How Configuration Choices Shape Environmental Impact of Static Analysis Tools

Quantifying Energy Usage of Different Static Analysis Tool Configurations used in Continuous Integration

Sophie van der Linden[1]

Supervisors: Carolin Brandt[1]**, Xutong Liu**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 25, 2026

Name of the student: Sophie van der Linden
Final project course: CSE3000 Research Project
Thesis committee: Carolin Brandt, Xutong Liu, Benedikt Ahrens

Abstract

Continuous Integration, CI, pipelines are widely used to ensure code quality through automated builds, tests, and static analysis. While prior work has examined the overall energy consumption of CI workflows, the energy consumption of individual phases remains unexplored. This study investigates the energy consumption of static analysis using SpotBugs across 10 open-source Java projects. SpotBugs was selected based on a systematic literature review, its widespread usage, and its configurable effort levels. Energy measurements were conducted for the four SpotBugs effort levels on both Gradle and Maven projects. The results show that energy consumption differs significantly between efforts, with the biggest differences between Min and Less, and Less and More. Performance analysis shows that increased effort improves bug detection in projects, but comes with a higher energy consumption. The most efficient effort is Min, which found the most warnings with regards to energy consumption.

1 Introduction

Continuous Integration, CI in short, is a common software engineering practice. This development philosophy proposes numerous sub-practices that are synergetic, such as automated tests, regular builds and frequent commits [1]. It has multiple benefits, such as consistency of projects and reliable bug detection [2].

Typically, each commit triggers a CI pipeline to ensure code quality, consisting of steps such as building, testing, and static analysis [3]. The focus of this paper is on the static analysis phase. Static analysis helps software engineers detect potential bugs in code, without actually executing the program [4]. There are multiple static analysis tools, SATs, available, all with their own options for configurations and rules.

Although executing this CI pipeline for a single commit may appear insignificant, it is important to recognise that CI processes typically occur in large-scale software development environments. For example, the Elasticsearch project recorded over 9.600 commits in the past year [5], corresponding to an equivalent number of CI pipeline executions. Frequent execution of the entire pipeline is computationally expensive [6].

To reduce these problems, one can look at different configurations of different SATs. These can be lightweight, therefore requiring relatively little time and resources, but also heavy, which might be able to provide a deeper analysis, but comes at the sacrifice of time and resources.

The use of resources also comes with another cost, namely the environmental impact. Little research has been done on the environmental implications of configurations of CI processes. Previous work has shown that the energy consumption of CI processes is significant [7]. However, this study focused on the footprint of CI workflows as a whole, without looking at contributions of the different phases with their different tools and configurations.

Before the environmental impact can be assessed, it is important to first identify which SATs are most widely used. Selecting a widely adopted tool for a commonly used programming language is essential for the research, as it could make a significant difference in environmental impact. To ensure the rest of the research is conducted with the right SAT, we answer the next Research Question, by conducting a literature survey:

RQ1: "What are the most commonly mentioned static analysis tools for Java in primary studies?"

Identifying the most frequently mentioned static analysis tool provides the foundation for analysing its empirical impact. Based on the results of RQ1, this study investigates how different tool configurations influence energy consumption.

RQ2: "How does energy consumption vary across static analysis tool configurations?"

As it is probable that energy efficient configurations come at the cost of performance, we also verify whether this is the case. We therefore measure and compare the tool's performance. We define performance as the total amount of warnings found by a tool with a certain configuration. The drawback will be taken into account with the last Research Question:

RQ3: "Is there a trade-off between performance and energy consumption of different configurations?"

We identified SpotBugs as the most frequently reported static analysis tool for Java, making it a suitable candidate for empirical evaluation. Our results indicate that SpotBugs' energy consumption increases with higher effort levels, with the most pronounced increases occurring between the lower effort configurations. Moreover, higher effort levels are less energy-efficient than the lowest effort setting, which detects the highest number of warnings per unit of energy consumed.

This paper first analyses previous work related to energy consumption and SATs in Section 2. Secondly, it discusses the methodology and the results of the systematic literature review in Section 3, as an answer to RQ1. This section also goes into detail on which SAT is discussed in this study, with its configuration options. Then, in Section 4, we discuss the experiment setup of the energy measurements, as well as the methodology of measuring performance. Section 5 reports the results of the energy and performance measurements. We discuss the found results, along with threats to validity and suggestions for future work in Section 6. Section 7 includes a discussion on the reproducibility and ethics of the research. Lastly, we report the conclusions in Section 8.

The data and software used for this project are available on Zenodo [8].

2 Related Work

Multiple research papers studied the combined field of sustainability and static analysis already. Faghih and Jalili investigated whether SATs can find energy defects in Android [9]. They proposed a framework, which can analyse an Android application and effectively detect energy anti-patterns, such as inefficient resource usage or unnecessary background operations. Their work focused on mobile applications and on improving energy efficiency through static analysis, but did not address the energy usage introduced by the execution of the SATs themselves.

Beyond using static analysis to reduce energy impact, Brosch examined the influence of implementing SAT annotations on the energy consumption of existing projects [10]. Specifically, he studied the Benchmarks Game project and evaluated whether SAT suggestions affected the energy consumption. He found that three out of eight algorithms showed an increase in energy consumption after implementation of the recommendations of the SAT. This research highlighted that SAT suggestions may have an unintended negative influence on the energy efficiency of software systems.

With regards to the sustainability of software engineering, Ailane et al. proposed a Green DevOps Guide, to integrate sustainable choices throughout the entire DevOps cycle [11]. On the subject of CI pipelines, they mentioned multiple optimisation strategies, namely:

- "Reducing redundant builds via change detection
- Leveraging caching layers for dependencies and containers
- Running tests in parallel or on-demand rather than blanket execution
- Scheduling non-critical builds during off-peak hours or low-carbon intensity periods" [11, p. 205]

However, Ailane et al. did not quantify energy consumption of specific CI steps.

As previously mentioned, Zaidman did perform an empirical investigation into the energy consumption of CI processes [7]. These results demonstrated that the energy required for CI pipelines is significant. However, he did not perform research into the energy consumption of the different phases of the CI pipeline, such as static analysis.

