Evaluating the Energy Consumption of Static Analysis Tools: A Sustainability Perspective"

Rafał Owczarski*
Delft University of Technology
Delft, Netherlands
R.R.Owczarski@student.tudelft.nl

Lászlo Roovers*
Delft University of Technology
Delft, Netherlands
L.Roovers@student.tudelft.nl

Athanasios Christopoulos*
Delft University of Technology
Delft, Netherlands
A.Christopoulos@student.tudelft.nl

Muhammad Zain Fazal*
Delft University of Technology
Delft, Netherlands
mfazal@tudelft.nl

* Equal contribution

***Abstract*—As software systems grow in scale and complexity, the tools we use to maintain them play an increasingly important role in the broader sustainability of software engineering. Among these tools, static analysis tools—such as PMD—are indispensable for identifying code issues early, improving maintainability, and ensuring quality standards. However, the energy footprint of such tools remains an overlooked aspect of software development.**

With rising awareness of climate change and environmental impact, the energy efficiency of software tooling has gained relevance. Developers and organizations alike are beginning to consider not only the functionality and performance of their development pipelines, but also their sustainability. While runtime energy efficiency has received considerable attention, the energy cost of development tools themselves—tools that run continuously in CI pipelines or are triggered repeatedly during development—deserves closer scrutiny.

This study aims to bridge that gap by focusing on the energy consumption of PMD under various configurations. Specifically, we investigate how different rulesets and code complexities affect PMD's energy usage. Through a systematic measurement process in a controlled environment, we aim to uncover patterns in energy usage that can inform sustainable tool configurations. Our results provide actionable insights for developers who want to balance code quality assurance with energy efficiency.

***Index Terms*—Sustainable software engineering, energy consumption, static analysis tools, PMD, software quality, energy efficiency, code analysis, green computing, software sustainability.**

I. Introduction

As software development increasingly prioritizes sustainability, understanding the energy consumption of development tools has become essential. Static analysis tools, such as PMD, play a crucial role in improving code quality by identifying potential issues early in the development process. However, their computational cost and energy footprint remain largely unexplored. This research seeks to bridge this gap by systematically evaluating the energy consumption of PMD across different configurations.

Our study focuses on measuring how varying rulesets and code complexities influence energy consumption. By conducting experiments in a controlled environment, we aim to provide actionable insights into optimizing static analysis for energy efficiency. The findings will enable developers to make informed choices when configuring PMD, balancing code quality assurance with sustainable computing practices.

As energy efficiency becomes a key concern in computing, it is crucial to understand the broader energy implications of software development tooling. While much attention has been given to optimizing software for runtime performance and energy use, less focus has been placed on the tools used during development. Static analysis tools are often executed frequently, sometimes integrated into automated workflows, where even marginal inefficiencies can accumulate into a notable energy footprint over time.

Furthermore, the trade-off between analysis depth and energy consumption is still poorly understood. Developers are often unaware of the energy cost of stricter or more comprehensive rulesets, and may unintentionally configure tools in ways that significantly increase energy usage. By shedding light on these dynamics, this study aims to empower more sustainable decision-making in tool configuration.

In a broader sense, our research contributes to the evolving field of sustainable software engineering. Quantifying the energy impact of development tools provides a foundation for optimizing not only the software being built, but also the processes and environments in which it is created. As awareness grows around the environmental cost of digital infrastructure, such insights become increasingly relevant for both academia and industry.

II. Problem Statement and Motivation

As software continues to evolve, energy efficiency has become an increasingly important consideration. With the growing emphasis on sustainability in computing, understanding the energy impact of software development tools is crucial. Static analysis tools like PMD play a key role in improving software quality by detecting code issues early. However, their own computational cost and energy consumption remain largely unexplored.

The motivation behind this research stems from several key concerns:

- **Sustainability in Software Development:** The carbon footprint of software is often overlooked, yet energy-intensive tools contribute to overall energy consumption in data centers and development environments.
- **Balancing Code Quality with Efficiency:** Static analysis improves maintainability and security, but different rulesets and configurations may lead to significantly different energy costs. Understanding these trade-offs allows developers to optimize rule selection without unnecessary energy overhead.
- **Scalability and Performance Considerations:** As codebases grow, static analysis tools may introduce performance bottlenecks. Measuring energy consumption helps assess whether an analysis process remains efficient as software projects scale.
- **Optimizing Developer Toolchains:** Developers use static analysis as part of CI/CD pipelines, where energy consumption can accumulate overtime. Insights from every measurements can contribute to designing more energy-efficient development workflows.

