**Architectural Smells and Their Impact on Software Evolution**

Siva Reddy Potlapati

[SivaReddyPotlapati@lewisu.edu](mailto:SivaReddyPotlapati@lewisu.edu)

Lewis University, Fall 2024

# ABSTRACT

Software structural design flaws known as "architectural smells" can have a negative impact on the program's performance, evolution, and maintainability over time. The goal of this study is to empirically evaluate the influence of architectural smells on software evolution by detecting and examining them in Python projects of all sizes. Nine Python programs are selected for the study and arranged according to their difficulty and size. To find possible architectural smells like God Classes, Cyclic Dependencies, and Feature Envy, cohesion percentages are computed for every class. Architectural smells are identified by automated tools like SonarQube and Radon, and metrics like defect rates, code churn, and maintainability index are used to gauge the evolutionary impact. The study aims to shed light on how architectural smells affect software system development and offer solutions for reducing their impact in Python-based projects. The results advance knowledge of recommended techniques for controlling architectural smells and guaranteeing software quality over the long run.

**Keywords**

Architectural Smells, Software Evolution, Python Projects, Cohesion Analysis, Code Maintainability, Software Metrics, Empirical Study

# 1 INTRODUCTION

The maintainability, scalability, and overall quality of software systems are significantly influenced by their architecture. As software evolves to meet changing requirements, inherent structural issues—termed "architectural smells"—can emerge. These smells indicate deeper systemic problems, potentially hindering long-term adaptability, increasing maintenance costs, and degrading system performance. Identifying and addressing architectural smells is therefore crucial for sustainable software evolution.

This study aims to detect and analyze architectural smells in Python projects of varying sizes—small, medium, and large. By conducting an empirical investigation, we seek to understand how these smells manifest in software systems, their prevalence across different project scales, and their impact on software evolution. Focusing on cohesion levels within classes, we examine architectural smells such as cyclic dependencies, God components, and inappropriate layering to assess their influence on software development.

Empirical research in software engineering has historically faced challenges in achieving widespread acceptance, often due to practical obstacles like issues with statistical design and difficulties in generalizing findings [1]. However, empirical methods remain essential for bridging theoretical knowledge with real-world applications, enabling researchers to gain valuable insights into software development practices. By systematically examining architectural smells, this study aims to enhance understanding of software architecture evolution and provide strategies to prevent architectural decay.

This research builds upon existing studies that emphasize the importance of addressing architectural quality for sustainable software engineering. The anticipated outcomes are expected to elucidate the relationship between architectural smells and software evolution, offering practitioners actionable guidance to improve software adaptability and maintainability [2].

**2 BACKGROUND AND RELATED WORK**

The architectural choices made throughout the design and development stages of software systems have a significant impact on their quality and evolution. Since a software architecture must be flexible enough to accommodate changing business requirements, it must be regularly checked for possible deterioration. With time, some "smells" or signs of architectural defects may emerge; these are often called architectural smells. Like code smells in programming, these smells indicate fundamental problems with the software's architecture that may impair performance, scalability, and long-term maintainability. To guarantee the system's resilience over time, it is crucial to recognize these smells and comprehend how they affect software development.

Researchers have described and classified architectural smels in a variety of ways. Inappropriate layering, where the design does not reflect an ideal separation of concerns, God components, where some components take on too many duties, and cyclic dependencies, where modules depend on each other in a circular pattern, are some such examples. It has been demonstrated that these problems reduce the software's adaptability, which raises maintenance expenses and slows down development as new features are introduced, or old ones are changed. The complexity of software projects rises with time, increasing the possibility that architectural smells will become more noticeable. Because large projects typically have more complex and convoluted dependencies, this is especially true.

Over time, research on the evolution of software architecture has accelerated, particularly considering the growing complexity of contemporary software systems. Numerous research has put forth methods for detecting architectural smells, many of which center on examining the coding and spotting deterioration patterns that point to underlying architectural issues. For instance, an empirical study by Sas, Avgeriou, and Uyumaz (2022) investigating the influence of architectural scents on software evolution discovered that these scents had a major impact on the scalability and maintainability of software. Their investigation shown how crucial it is to treat these smells at an early stage of development to prevent long-term problems [1].