In contrast to aforementioned studies, this research contributes to initial empirical insights into the energy consumption of the static analysis phase of a CI pipeline.

3 Systematic Literature Review on SATs

3.1 Methodology

We performed a systematic literature review to answer RQ1. There already exists a systematic literature review on the mentions of SATs in literature, done by Stefanović et al. [12]. The existing literature review was conducted in 2020 and is therefore no longer fully representative of the current state of the field. Therefore, we conducted a new systematic literature survey for this research.

This literature review focused on tools for Java, as Stevanoic et al. [12] discovered that the most supported programming language in previous papers on SATs is Java. Java is also one of the most used programming languages on GitHub [13].

We used Scopus as the database to be searched. It was one of the two databases used in the previous systematic literature review by Stefanović et al. [12], but has less non-journal and non-English papers than Google Scholar [14], the other database used in the previous literature survey.

We used the following query, in title, abstract and keywords: *"Static code analysis" AND "tools" AND "detection" AND "Java"*. We chose this query as it is the same as in the previous literature study, with the exception of Java being added.

We assumed that the previous literature review covers the time period before 2020. Therefore, for this research, the publication year had to be in or after 2020. This search resulted in a total of 89 candidate studies.

The inclusion and exclusion criteria used were the same as used in the literature review by Stefanović et al. [12], except when explicitly stated otherwise. Inclusion criteria applied to these papers were:

- Paper has to be limited to Computer Science. This is not one of the inclusion criteria in the previous literature survey, but was used to make the search simpler and faster.
- Paper has to be written in English.
- Paper has to present static analysis tools.

Exclusion criteria applied to these papers were:

- Duplicate papers should be removed.
- Paper should not already have been included in the systematic literature by Stefanović et al. [12]. This is not one of the exclusion criteria in the previous literature study, but was used for deduplication.
- If one author had more than one paper regarding the same tool, only one paper should be included in the review. This should be the paper that contains the most tools.
- Papers focusing exclusively on static analysis security tools should be excluded. This is not one of the exclusion criteria in the earlier literature study, but was used because security analysis sometimes represents a distinct category within the CI pipeline and falls outside the scope of this study.

This resulted in the following flow:

- 89 after search ->
- 85 after limiting to Computer Science ->
- 81 after limiting to English ->
- 79 after removing duplicates ->
- 78 after removing papers already included in the previous literature study ->
- 50 after reading title and abstract ->
- 21 after reading ->
- 17 after removing papers of authors with more than one paper.

After applying the inclusion and exclusion criteria, we extracted SATs from 17 papers. The tools were not extracted if:

- They are mentioned in related work, or
- They are not a static analysis tool, or
- They are a static analysis **security** tool, or
- They are not a tool for Java.

3.2 Results

Table 1 covers the papers of which tools were extracted with their corresponding publication year.

Year	Articles
2020	[15] [16]
2021	[17] [18] [19]
2022	[20] [21]
2023	[22] [23] [24] [25] [26] [27] [28]
2024	[29] [30]
2025	[31]

Table 1: Publication years of articles

Table 2 summarises the three most frequently mentioned static analysis tools in the reviewed literature.

Tool	Articles	Mentions in [12]	Total
SpotBugs or FindBugs	[15] [17] [18] [19] [20] [22] [25] [28] [26]	4	13
PMD	[17] [18] [19] [22] [25] [26] [27] [28]	3	11
SonarQube	[17] [20] [25] [26] [28] [30] [29] [31]	3	11

Table 2: Static analysis tools mentions

Overall, 9 of the papers contain the tool SpotBugs, or its predecessor FindBugs [32]. Together with the 4 mentions of SpotBugs by Stefanović et al. [12], SpotBugs was the most often mentioned tool with 13 mentions.

In addition to frequency of use, SpotBugs stands out from other SATs by offering an easy to set configuration option. It allows users to set the analysis scope through an effort option [33]. More information about the effort parameter can be found in Section 3.3.

In contrast, PMD and SonarQube present several limitations with regards to configurations. PMD has more than 400 built-in rules, which can be extended with custom rules [34]. As a result, choosing a set of configurations is harder, as the rulesets would mainly differ in number and types of rules, not analysis precision for the same task, which is the goal of comparing configurations.

SonarQube, on the other hand, operates as a platform, rather than an SAT alone [35]. The analysis is done on servers, making it more opaque and more difficult to set up for reliable energy measurements. Furthermore, the inclusion of communication with servers and the internet make it harder to create a controlled environment for energy measurements.

The benefits of SpotBugs having an easy configuration option through which analysis precision can be compared and it being a stand-alone tool works well for this research. The research is specifically meant to compare different configuration options and measure the impact of those options. Therefore, Spotbugs was chosen as the tool to use for RQ2 and RQ3.

3.3 SpotBugs

SpotBugs [36] is an open-source static analysis tool for Java that supports ten categories of bug patterns, including bad practices and performance-related issues. It provides plugins for common build tools and development environments, such as Ant, Eclipse, Gradle and Maven.

Configuration

SpotBugs provides an effort configuration option with four levels: *"Min", "Less", "More"* and *"Max"*. The documentation reports that "Effort value adjusts internal flags of SpotBugs, to reduce computation cost by lowering the prediction." [33]. Table 3 shows the SpotBugs documentation on the differences between each effort [33].

