Evaluating the Impact of Modularity in Python Applications Across Various Domains

By

Franklin Antony

Manikanta Iskala

Samba Siva Rao Komerisetti

Spandana Peravali

Sravan Kumar Pitta

Software Architect and Design

Contents

Abstract

Modularity is a key design principle in software engineering that enhances code readability, scalability, and maintainability. This study evaluates modularity and its impact on code quality across seven Python projects spanning web development and data science domains. Using static code analysis tools such as Radon, we extracted key metrics including cyclomatic complexity, maintainability index (MI), and structural code breakdown (LOC, comments, blank lines). Our comparative analysis reveals notable differences in complexity and maintainability patterns between data science and web-based projects. Projects like Django and Scikit-learn exhibit high code volume and complexity, while Flask and FastAPI reflect more concise and modular designs. Maintainability was generally strong across all repositories, though some exhibited high-complexity functions. These insights provide actionable evidence for developers to adopt modular design more effectively, especially in large-scale collaborative codebases. This work underscores the value of empirical software analysis in shaping best practices and optimizing software development workflows.

Introduction

Modular programming is a cornerstone of robust software architecture. By dividing a large program into smaller, self-contained modules, developers can achieve greater code clarity, reuse, and maintainability. In the Python ecosystem, which is widely used across domains such as web development, data science, and automation, modularity plays a pivotal role in managing growing codebases. However, while modular design is well-understood conceptually, its actual implementation and impact in real-world Python projects remain underexplored.

This study focuses on quantitatively assessing modularity in seven open-source Python repositories, including frameworks like Flask and Django, and data-focused libraries like Scikit-learn and Statsmodels. We utilized static analysis tools such as Radon to compute cyclomatic complexity, maintainability index, and raw metrics like lines of code and comment density. These metrics help identify code segments that are complex, poorly structured, or potentially less maintainable.

By comparing projects across different domains, we aim to uncover patterns that reflect how modular design influences software quality. The analysis not only highlights the technical aspects of modularity but also draws attention to development practices that either foster or hinder maintainability. Our goal is to offer practical insights that Python developers can apply to design cleaner, more modular code structures regardless of the application domain.

Methodology

To conduct a systematic and reproducible analysis of modularity in Python projects, we adopted an empirical approach involving data collection, metric extraction, parsing, and comparative visualization. This section outlines the methodology in detail, spanning repository selection, command-based metric extraction, and Python-based result parsing.

1. Project Selection

Seven open-source Python projects were selected from GitHub to represent both **web development** and **data science** domains:

- **Web development**: django, flask, fastapi, sanic, tornado
- **Data science**: scikit-learn, statsmodels

These projects vary in size, architecture, and popularity, making them ideal for analyzing modularity across contexts. All repositories were cloned locally to enable static analysis without network dependencies.

2. Metric Extraction with Radon

To evaluate code complexity and maintainability, we used **Radon**, a Python tool for static code analysis. Key metrics extracted included:

- **Cyclomatic Complexity (CC)**
- **Maintainability Index (MI)**
- **Raw Metrics**: LOC, LLOC, SLOC, comment lines, blank lines

All metrics were extracted using batch PowerShell scripts for automation. Below is a sample script:

