Measuring Energy Consumption of JUnit tests

Jeroen Janssen*
Delft University of Technology
Delft, Netherlands
J.D.Janssen@student.tudelft.nl

Roelof van der Geest*
Delft University of Technology
Delft, Netherlands
R.R.L.vanderGeest@student.tudelft.nl

Joaquín Cava*
Delft University of Technology
Delft, Netherlands
jcava@tudelft.nl

Elena Ibañez*
Delft University of Technology
Delft, Netherlands
E.Ibanez@student.tudelft.nl

* Equal contribution

***Abstract*—Unit testing is a fundamental part of software development, especially in projects that rely on continuous integration and delivery (CI/CD) pipelines. However, the energy consumption associated with automated test executions has received limited attention. This paper investigates the energy impact of running JUnit test suites across various Java based open-source projects using JoularJX, a Java agent that monitors energy consumption via the RAPL interface.**

Our experimental setup automates the cloning, configuration, and repeated execution of Maven-based projects. Each project's test suite is executed 30 times to calculate average energy usage and filter outliers. The environment includes Java JDK versions 17 and 23, Maven 3.9.x, and Python 3.10+, with all dependencies documented for reproducibility.

We analyzed projects such as JUnit4, SQLLine, Togglz, and Okta Spring Boot. The results showed that while most tests consume very little energy, a small number of them use disproportionately more especially those involving concurrency or complex parsing. We then devise categories for classifying energy intensive tests. This study highlights the potential for integrating sustainability into software testing practices and opens the door to more sustainable development workflows.

***Index Terms*—Power Monitoring, Software Testing, Energy Analysis, Sustainable development**

I. Introduction

In recent years, environmental sustainability has become an increasingly critical topic within the field of software engineering. The Information and Communication Technology (ICT) sector, driven by the exponential growth of software systems and digital services, has witnessed significant increases in energy consumption. Current estimates suggest that ICT may be responsible for up to 20% of global electricity consumption by 2025 [1]. Consequently, the software engineering community is being urged to incorporate sustainability considerations explicitly into development practices, from design through deployment.

While extensive research has been conducted on optimizing energy consumption during software runtime, the energy overhead associated with software testing, particularly unit testing, has received comparatively little attention. However, unit testing frameworks, such as JUnit, have become essential components in the software development lifecycle, frequently integrated into continuous integration and continuous delivery (CI/CD) pipelines. As these frameworks execute test suites repeatedly, the cumulative energy consumption from frequent test runs can significantly contribute to a project's overall energy footprint. For instance, [1] demonstrated that the Elasticsearch project alone, through over 5000 builds executed in a single year, consumed approximately 161.5 kWh—equivalent to nearly 10% of an average European household's annual electricity usage.

This research gap motivates the current study, aiming to rigorously quantify the energy consumption associated with executing JUnit tests in diverse Java-based open-source software projects. By analyzing a representative selection of projects—including JUnit4, Togglz, SQLLine, and Okta Spring Boot to name a few, all the projects used in this paper were taken from the Github repository of [2]. This research seeks to systematically identify the energy impact of automated testing procedures and highlight opportunities for enhancing sustainability practices within software development.

To obtain precise, method-level energy consumption data, this study utilizes JoularJX [3], a Java agent specifically designed to monitor energy consumption transparently within the Java Virtual Machine (JVM). JoularJX leverages the Running Average Power Limit (RAPL) interface, available on modern Intel and AMD processors, to collect detailed energy consumption data in real-time at granular code levels without requiring manual code instrumentation [4] [5]. This methodological choice is justified by JoularJX's ability to transparently integrate into existing software projects, providing detailed insights into which test methods and underlying software components constitute significant energy "hotspots."

Although external hardware measurements offer high accuracy, they typically lack the granularity necessary to attribute energy consumption to specific methods or tests [6], that is why this measurement was discarded from the project. Conversely, model-based estimation approaches, which rely on proxy metrics such as CPU usage or system calls, often sacrifice accuracy [7]. Consequently, JoularJX represents an optimal balance between ease of use, measurement granularity,

and precision [6] [4] [5].