Flags in		Effort Level			
FindBugs.java	Description	Min	Less	More	Max
Accurate Exceptions	Determine (1) what exceptions can be thrown on exception edges, (2) which catch blocks are reachable, and (3) which exception edges carry only "implicit" runtime exceptions.		✓	✓	✓
Model Instanceof	Model the effect of instanceof checks in type analysis.		✓	✓	✓
Track Guaranteed Value Derefs in Null Pointer Analysis	In the null pointer analysis, track null values that are guaranteed to be dereferenced on some (non-implicit-exception) path.			✓	✓
Track Value Numbers in Null Pointer Analysis	In the null pointer analysis, track value numbers that are known to be null. This allows us to not lose track of null values that are not currently in the stack frame but might be in a heap location where the value is recoverable by redundant load elimination or forward substitution.			✓	✓
Interprocedural Analysis	Enable interprocedural analysis for application classes.			✓	✓
Interprocedural Analysis of Referenced Classes	Enable interprocedural analysis for referenced classes (non-application classes).				✓
Conserve Space	Try to conserve space at the expense of precision, e.g., prune unconditional exception thrower edges for control flow graph analysis, to reduce memory footprint.	✓			

Table 3: Complete list of differences between SpotBugs efforts

Flags in		Effort Level			
FindBugs.java	Description	Min	Less	More	Max
Skip Huge Methods	Skip method analysis if length of its bytecode is too long (6,000).	✓	✓	✓	

Table 3: Complete list of differences between SpotBugs efforts, continued

4 Methodology

All of the code used for this research can be found on Zenodo [8].

4.1 Projects and Versions

We conducted the measurements on the same set of projects as the projects Zaidman examined [7]. These projects, their build systems, the exact commit hashes used, and the total amount of commits of the project can be found in Table 4. For each project, the most recent commit with a successfully passing CI pipeline was selected, as such commits provide a higher degree of confidence in successful compilation.

Project	Source	Build System	Commit	Amount of Commits
Cruise-control	[37]	Gradle	#e30eaf3	1014
Elasticsearch	[38]	Gradle	#39d2bb8	91711
JUnit Framework	[39]	Gradle	#464022d	10262
OpenEMS	[40]	Gradle	#fd6a70d	6342
Spring Boot	[41]	Gradle	#0e56cd0	58761
Spring Framework	[42]	Gradle	#8642a39	34130
Apache Flink	[43]	Maven	#b152cde	37297
Apache Maven	[44]	Maven	#a336a2c	15880
Apache Seatunnel	[45]	Maven	#ca6447f	5149
Google Guava	[46]	Maven	#782206b	7121

Table 4: Projects with build system, specific commits used for measurements, and total amount of commits, sorted by build system

Most projects work with Java 17, except for OpenEMS, which was configured to use Java 21, and Spring Boot, which was configured to use Java 25.

Gradle version 9.2.1 was used, as it is the most recent version of Gradle as of this moment and worked for all Gradle projects. Maven version 3.8.6 was used, as this was required by Apache Flink and worked for the other Maven projects.

For each project, the SpotBugs plug-in was added. SpotBugs version 4.9.8 was used, along with plug-in versions 6.4.7 for Gradle and 4.9.8.2 for Maven. These versions are all the current latest versions. Both the main and test files are analysed.

For the Maven-based projects, parameter `includeTests` was set to `true`, as to include the test files in the SpotBugs analysis. Before the first run, we executed the command `mvn clean install -DskipTests`, to compile all the code. Dependencies were also downloaded and cached beforehand, to increase speed and reduce dependence on the internet. For the measurements, we ran command `mvn -Dspotbugs spotbugs:spotbugs`.

Gradle makes use of incremental builds, which means that it only runs the tasks which are not up-to-date due to a change in input or output [47]. As the code is not changed in between runs, this is not a fair representation of the use of a static analysis tool in a CI pipeline, where at least some code is changed. In addition, a measurement of an entirely up-to-date task would only measure the energy consumption of checking whether all tasks are up-to-date, but not of running SpotBugs with a certain configuration. Secondly, Gradle runs all the tasks the current task depends on, due to its inferred task dependencies functionality [47]. This means that for a SpotBugs task, the build of the project is also included, whilst this is a separate task in the CI pipeline. Because of these two reasons, we made the decision to first run `.\gradlew clean build -x test`. Subsequently, for the energy measurements, we ran `.\gradlew spotbugsMain spotbugsTest --rerun-tasks -x compileJava -x compileTestJava -x classes -x processResources -x testClasses -x processTestResources -x jar`. The command often excluded extra, project specific tasks that were already run with `.\gradlew clean build`.

4.2 Setup for Energy Measurements

We performed the measurements with the tool EnergiBridge [48]. It is simple, makes for reproducible work and is available for multiple platforms [49]. It measures energy by accessing the energy information by reading the Model-specific Register. EnergiBridge is known to panic sometimes due to an overflow error under high-load conditions [50]. Therefore, a small change was made to the source code. This change is explained as an answer to the GitHub issue [50].

We measured the energy consumption on one computer. This was the Dell OptiPlex 7060 on Windows 10, with an Intel i5 8th Gen core. We chose a desktop PC over a laptop, as the desktop is not battery powered, therefore having more stable measurements. The computer had a cabled internet connection. We chose cabled internet as the cabled connection has less interference and has a lower power consumption than WiFi [51]. Moreover, energy consumption from a cable connection is more stable than the consumption from a wireless connection.

We placed the computer in an indoor room within a university building, where external temperature fluctuations are limited. We conducted measurements during weekdays when the building heating system operates continuously, ensuring relatively stable ambient temperature conditions throughout the measurement period.

We closed all non-essential user-level applications prior to the measurements. Furthermore, we turned off notifications and killed unnecessary services that were running in the background. Lastly, we removed all external hardware peripherals, except for the power and Ethernet cables.

Before each measurement, a 5-minute warm-up of the CPU was performed. We performed the measurements for each single ruleset 30 times. A ruleset consists of a project, tool and configuration. The amount of 30 measurements accounts for enough measurements to take a reliable average of [52]. One minute of rest was taken between each measurement, to prevent tail measurements.

One run of measurements consisted of running all the projects with one effort, with one run being repeated 30 times. Measurements were interleaved across projects rather than being conducted in sequence for a single project. This makes the measurements even less biased to external conditions. We repeated the runs four times, each with a different effort.

We created automation batch scripts to streamline the execution of measurements and reduce manual effort.

4.3 Performance

To answer RQ3, we also considered the effectiveness of each configuration. We assessed the effectiveness by comparing the number of warnings reported by SpotBugs, which serves as an indicator of the amount of detected issues. We extracted the total number of warnings from the SpotBugs reports and used it for comparison across configurations.