III. Research Questions

To address these concerns, our research is guided by the following key questions:

RQ1 Ruleset Complexity vs. Energy Consumption
How does the number and complexity of rules in a PMD ruleset impact the energy consumption of static code analysis?

RQ2 Error Detection vs. Energy Efficiency
Is there a correlation between the number of detected errors and the energy consumption of the analysis process?

RQ3 Popular Rulesets and Their Energy Cost
How do commonly used PMD rulesets compare in terms of energy efficiency, and which provide the best balance between detection capability and power consumption?

By systematically analyzing these aspects, this research aims to provide actionable insights that help software engineers make informed choices about static analysis configurations, balancing software quality with sustainable computing practices.

IV. Background and Related Work

With climate change and overall environmental concerns being on the rise in the last few years, the aspect of sustainability among all engineering sectors occupies an increasingly larger part of recent research. Software engineering is no exception to this and a large amount of research has already been conducted to reduce the energy consumption of various aspects of software engineering, therefore increasing its sustainability. However, the number of developer tools and frameworks being so large, it is extremely complicated to cover them all, meaning that in the context of sustainability, most of them remain unexplored.

The purpose of this research is to deepen our knowledge regarding the energy consumption of different rulesets of a static code analysis tool, ran on open source projects of varying lengths. This research is intended to present a well-rounded understanding of the differences in energy consumption of static analysis rulesets, allowing developers to make more informed and sustainable choices during development.

Unfortunately, approaches such as Marantos et al. (2021)[1] and Lopez-Garcia et al. (2015)[2] keep their main focus on focus using static analysis tools to measure the energy consumption of running code rather than the energy usage of the action tool.

Despite the widespread adoption of static analysis tools, there is a lack of research quantifying their energy consumption. Some existing studies focus on the performance overhead of static analysis tools, measuring their execution time and CPU usage. For instance, a study by V Lenarduzzi et al[3] compared the performance impact of different static analyzers but did not explicitly assess their power consumption.

A notable exception is the work of Sahin et al. [4], who explored the energy consumption of automated software testing frameworks and found that test coverage tools can significantly increase energy usage, especially in large codebases. While their study does not focus on static analysis, it suggests that software quality assurance tools can have substantial energy implications.

Thus, research focused on the impact on energy consumption of rules and the size of the code analysed remains lacking. As we attempted to bridge this gap, different static analysis tools were at our disposal, such as Checkstyle, jQAssitant, PMD and FindBugs but it was decided to focus on PMD, due to its flexibility and configurable rules, as well as its ease of integration with popular IDEs such as IntelliJ and VS Code.

V. Methodology

To evaluate the energy consumption of the PMD static analysis tool under different rulesets and varying code and rule complexities, we designed an experiment aimed at capturing power usage during execution. The key objective was to measure how different configurations impact energy efficiency under controlled conditions.

All measurements were conducted in a consistent computing environment to minimize variability. The hardware setup used for the experiment was as follows:

- Processor: Intel Core i7-9750H CPU @ 2.60GHz
- Memory: 16GB RAM
- Operating System: Windows 10 Home

- Power Monitoring Tool: EnergiBridge
- Static Analysis Tool: PMD

This setup ensured that all runs were executed under identical conditions, reducing potential noise in the results.

A. *Evaluation Metrics*

To quantify the energy consumption of PMD across different configurations, we compared multiple evaluation metrics commonly used in energy-aware computing research:

- Total Energy Consumption (joules): The total enery consumed during the execution of PMD.
- Execution time (seconds): The duration required to complete the static analysis task, providing helpful insights into performance efficiency.
- Energy Delay Product (EDP): A combined metric considering both energy consumption and execution time, calculated as

  $$EDP = Energy \times Time,$$

  emphasizing energy efficiency relative to runtime.

B. *Open Source Projects*

To provide a diverse and representative analysis, we selected a mix of well-known and lesser-known open-source Java projects. The selection criteria included project size, complexity, real-world relevance, and code structure diversity. The following projects were chosen:

- Well-known projects
  - **Spring Framework:** A widely used Java framework for enterprise applications offering a large and complex codebase.
  - **JUnit 5:** A popular unit testing framework, useful for analyzing the energy consmption for test-related code.
- Lesser-known projects
  - **JabRef:** A reference manager for academic papers, featuring a mix of GUI and backend logic.
  - **Terasology:** A voxel-based game engine with performance-critical code, providing insights into energy usage in computationally intensive applications.

VI. Experiment Setup

- Each scenario (a unique combination of a source code directory and ruleset) was executed **30 times**.
- The execution order was randomized to eliminate bias.
- Energy measurements were collected using EnergiBridge and stored for analysis.
- A cooldown period was introduced between runs to reset system state.

