On the Energy Cost of Static Analysis Precision: An Empirical Study of SpotBugs Effort Levels

Sophie van der Linden
sophievanderlinden001@gmail.com
Delft University of Technology
Delft, The Netherlands

Xutong Liu
x.liu-14@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Carolin Brandt
c.e.brandt@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Abstract

Static analysis tools are commonly used in continuous integration (CI) pipelines to detect potential defects without executing the program. Such tools offer configuration options that control how thoroughly the code is analyzed. In SpotBugs, this is done through an effort level setting (Min, Less, More, Max), where higher levels enable deeper and more computationally intensive analysis. In this paper, we study how these effort levels affect energy consumption and analysis results. Our results show that higher effort levels generally consume more energy, which fits our intuition. However, the increase is not uniform: deeper static analysis comes with additional energy cost, but the marginal benefit of higher effort levels may be limited in some cases. This suggests that static analysis tool designers could make energy consumption more transparent and provide configuration options that explicitly expose trade-offs between analysis depth and energy cost.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

static analysis, energy consumption, software sustainability, continuous integration

ACM Reference Format:
Sophie van der Linden, Xutong Liu, Andy Zaidman, and Carolin Brandt. 2026. On the Energy Cost of Static Analysis Precision: An Empirical Study of SpotBugs Effort Levels. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805601

1 Introduction

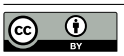
Continuous Integration (CI) pipelines automatically build, test, and analyze software after commits [7, 16]. In large-scale industrial settings, CI and continuous testing infrastructures execute builds and analyses repeatedly to maintain quality [14]. Although a single execution may appear inexpensive, the cumulative computational cost can be substantial. Recent research has shown that automated quality assurance processes, including CI and testing, consume considerable amounts of energy [13, 19]. However, prior work typically evaluates the pipeline as a whole, without isolating the contribution of individual phases. It therefore remains unclear where within CI workflows meaningful opportunities exist to reduce energy consumption.

Static analysis is one phase in CI pipelines whose energy implications remain underexamined. Static analysis tools (SATs) inspect source code without executing it and report warnings about potential defects or code quality issues. Deeper analysis explores more program paths and relationships, which can produce more warnings, but also requires more computation. At the same time, warnings are not always correct, and false positives are common [12]. For this reason, many tools allow the analysis depth to be configured [6]. Developers can therefore choose how strict the analysis should be, which may affect both the number of reported warnings and the computational cost.

Prior empirical research shows that static analysis tools are often configured more strictly in CI than in local environments [18]. Since static analysis depth is configurable rather than correctness-constrained, CI policies can deliberately trade analysis precision against computational cost. This creates a practical space where CI policies can trade analysis precision against computational cost. Furthermore, Beller et al. found that configuration settings of SATs tend to remain stable once adopted by development teams [4], and that default configurations are frequently used in practice. This suggests that configuration choices can have long-term impact, including on energy consumption in CI environments.

Despite the potential long-term impact and the configurable effort levels as a built-in mechanism for some SATs, there is limited empirical evidence on how different static analysis configurations influence energy consumption. To address this gap, in this paper, we focus on SpotBugs [1], an open-source SAT for Java, since it is the most frequently used SAT in our literature review for SAT pulications from 2020 to 2025 [1]. SpotBugs provides a clearly defined configuration parameter called *effort*, with four settings: Min, Less, More, and Max, which can be used to investigate how changes in analysis depth relate to energy usage and reported warnings. These levels adjust the internal analysis depth while keeping the tool and

[1] https://github.com/spotbugs/spotbugs



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805601

Table 1: Differences between SpotBugs effort levels

Feature	Description	1	2	3	4
Accurate Exceptions	More precise exception flow analysis		✓	✓	✓
Model Instanceof	Model `instanceof` in type analysis		✓	✓	✓
Track Guaranteed Dereferences	Track guaranteed null dereferences			✓	✓
Track Null Value Numbers	Track recoverable null values			✓	✓
Interprocedural Analysis	Across application classes			✓	✓
Interprocedural (Referenced)	Across referenced classes				✓
Conserve Space	Reduce memory use at lower precision	✓			

rule set consistent. This allows energy consumption comparison of different effort levels within the same environment. We study two research questions on how configuration choices affect energy use:

RQ1: How does energy consumption vary across different effort levels of SpotBugs?