5 Results

5.1 Energy Measurements

For each project and effort, we conducted 30 runs. Runs in which EnergiBridge failed to report a total energy consumption value were considered invalid and excluded from the analysis.

The collected data was visualised using a combined box and violin plot, to illustrate both descriptive statistics and the shape of the distribution of the data. An example of one of these can be seen in Figure 1.

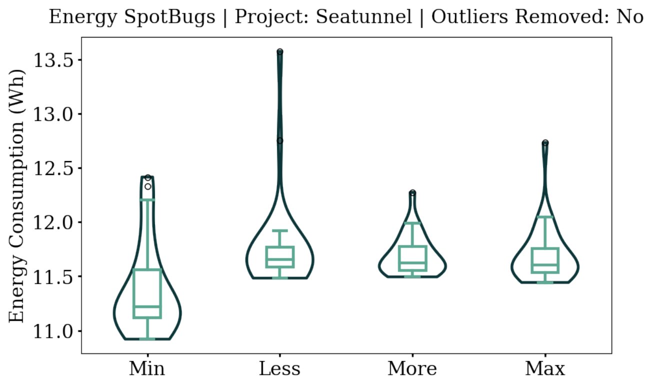

Figure 1: Energy consumption of different SpotBugs efforts on project Apache Seatunnel, before outlier removal. Lower and upper fences are first quartile and third quartile, with a line at the median. The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range from the box.

We removed outliers using the z-score outlier removal, excluding points that deviate more than three standard deviations from the mean, $|\bar{x} - x| > 3s$. This resulted in distributions closer to normality, as can be seen in Figure 2.

After outlier removal, we compared sample means using a bar chart. This graph is shown in Figure 3. The means per project and effort can also be seen in Table 5.

We assessed statistical significance between configurations using an unbalanced Friedman test. This non-parametric, repeated-measures test accounts for paired runs across efforts

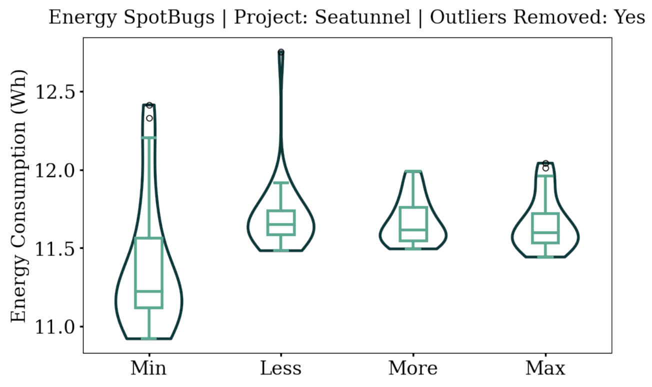

Figure 2: Energy consumption of different SpotBugs efforts on project Apache Seatunnel, after outlier removal. Visualisation is the same as Figure 1.

while handling the possible unequal number of measurements per effort. The test produced a p-value for each project, which can be seen in Table 5.

Project	**p-value Friedman test**	**Avg. Energy Min (Wh)**	**Avg. Energy Less (Wh)**	**Avg. Energy More (Wh)**	**Avg. Energy Max (Wh)**
Cruise-control	$1{,}2 \times 10^{-4}$	0,5581	0,5735	0,6058	0,5837
Elastic-search	$3{,}4 \times 10^{-3}$	48,27	49,12	50,47	49,90
JUnit	$7{,}0 \times 10^{-4}$	1,627	1,632	1,665	1,668
OpenEMS	$2{,}8\times10^{-12}$	6,988	7,059	7,223	7,194
Spring Boot	$8{,}5 \times 10^{-4}$	18,66	18,89	19,05	19,03
Spring Framework	$1{,}7\times10^{-10}$	4,895	5,011	5,229	5,207
Apache Flink	$1{,}2\times10^{-12}$	7,366	7,698	12,13	12,24
Apache Maven	$9{,}9\times10^{-10}$	2,318	2,452	2,428	2,440
Apache Seatunnel	$9{,}5 \times 10^{-3}$	11,37	11,71	11,67	11,66
Google Guava	$5{,}8\times10^{-11}$	0,9509	1,052	1,026	1,059

Table 5: Result Friedman test and average energy consumption per project

Across all ten projects, effort level shows a statistically significant effect on energy consumption. The Friedman test indicates that effort configuration influences energy usage, with $p < 0,0095$ for all projects. This indicates that, in principle, effort configuration influences energy usage across projects.

Post-hoc pairwise comparisons using Wilcoxon signed-rank tests with Holm correction identify where those differences arise. The results of the Wilcoxon tests between increasing effort pairs can be found in Table 6.