The empirical component of our research involves automating the execution of selected open-source projects under controlled conditions. Our experimental pipeline clones each repository, executes their JUnit test suites multiple times using Maven, and captures energy consumption through JoularJX. By repeatedly executing test suites under controlled conditions, we aim to achieve statistically significant results, reducing variability inherent in energy measurements and thereby ensuring robust conclusions.

The overarching goal of this study is to provide empirical insights into the energy consumption patterns of unit testing practices, thus contributing to the broader objective of sustainable software engineering. Findings from this study could inform the development of energy-aware software testing methodologies, highlight potential areas for optimization in test code design, and suggest best practices for energy-efficient CI/CD pipelines.

The remainder of this paper is structured as follows. First, we review related work in sustainable software engineering and existing power measurement methodologies. We then present our research methodology, detailing project selection criteria, experimental setup, and data collection procedures. Subsequently, we discuss the experimental findings, analyze their implications, and suggest practical recommendations for energy-efficient unit testing practices. Finally, we conclude by identifying directions for future research to further advance sustainable practices within software engineering.

II. Problem Statement and Motivation

Software testing is a key part of development, especially in projects that use continuous integration and delivery (CI/CD). These tests are run automatically and while each test run uses a small amount of energy, the total impact can be large over time. Most research on energy efficiency in software has focused on improving how software runs. But much less attention has been given to testing, even though it's repeated so often. We think this is a missed opportunity. Our team was interested in understanding how much energy is used during unit testing and whether some tests or projects use more energy than others. By measuring and comparing this, we hope to understand and make our future JUnit tests more energy efficient and support more sustainable software practices.

A. Research Questions

To investigate the energy consumption of unit testing in Java-based open-source projects, we define twp research questions. These aim to explore how much energy is consumed during test execution, a routine part of CI/CD pipelines that is often overlooked in sustainability discussions. Together, these questions provide an overview of energy usage in automated unit testing:

- How does the energy consumption results differ across multiple test executions for the same project and test suite?
- What are the observed energy consumption patterns of JUnit test executions across different Java-based open-source projects?

III. Background and Related Work

Recent research has increasingly focused on the environmental impact of software development practices, as part of the broader push toward sustainable software engineering. For example, [8] analyzed the energy consumption of Continuous Integration (CI) pipelines in Java projects and revealed that frequent triggers, such as pull requests, pushes, and scheduled builds—can lead to substantial cumulative energy usage. Their study found that a single project can consume, on average, 22 kWh, equating to approximately 10.5 kg of CO2 emissions, or the emissions from driving 100 kilometers in a typical European car [8]. While their work does not focus on unit testing specifically, it underscores the need to examine common development activities through a sustainability lens.

More closely related to unit testing,[7] explored the energy consumption of test generation and execution using EvoSuite. They employed JoularJX to measure the energy used by both automatically and manually created test cases. Their findings show that test generation is more energy-intensive than test execution, and that automatically generated tests tend to be more energy-efficient despite being potentially shorter or less complex [7]. The energy monitoring tool JoularJX and PowerJoular has been introduced to support green software development by providing cross-platform power consumption insights [3]. JoularJX helps developers see which Java methods use the most energy when running. The study shows that energy consumption changes more between platforms than between programming languages. These tools make it easier to build software that works across platforms and uses less energy. Similarly, [9] investigated whether theoretical time complexity can predict the energy consumption of sorting algorithms implemented in Java. Their study revealed strong correlations between time complexity, wall-clock time, and energy usage, suggesting that algorithmic complexity can serve as a reliable proxy for estimating energy consumption in Java-based sequential programs [9].

Despite growing research on energy consumption in software systems, energy efficiency is still rarely prioritized in daily development practices. [10] point out that software energy consumption is often overlooked in the daily work of software engineers due to lack of time and resources. The authors argue that companies play a key role in changing this, by setting clear objectives and KPIs around software energy consumption (SEC), allocating time for developers to address it, and providing the tools and support needed to do so [10]. While these studies give helpful insights into sustainable software engineering, there is still little research on the energy use of everyday unit testing, especially in real-world projects using tools like JUnit. In this project we aim to explore how improvements can be made to measure the energy consumption of JUnit tests.