RQ2: How does the change in energy consumption relate to the number of reported warnings?

To answer these questions, we conduct energy consumption measurements of four effort levels across ten open-source Java projects. Energy consumption is measured using EnergiBridge [15].

The remainder of this paper is structured as follows. Section 2 positions our study within existing research on sustainability and static analysis . Building on this background, Section 3 details the experimental design and measurement setup. Section 4 then presents the empirical results addressing our research questions. Finally, Section 5 concludes the paper and discusses the future work.

2 Related Work

Several studies have examined the relationship between sustainability and static analysis. Faghih and Jalili investigated whether SATs can detect energy-related defects in Android applications [11]. They proposed a framework to identify energy anti-patterns, such as inefficient resource usage. Their work focuses on improving the energy efficiency of applications through static analysis, but does not consider the energy consumption of the analysis tools themselves. Brosch studied the effect of applying static analysis recommendations on software energy consumption [5]. Using the Benchmarks Game project, he observed that implementing suggested changes sometimes increased energy usage. This highlights that static analysis results do not always lead to improved energy efficiency. However, the study does not measure the energy cost of running the analysis tools.

At the CI level, Ailane et al. proposed a Green DevOps guide with strategies to reduce the environmental impact of development workflows [2]. Their recommendations include reducing redundant builds, leveraging caching, and scheduling tasks during low-carbon periods. While these strategies address CI sustainability at a process level, they do not quantify the energy consumption of specific CI phases. Zaidman conducted an empirical study on the overall energy consumption of CI pipelines [19]. The results show that CI processes consume considerable amounts of energy. However, the analysis treats the pipeline as a whole and does not isolate the contribution of individual phases such as static analysis.

In summary, prior work has explored how static analysis can improve software energy efficiency and how CI pipelines consume energy. However, the energy cost of configurable static analysis itself within CI workflows remains largely unexamined.

Table 2: Projects used in the empirical study

Project (clickable)	Source	Build	Commit	Commits
Cruise-control	GitHub	Gradle	#e30eaf3	1,014
Elasticsearch	GitHub	Gradle	#39d2bb8	91,711
JUnit Framework	GitHub	Gradle	#464022d	10,262
OpenEMS	GitHub	Gradle	#fd6a70d	6,342
Spring	GitHub	Gradle	#0e56cd0	58,761
Spring Framework	GitHub	Gradle	#8642a39	34,130
Apache Flink	GitHub	Maven	#b152cde	37,297
Apache Maven	GitHub	Maven	#a336a2c	15,880
Apache Seatunnel	GitHub	Maven	#ca6447f	5,149
Google Guava	GitHub	Maven	#782206b	7,121

3 Methodology

3.1 Subject Tool: SpotBugs

SpotBugs was selected because it is widely used in Java projects and frequently referenced in empirical studies on static analysis [3, 6, 8–10, 17]. It is an open-source SAT for Java that integrates with common build systems such as Maven and Gradle. The effort configuration in SpotBugs provides four levels that adjust the depth of analysis while keeping the detection rules consistent, thereby enabling an empirical investigation of the energy implications of different precision trade-offs. Table 1 summarizes the documented differences between effort levels (1,2,3,4 means Min, Less, More, and Max).

3.2 Projects and Versions

We conducted the measurements on the same set of projects studied by Zaidman [19]. Table 2 lists the projects, their build systems, selected commit hashes, and total number of commits. For each project, we selected the most recent commit with a successfully passing CI pipeline to ensure that the code compiled without modification.

Gradle and Maven are widely used Java build automation tools. In our experiments, most projects use Java 17. OpenEMS was configured with Java 21, and Spring Boot with Java 25, as required by their build configurations. We used Gradle 9.2.1 for all Gradle-based projects and Maven 3.8.6 for Maven-based projects. SpotBugs version 4.9.8 was used with plug-in version 6.4.7 (Gradle) and 4.9.8.2 (Maven). Both main and test source files were analyzed.

For `Maven` projects, the parameter `includeTests` was set to `true`. Before measurements, we executed `mvn clean install -DskipTests` to compile the projects and cache dependencies. Then using `mvn -Dspotbugs spotbugs:spotbugs` to conduct energy measurements. `Gradle` uses incremental builds, meaning that tasks are skipped if inputs and outputs are unchanged[2]. Since our experiments require repeated execution of SpotBugs without code

[2] https://docs.gradle.org/current/userguide/incremental_build.html

changes, incremental behaviour would underestimate energy consumption. To ensure consistent execution, we first ran `gradlew clean build -x test`. Energy measurements were then performed using `gradlew spotbugsMain spotbugsTest –rerun-tasks` while excluding compilation and packaging tasks. This isolates the static analysis phase from the build process.