Each scenario's execution consisted of the following steps:

1) **Initialization**: The energy measurement tool was started to record baseline consumption.
2) **Idle Energy Measurement**: The measurement of idle energy lasting 5 seconds.
3) **PMD Execution**: The tool analyzed the source code using a predefined ruleset.
4) **Energy Logging**: The total energy consumption was recorded.
5) **Cooldown Period**: A short pause was introduced before the next run to mitigate thermal fluctuations.
6) **Data Storage**: Results were appended to a dataset stored in Parquet format for further analysis.

A. *Approach Considerations*

To ensure reliable and repeatable results, the following precautions were taken:

- **Repeated trials**: Each test was conducted 30 times to account for fluctuations.
- **Controlled environment**: The experiments were executed on a dedicated machine without background processes interfering.
- **Temperature monitoring**: CPU temperature was tracked to ensure thermal effects did not distort energy readings.
- **Randomized execution**: The order of runs was shuffled to prevent systematic bias.
- **Idle energy offset**: Measuring the idle energy in the beginning of the trial is offset to minimize the influence of energy fluctiations between runs aiming to measure only the energy directly linked to the execution of PMD

B. *Configuration and Implementation Details*

To run the experiments and generate the visualisations, a replication package was created [5]. To configure and run the experiment, the `pipeline.py` script was used. This script defines scenarios using Java projects and rulesets to consider, along with logic to randomize the execution order.

The visuals used in subsection VII-A were generated using `total_energy_analyzer.py`. Energy Delay Product (EDP) and energy-over-time analysis in subsection VII-B and subsection VII-C were performed using the `edp.py` script. Finally, the error influence on the energy distribution in subsection VII-D was visualized using a Python script `errorVisuals.py`. Unfortunately we were unable to ensure that the data used in `edp.py` include data without the idle data. Therefore all graphs generated in the above mentioned sections use data that were not offset the same way as in subsection VII-A

VII. Results

To split the PMD rules in different rulesets, it was decided to use the rule subcategories of PMD:

- Best practices
- Codestyle
- Design
- Documentation
- Error prone
- Multithreading
- Performance
- Security

The energy consumption of the computer was measured when running PMD with each aforementioned category as a ruleset. This was done on each of the 4 open source projects. After running each scenario 30 times the following results were observed.

A. *Total Energy Consumption*

To obtain the total energy consumption across the entire `PMD` run, the energy output of `energibridge` over time was summed. As a result, total `PP0`, `PP1`, and `Package` energy could be compared. The analysis focuses on `PP0` and `Package` results, as `PP1` was not expected to be influential, given that `PMD` should not rely on the GPU for calculations which was confirmed as the median total `PP1` energy consumption across runs was around 1J . After the totals per run were calculated, some outliers were observed. These outliers consisted of abnormally long `PMD` runs or instances where `energibridge` stopped instantly. The causes of these outliers were analyzed manually. To minimize their influence, the 3 maximum and 3 minimum total energy values were excluded when generating the plots in this section. However, plots including all obtained data can be accessed through the replication package.

Figure 1 and Figure 2 illustrate the average energy consumption on running PMD analysis on the open source projects excluding idle energy consumption and above mentioned outliers. The results are focued on energy variations across the projects and rulesets connected to `PP0` and `Package` energy.