Similarly, the evolution of software-architecture smells over software system versions was investigated by Gnoyke, Schulze, and Krüger (2024). Their research emphasized the dynamic character of architectural deterioration and the necessity of continuous observation and action to stop it. Both studies advance our knowledge of how architectural scents affect software systems, and their conclusions highlight the necessity of conducting empirical research in authentic environments to examine the practical effects of these smells [2].

Gokhale et al. (2019) made another noteworthy contribution by investigating the connection between architectural smells and overall software quality, with a focus on how smells impact system scalability and maintainability. They presented a system for measuring the effects of architectural defects, arguing that many common software quality problems might be avoided by identifying and controlling architectural smells. Their research showed how important it is to address these smells as part of a larger initiative to enhance software architecture and software evolution procedures [3].

Several techniques and approaches targeted at enhancing software architecture over time have also incorporated the detection of architectural smells. To help developers understand how their choices affect the development of software systems, tools such as ArchUnit, Structure101, and SonarQube have been developed to identify and display architectural flaws. Both industry and academics make extensive use of these technologies, which help to uphold high architectural standards as projects develop and expand.

Even with the advancements, much more must be discovered about the complex ways that architectural smells impact software systems of all sizes. Fewer research has systematically investigated the presence and influence of architectural scents across different project scales, with the majority of studies concentrating on large projects. By looking at Python projects of different sizes—small, medium, and large—this study aims to close this gap and provide a better understanding of how these smells change and appear in projects of different sizes.

This study attempts to advance knowledge of architectural smells' function in software evolution by building on earlier research on the subject. It will shed light on how architectural smells impair software systems' scalability and maintainability and make suggestions for improving architectural quality management.

# 3.METHODOLOGY

The purpose of this research is to evaluate empirically how architectural smells affect the development of Python software projects. To do this, I chose a wide range of Python projects and arranged them according to size and complexity to investigate how architectural defects appear and change across various software systems. The procedures used to carry out this empirical investigation are described in the methodology section that follows.

**3.1 Selection of Python Projects**

I selected nine Python repositories from GitHub, covering a variety of subjects, project sizes, and community activity, to perform a significant empirical investigation. The chosen projects range in size from tiny utilities to massive frameworks. Every project was selected to offer a fair assessment of the ways in which architectural smells can impact both small and big systems. The selection criteria for the repositories are as follows:

* **Project Size**: Both small-scale tools and huge frameworks were included in the selection of projects of various sizes. This enables us to investigate the ways in which architectural scents appear in various settings.
* **Community Involvemen**t: To investigate how architectural scents change in collaborative settings, projects with different levels of commitment (number of commits and contributors) were chosen.
* **Domain Diversity**: Cryptography, gaming, machine learning, and mobile security are just a few of the subjects covered by the selected repositories. This variety sheds light on how architectural scents may vary depending on the context.
* **Popularity and Activity**: The repositories were chosen according to their recent activity (such as the quantity of commits) and popularity (such as the number of stars and observers). This makes it easier to guarantee that the projects are kept up to date and that the information gathered is pertinent to contemporary software development techniques.

The chosen projects cover a variety of domains, such as cryptography, gaming, machine learning, mobile security, and more. Below is a brief description of each project selected for this study:

**Ethereum Consensus Specs**: A collection of specifications for Ethereum 2.0, focusing on the consensus layer [5].

**Shutit**: A tool for automating the setup of virtual environments, with support for Docker and various configurations [6].