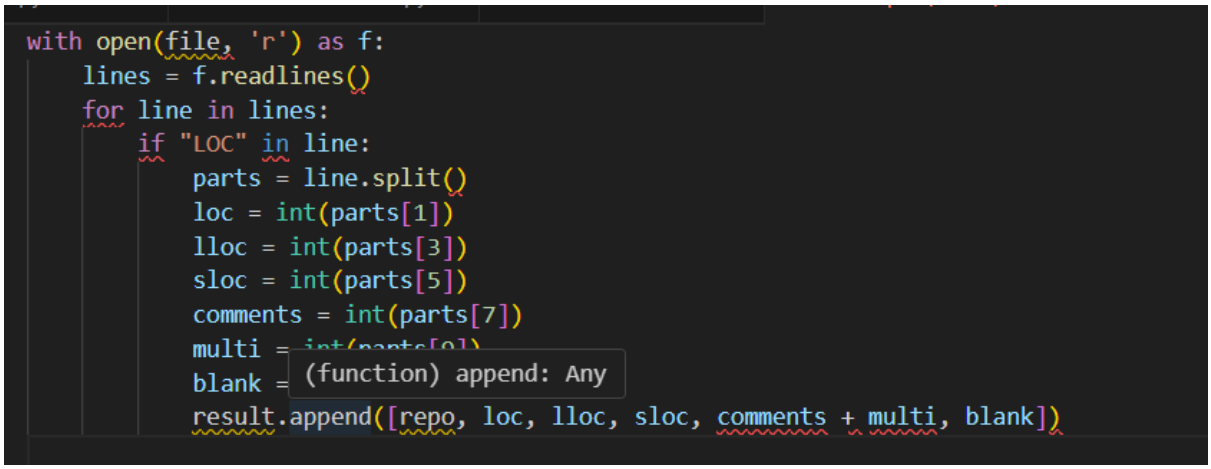
```
# Define repositories
$repos = @("flask", "fastapi", "django", "tornado", "sanic", "scikit-learn", "statsmodels")
# Extract Cyclomatic Complexity
foreach ($repo in $repos) {
    radon cc $repo -s > "$repo`_cc.txt"
    radon cc $repo -a > "$repo`_cc_avg.txt"
    radon cc $repo -s --show-closures > "$repo`_cc_funcs.txt"
}
# Extract Maintainability Index
foreach ($repo in $repos) {
    radon mi $repo -s > "$repo`_mi.txt"
}
# Count Python files
foreach ($repo in $repos) {
    $count = (Get-ChildItem -Recurse $repo -Include *.py -File).Count
    "$count" | Out-File "$repo`_pycount.txt"
}
# Extract Raw Metrics
foreach ($repo in $repos) {
    radon raw $repo > "$repo`_raw.txt"
}
```

3. Parsing Extracted Files with Python

After raw text files were generated, we used custom Python scripts to parse them into structured Excel sheets for analysis. Four separate parsing scripts were used:

- **Parse CC Average**
- **Parse Detailed CC**
- **Parse MI Grades**
- **Parse Raw Metrics**

Example snippet from Parse Raw Metrics.py:



All parsed data were saved in structured Excel files:

- cc_avg_summary.xlsx
- cc_detailed_summary.xlsx
- mi_grade_summary.xlsx
- raw_metrics_summary.xlsx

4. Data Consolidation and Visualization

Once parsed, the datasets were merged and prepared for visualization. For example, maintainability data from all sheets in mi_grade_summary.xlsx were combined using pandas:

```
mi_combined = []
for sheet_name, df in pd.read_excel("mi_grade_summary.xlsx", sheet_name=None).items():
    df["Project"] = sheet_name
    mi_combined.append(df)
```

mi_df = pd.concat(mi_combined, ignore_index=True)

5. Tools and Libraries Used

- **Radon**: For code metric extraction
- **PowerShell**: For automating metric extraction commands
- **Python (pandas, matplotlib, seaborn)**: For parsing, transforming, and visualizing the data
- **Excel**: For manual review and backup

6. Reproducibility and Scalability

All scripts were modular and parameterized, enabling reuse across other projects. This modularity ensured consistency across metric extraction, supported debugging, and allowed easy updates for future analysis extensions.

Results & Analysis

This section presents a comprehensive examination of the modularity, complexity, and maintainability of selected Python projects. Seven open-source repositories were analyzed: Django, Flask, FastAPI, Tornado, Sanic, Scikit-learn, and Statsmodels. Metrics such as cyclomatic complexity, code size, and maintainability index were extracted and visualized to identify trends and evaluate project structure quality.

1. Cyclomatic Complexity Summary

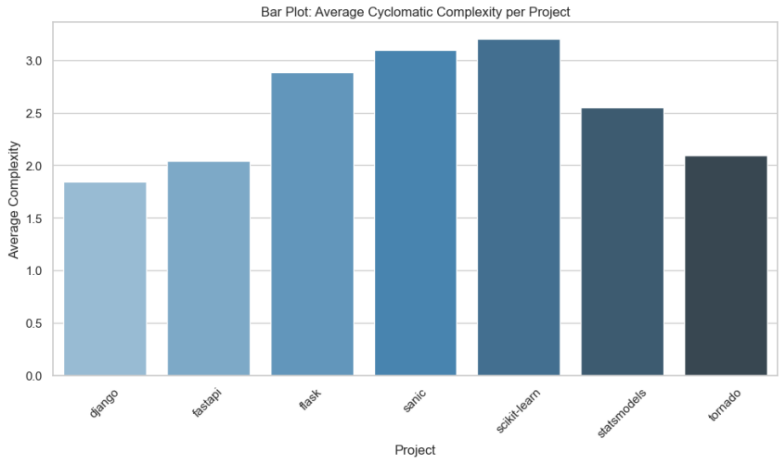
Cyclomatic complexity reflects the logical complexity of functions. Projects with higher complexity may be harder to maintain and test.