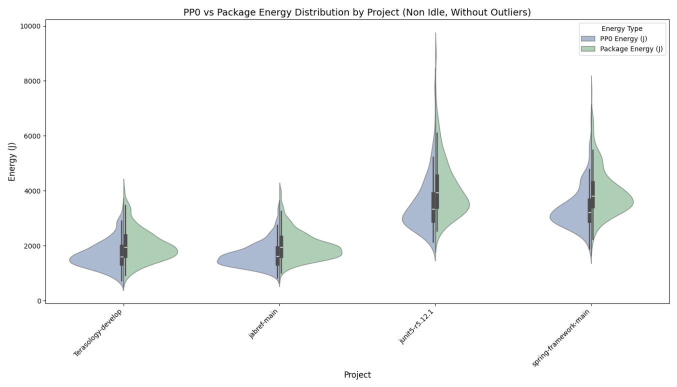

Fig. 1: Average Energy Consumption per Project excluding idle energy consumption

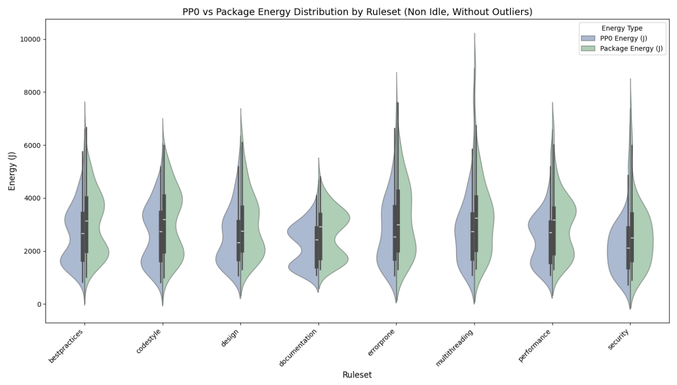

Fig. 2: Average Energy Consumption per Ruleset excluding idle energy consumption

Figure 1 showcase differences between distributions of the total energy consumption between Java projects. In all projects the `Package` violin plot has the same shape as `PP0` with a shift of around 500J. However, the overall shapes of the violin plots differ across projects. `Terasology-develop` and `jabref-main` display more dense accumulation near the median and upper interquartile range, while `junit5-r5.12.1` and `spring-framework-main` display higher standard deviation and higher mean.

Figure 2 reveals some differences between the distributions of the total energy consumption between rulsets. For the `Package` energy consumption the violin plots have similar shape with elongated tail toward higher values and values concentrated in the lower range. All rulesets exhibit similar median values and interquartile ranges. The shapes of the `PP0` violin plots are similar to the `Package` counterparts but the tails are shorter and we can distinguish two widths around the median, `errorprone` and `multithreading` are clearly elongated with less values near the median and more higher vaules in the tail, while the rest rulesets have clear condensation of values near the median and interquartile range. One difference can be obseved in case of `documentation` ruleset for which there are two visible condensations of values.

B. *Energy Delay Product*

The Energy Delay Product (EDP) metric provides information on the trade-off between energy consumption and execution time. As shown in Figure 4, the EDP distribution varies significantly across the four projects.

Among the projects, junit-5.9.2.1 exhibits the highest variability in EDP, with some instances reaching notably higher values. This suggests that the execution time or energy consumption for this project can be highly inconsistent, potentially due to differences in project complexity or workload distribution. Moreover, spring-framework-main consistently shows the lowest EDP values, indicating a more efficient execution with lower energy consumption and shorter runtime. The distributions for Terasology-develop and jabref-main are relatively similar, though Terasology-develop appears to have a slightly wider spread, suggesting greater fluctuations in energy and time requirements.

These results highlight the impact of project characteristics on the efficiency of static analysis tools. Larger or more complex codebases may introduce greater variability in EDP, affecting overall energy efficiency.