**Lutris**: A gaming platform that allows users to manage and play games from different sources on Linux [7].

**Cryptography**: A Python library that provides cryptographic recipes and primitives [8].

**Picard**: A music tagging application that organizes audio files by reading metadata [9].

**Cobbler**: A Linux provisioning server tool that simplifies the management of operating system installations [10].

**Open Model Zoo**: A repository with pre-trained deep learning models, optimized for Intel hardware [11].

**OWASP Mobile Application Security Testing Guide (MASTG)**: A comprehensive guide for mobile application security testing [12]

**DeepChem**: A library that facilitates deep learning for drug discovery and molecular science [13]

**3.2 Metrics and Tools for Architectural Smell Detection**

This study uses automated methods and particular software metrics to identify architectural smells in the chosen Python programs. Radon, which makes it easier to identify code smells, and the computation of cohesiveness metrics are two essential techniques for identifying architectural smells. To assess the software systems' development and maintainability, the following metrics were computed:

Inorder to compute the cohesion and codesmells I have set up my virtual environment and have installed both tools by using pip command. The radon was installed by using pip install radon and to extract the metrics **radon raw.** command is used. In the same way cohesion is installed using pip install cohesion and then cohesion metrics are extracted for each project by using command **cohesion -d pythonCodeDirectory/ | grep Total |sed 's/\s\{1,\}Total: \(.\*\)\%$/\1/'**

**Cohesion**: This metric assesses how closely connected the elements (like classes) of a project or module are to one another. Low cohesiveness is frequently an indication of prevalent architectural smells like "God Classes" or "Feature Envy." A high cohesiveness value suggests a better design since it shows how closely related the components of a module or class are.

**Code Smells (via Radon):** Radon [4] was used to identify common code smells, including duplicated code and complexity (such as excessive cyclomatic complexity). These smells can be a symptom of architectural problems in the system's design since they highlight parts of the code that might be too complex or challenging to maintain. The Radon tool assists in identifying these problems for additional investigation.

**3.3 Analyzing and Visualizing Data Collected**

The data collected from the above tools are aggregated based on the project type (Small, Medium and Large).The outcomes are displayed to give a better grasp of the architectural problems following the computation of the cohesion metrics and the identification of code smells. Among the visualizations will be:

**Cohesion Distribution**: A visual depiction of the cohesiveness values in each project's various modules and classes. This aids in locating regions of weak unity that might indicate possible God Classes or Feature Envy.

**Code Smell Frequency**: A Radon-detected code smell frequency and type visualization that illustrates how common problems like excessive cyclomatic complexity and duplicated code manifest themselves across repositories.

**3.4 Research Goals and Questions**

Examining how architectural smells affect the maintainability and development of Python software systems is the main objective of this study. Specifically, it seeks to pinpoint typical architectural problems in Python projects, evaluate how they impact software quality, and offer solutions. The research questions listed below are put forth to direct this study:

**RQ1:** What is the relationship between the frequency of architectural smells like Feature Envy and God Classes and the average cohesiveness of small, medium, and big Python projects?

This question examines the connection between the prevalence of architectural smells in projects of different sizes and coherence (as an indication of design quality).

**RQ2**: How much are architectural smells identified by Radon (such as code duplication and cyclomatic complexity) affect the maintainability of Python projects?

This question investigates the relationship between maintainability elements including readability, complexity, and ease of updating and Radon's criteria for architectural smells.

**RQ3:** What are the primary variations in the way architectural smells change over time in Python projects across different project sizes and domains?

This question examines how architectural scents evolve with time, highlighting size- and domain-specific variances in their patterns of resolution and appearance.

**RQ4:** Are code churn and defect rates in Python software systems accurately predicted by cohesion metrics and Radon-detected architectural smells?

This question assesses how well cohesion and smell measurements predict quantifiable outcomes like as defect rates (number of defects) and code churn (frequency of code changes).

# 4 RESULTS AND DISCUSSION

**RQ1**: What is the relationship between the frequency of architectural smells like Feature Envy and God Classes and the average cohesiveness of small, medium, and big Python projects?