Table 1: Cyclomatic Complexity Summary

Project	**Functions Analyzed**	**Avg Complexity**	**Max Complexity**	**Functions >10 CC**
django	37551	1.82	94	416
fastapi	4270	2.01	44	31

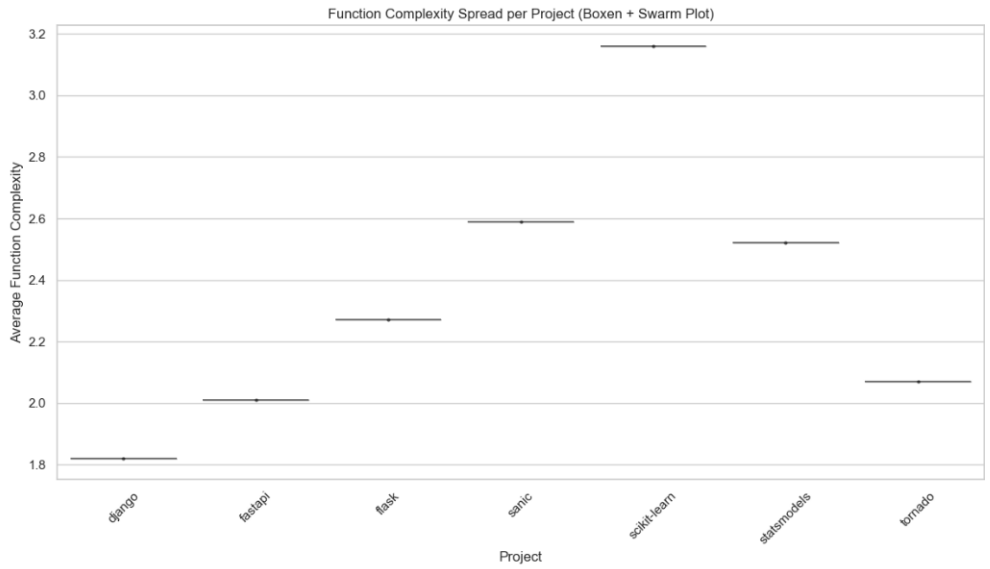
flask	1413	2.27	27	22
sanic	3747	2.59	39	96
scikit-learn	11354	3.16	71	473
statsmodels	14638	2.52	77	431
tornado	3669	2.07	50	44

Figure 1: Average Cyclomatic Complexity per Project



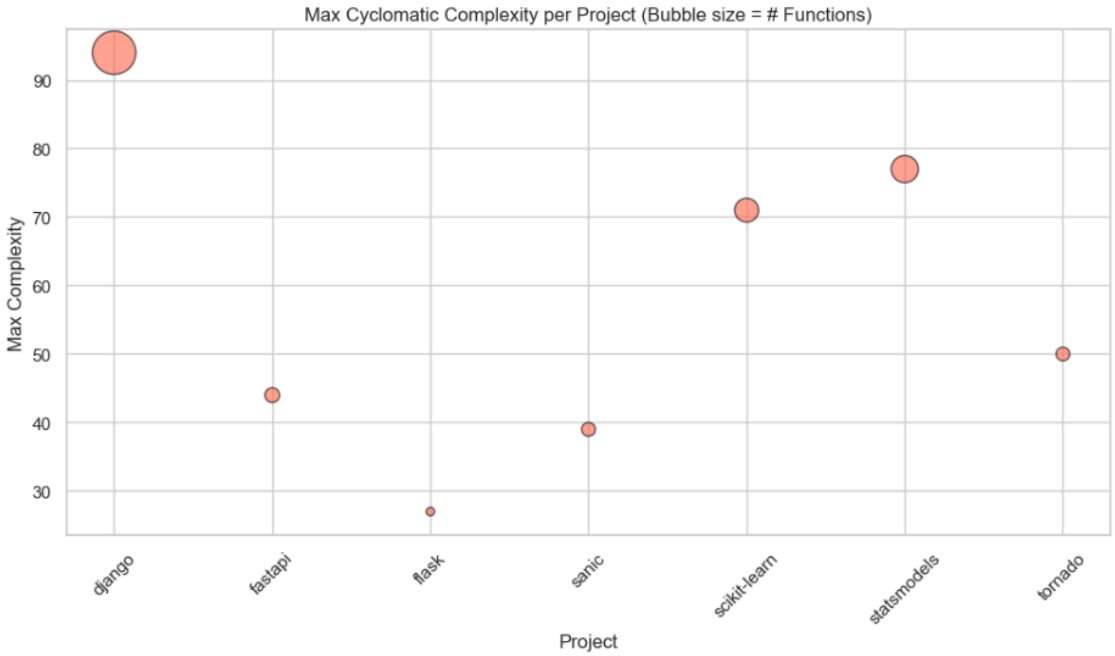
This bar plot shows the comparative average complexity of functions. Scikit-learn, Sanic, and Statsmodels appear more complex on average, while Django and FastAPI exhibit lower complexity, reflecting more modular or concise logic.