3.3 Setup for Energy Measurements

Energy consumption was measured using EnergiBridge, a cross-platform tool that retrieves CPU energy data from model-specific registers [15]. We therefore measure CPU-attributed energy rather than full-system energy consumption.

All experiments were conducted on a dedicated desktop machine (Dell OptiPlex 7060, Intel i5 8th Gen, Windows 10). The machine was connected via Ethernet to ensure stable network conditions. Before each measurement session, non-essential applications and background services were closed.

Each configuration was executed 30 times. Prior to each run, a 5-minute CPU warm-up was performed, followed by a 1-minute cool-down period between runs to reduce thermal and power-state interference. Outliers were removed using a z-score criterion, excluding observations that deviated more than three standard deviations from the mean ($|\bar{x} - x| > 3s$). Subsequent statistical analyses were performed on the cleaned data.

3.4 Performance Metric

To answer RQ1, we used the total number of warnings reported by SpotBugs as an indicator of configuration effectiveness. For both energy consumption and warning counts, we tested whether effort levels differed within each project using the Friedman test. When the test indicated significant differences, we performed pairwise Wilcoxon signed-rank tests with Holm correction for multiple comparisons.

To answer RQ2, we also considered the effectiveness of each configuration. As this work is exploratory in nature, we treat the total number of reported warnings as a coarse proxy for the analytical power of SpotBugs. We do not, at this stage, differentiate between true and false positives or evaluate additional validity-related metrics such as recall or false alarm rate.

4 Results

4.1 RQ1: How does energy consumption vary across effort levels?

Table 3 reports the average energy consumption per effort level and the corresponding Friedman test results. Figure 1 visualises the mean energy consumption across projects. For all ten projects, the Friedman test indicates a statistically significant effect of effort level on energy consumption ($p < 0.01$ for all projects). This shows that effort configuration influences energy usage.

Post-hoc Wilcoxon signed-rank tests with Holm correction (Table 4) reveal where these differences occur. Significant increases in energy consumption are most frequently observed when moving from *Min* to *Less* and from *Less* to *More*. Several projects, including OpenEMS, Spring Framework, Apache Flink, Apache Maven, and Guava, show strong statistical significance for these comparisons.

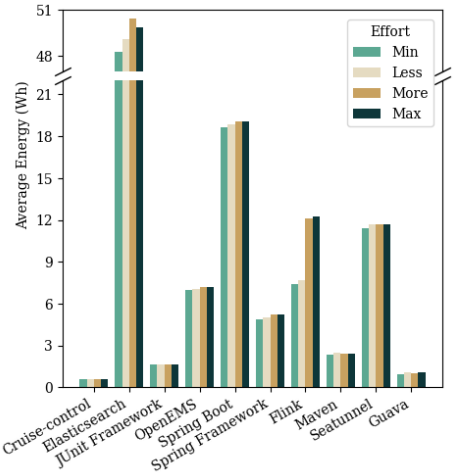


Figure 1: Average energy consumption per project across SpotBugs effort levels.

In contrast, differences between *More* and *Max* are often not statistically significant. For most projects, the increase from *More* to *Max* results in limited or negligible additional energy consumption. This suggests diminishing marginal energy costs at the highest effort level. A small number of cases show non-monotonic behaviour, where higher effort levels do not always correspond to higher energy consumption. These cases appear to be project-specific.

Answer to RQ1: Increasing the SpotBugs effort level generally increases energy consumption, particularly when moving from lower to moderate effort levels. However, the additional energy cost from *More* to *Max* is typically limited, and the relationship is not strictly monotonic for all projects.

4.2 RQ2: How does energy consumption relate to the number of reported warnings?

Across most projects, higher effort levels result in a larger number of reported warnings. This aligns with the design of SpotBugs, where higher effort increases analysis depth.

To evaluate the trade-off between energy use and detection output, we computed the number of warnings per unit of energy (warnings per Wh). The results are shown in Figure 2. For nine out of ten projects, the *Min* effort level provides the highest number of warnings per unit of energy. For Spring Framework, *Less* is the

Table 3: Friedman test p-values and average energy consumption (Wh) per effort level.