To identify an appropriate tool that aligns with the requirements of our project, we conducted a comprehensive evaluation and compared two prominent solutions: JalenUnit [11] and JoularJX. Although JalenUnit provides robust capabilities for generating detailed energy consumption models, its reliance on elaborate configuration procedures and extensive benchmarking processes reduces its suitability for agile development environments and rapid iteration cycles. Conversely, JoularJX emerged as a preferable choice due to its seamless integration with standard software development workflows, particularly through its extension of the widely-used JUnit testing framework.

Next to these two prominent solutions, we note one last briefly explored option. It is also possible to run a single test with a maven command. The energy usage of this process can then be measured using a tool like EnergiBridge [12]: `energibridge mvn -Dtest=TestClass#testName test`. This can be repeated for every test in every project to obtain the required results. This approach was not chosen for the following reasons:

- As the name implies, unit tests evaluate small, individual units of a Java program. This means that a unit test may finish in a couple milliseconds (or less). This is also confirmed by our results, that show that the vast majority of unit tests consume very little energy. However, when measuring the entire process using a tool such as EnergiBridge, the energy consumed during starting and stopping of the JVM is also measured. Starting the JVM without doing anything varies in energy consumption every time, but is already close to 100J. When a unit test only consumes a few joules at most, the energy usage of the unit test gets lost as noise in the measurements. We therefore lose the required level of granularity in the measurements.
- Starting and stopping a JVM for every test adds a lot of extra overhead. On our machine, starting a JVM takes a couple seconds. Because some projects have thousands of tests, running a single project 30 times would already take over a week. Because JoularJX is integrated directly, we can instead run a project with thousands of tests 30 times in just a few minutes.

IV. Experiment Setup

In this section, we describe our experimental setup, where the goal is to measure the energy usage of individual JUnit tests in various open-source Java projects. To measure energy consumption, we use JoularJX, a Java agent. We explain how JoularJX works and how we integrated it into the open-source Java projects in section IV-A.

To test the energy usage of various Maven projects, we developed a Python script. This script builds each project and runs its tests 30 times to compute average energy usage and facilitate outlier removal during analysis. We provide a replication package containing all source code and projects used for these measurements on GitHub.

In the sections below, we outline the general workings of our measurement script and explain how JoularJX is integrated into our measurements. Additionally, we describe the information generated using JoularJX and specify the dependencies used, along with their versions, to ensure the reproducibility of our measurements.

A. JoularJX

This project requires the ability to measure the energy consumption of a Java project at method-level. Because of this granularity requirement, JoularJX was selected as the best measuring tool for this purpose. JoularJX is a Java agent, meaning it can modify Java bytecode at runtime. It leverages this capability to insert energy-measuring bytecode into the program. Together with filtering capabilities offered by JoularJX, we are able to filter specifically for the test methods in Java projects, allowing us to measure the energy consumption of individual test cases in various open-source Java projects.

B. Python Script

The Python code can be divided into three stages: the setup phase, measuring phase, and the evaluation phase, as indicated in figure 1. We also briefly explain each section below:

1) Setup phase: In the setup phase, we start off by cloning JoularJX from its GitHub repository and building it on the machine used for the experiment. We then clone all projects selected for measuring and modify their `pom.xml` files to insert JoularJX. After that, we parse all Java files in the project to find all test methods, so that we can filter those for energy consumption measurement with JoularJX. Finally, the whole project is built and the tests are run once to confirm the operations were successful.

2) Measuring phase: During the measuring phase, all projects are ready to be measured for energy consumption. Before every project, the machine used for measuring is warmed up by performing some Fibonnaci sequence calculations to keep the temperature of the machine consistent throughout the experiment. After each test run, the machine is left to rest for half a minute, allowing background or cleanup processes to finish before the next run.

3) Evaluation phase: The evaluation phase is the final phase of the experiment. In this phase, all measurement from the 30 separate runs are aggregated. We then remove outliers and compute or generate the relevant metrics, statistical tests, graphs and tables. These can be found in appendix A.