Figure 2: Function Complexity Spread (Boxen + Swarm)



The boxen plot with swarm overlay provides a deeper look at function complexity variation. Scikit-learn displays a wide range of complexities, suggesting both simple and highly intricate function blocks, whereas Django and FastAPI maintain more consistent low-complexity patterns.

Figure 3: Max Cyclomatic Complexity vs Function Count (Bubble Plot)



The bubble plot relates maximum complexity and the number of functions. Django stands out with both high function volume and maximum complexity. Statsmodels and Scikit-learn follow, reinforcing the correlation between large codebases and complexity hotspots.

2. Code Structure Metrics

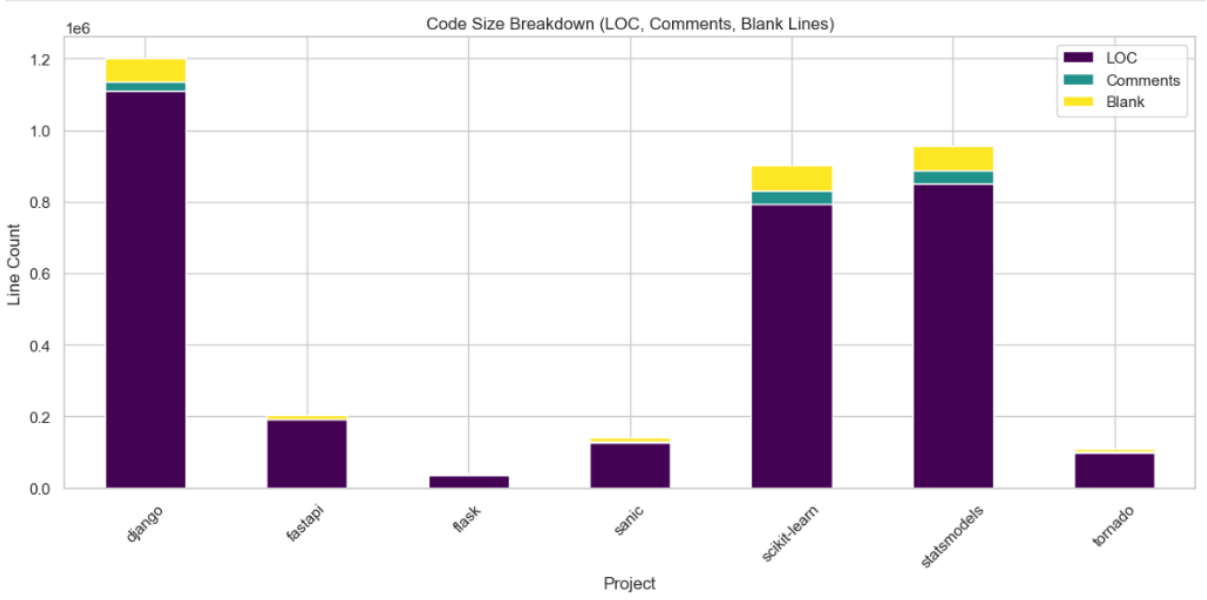
Understanding the breakdown of LOC (Lines of Code), comments, and blank lines provides insight into documentation quality and structure.