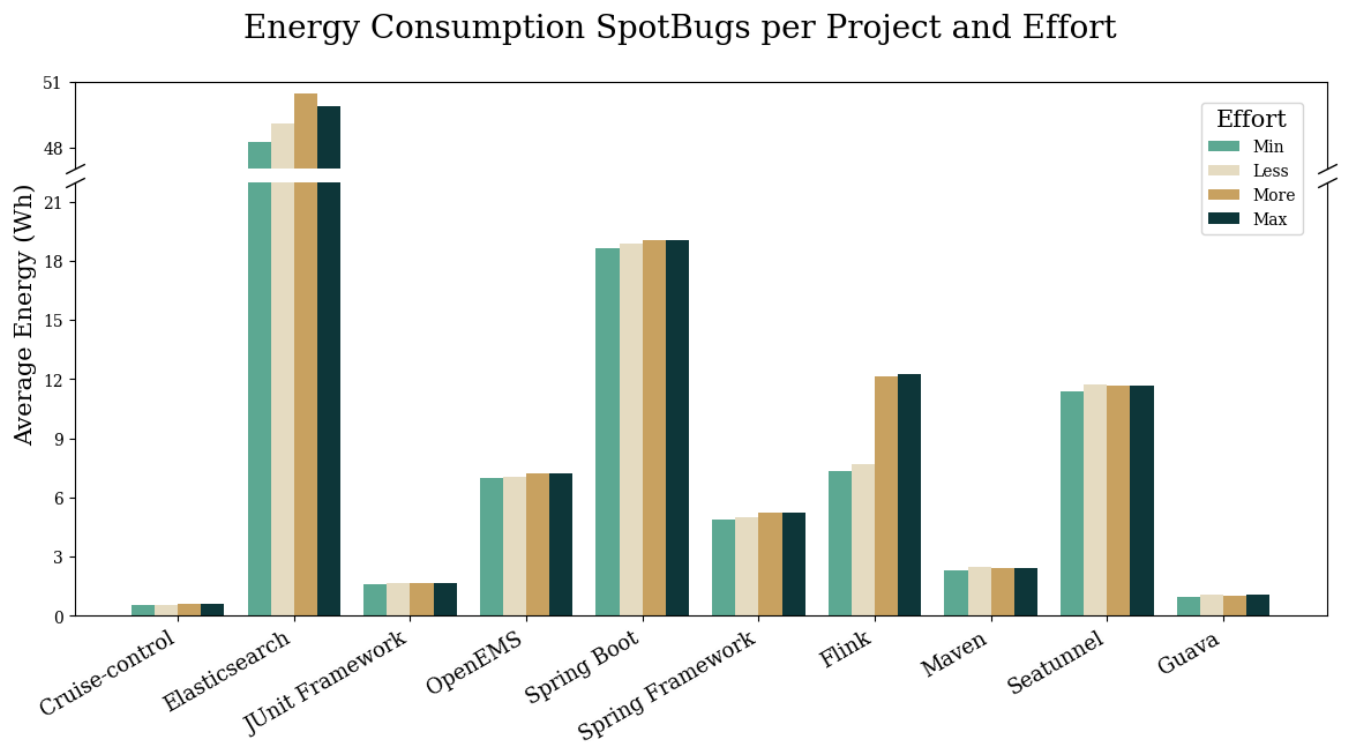

Figure 3: Average energy consumption per project per effort

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	$\mathbf{1{,}5 \times 10^{-3}}$	$\mathbf{2{,}3 \times 10^{-8}}$	0,187
Spring Boot	**0,014**	0,234	0,851
Spring Framework	**0,013**	$\mathbf{7{,}7 \times 10^{-7}}$	0,413
Apache Flink	$\mathbf{6{,}4 \times 10^{-4}}$	$\mathbf{2{,}0 \times 10^{-33}}$	0,060
Apache Maven	$\mathbf{1{,}7 \times 10^{-11}}$	$\mathbf{3{,}4 \times 10^{-3}}$	0,311
Apache Seatunnel	**0,047**	1,000	1,000
Guava	$\mathbf{2{,}1 \times 10^{-7}}$	$\mathbf{2{,}0 \times 10^{-5}}$	$\mathbf{2{,}8 \times 10^{-6}}$

Table 6: Holm-corrected Wilcoxon signed-rank post-hoc p-values for adjacent effort level comparisons. Statistically significant results ($p < 0,05$) are shown in bold.

Statistically significant differences in energy consumption were primarily observed for the Min/Less and Less/More comparisons. Several projects, including OpenEMS, Spring Framework, Apache Flink, Apache Maven and Guava, exhibited highly significant differences for these pairs. In contrast, most projects did not show a statistically significant increase in energy consumption when moving from More to Max. This suggests diminishing energy costs at the highest effort levels. This pattern was consistent across both Maven and Gradle builds, although statistically significant effects were more frequent and pronounced in Maven-based projects.

In a small number of cases, energy consumption decreased at higher effort levels, and these decreases were sometimes statistically significant, such as for the More/Max comparison in Cruise-control and the Less/More comparison in Apache Maven and Guava. These anomalies indicate that the relationship between effort level and energy consumption is not strictly monotonic and may be influenced by project-specific characteristics or build behaviour.

Collectively, these results indicate that the most consequential energy increases tend to occur when moving from lower to moderate effort levels, while the marginal energy cost of escalating from More to Max is typically limited.

5.2 Performance

To address performance for RQ3, we measured the amount of SpotBugs warnings found per project across effort levels. In most projects, higher effort yields more warnings, consistent with deeper analysis uncovering additional findings. In combination with the higher energy consumption for higher efforts, a trade-off is established between energy consumption and performance.

We divided the warnings by the energy consumption for each project and each effort, to establish which effort is the most efficient. These results can be found in Figure 4. The most efficient effort is Min, which found most warnings per unit of energy for all projects except Spring Framework. For Spring Framework, effort Less was the most efficient.