**Cohesion Distribution by Project Size:**

* **Small Projects**: As shown in Figure 1 Cohesion values vary widely, with an average of 24.94 across the three listed projects. The range includes low cohesion (e.g., Ethereum/Consensus-Specs: 10.08) and higher cohesion (Ianmiell/Shutit: 37.08).
* **Medium Projects**: Cohesion values are more consistent, averaging around 28.45. Projects like Metabrainz/Picard (32.18) and Pyca/Cryptography (24.29) show moderate cohesion levels (see Figure 1)
* **Large Projects**: Cohesion is polarized. DeepChem/DeepChem exhibits very high cohesion (77.86), while OWASP/OWASP-Mastg has zero cohesion. The average is influenced by these extremes.

**Frequency of Code Smells by Project Type:**

* **Small Projects**: Feature Envy is significantly higher (average 5 instances) compared to medium and large projects. This suggests that smaller projects are more prone to architectural smells indicating poor modularity and cohesion.
* **Medium Projects**: Feature Envy is absent, but occurrences of Large Classes (6) and Long Methods (3) indicate moderate architectural issues (see Figure 2)
* **Large Projects**: As shown in Figure 2, Feature Envy (0.67) and Low Cohesion (3.33) are slightly present, suggesting larger projects occasionally struggle with maintaining modularity and cohesiveness due to their size and complexity.

***Figure1: Distribution of Average Cohesion across 9 projects***

The data shows that the association between architectural smells and cohesiveness is influenced by project size. Low cohesion and the occurrence of smells like Feature Envy are more strongly correlated in smaller projects. Medium-sized projects encounter fewer architectural problems and maintain greater coherence. Even though large projects exhibit less frequent Feature Envy and varying cohesiveness, their complexity causes structural smells like Low cohesiveness and Duplicated Code.

***Figure 2: Distribution of Code Smells by Project Type***

This link emphasizes how crucial scalable design concepts are for reducing architectural smells and enhancing cohesiveness as project sizes grow.

**RQ2**: How much are architectural smells identified by Radon (such as code duplication and cyclomatic complexity) affect the maintainability of Python projects?

***Figure3: Distribution of Frequency of Code Smells***

Radon found that architectural smells like cyclomatic complexity and code duplication have a big impact on how maintainable Python applications are. These smells create difficulties by making it harder to comprehend, alter, and test the code. The following conclusions can be made using the data and the related case studies.

**Effects of Cyclomatic Complexity and Long Methods:**

Long methods make code more difficult to comprehend and change, as demonstrated by projects like ianmiell/shutit (25 instances) as shown in Figure 3. Elevated cyclomatic complexity makes testing more challenging, which raises maintenance expenses and fault rates.

This conclusion is supported by research, which demonstrates that these smells are associated with increased insect densities and lower maintainability.

**Impact of Big Classes:**

Projects like deepchem/deepchem (11 as big classes) show how large classes make it difficult to maintain and impede modularization. Even in very cohesive projects, this issue still exists.

**Poor Cohesion:**

Maintainability is directly impacted by cohesiveness. Architectural smells are more common in projects with low cohesiveness as shown in Figure 1 (e.g., Ethereum/consensus-specs: 10.08), which exacerbates maintainability problems. It is more difficult to comprehend and refactor classes that execute unrelated functions (poor coherence).

**Code Duplication:**

Because changes must be copied across several locations, duplicate code—like that seen in deepchem/deepchem (10 instances)—raises maintenance burden and increases the possibility of errors and inconsistencies.

**Feature Envy:**

As shown in Figure 3, this smell is common in lutris/lutris (15 occurrences), which is indicative of bad design because methods rely too heavily on data from other classes. Tighter coupling and decreased maintainability result from this.

**Varying Impact by Project Size**:

While bigger projects maintain better balance with fewer smells, small projects show comparatively higher feature envy and long procedures (e.g., ianmiell/shutit).

Due to their scale and complexity, large projects with high cohesion (as deepchem with 77.86) suffer from several smells, which severely impair maintainability.

To include RQ2, regardless of size or degree of cohesiveness, the investigation shows that architectural smells have a detrimental effect on Python projects. Cohesion is important, however the amount of work required for maintenance is disproportionately increased when smells like lengthy methods, huge classes, and duplicated code are present.