Table 2: Code Size and Comments Summary

Project	LOC	LLOC	SLOC	Comments	Blank Lines
django	1109426	246974	372567	24724	65456
fastapi	191378	29219	74163	966	12413

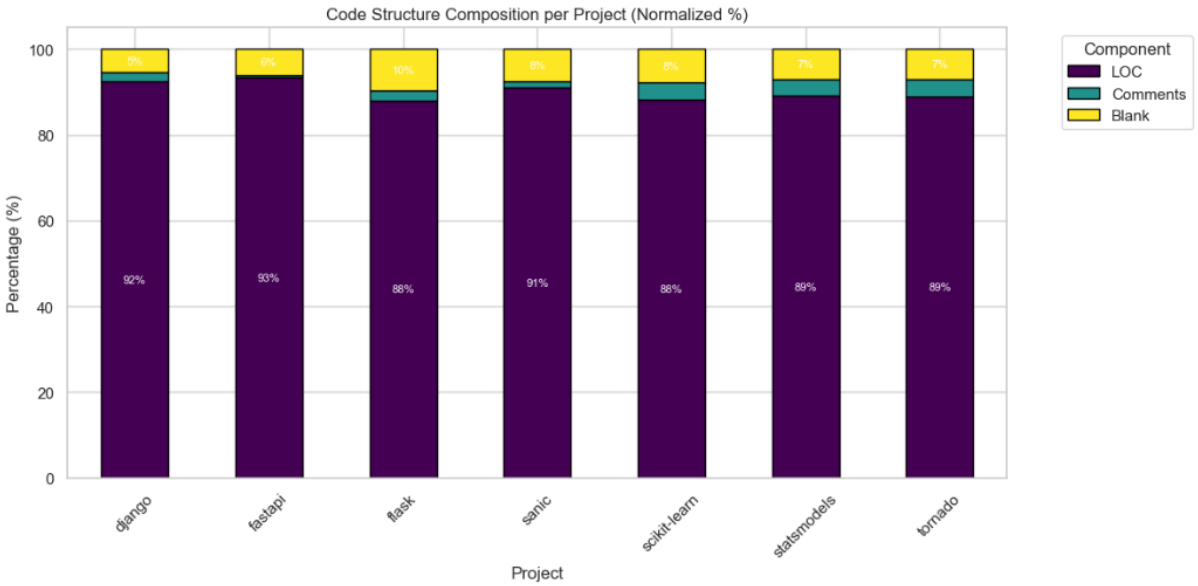
flask	37088	9076	10193	952	4097
sanic	128282	27529	43109	1973	10593
scikit-learn	794684	150940	227679	36132	69966
statsmodels	851166	174197	247380	36393	67357
tornado	97952	23167	28651	4513	7609

Figure 4: Grouped Bar Chart - LOC, Comments, Blank Lines



This grouped bar chart shows clear codebase size differences. Django, Scikit-learn, and Statsmodels dominate in LOC, with healthy proportions of comments and blank lines, indicating potential for maintainability. Lightweight frameworks like Flask and Tornado remain compact.