C. Dependencies

The Python script developed for this experiment does not require many dependencies. The required dependencies are listed in ‘requirements.txt‘. During this research, Python 3.10 was used, but other versions may also be supported.

Since we measure Java projects the environment needs to have a JavaJDK installed. Furthermore, since the projects being tested use Maven, the environment also needs to include Maven version 3.9 or newer. We have run with Java versions

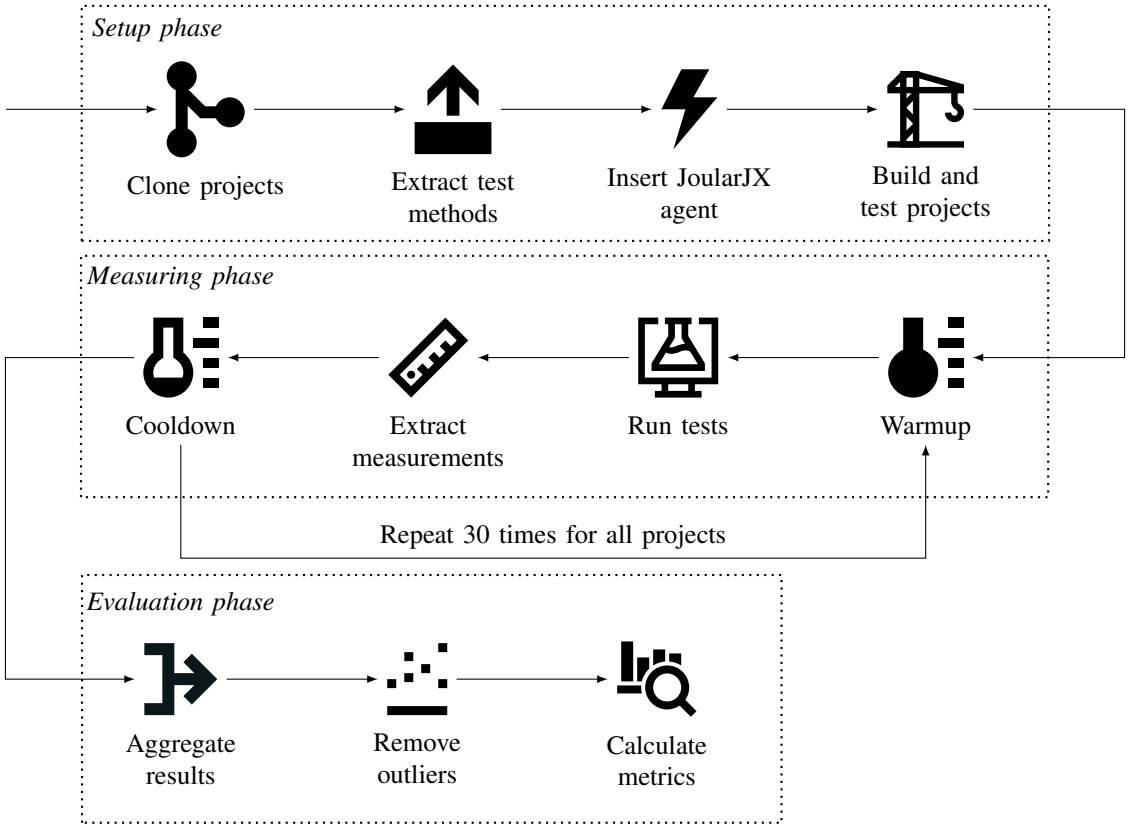

Fig. 1: Pipeline used for this project

17 and 23. For each tested project we also noted down the current git commit for which we ran the tests. As the Java version requirements for the projects being tested may be changed by their respective authors, running newer versions of the projects under test may require different Java versions.

V. Results

We measured the energy consumption of unit tests across seven open-source Java projects using JoularJX. To reduce variability and capture consistent patterns, each test was run 30 times. The boxplots show significant differences in energy usage both between and within projects. In some cases, a few individual tests consumed much more energy than others, highlighting potential areas where test design could be optimized. The plots of energy usage can be found in appendix A, but we highlight the most important parts below.