**RQ3:** What are the primary variations in the way architectural smells change over time in Python projects across different project sizes and domains?

Because of the nature of development and the difficulties encountered at every stage, architectural smells, such as lengthy methods, huge classes, and duplicated code, tend to differ among Python projects of varying sizes. Architectural smells are more noticeable for smaller projects, such as ianmiell/shutit and ethereum/consensus-specs, frequently due to experimentation, quick development, or a lack of early design planning. Although these projects can usually be implemented quickly, they may eventually have issues with scalability and maintainability. Early-stage initiatives often have more architectural smells because to the emphasis on functionality over structure, according to research on small projects [14]

Refactoring efforts and an emphasis on maintainability usually result in a decrease of architectural smells in medium-sized projects, like cobbler/cobbler. Developers frequently try to improve cohesiveness and modularize the code as the project expands. Nevertheless, as things are added gradually, medium-sized projects may still have issues with smells like poor coherence or redundant code. This finding is supported by studies like those conducted by Jureczko [15], which show that medium-sized projects frequently strike a compromise between maintaining a cleaner code structure and adding new features, but not without developing distinct architectural smells.

The evolution of architectural smells tends to be more complex in larger projects, such as deepchem/deepchem. As demonstrated in our dataset, smells like feature envy, huge classes, and duplicated code frequently build up as a project gets bigger. Because more developers work on larger projects and complicated features are added over time, keeping clean architecture is more likely to be difficult. To counteract some of these smells, these projects also benefit from more organized development procedures including code reviews, automated testing, and continuous integration. The results of O'Keeffe study [16] on large-scale software systems indicate that although bigger projects tend to have more architectural smells, they also spend more money on code management and refactoring to encounter maintanbility issues.

**RQ4:** Are code churn and defect rates in Python software systems accurately predicted by cohesion metrics and Radon-detected architectural smells?

|  |  |  |
| --- | --- | --- |
| **Project Name** | **Project Type** | **Cohesion (Avg)** |
| ethereum/consensus-specs | Small | 10.08576923 |
| ianmiell/shutit | Small | 37.0868 |
| lutris/lutris | Small | 25.66219512 |
| pyca/cryptography | Medium | 24.29049618 |
| metabrainz/picard | Medium | 32.18261438 |
| cobbler/cobbler | Medium | 28.8752381 |
| openvinotoolkit/open\_model\_zoo | Large | 37.8223 |
| OWASP/owasp-mastg | Large | 0 |
| deepchem/deepchem | Large | 77.86760736 |

***Table 1: Average Cohesion Values of all projects***

Cohesion, architectural smells, and defect rates or code churn in Python software systems were all clearly correlated when I verified the Git repositories. Defect rates and code changes were greater in projects with low cohesiveness and strong architectural smells, such as `ianmiell/shutit`. Projects with strong coherence, like `deepchem/deepchem`(see Table 1), on the other hand, showed fewer flaws and lower churn, indicating that higher design quality leads to more stability and maintainability.

I discovered that longer methods, huge classes, and duplicated code—all which Radon identified as architectural smells—are associated with greater defect rates and more code churn. Projects like `pyca/cryptography`, for instance, that had fewer architectural smells were easier to maintain. Projects with more smells, like `ianmiell/shutit` and `deepchem/deepchem`, on the other hand, had greater defect rates and more frequent updates, underscoring the effect of subpar design on software quality.

Defect rates and churn are influenced by a variety of parameters, even though cohesiveness and architectural smells offer important insights about maintainability. These results are also influenced by other factors, like team size and development procedures. Nonetheless, it is evident from the data that architectural smells and a lack of cohesiveness are highly correlated with greater rates of defects and code churn, suggesting that fixing these problems could result in more stable and maintainable projects.

# 5 THREATS TO VALIDITY

# Internal, external, construct, and conclusion validity are some of the aspects that may affect the validity of the findings in this empirical investigation.

# 5.1 Internal Validity

# Validity within the extent to which the observed effects may be ascribed to the causes under investigation rather than outside influences is known as internal validity.