Figure 5: Code Structure Composition (Normalized %) per Project

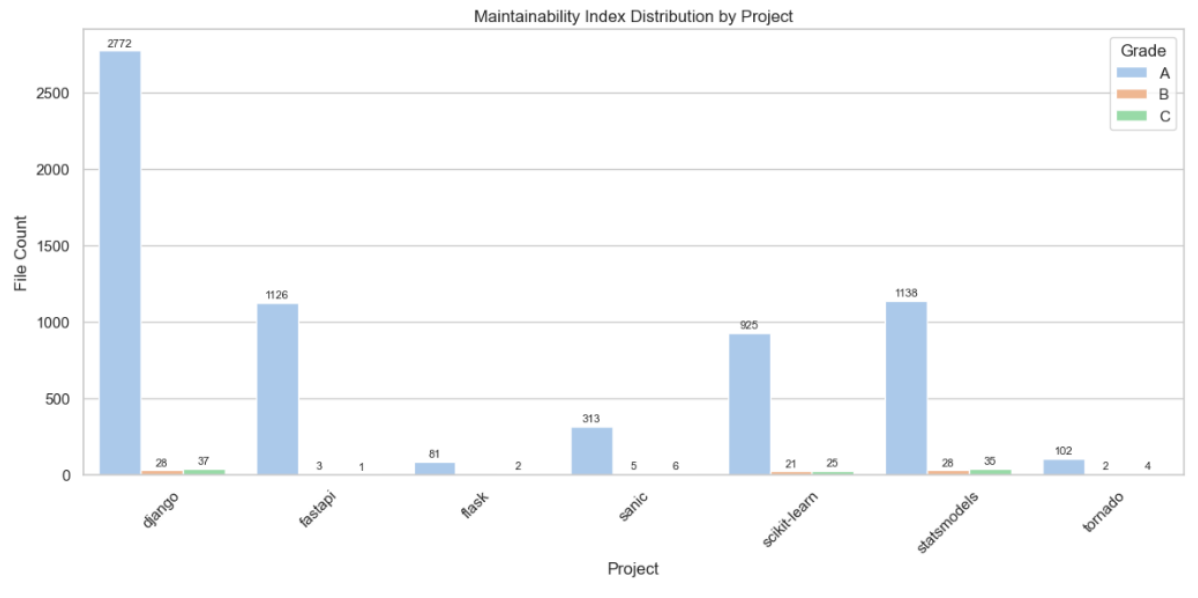


Although total LOC varies greatly, normalized percentages show that most projects allocate 85–93% of code to logic (LOC), and only a small fraction to comments and whitespace. Flask has a slightly higher comment and blank line share, which may aid readability.

3. Maintainability Index (MI) Grade Distribution

MI grades provide a high-level evaluation of how maintainable each project's codebase is.

Figure 6: Maintainability Index Distribution by Project



Most projects show an overwhelming dominance of Grade A (high maintainability), particularly Django and FastAPI. A small proportion of Grade B and C exists in larger projects like Scikit-learn and Statsmodels, often due to their dense analytical logic.

4. Summary Observations

- Projects with large function counts (Django, Scikit-learn) show more complexity variation but maintain strong MI grades.
- Frameworks like Flask and FastAPI maintain low average complexity and lean structure—hallmarks of good modular design.
- High LOC doesn't always mean poor maintainability; documentation (comment density) and function design play a significant role.
- Data science libraries tend to be more complex due to analytical depth but are still well-maintained overall.

This comprehensive comparison reinforces the importance of managing complexity and modularity, especially in large-scale or collaborative Python projects.

Discussion

The comparative evaluation of modularity in Python projects highlights several recurring themes. First, the results affirm the hypothesis that modularity—while influenced by domain—often reflects deliberate architectural decisions. For example, web development projects such as Flask and FastAPI show relatively lower cyclomatic complexity and smaller codebases, supporting the idea that these frameworks are optimized for simplicity and readability. Their modular nature makes them highly maintainable and suitable for agile development cycles.

In contrast, data science libraries like Scikit-learn and Statsmodels display higher average and maximum complexity values. This increased complexity stems from the inherent requirements of mathematical modeling, statistical computation, and model orchestration. However, despite their size and intricacy, these projects still demonstrate high maintainability grades. This

suggests that complexity does not necessarily hinder maintainability if offset by strong documentation, well-separated logic, and reusable function structures.

The bubble and boxen plots also reveal that while projects like Django and Statsmodels have high function volumes, their internal complexity is well distributed. This modular balance is likely a result of established coding conventions and community contributions. Furthermore, the normalized code composition chart indicates that while code volume varies widely, the proportion of comments and whitespace remains consistent across projects. This consistency reflects an adherence to documentation practices that foster long-term collaboration and clarity.

Ultimately, the patterns indicate that modularity is not just about reducing complexity, but about structuring complexity in an intentional and transparent way. Projects that excel in maintainability often exhibit traits such as smaller, well-defined functions, strategic code grouping, and consistent commenting. These findings provide actionable benchmarks for developers aiming to improve software design across varying scales and domains.

Conclusion

This study explored the impact of modularity across seven open-source Python projects spanning web development and data science. Using metrics like cyclomatic complexity, maintainability index, and structural code indicators (LOC, comments, blank lines), we systematically evaluated how modularity shapes software quality.

Our findings demonstrate that modularity positively correlates with maintainability but does not always mean reduced complexity. High-performing projects often manage complexity through separation of concerns, documentation, and reusable components. Lightweight frameworks like Flask and FastAPI exemplify lean modularity, while larger libraries such as Scikit-learn and Statsmodels illustrate how even complex ecosystems can maintain structure and clarity.

The methodology used including static analysis via Radon and structured parsing via Python ensures this approach is reproducible and scalable. The results serve as a reference point for Python developers seeking to enhance code quality and scalability. Overall, modular programming remains a cornerstone of sustainable software engineering, adaptable to various domains and project scopes.

References

- Fowler, M. (2004). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.
- Radon Python Tool Documentation: https://radon.readthedocs.io
- GitHub repositories: Django, Flask, FastAPI, Tornado, Sanic, Scikit-learn, Statsmodels
- PEP 8 – Style Guide for Python Code: https://peps.python.org/pep-0008/
- Lutz, M. (2013). *Learning Python*. O'Reilly Media.