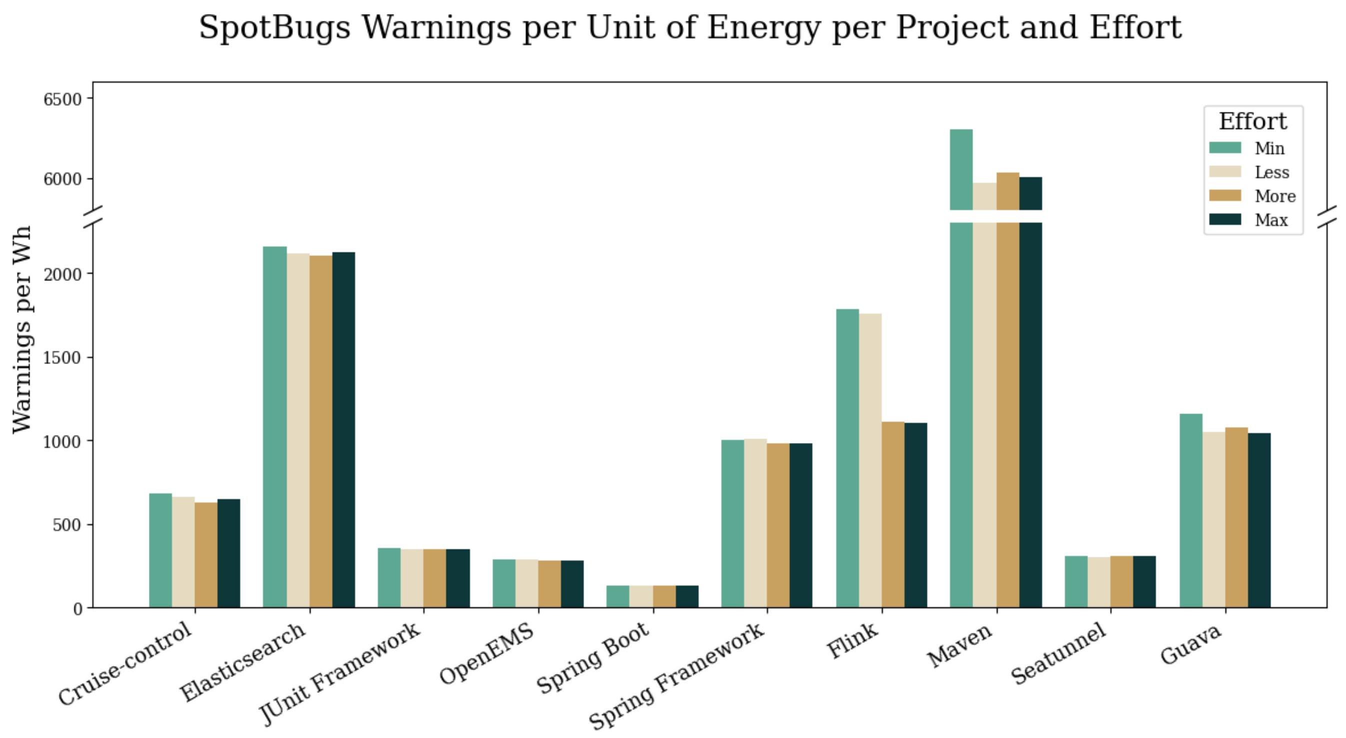

Figure 4: Amount of warnings divided by energy consumption per project per effort

6 Discussion

SpotBugs Most Commonly Mentioned SAT

Whilst SpotBugs was found as the most commonly mentioned tool, it is good to note that it is a tool only meant for Java. Because RQ1 focused exclusively on Java, it is likely that static analysis tools targeting other programming languages are more prominent outside the context of Java. Future work could therefore extend this research by investigating the energy impact of additional static analysis tools across a broader range of programming languages.

True Representation Execution CI Pipeline

In this study, Gradle's incremental build feature was deliberately disabled. However, in real-world CI pipelines, incremental builds can be employed, meaning that the experimental setup does not fully reflect typical CI executions. Future work could investigate the impact of incremental builds on energy consumption, comparing scenarios with and without caching to make CI more sustainable.

Comparison to Previous Work

The amounts of energy consumption found in this research are similar to the amounts found by Zaidman [7]. The energy consumption measured in this research show a rough correspondence with project activity: larger projects with more commits, such as Elasticsearch and Spring Boot, tend to require more energy for both building and static analysis, whereas smaller projects like Cruise-control and JUnit Framework require considerably less.

Energy Consumption in Context

To put the energy consumption as displayed in Table 5 into context, we took the the largest measured energy consumption and multiplied it by the amount of commits per year. The largest energy consumption was the consumption of Elasticsearch with effort More. Together with the amount of yearly Elasticsearch commits [5], we end up with an energy consumption of roughly 485 kWh/year. This represents roughly 32% of average EU household consumption [53].

6.1 Threats to Validity

We used EnergiBridge, a software-based energy measurement, in this study, for its practicality and repeatability. However, software-based measuring is less accurate than direct hardware measurements [54]. Consequently, while the results are suitable for comparative evaluation, the absolute energy values should be interpreted cautiously.

Furthermore, whilst we attempted to minimise temperature fluctuations, the temperature could still have influenced the measurements. The same holds for background tasks.

As EnergiBridge occasionally failed to report the total energy consumption for a task, it was not always possible to obtain 30 valid measurements per ruleset. Additional measurements can be conducted in future work to ensure a consistent set of 30 reliable observations for each ruleset.

7 Responsible Research

In terms of sustainability, this research required a substantial number of SpotBugs executions. Most of them were necessary to ensure both the tool and the automation worked, so only a few were actually conducted as measurements. As an estimate, the total amount of executions combined to an energy consumption of over 8.000 Wh. From a sustainability perspective, this approach is suboptimal, as a smoother implementation would have reduced the number of runs, and consequently, reduced the environmental impact. Additionally, a more in-depth manual inspection of bugs, rather than repeated executions following incremental fixes, could have further reduced the number of runs. However, the insights from this research can contribute to more sustainable software engineering.

The arguments for excluding and including papers in the literature review are clearly stated and decisions were made critically. However, it is still human-decision making, which may differ per person. To ensure the decision making on whether or not to include a paper is less biased, this can be replicated independently by another researcher.

In terms of reproducibility, all relevant configuration and setup details are fully documented. The scripts and code used for the research are publicly available on Zenodo [8], which makes the methodology very reproducible. The collected data is available on Zenodo [8] as well, as to contribute to open science.

8 Conclusions

This study provides initial empirical insights into the energy consumption of static analysis in CI pipelines. The systematic literature review showed that SpotBugs is the most frequently mentioned Java static analysis tool. Furthermore, it offers configurable effort levels suitable for energy measurements. Energy measurements across 10 projects revealed that energy consumption is sensitive to SpotBugs effort in all projects, with almost all higher efforts resulting in higher energy consumptions. The largest energy differences occur at Min/Less and Less/More, whereas the increase More/Max is rarely significant. Performance analysis showed that increased effort improves bug detection in projects, but comes with a higher energy consumption. The most efficient effort is Min, which found the most warnings with regards to energy consumption for 9 out of 10 projects.

These results highlight a clear trade-off between energy consumption and analysis depth in static analysis, emphasising the potential for energy-efficient configurations in CI pipelines. Future work could extend this study to other programming languages, additional static analysis tools, and real-world CI setups employing incremental builds and caching strategies, contributing to more sustainable software engineering practices.

References

[1] E. Soares, G. Sizilio, J. Santos, D. A. Da Costa, and U. Kulesza, "The effects of continuous integration on software development: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 3, p. 78, May 2022. DOI: 10.1007/s10664-021-10114-1.