# Tool Restrictions: SonarQube and Radon, the technologies used to identify architectural smells, have built-in restrictions. For example, because Radon focuses on code-level smells rather than architectural-level issues, it may overlook some intricacies, and SonarQube may not detect all kinds of smells. The thoroughness of the identified architectural smells may be impacted by these restrictions.

# Human error in interpretation: When classifying smells or interpreting cohesiveness levels, the manual examination of detected smells may introduce subjectivity. This raises the possibility of bias in the findings, which can affect the judgments made regarding the projects' architectural merit.

# 5.2 External Validity

The degree to which the results can be applied to different situations, places, or systems is known as external validity.

**Project Selection**: Python-based projects chosen from GitHub were the focus of the study. The ecology of software development may not be fully represented by this option. Projects created in other languages or in private company settings may differ from Python projects, particularly open-source ones. As a result, the results could not apply to all software languages or systems.

**Community Involvement and Project Activity**: The chosen projects range widely in terms of community involvement and project activity. While some projects might be abandoned or receive only occasional contributions, others might be maintained more actively. Depending on how often the project is updated or maintained, the architecture quality and odor intensity may change, which could restrict the results' applicability to other projects with varying degrees of community involvement.

**5.3 Construct Validity**

The precision of the measurements employed in the research is the focus of construct validity.

**Architectural Quality Substituted by cohesiveness**: In this study, architectural quality was substituted by the cohesiveness metric. Although cohesiveness is an important indicator of modularity and design, it may not take into consideration all facets of software architecture, such as layering, coupling, and component interactions. Therefore, relying only on cohesion may miss further architectural issues that could impact software's growth and maintainability.

**Using Code Smells to Identify Architectural Smells**: Radon recognized code smells as markers of architectural smells, including cyclomatic complexity and duplicated code. These code smells were then used as stand-ins for architectural smells. Although there is considerable overlap between architectural problems and code smells, not all code smells are indicative of architectural concerns. In certain situations, this could result in an overestimation or incorrect classification of architectural smells.

**5.4 Conclusion Validity**

The degree to which the observed associations in the study represent real relationships rather than random or confounding variables is known as conclusion validity.

**Correlation versus Connection**: The focus of this study is to find relationships between architectural smells and software evolution measures like code churn and defect rates. Correlations do not necessarily indicate causation, but they do indicate a relationship. Future research should look on the causal relationships between these software evolution measures and architectural smells.

**Data Quality**: The results may be impacted by the completeness of the commit history, the correctness of the defect rates, and the consistency of the coding processes, among other aspects of the data obtained from GitHub repositories. Results may vary depending on how changes to the codebase are tracked or how issues are reported.

# 6 FUTURE DIRECTIONS FOR RESEARCH

Building on the results of this study, future research can investigate several directions to learn more about how architectural smells affect software development. Future study could go in the following directions:

**6.1 Increasing the Toolkit and Smell Detection Methods**

* **Including More Detection technologies**: To increase the thoroughness of the investigation, future research could make use of a greater range of architectural smell detection technologies. A more comprehensive understanding of software architecture may be possible by combining static and dynamic analysis technologies.
* **Automated Smell Classification**: Subjectivity in detecting and classifying architectural smells may be decreased by improving the automation of smell classification and categorization, maybe with the use of machine learning techniques. This might result in a more scalable and reliable method of detecting smells.

**6.2 A Wider Range of Languages and Projects**

* **Examining Projects Written in Other Programming Languages**: To find out if the architectural smells and their effects are the same in various ecosystems, future research could build on this study by looking into projects written in other programming languages (such as Java, C++, and JavaScript).
* **Private and Enterprise Projects**: Since the scale and maintenance procedures of private or enterprise-grade software projects may differ greatly from those of open-source projects, including them in the study could shed light on how architectural smells impact systems in various organizational contexts.

**6.3 Examining Causal Relationships**

* **Casual Studies**: To determine causative correlations between architectural smells and software evolution measures, future research could go beyond correlation. This could entail altering the architecture using experimental or quasi-experimental means and monitoring the effects on evolution metrics such as maintainability and defect rates.