Project	p-value	Min	Less	More	Max
Cruise-control	1.2e-4	0.5581	0.5735	0.6058	0.5837
Elasticsearch	3.4e-3	48.27	49.12	50.47	49.90
JUnit	7.0e-4	1.627	1.632	1.665	1.668
OpenEMS	2.8e-12	6.988	7.059	7.223	7.194
Spring Boot	8.5e-4	18.66	18.89	19.05	19.03
Spring Framework	1.7e-10	4.895	5.011	5.229	5.207
Apache Flink	1.2e-12	7.366	7.698	12.13	12.24
Apache Maven	9.9e-10	2.318	2.452	2.428	2.440
Apache Seatunnel	9.5e-3	11.37	11.71	11.67	11.66
Google Guava	5.8e-11	0.9509	1.052	1.026	1.059

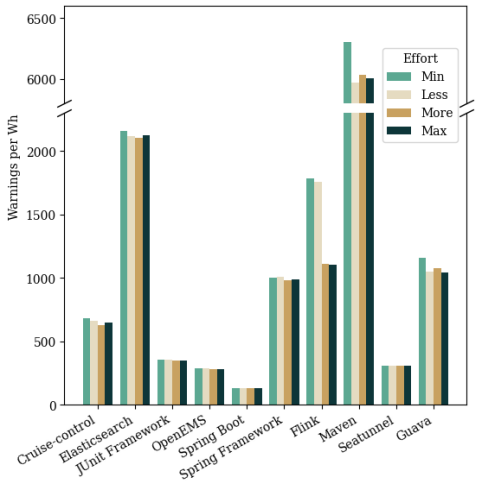


Figure 2: Number of warnings divided by energy consumption per project per effort

most efficient configuration. Although higher effort levels may report more warnings in absolute terms, the additional findings often come at a disproportionately higher energy cost. As a result, energy efficiency decreases for higher effort levels.

Answer to RQ2: Higher effort levels generally produce more warnings, but they are less energy-efficient. In most projects, the *Min* effort level yields the highest number of warnings per unit of energy.

4.3 Threats to Validity

Internal validity. Our results may be influenced by measurement noise and system-level variability. Although we followed a experiment protocol with repeated runs, CPU warm-up phases, and cool-down periods, our measurements remain sensitive to background processes and thermal fluctuations. To mitigate this, we performed multiple runs and applied outlier filtering. However, some variability may remain. In addition, energy consumption was measured using EnergiBridge, which estimates CPU-attributed energy; therefore, unmeasured factors may still influence the observed differences.

Table 4: Holm-corrected Wilcoxon signed-rank post-hoc p-values for adjacent effort comparisons. Bold values indicate statistically significant differences ($p < 0.05$).

Project	Min–Less	Less–More	More–Max
Cruise-control	0.358	**0.036**	**0.036**
Elasticsearch	0.236	**0.009**	0.192
JUnit Framework	1.000	0.857	1.000
OpenEMS	**1.5e-3**	**2.3e-8**	0.187
Spring Boot	**0.014**	0.234	0.851
Spring Framework	**0.013**	**7.7e-7**	0.413
Apache Flink	**6.4e-4**	**2.0e-33**	0.060
Apache Maven	**1.7e-11**	**3.4e-3**	0.311
Apache Seatunnel	**0.047**	1.000	1.000
Guava	**2.1e-7**	**2.0e-5**	**2.8e-6**

External validity. All experiments were conducted on a dedicated Windows-based desktop machine, which may not fully represent typical CI environments that often rely on Linux-based or containerized server. While we expect the relative trends across effort levels to generalize, absolute energy consumption may differ under other hardware and system configurations.

Construct validity. We measured energy consumption using a software-based tool that focuses on CPU energy, which does not capture the full-system energy footprint (e.g., memory, storage, or cooling). In addition, we used the number of reported warnings as a proxy for analysis effectiveness. Since static analysis tools may produce false positives, warning counts do not necessarily reflect the true usefulness or correctness of detected issues. Therefore, our results should be interpreted as reflecting trade-offs between energy consumption and warning volume rather than overall analysis quality.