- **Taj4**: In the ta4j project, the tests `randomWalkIndexHigh` and `sqrtLudicrousPrecision` use the most energy. The test randomWalkIndexHigh uses about 14 to 15 joules on average. It loads a full time series from an Excel file and calculates values using indicators like RWIHighIndicator or RWILowIndicator. Then it checks if the results are correct using a helper method called `assertIndicatorEquals`, which compares every value in the series one by one which explains its high energy use. Moreover, the test `sqrtLudicrousPrecision` also uses a lot of energy around 12 to 13 joules. It works with very large numbers and tries to calculate their square root using 100,000 digits. The test also runs several checks to make sure the result is accurate, which adds even more processing. That explains why this test uses so much energy. Both tests are heavy because they either process a lot of data or do complex math. On the other hand, the remaining methods stay below 5 joules, and many use less than 1 joule. This means that a small number of methods account for the majority of the energy usage in this project.
- **SQLLine**: The test that uses the most is `testSchemaTableColumnCompletions`, which uses around 8.5 joules. This test checks if the command-line suggestions work properly. It sets up a fake line reader, adds a completer, runs a command to refresh suggestions, and then checks if the output matches what's expected. Because it's a parameterized test, it runs several times with different inputs. Each run processes commands and suggestions, which takes time and resources—this explains why it uses more energy than most other tests. The remaining test methods stay below 2 joules, indicating that the majority of energy consumption is concentrated in just a few methods.
- **JUnit 4**: In the JUnit4 project, most tests use very

little energy less than 0.1 joules. However, testCountsStandUpToFiltration uses the most energy by far, averaging around 0.2 joules. Overall, the JUnit4 test suite is very lightweight in terms of energy use.

- **Java API for Github**: In the github-api project, the test that uses the most energy is `testGetDeployKeys`, with an average of around 4.5 joules. This test connects to a repository and loads a list of deploy keys, then performs several detailed checks on each key. It filters through the keys using streams, checks creation dates, who added the keys, their access rights, and other properties. These operations involve multiple object calls, date conversions, and string comparisons. Since it processes and verifies many fields for two keys, the test ends up doing more work than others, which explains why it consumes the most energy. The remaining tests in the GitHub API project consume on average between 0.3 and 1 joules indicating relatively low energy usage compared to the top-consuming test.
- **Cucumber**: One method stands out in terms of energy use: `DeserializeEncodedDataReturnsEmbedding`, which consumes the most energy, averaging around 2.8 to 3 joules. A couple of other tests, like `getFileName_ReturnsFileName`, use a small amount of energy around 0.2 joules, while the remaining tests consume nearly none close to 0 joules.
- **eo-yaml**: The test `complainsOnEmptyKey` uses a mocked BaseYamlNode object and checks whether the RtYamlMappingBuilder correctly throws an IllegalArgumentException when an empty key is added. It consumed around 4 joules on averagy significantly more than the rest of the tests in this project probably because of the mock interactions, and exception handling logic inside the RtYamlMappingBuilder and `build()` method. The remaining tests consume much less energy, usually between 0.2 and 0.6 joules, suggesting that they perform lighter operations
- **Cactoos**: For the project cactoos The method `correctValuesForConcurrentNextNext` consumes a lot of energy not because the unit test or the Synced class is inherently energy-intensive, but due to the way the test is structured. Specifically, it runs a loop 5000 times and executes assertions in two concurrent threads using RunsInThreads. This results in a total of 10,000 thread executions, each involving thread synchronization and assertions. The high number of iterations combined with the repeated creation and execution of threads leads to significant CPU usage and energy consumption. Therefore, the energy cost is primarily due to the large scale of concurrent operations in the test, not the complexity of the logic being tested. The remaining tests consume between 0.1 and 0.6 joules on average, with most of them staying below 0.3 joules, showing that they are relatively low in terms of energy consumption.
- **testing 2**: In the testing-2 project, the test `calculateTest` consumes a lot of energy. This is because it runs a custom method called `doStuff`, which calculates a large Fibonacci number using BigInteger. Inside doStuff, there's a loop that repeats the entire Fibonacci calculation five times. Each time, it runs a loop up to 200,000, performing heavy BigInteger math and adding results to a list. Since these calculations are large and repeated multiple times, the test becomes very CPU- and memory-intensive. That's why `calculateTest` shows high energy use -around 7 joules on average, with some outliers reaching above 13 joules.