**6.4 Examining Strategies for Mitigation**

* **Smell Mitigation and Refactoring**: To assess how well automated refactoring techniques and other mitigation strategies for architectural smells improve software maintainability, defect rates, and other evolutionary features, research could concentrate on creating or evaluating these strategies.

**6.5 Assessing Long-Term Effects**

* **Longitudinal Studies**: Longitudinal studies that monitor the effects of architectural scents throughout time may be a part of future research. This would make it easier to evaluate how architectural choices affect software evolution over the long run, including how technical debt builds up and how it affects system scalability and adaptability.

**7 CONCLUSIONS**

This study investigated the connection between software evolution in Python-based projects and architectural smells. We examined the relationship between architectural and code smells and software evolution measures like defect rates and code churn by using tools like SonarQube and Radon to identify them. The findings imply that the maintainability and defect rates of software projects are significantly impacted by architectural smells, especially those associated with weak cohesiveness. However, several validity risks, such as tool limits, human error, and the results' generalizability, point to the necessity of additional study to support and build upon these findings.

Architectural smells can be early warning signs of possible problems in software development, but detecting and reducing them calls for constant study and tool creation. The study's conclusions emphasize the value of identifying architectural issues early on and the necessity of efficient methods for handling architectural smells throughout the software development process.

# REFERENCES

1. Sas, D., Avgeriou, P., & Uyumaz, U. (2022). On the evolution and impact of architectural smells—an industrial case study. Empirical Software Engineering, 27(86). <https://doi.org/10.1007/s10664-022-10132-7>
2. Gnoyke, P., Schulze, S., & Krüger, J. (2024). Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells. Journal of Systems and Software, 217, 112170. https://doi.org/10.1016/j.jss.2024.112170 .
3. Gokhale, A., Kamsu-Foguem, B., & Komuravelli, S. (2019). A framework for detecting architectural smells and its impact on software quality. International Journal of Software Engineering and Knowledge Engineering, 29(8), 1093-1111. <https://doi.org/10.1142/S0218194019500560>
4. Radon Documentation. (2024). Radon Documentation. Retrieved from <https://radon.readthedocs.io/>
5. Ethereum Consensus Specs: Ethereum Foundation. (n.d.). Ethereum consensus specs. GitHub. <https://github.com/ethereum/consensus-specs>
6. Shutit: Miell, I. (n.d.). Shutit. GitHub. <https://github.com/ianmiell/shutit>
7. Lutris: Lutris. (n.d.). Lutris. GitHub. <https://github.com/lutris/lutris>
8. Cryptography: Pyca. (n.d.). Cryptography. GitHub. <https://github.com/pyca/cryptography>
9. Picard: MetaBrainz Foundation. (n.d.). Picard. GitHub. <https://github.com/metabrainz/picard>
10. Cobbler: Cobbler. (n.d.). Cobbler. GitHub. <https://github.com/cobbler/cobbler>
11. Open Model Zoo: OpenVINO Toolkit. (n.d.). Open model zoo. GitHub. <https://github.com/openvinotoolkit/open_model_zoo>
12. OWASP MASTG: OWASP Foundation. (n.d.). OWASP MASTG. GitHub. <https://github.com/OWASP/owasp-mastg>
13. DeepChem: DeepChem. (n.d.). DeepChem. GitHub. <https://github.com/deepchem/deepchem>
14. Wang, S., et al. (2017). "A Study of Architectural Smells in Software Systems: Characteristics, Causes, and Consequences." Empirical Software Engineering, 22(5), 2394-2420.
15. Jureczko, M., et al. (2012). "A Study on the Impact of Architectural Smells on Software Maintainability." Proceedings of the 6th IEEE International Conference on Software Engineering and Service Science (ICSESS).
16. O'Keeffe, M., et al. (2016). "Managing Architectural Smells in Large-Scale Software Systems: A Case Study." Journal of Software Maintenance and Evolution: Research and Practice, 28(7), 524-541.