[2] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the Benefits and Challenges of Continuous Integration Practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, Jul. 2022. DOI: 10.1109/TSE.2021.3064953.

[3] A. Fakokunde and I. Kolawole, "Improving Software Development with Continuous Integration and Deployment for Agile DevOps in Engineering Practices," *International Journal of Computer Applications Technology and Research*, pp. 25–39, 2025. DOI: 10.7753/IJCATR1401.1002.

[4] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden, "Explaining Static Analysis - A Perspective," in *34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, Nov. 2019, pp. 29–32. DOI: 10.1109/ASEW.2019.00023.

[5] *Commits · elastic/elasticsearch*, Jan. 2026. Accessed: Jan. 13, 2026. [Online]. Available: https://github.com/elastic/elasticsearch.

[6] A. Memon et al., "Taming Google-scale continuous testing," in *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, Buenos Aires, May 2017, pp. 233–242. DOI: 10.1109/ICSE-SEIP.2017.16.

[7] A. Zaidman, "An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, Jun. 2024, pp. 214–218. DOI: 10.1145/3644032.3644461.

[8] S. van der Linden, *Sophievdl/brp-energy-sat-configurations: 1.0.0*, Jan. 2026. DOI: 10 . 5281 / zenodo.18368950.

[9] M. Jalili and F. Faghih, "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)*, May 2022, pp. 1–8. DOI: 10.1109/RTEST56034.2022.9850164.

[10] C. Brosch, "Influence of Static Code Analysis on Energy Consumption of Software," in *Lecture Notes in Informatics (LNI), Gesellschaft für Informatik*, 2023, pp. 111–120. Accessed: Jan. 13, 2026. [Online]. Available: https://dl.gi.de/handle/20.500.12116/43330.

[11] M. T. Ailane, C. Rubner, and A. Rausch, "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*, Cham, 2026, pp. 195–211. DOI: 10.1007/978-3-032-04114-2_13.

[12] D. Stefanović, D. Nikolicć, D. Dakicć, I. Spasojevicć, and S. Risticć, "Static Code Analysis Tools: A Systematic Literature Review," in *DAAAM Proceedings*,

2020, pp. 0565–0573. DOI: 10.2507/31st.daaam.proceedings.078.

[13] GitHub Staff, *Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1*, Oct. 2025. Accessed: Dec. 4, 2025. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/.

[14] A. Martín-Martín, E. Orduna-Malea, M. Thelwall, and E. Delgado López-Cózar, "Google Scholar, Web of Science, and Scopus: A systematic comparison of citations in 252 subject categories," *Journal of Informetrics*, vol. 12, no. 4, pp. 1160–1177, Nov. 2018. DOI: 10.1016/j.joi.2018.09.002.

[15] K. Hjerppe, J. Ruohonen, and V. Leppänen, "Annotation-Based Static Analysis for Personal Data Protection," in *Privacy and Identity Management. Data for Better Living: AI and Privacy*, 2020, pp. 343–358. DOI: 10.1007/978-3-030-42504-3_22.

[16] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proceedings of the ACM on Programming Languages*, vol. 4, 2020. DOI: 10.1145/3428205.

[17] A. Dura, C. Reichenbach, and E. Söderberg, "JavaDL: Automatically incrementalizing Java bug pattern detection," *Proceedings of the ACM on Programming Languages*, vol. 5, 2021. DOI: 10.1145/3485542.

[18] A. Groce et al., "Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 207–218. DOI: 10.1109/QRS54544.2021.00032.

[19] J. Hua and H. Wang, "On the effectiveness of deep vulnerability detectors to simple stupid bug detection," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 530–534. DOI: 10.1109/MSR52588.2021.00068.

[20] M. Alqaradaghi and T. Kozsik, "Inferring The Best Static Analysis Tool for Null Pointer Dereference in Java Source Code," in *CEUR Workshop Proceedings*, vol. 3237, 2022. Accessed: Nov. 13, 2025. [Online]. Available: https://ceur-ws.org/Vol-3237/paper-alq.pdf.

[21] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "HBSNIFF: A static analysis tool for Java Hibernate object-relational mapping code smell detection," *Science of Computer Programming*, vol. 217, 2022. DOI: 10.1016/j.scico.2022.102778.

[22] R. Amankwah, J. Chen, H. Song, and P. K. Kudjo, "Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites," *Software - Practice and Experience*, vol. 53, no. 5, pp. 1125–1143, 2023. DOI: 10.1002/spe.3181.

[23] A. Costea, A. Tiwari, S. Chianasta, R. Kishore, A. Roychoudhury, and I. Sergey, "Hippodrome: Data Race Repair Using Static Analysis Summaries," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–33, 2023. DOI: 10.1145/3546942.

[24] P. Gnoyke, S. Schulze, and J. Kruger, "On Developing and Improving Tools for Architecture-Smell Tracking in Java Systems," in *IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023, pp. 248–253. DOI: 10.1109/SCAM59687.2023.00034.

[25] S. Gunawardena, E. Tempero, and K. Blincoe, "Concerns identified in code review: A fine-grained, faceted classification," *Information and Software Technology*, vol. 153, 2023. DOI: 10.1016/j.infsof.2022.107054.

[26] 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, 2023. DOI: 10.1016/j.jss.2022.111575.

[27] A. Trautsch, J. Erbel, S. Herbold, and J. Grabowski, "What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes," *Empirical Software Engineering*, vol. 28, 2023. DOI: 10.1007/s10664-022-10257-9.

[28] J. Yeboah and S. Popoola, "Efficacy of Static Analysis Tools for Software Defect Detection on Open-Source Projects," in *International Conference on Computational Science and Computational Intelligence (CSCI)*, 2023, pp. 1588–1593. DOI: 10.1109/CSCI62032.2023.00262.

[29] S. Motogna, D. Cristea, D.-F. Şotropa, and A.-J. Molnar, "Uncovering Bad Practices in Junior Developer Projects Using Static Analysis and Formal Concept Analysis," in *International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE - Proceedings*, 2024, pp. 752–759. DOI: 10.5220/0012739500003687.