5 Conclusions and Future Work

This study shows that the largest energy increases occur when moving from *Min* to *Less* and from *Less* to *More*, while the additional cost from *More* to *Max* is often small. When considering warnings per unit of energy, the *Min* configuration is the most efficient in nine out of ten projects. Overall, our results suggest a trade-off between analysis depth and energy consumption: higher effort levels may yield more warnings, but not always in proportion to their additional energy cost. Given the limited scope of the study and the use of warning counts as a coarse effectiveness proxy, these findings should be interpreted as preliminary observations rather than definitive configuration recommendations.

Future work can extend this study beyond the predefined effort levels offered by SpotBugs. A more fine-grained insight into how specific analysis features contribute to energy consumption can be interesting.

Besides, in this study, configuration effectiveness was approximated using the total number of reported warnings. However, warning counts do not distinguish between true positives and false positives. As a result, the observed trade-off between energy consumption and detection output does not fully capture analysis quality. Future work could evaluate static analysis configurations on labelled datasets with known defects, enabling assessment of metrics such as false positive rates. This would enable a more informed analysis of the energy–quality trade-off.

Data Availability

The datasets and research artifacts are available at [1].

References

[1] 2026. *Replication kit of: On the Energy Cost of Static Analysis Precision An Empirical Study of SpotBugs Effort Levels.* doi:10.5281/zenodo.18516019

[2] Mohamed Toufik Ailane, Carolin Rubner, and Andreas Rausch. 2026. Enabling a Green Life-Cycle Approach to Software Sustainability: A Guideline-Driven Framework. In *Energy-Efficient Algorithms and Systems in Computing: Optimizing Performance and Sustainability Through Advanced Computational Methods.* 195–211.

[3] Midya Alqaradaghi and Tamás Kozsik. 2022. Inferring The Best Static Analysis Tool for Null Pointer Dereference in Java Source Code. In *CEUR Workshop Proceedings*, Vol. 3237. https://ceur-ws.org/Vol-3237/paper-alq.pdf

[4] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source

Software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 470–481.

[5] Christoph Brosch. 2023. Influence of Static Code Analysis on Energy Consumption of Software. In *Lecture Notes in Informatics (LNI), Gesellschaft für Informatik*. 111–120. https://dl.gi.de/handle/20.500.12116/43330

[6] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: Automatically incrementalizing Java bug pattern detection. *Proceedings of the ACM on Programming Languages* 5 (2021).

[7] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. 2022. Uncovering the Benefits and Challenges of Continuous Integration Practices. *IEEE Trans. on Softw. Engineering* 48, 7 (2022), 2570–2583.

[8] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qihong Chen. 2021. Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 207–218.

[9] Kalle Hjerppe, Jukka Ruohonen, and Ville Leppänen. 2020. Annotation-Based Static Analysis for Personal Data Protection. In *Privacy and Identity Management. Data for Better Living: AI and Privacy*. 343–358.

[10] Jiayi Hua and Haoyu Wang. 2021. On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 530–534.

[11] Mohammad Jalili and Fathiyeh Faghih. 2022. Static/Dynamic Analysis of Android Applications to Improve Energy-Efficiency. In *2022 CPSSI 4th International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. 1–8.

[12] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023).

[13] Xutong Liu, Robert Arntzenius, and Andy Zaidman. 2026. On The Energy Consumption of Continuous Integration in Open-Source Java Projects. In *Proc. 1st Int'l Workshop on Green Software Evolution (Greenvolve)*. IEEE, 181–188.

[14] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *Int'l Conf. on Software Engineering: Softw. Engineering in Practice (ICSE-SEIP)*. 233–242.

[15] June Sallou, Luís Cruz, and Thomas Durieux. 2023. EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement. doi:10.48550/arXiv.2312.13897

[16] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar Da Costa, and Uirá Kulesza. 2022. The effects of continuous integration on software development: a systematic literature review. *Empirical Software Engineering* 27, 3 (2022).

[17] Darko Stefanović, Danilo Nikolicć, Dusanka Dakicć, Ivana Spasojevicć, and Sonja Risticć. 2020. Static Code Analysis Tools: A Systematic Literature Review. In *DAAAM Proceedings*. 0565–0573.

[18] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25, 2 (2020), 1419–1457.

[19] Andy Zaidman. 2024. An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects. In *Proc. Int'l Conf. on Automation of Software Test (AST)*. ACM, 214–218.