VI. Discussion

Our analysis of energy consumption across seven open-source Java projects using JUnit tests revealed significant variation in energy usage, both within and between projects. These results point to opportunities for improving test design and execution efficiency, especially in the context of continuous integration pipelines where tests are run frequently.

A. Key observations

From all the projects we looked at, the tests that used the most energy usually fell into two groups: ones that handled a lot of data or did really detailed calculations, and ones that ran many times or used multiple threads. For example, in ta4j and testing-2, the tests worked with long time series or big numbers, which made them heavy. In cactoos, the test wasn't doing anything too complex, but it ran 5000 times with multiple threads, which added up and used a lot of energy. In SQLLine, the test ran several times with different inputs, which also made it more energy-consuming. These examples show that high energy use in tests doesn't always come from difficult tasks—it can also come from repeating tasks too many times or using lots of threads. Understanding this can help developers improve their tests by making them simpler or running them fewer times.

From the results, we attempt to define categories of tests that tend to use a lot of energy. We list the categories that we were able to define below.

- **God Tests**; we define this category similarly to the 'God Classes' or 'God Functions' in normal code. They are tests that do too much. For instance, have enormous for-loops or access many classes/functions to test this unit.
- **Data-Intensive Tests**; we define this category as tests that load in significantly large datasets to test a unit.
- **Mocking Tests**; this category simply defines all the tests that make use of mocking (or stubbing) of classes/functions.
- **Side-Effect Tests**; defined as the category where a unit test has side effects. Such as downloading external resources or making real api-calls.

B. Limitations

Our study contains a number of significant limitations that impact the interpretation and generalizability of our findings, despite the fact that it produced insightful information. To begin with, the measuring instrument we employed, JoularJX, does not offer a thorough, cumulative view of system-wide energy usage; instead, it depends on the RAPL interface for CPU-level information. Disk, network, and GPU components are still mostly ignored, thus any energy use above and beyond CPU operations is only partially recorded. Additionally, JoularJX adds method-level instrumentation overhead, which could distort reported CPU utilization or impact execution durations, altering the measured energy profiles in ways that aren't indicative of sheer application overhead.

Second, the external validity of our tests was limited because we only used one hardware configuration. Power management technologies like frequency scaling and thermal throttling are used by contemporary CPUs, and these can vary widely between different architectures. Warm-up and cool-down times helped to reduce some of these effects, but resource contention or background operating system activities can still introduce measurement noise. This highlights the hardware reliance of our results, as comparing the energy consumption of JUnit tests across machines with various CPU types, clock speeds, or memory configurations may produce distinct patterns.

Thirdly, a significant portion of our results does not fit a normal distribution, suggesting that there is skewness and heavy tails in the data. This is most likely due to the abnormally high resource usage of a small sample of tests. This unequal distribution complicates statistical analysis, particularly if standard parametric approaches presume normality. Furthermore, because JoularJX evaluates CPU-oriented metrics rather than whole system-level energy, strong CPU-bound workloads may occasionally have a disproportionate impact on these hefty tails. Therefore, without strong outlier and distribution research, as we did, interpreting or comparing average energy use across projects might be deceptive.

Finally, although our open-source Java project sample sheds light on unit-test energy consumption trends, it is still a small sample. The lack of real-world CI / CD conditions, where tests can run on virtualized or containerized environments, also limits the generalizability of these findings.

C. Future Work

While this paper gives a broad view of how much energy is used when running JUnit tests, there's still for improvements and deeper understanding. Firstly, JoularJX relies on the RAPL interface, which mostly tracks the CPU's energy use but it doesn't cover other parts of the system like the disk, network, or GPU. Future research could look into combining JoularJX with other tools to get a more complete view of the energy consumption during test execution. Second, expanding the range of analyzed projects would strengthen the generalizability of our findings. Our experiment focused on a small set of Java-based, Maven-managed open-source projects. Including projects that uses Python or JavaScript and so on or using different testing frameworks could reveal whether similar patterns hold across languages and environments. Third, it would be interesting to investigate the relationship between test design and energy usage in more depth. For example, analyzing which code patterns, library calls, or dependencies tend to increase energy consumption. This could help developers to write more efficient tests from the start. Overall, this paper highlights that measuring test energy use could lead to more awareness in terms of sustainable development practices in everyday lives.