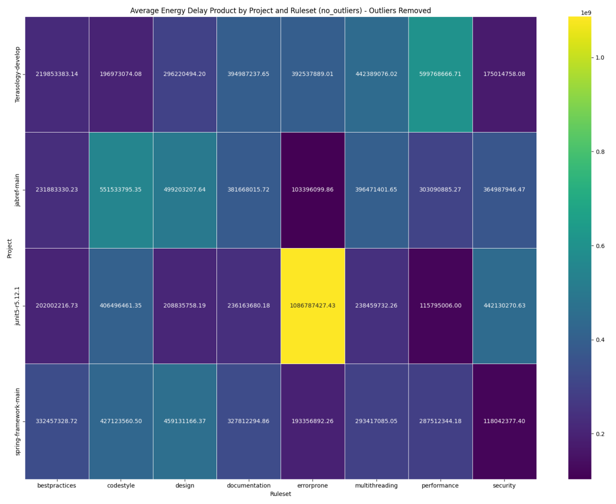

Fig. 3: EDP Heatmap by Project and Ruleset

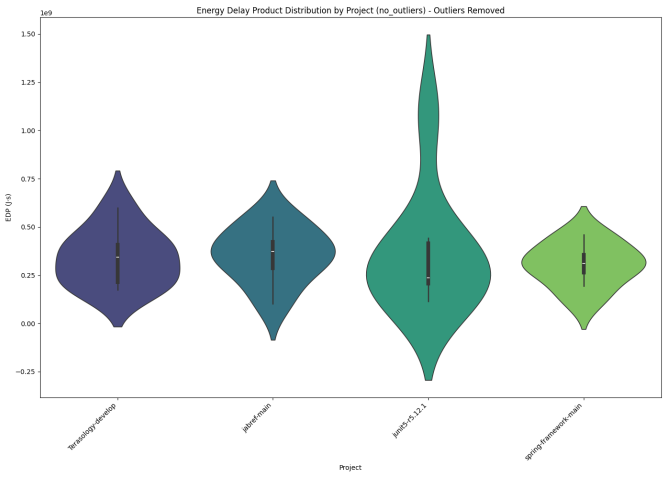

Fig. 4: EDP Distribution by Project

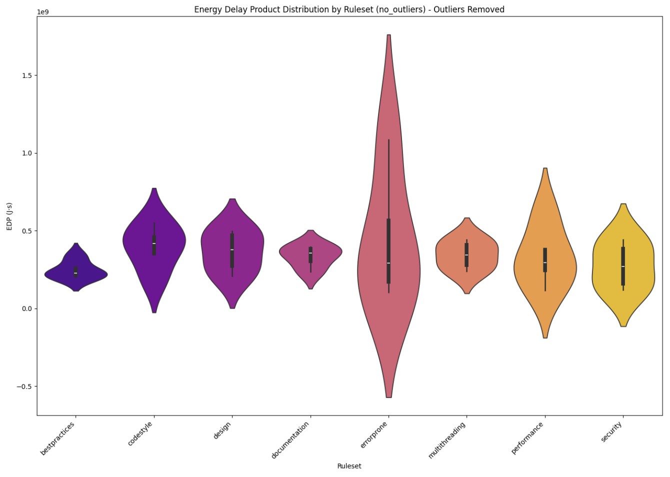

Fig. 5: EDP Distribution by Ruleset

C. Average Energy over Time

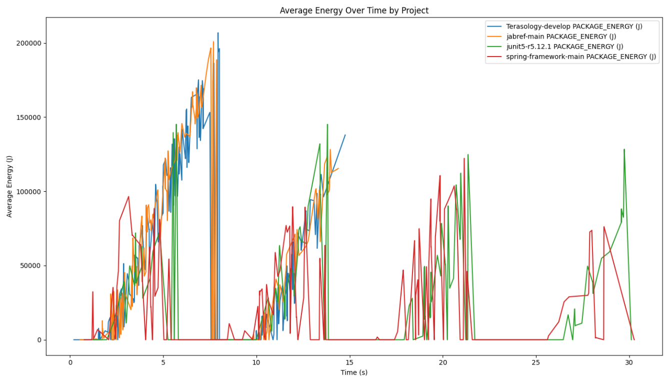

Fig. 6: Average Energy over Time by Project

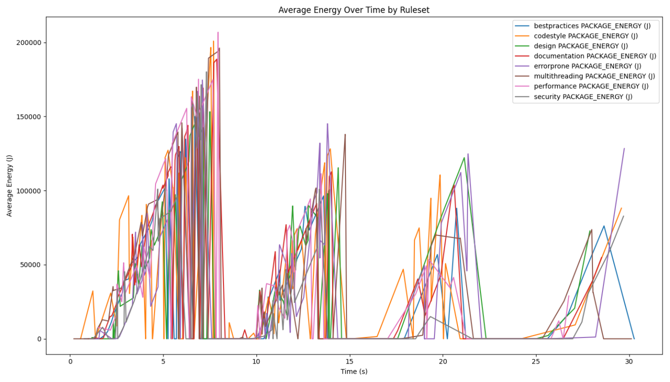

Fig. 7: Average Energy over Time by Ruleset

The energy consumption of PMD across different projects exhibits noticeable variations in both intensity and distribution over time. As shown in Figure 6, all four projects follow a very similar initial pattern, where energy usage steadily increases during the first ten seconds. This phase likely corresponds to PMD's parsing and rule execution processes, which demand significant computational resources. The Terasology-develop and JabRef-main projects show the steepest increase, peaking near 200kJ, whereas JUnit 5.12.1 and Spring Framework-main exhibit more moderate increases, remaining below 150kJ at this stage.

Following this initial phase, a distinct drop in energy consumption occurs around the ten-second mark for most projects, suggesting a transition between different processing stages. While Terasology and JabRef maintain relatively stable energy usage, JUnit and Spring Framework display more occasional spikes, likely indicating intermittent bursts of computational effort due to complex code structures or rule evaluations.

In the final phase of execution, energy usage patterns change even further. JUnit and Spring Framework experience additional short-lived surges in energy consumption before completion, possibly due to the fact that these are larger projects. In contrast, Terasology and JabRef show a more gradual decline, suggesting a continuous workload until completion. This in turn shows us that the execution time of PMD also varies depending on the project, with almost double execution time for jUnit and Spring, contributing to their large energy consumption, without jUnit and Spring being larger than the other projects.

The observed differences in energy consumption can likely be attributed to variations in codebase size, structure, and complexity. Projects such as Terasology and JabRef, tend to show a higher and more sustained initial energy usage, with a run stopping at around 15 seconds, whereas projects with more modular or less complex structures, like JUnit and Spring Framework, show intermittent patterns of computation, with smaller spike but a longer overall runtime. Additionally, the presence of idle periods and sharp drops suggests that PMD's execution process is most likely not completely uniform and may depend on how individual rules interact with different code structures.

