of the selected projects—Apache Airflow, SaltStack, and Spyder IDE—it's evident that their architectural choices significantly influence their modularity and overall software quality. Apache Airflow adopts a microservices-like architecture, allowing tasks to be decoupled and managed independently, which promotes high modularity and maintainability. This modular structure is crucial for its functionality as a workflow management system, enabling developers to easily add, modify, or remove components without affecting the entire system. In contrast, SaltStack employs a more modular monolithic architecture that organizes the code into manageable packages but still exhibits higher complexity in its modules, suggesting a need for further decomposition to enhance maintainability. Spyder IDE, with its plug-in architecture, exemplifies modularity aimed at extending the software's functionality through independent modules. However, the compact yet complex functions within its codebase hint at a trade-off between modular flexibility and readability, requiring careful management of dependencies to avoid tight coupling. Across these projects, adherence to design principles like DRY and SOLID is evident, although varying levels of implementation influence their maintainability outcomes. A closer look at their dependency management and modular decomposition strategies reveals that while modular architectures generally enhance maintainability, they must be supported by clear code organization practices and effective dependency handling to maximize their benefits. This observation highlights the critical role of architectural decisions and design patterns in determining the maintainability and scalability of Python applications across different domains.

V. Conclusion

This project effectively explores the impact of modularity on software quality across different Python application domains, utilizing empirical analysis of three significant open-source projects: Apache Airflow, SaltStack, and Spyder IDE. The findings indicate a clear correlation between modularity and key software quality metrics, such as cyclomatic complexity and maintainability index. Airflow stands out as the most maintainable and least complex project, demonstrating the benefits of thoughtful modular design. In contrast, Salt and Spyder, while functional, exhibit higher complexities and lower maintainability ratings, suggesting opportunities for refactoring and improvement. The insights gained from this study underscore the importance of modularity in software development, particularly within the context of Python applications. By illustrating how modular design can lead to lower complexity and enhanced maintainability, this research provides valuable guidance for developers and organizations aiming to optimize their codebases. As software continues to evolve, adopting modular practices will be crucial for ensuring adaptability, scalability, and long-term sustainability in increasingly complex programming environments.

References

[1] H. Koziolek, "Sustainability Evaluation of Software Architectures: A Systematic Review," *Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '11*, 2011. Available: https://doi.org/10.1145/2000259.2000263.

[2] P. Avgeriou, P. Kruchten, P. Lago, and A. Grubb, "An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt," *Proceedings of the 10th International Conference on Quality Software*, 2007. Available: https://doi.org/10.1109/QSIC.2010.58.

[3] Apache Software Foundation, "Apache Airflow," Retrieved from: https://airflow.apache.org/.

[4] SaltStack, "Salt Open," Retrieved from: https://github.com/saltstack/salt.

[5] Spyder IDE, "Spyder - The Scientific Python Development Environment," Retrieved from: https://github.com/spyder-ide/spyder.

[6] Python Code Quality Authority, "Radon: Code Metrics," Retrieved from: https://radon.readthedocs.io/en/latest/.

[7] Lizard, "Lizard - A Code Complexity Analyzer," Retrieved from: https://github.com/terryyin/lizard.