D. Answers to Research Questions

With the presented results and the discussion points above, we can answer our research questions.

- *RQ1, How does the energy consumption results differ across multiple test executions for the same project and test suite?* We found that there was a spread in energy usage through the different test runs. For some tests, this meant that the energy usage distribution was not normal, given the 30 runs. However, for most tests the spread was within 1 Joule meaning that one can still gain a good understanding of which tests are energy intensive.
- *What are the observed energy consumption patterns of JUnit test executions across different Java-based open-source projects?* We were able to define a few categories of tests that are a strong indicator that a test will be energy intensive. These were the God Tests, Data-Intensive Tests, Mocking Tests and Side-Effect tests.

VII. Conclusion

This paper explored the energy consumption of JUnit test executions across several Java-based open-source projects. Using JoularJX, we measured the energy usage of individual test cases and identified patterns in how different types of tests consume energy. Our results show that while most tests use very little energy, a few can be significantly more demanding especially those involving concurrency or complex tasks.By identifying and optimizing high-consumption JUnit tests in terms of energy consumption, developers can make a difference in contributing to sustainability. While we focused on a small set of Java projects, the approach we used could work just as well on other kinds of projects and tools. As the field of sustainable software engineering grows, we hope this paper encourages further research into how everyday development practices like unit testing can be improved in terms of energy consumption.

References

[1] 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)*, ser. AST '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 214–218. [Online]. Available: https://doi.org/10.1145/3644032.3644461

[2] A. Khatami, "sop-in-qa-replication-package," https://github.com/akhatami/sop-in-qa-replication-package, 2022.
[3] A. Noureddine, "Powerjoular and joularjx: Multi-platform software power monitoring tools," in *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, Jun 2022.
[4] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3177754
[5] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," *Automated Software Engg.*, vol. 22, no. 3, p. 291–332, Sep. 2015. [Online]. Available: https://doi.org/10.1007/s10515-014-0171-1
[6] A. Katsenou, X. Wang, D. Schien, and D. Bull, "Comparative study of hardware and software power measurements in video compression," in *2024 Picture Coding Symposium (PCS)*, 2024, pp. 1–5.
[7] F. Kifetew, D. Prandi, and A. Susi, "On the energy consumption of test generation," 2025. [Online]. Available: https://arxiv.org/abs/2501.09657
[8] Q. Perez, R. Lefeuvre, T. Degueule, O. Barais, and B. Combemale, "Software frugality in an accelerating world: the case of continuous integration," 2024. [Online]. Available: https://arxiv.org/abs/2410.15816
[9] K. Carter, S. M. G. Ho, M. M. A. Larsen, M. Sundman, and M. H. Kirkeby, "Energy and time complexity for sorting algorithms in java," 2024. [Online]. Available: https://arxiv.org/abs/2311.07298
[10] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "On reducing the energy consumption of software: From hurdles to requirements," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3382494.3410678
[11] A. Noureddine, R. Rouvoy, and L. Seinturier, "Unit testing of energy consumption of software libraries," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1200–1205. [Online]. Available: https://doi.org/10.1145/2554850.2554932
[12] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," 2023. [Online]. Available: https://arxiv.org/abs/2312.13897

APPENDIX

A. Results from test runs

In this appendix we provide the boxplots showing the energy consumption across multiple test runs for all projects. We also provide the p-value for the Shapiro-Wilk test for normality. Because most project include hundreds or thousands of tests, we only show the 15 most energy consuming tests. The vast majority of tests barely use any energy.