[30] U. Zarić and G. Rakić, "Software Metrics and Multilingual Code: A Preliminary Study Based on Java and C," in *SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*, 2024. Accessed: Nov. 13, 2025. [Online]. Available: https://ceur-ws.org/Vol-3845/paper25.pdf.

[31] A. Nizam, E. Islamoglu, O. Kerem Adali, and M. Aydin, "Optimizing Pre-Trained Code Embeddings with Triplet Loss for Code Smell Detection," *IEEE Access*, vol. 13, pp. 31 335–31 350, 2025. DOI: 10.1109/ACCESS.2025.3542566.

[32] *Spotbugs/spotbugs*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/spotbugs/spotbugs.

[33] *Effort — spotbugs 4.9.8 documentation*, 2025. Accessed: Nov. 18, 2025. [Online]. Available: https://spotbugs.readthedocs.io/en/latest/effort.html#effort.

[34] *Documentation Index — PMD Source Code Analyzer*, 2025. Accessed: Jan. 13, 2026. [Online]. Available: https://pmd.github.io/pmd/.

[35] *Home — Sonar Documentation*, Dec. 2025. Accessed: Jan. 14, 2026. [Online]. Available: https://docs.sonarsource.com.

[36] *SpotBugs manual — spotbugs 4.9.8 documentation*, 2025. Accessed: Nov. 18, 2025. [Online]. Available: https://spotbugs.readthedocs.io/en/latest/index.html.

[37] *Linkedin/cruise-control*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/linkedin/cruise-control.

[38] *Elastic/elasticsearch*, 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/elastic/elasticsearch.

[39] *Junit-team/junit-framework*, 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/junit-team/junit-framework.

[40] *OpenEMS/openems*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/OpenEMS/openems.

[41] *Spring-projects/spring-boot*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/spring-projects/spring-boot.

[42] *Spring-projects/spring-framework*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/spring-projects/spring-framework.

[43] *Apache/flink*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/apache/flink.

[44] *Apache/maven*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/apache/maven.

[45] *Apache/seatunnel*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/apache/seatunnel.

[46] *Google/guava*, Jan. 2026. Accessed: Jan. 24, 2026. [Online]. Available: https://github.com/google/guava.

[47] *Incremental build*, 2025. Accessed: Jan. 18, 2026. [Online]. Available: https://docs.gradle.org/current/userguide/incremental_build.html.

[48] J. Sallou, L. Cruz, and T. Durieux, *Energibridge*, Mar. 2024. Accessed: Nov. 10, 2025. [Online]. Available: https://github.com/tdurieux/EnergiBridge.

[49] J. Sallou, L. Cruz, and T. Durieux, *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*, Dec. 2023. DOI: 10.48550/arXiv.2312.13897.

[50] RobertoN0, *Overflow Error When Subtracting Durations in Energibridge*, Mar. 2025. Accessed: Jan. 12, 2026. [Online]. Available: https://github.com/tdurieux/EnergiBridge/issues/20.

[51] R. Sharma, "A Comparative Study of Wired and Wireless Communication Technologies," in *International Journal of Emerging Technologies and Innovative Research*, vol. 12, May 2025. Accessed: Dec. 10, 2025. [Online]. Available: https://www.jetir.org/view?paper=JETIR2505114.

[52] L. Cruz, C. Brandt, and E. Barba Roque, *3. Scientific Guide for Reliable Energy Experiments*, 2025.

[53] Eurostat, *Electricity and heat statistics*, 2023. Accessed: Jan. 14, 2026. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Electricity_and_heat_statistics.

[54] U. Asgher and T. Malik, "Evaluating Hardware and Software Power Measurement Tools: Assessing Accuracy in Measuring Application Energy Consumption for Data-Parallel Workloads," in *Proceedings of the Fourth International Conference on Innovations in Computing Research (ICR'25)*, 2025, pp. 460–470. DOI: 10.1007/978-3-031-95652-2_39.

A LLM Prompts

This section discusses prompts that were given to Large Language Models for certain tasks during this project.

A.1 Installing EnergiBridge

"Write a ready-to-run PowerShell script that downloads and installs Rust, verifies it, and builds EnergiBridge."

A.2 Added SpotBugs plug-in

The error messages that appeared after attempting to run SpotBugs were given to ChatGPT and possible fixes were implemented.

A.3 Creating Automation Script

"I want to create a Windows batch script that runs energibridge -o gauva_spotbugs.csv –summary guava_spotbugs.bat and then energibridge -o maven_spotbugs.csv –summary flink_spotbugs.bat. I want it to repeat that 5 times. Every time one of the EnergiBridge commands is executed, it leaves a line like Energy consumption in joules: 2667.103759765625 for 46.30371 sec of execution. at the end. I want to store all of the lines for the guava command in guava_summary.csv and for the maven command in maven_summary.csv, with each run adding one line to the csv."

After that, we manually expanded the script. Found bugs were reported to ChatGPT and recommended fixes were implemented after critical assessment.

A.4 Warming Up CPU

"Can you create a line for a batch script that makes sure the CPU is warmed up for five minutes?"

A.5 Plots

Box+violin

"Can you create a Python script for a Jupyter notebook that creates a box+violin plot of the data in the csv column "energy", which is in the file "guava_summary_extracted", which is in the folder spotbugs/effort_min?"

We asked further prompts to make the plots usable for the poster, with a transparent background and different colours, and for this paper. After that, the following prompt was asked to create one figure with four box plots per project:

"I have this ipynb, can you change it such that it creates one figure per project with four box plots in that for each of the efforts?"

Grouped Bar Charts

"Can you now create a bar chart of these dfs? There should be one bar chart, grouped per project, and the four categories of that should be min less more and max."

"Can you change it to have a broken y-axis such that the differences are clearer? See the image for the cutoff points"

"Can you create a bar chart that's the same as the previous ones but divides the warnings by the energy means?"

Statistical Analysis

"Can you give a python script that prints the results of paired one way ANOVA?"

After providing error messages to ChatGPT, it wrote a script for Friedman test with post-hoc analysis.