These findings highlight the non-linear nature of PMD's energy consumption and emphasize the impact of project-specific characteristics on static analysis performance.

The energy consumption trends for different PMD rulesets, which can be seen in Figure 7, closely resemble those observed across different projects. All rulesets exhibit a sharp increase in energy usage within the first ten seconds, peaking at over 200kJ. This suggests that most rulesets are applied early in the analysis process, consuming significant computational resources. Among them, best practices, code style, and performance show the highest energy demands, while security and documentation remain lower, most likely due to their smaller size.

After this initial peak, energy usage declines sharply, followed by intermittent spikes beyond the first 15 seconds. Some rulesets, such as multithreading and error prone, display extended energy consumption, likely due to their complexity. In the final phase, energy patterns become irregular, suggesting that certain checks or reporting processes continue even after the bulk of analysis is completed.

Overall, the results indicate that rule selection significantly impacts energy consumption. More complex and larger rulesets require greater processing power, and thus increase the average energy consumption, while simpler rulesets show lower activity. This shows that ruleset selection could be optimized in order to best fit the requirements of the project, while reducing energy consumption

D. Error Frequency and relation to Energy Consumption

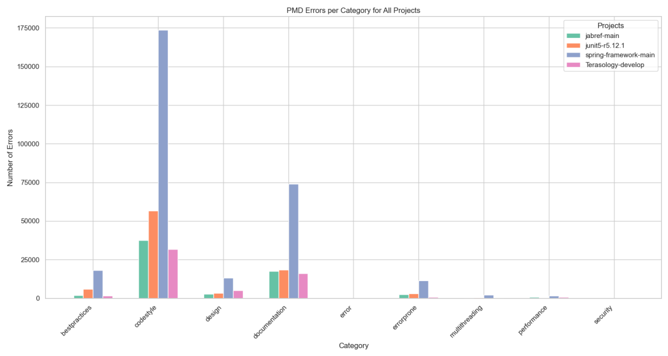

Fig. 8: Project Errors per Ruleset

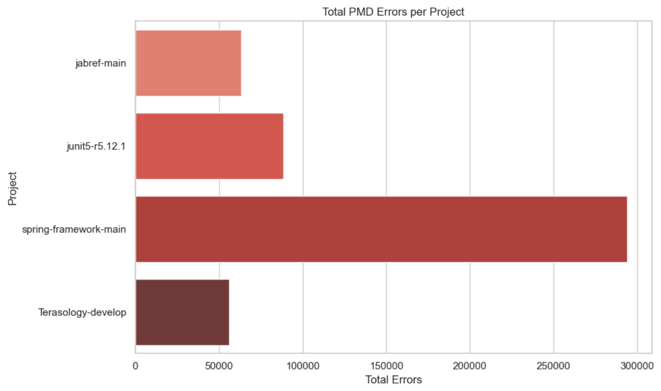

Fig. 9: Total Errors per Project

As shown previously, the energy consumption across different projects reveals that certain projects exhibit higher energy usage, which actually seems to correlate with the number of errors detected. In fact, Spring-framework-main shows the highest energy consumption, aligning with its significantly higher error count (294,220 errors) as seen on Figure 9. Similarly, junit5-r5.12.1, which ranks second in error occurrences (113,258 errors), also demonstrates increased energy usage. In contrast, Terasology-develop and jabref-main, which have fewer violations, consume comparatively lower energy.

This correlation suggests that projects with a larger number of static analysis violations require more computational processing, directly impacting energy consumption. The processing overhead of evaluating and reporting each violation, particularly when rulesets are applied extensively, leads to increased CPU activity, resulting in greater power usage.

Among the different PMD rulesets, codestyle and documentation errors are the most frequently occurring, contributing significantly to energy consumption. These rulesets include checks for variable finalization, unused imports, and required comments, which are relatively simple but occur in large volumes. The continuous parsing of large codebases to enforce these rules results in higher energy expenditure, as seen in projects like spring-framework-main and junit5-r5.12.1. In fact, while the number of errors seems to correlate to energy consumption, since jUnit and Spring have similar energy consumptions but very different amounts of errors, it seems like the size of the project matters is the determining factor.

Moreover, less frequent but computationally intensive rules, such as those under multithreading and security, have a different impact on energy usage. While these rules generate fewer violations, their complexity requires deeper analysis, potentially increasing per-instance processing time. However, because they occur less frequently, their overall contribution to total energy consumption remains lower than that of highly repetitive codestyle checks.

VIII. Limitations

While this study provides valuable insights into the relationship between software projects, rulesets, and energy consumption, several limitations should be acknowledged.

First, the energy measurements were conducted in a specific hardware and software environment, which may influence the observed results. Factors such as CPU architecture, background processes, and power management settings could introduce variability in energy consumption. Future studies could extend this work by running experiments on a broader range of hardware configurations to ensure generalizability.

Another limitation relates to the estimation of idle energy. In this study, idle energy was subtracted using a fixed baseline period; however, idle power consumption can fluctuate due to system activities beyond our control. A more adaptive approach that dynamically accounts for variations in idle energy could improve measurement accuracy.

Furthermore, the study primarily examines package-level energy consumption, focusing on the energy domains PP0, PP1 and Package. This approach does not isolate the energy impact of specific system components such as memory, I/O operations, or network activity. Future research could employ more fine-grained energy profiling techniques to capture a detailed breakdown of energy usage across different subsystems.

The complexity and computational demands of different software projects also present a challenge. While the study analyzes energy consumption based on project and ruleset, additional factors such as code structure, execution time, and workload intensity were not explicitly accounted for. Further investigation into how these factors influence energy consumption could refine our understanding of energy-efficient software development.

Additionally, the scope of rulesets considered in this study is limited to predefined categories such as best practices, security, and code style. Other aspects of software quality, including performance optimizations, architectural design patterns, and compiler optimizations, could also play a significant role in energy efficiency. Expanding the analysis to include a broader range of rulesets may yield deeper insights into how software development practices affect energy consumption.

Another critical limitation is the reliance on EnergiBridge, the energy and power monitoring tool used in this study. While EnergiBridge provides useful estimations, no single measurement tool is completely free from inaccuracies. Factors such as sensor precision, sampling rate, and power estimation algorithms can impact the reported results. To increase reliability, future studies should incorporate multiple energy measurement tools and compare their outputs to obtain a more comprehensive and accurate assessment of energy consumption.

Finally, the generalizability of our findings to real-world applications remains an open question. The experiments were conducted in a controlled environment with specific configurations, whereas real-world applications often exhibit varying energy consumption patterns due to user interactions, dynamic workloads, and external dependencies. Future research could explore energy profiling in production environments to bridge the gap between controlled experiments and real-world scenarios.

Despite these limitations, this study contributes to the growing body of research on software energy efficiency. It highlights key areas where energy consumption can be optimized through static analysis and best practices, providing a foundation for further exploration in this field.

IX. Discussion

The results suggest that PMD's Security ruleset is significantly more energy-intensive than other rulesets. This could be due to the complexity of security-related static analysis, which may involve additional computational checks, such as vulnerability detection and deep code inspections, leading to increased processing time and energy use.

Similarly, the Terasology (develop branch) project showed the highest energy consumption among the evaluated projects. Given that Terasology is a large-scale game engine, it likely contains a complex codebase with numerous interdependencies, resulting in higher computational overhead during static analysis. This aligns with the expectation that larger and more intricate codebases demand more processing power and, consequently, more energy.

Interestingly, when idle energy was included, differences between rulesets and projects were negligible. This highlights the importance of accurately isolating the energy cost of static analysis from background system consumption. Without subtracting idle energy, the relative impact of different PMD rulesets and projects on energy consumption would have been masked.

These findings indicate that optimizing static analysis tools, particularly for security checks, could lead to significant energy savings. Future research could explore alternative configurations, optimizations, or parallel processing techniques to reduce the energy footprint of static analysis while maintaining thorough code inspections.

A. Broader Implications

The results of this study reinforce the broader role of energy-efficient tooling in sustainable software engineering. As software development workflows increasingly rely on automated analysis tools, often integrated into CI/CD pipelines, their cumulative energy cost becomes non-trivial. Frequent and unnecessary runs of static analysis tools in development environments can lead to unnecessary energy expenditure, making it important to explore smarter execution strategies.

One practical takeaway is that developers and DevOps engineers should carefully configure static analysis rulesets based on project needs. While comprehensive security checks are essential, running them excessively or indiscriminately on every minor code change can be wasteful. A potential solution is to prioritize selective analysis, where security checks are triggered less frequently, for instance, only before major releases or when security-sensitive files are modified.

Another key insight relates to tool design and optimization. If static analysis tools were built with energy efficiency as a first-class concern, developers could leverage optimized execution paths, caching mechanisms, and parallel processing to reduce their impact. Tools like PMD could integrate energy-aware modes that adapt their execution strategy based on available computational resources, reducing energy waste while still delivering useful insights.

B. The Main Trade-off

A notable consideration emerging from this study is the potential trade-off between rigorous static analysis and energy efficiency. Security rulesets tend to be the most computationally expensive, yet they provide critical vulnerability insights. As such, reducing their execution frequency to save energy must be carefully weighed against the risk of overlooking security flaws.

This trade-off suggests that software teams should evaluate energy consumption in the context of project priorities. For security-critical applications, such as financial systems or healthcare software, thorough static analysis might justify higher energy costs. In contrast, for non-critical projects, a more lightweight analysis approach could be appropriate.

Moreover, ruleset complexity should be assessed in relation to diminishing returns. If certain security rules contribute disproportionately to energy use while detecting only a marginal number of additional issues, software engineers might consider refining or simplifying them. Developing energy-aware heuristics that balance analysis depth with computational cost could provide a viable path forward.

C. Future Work

Several future research directions arise from this study:

- **Optimizing Static Analysis Execution**: Future research could explore techniques such as incremental analysis, where only modified portions of the codebase are analyzed instead of the entire project. This could significantly reduce energy consumption without compromising software quality.
- **Comparative Studies with Other Static Analysis Tools**: While this study focuses on PMD, analyzing a broader set of tools—including Checkstyle, FindBugs, SonarQube, and ESLint—could provide a more comprehensive understanding of how different analysis techniques influence energy consumption.
- **The Role of Hardware in Energy Efficiency**: Hardware plays a crucial role in determining energy efficiency. Investigating how different CPU architectures, cloud-based environments, or energy-efficient hardware configurations impact static analysis could yield valuable insights.
- **Integration of Energy Profiling into Developer Workflows**: There is currently limited visibility for developers regarding the energy cost of their tools. Monitors and sensors are quite hard to find, let alone comine with existing tools. Future research could explore real-time energy feedback mechanisms that inform developers about the energy impact of their analysis settings, helping them make more sustainable choices.
- **Energy-Aware CI/CD Pipelines**: Modern software development relies on frequent automated testing and analysis. Exploring adaptive scheduling techniques—where static analysis runs are dynamically adjusted based on energy consumption thresholds—could help balance software quality with environmental impact.

X. Conclusion

In this paper, we investigated the energy consumption of the static analysis tool PMD, focusing on how different rulesets and code characteristics influence power usage. Through controlled experiments, we measured PMD's energy footprint using various evaluation metrics, including total energy consumption, CPU energy usage, execution time, energy delay product, and power consumption. Our results highlight the trade-offs between ruleset complexity and energy efficiency, providing insights into optimizing static analysis for sustainable software development.

The findings indicate that while static analysis tools are essential for maintaining software quality, their configuration can significantly impact energy consumption. Developers can leverage this knowledge to balance code quality assurance with sustainability by selecting energy-efficient rulesets and minimizing unnecessary computations.

To directly address the research questions posed in this study, our findings reveal that the number and complexity of rules in a PMD ruleset significantly impact energy consumption, with complex rulesets like Security, Errorprone and Multithreading showing higher energy demands due to increased processing requirements. There is a partial correlation between the number of detected errors and energy consumption, though project size appears to be a more dominant factor, as seen in the high energy usage of Spring Framework despite varying error counts. Commonly used rulesets vary in energy efficiency, with simpler rulesets like Documentation and BestPractices offering lower energy costs, while Errorprone ruleset provides critical detection capabilities at a higher energy expense.

Future work could extend this research by analyzing additional static analysis tools, exploring energy optimizations within PMD, and evaluating the impact of different hardware architectures. By incorporating energy efficiency into tool selection and configuration, developers can contribute to more sustainable software engineering practices.

References

[1] C. Marantos, K. Salapas, L. Papadopoulos, P. K. Linos, and G. Goumas, “A flexible tool for estimating applications performance and energy consumption through static analysis,” *SN Computer Science*, vol. 2, no. 1, p. 21, 2021. [Online]. Available: https://doi.org/10.1007/s42979-020-00405-7

[2] P. López-García and D. Moreira, “Open questions on the origin of eukaryotes,” *Trends in Ecology & Evolution*, vol. 30, no. 11, pp. 697–708, 2015. [Online]. Available: https://doi.org/10.1016/j.tree.2015.09.005

[3] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, “A critical comparison on six static analysis tools: Detection, agreement, and precision,” *Journal of Systems and Software*, vol. 198, p. 111575, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222002515

[4] D. Li, C. Sahin, J. Clause, and W. G. Halfond, “Energy-directed test suite optimization,” in *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, 2013, pp. 62–69.

[5] R. Owczarski, L. Roovers, A. Christopoulos, and M. Z. Fazal, “Sse_2: Static software energy analysis,” 2025, accessed: 2025-04-04. [Online]. Available: https://github.com/MuhammadZainFazal/SSE_2