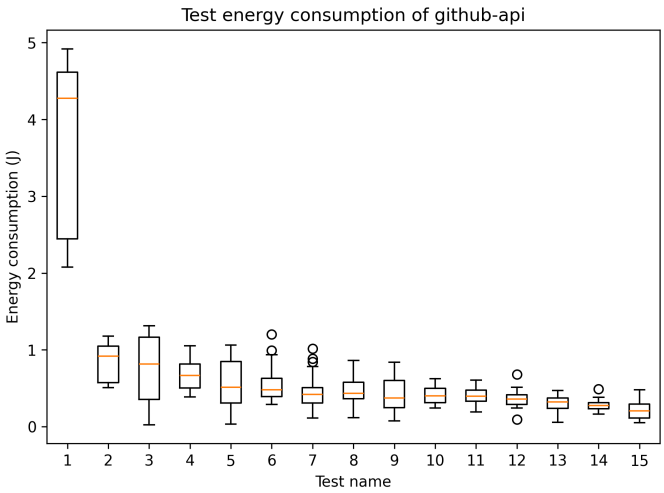

Fig. 2: Energy consumption for github-api test suite. Corresponding tests can be found in table I

Index	Test name	Mean J	Std. dev.	p-value
1	GHDeployKeyTest.testGetDeployKeys	3.6754	1.084	$4.45e-05$
2	GHRepositoryForkBuilderTest.testForkDefaultBranchOnly	0.8428	0.245	$7.41e-04$
3	AppTest.testOrgRepositories	0.7429	0.445	$1.73e-02$
4	AppTest.notifications	0.6758	0.188	$2.41e-01$
5	AppTest.testUserPublicEventApi	0.5744	0.286	$6.77e-02$
6	DefaultGitHubConnectorTest.testCreate	0.5555	0.23	$2.51e-03$
7	GHWorkflowTest.testDispatch	0.4655	0.211	$1.36e-02$
8	GHRepositoryTest.getCommitsBetweenPaged	0.4605	0.158	$9.50e-01$
9	AppTest.testMyTeamsShouldIncludeMyself	0.4229	0.212	$2.76e-01$
10	AppTest.testGetIssues	0.4114	0.111	$1.35e-01$
11	AppTest.testCommitSearch	0.399	0.099	$9.41e-01$
12	AppTest.testIssueSearch	0.3583	0.109	$2.36e-01$
13	AppTest.testListIssues	0.2993	0.111	$3.41e-01$
14	AppTest.testEventApi	0.2791	0.069	$2.67e-01$
15	AppTest.testGetDeploymentStatuses	0.2175	0.12	$2.74e-01$

TABLE I: Detailed energy usage for github-api

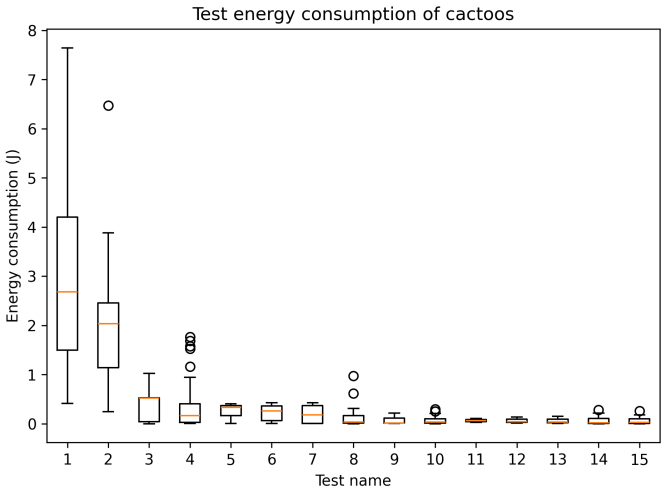

Fig. 3: Energy consumption for cactoos test suite. Corresponding tests can be found in table II

Index	Test name	Mean J	Std. dev.	p-value
1	SyncedTest.correctValuesForConcurrentNextNext	3.0017	1.909	$1.74e-01$
2	SyncedTest.correctValuesForConcurrentNextHasNext	2.0682	1.392	$2.03e-02$
3	RetryTest.runsScalarMultipleTimes	0.4215	0.374	$4.21e-01$
4	SolidTest.worksInThreads	0.3613	0.474	$5.74e-08$
5	ShuffledTest.shuffleIterable	0.2463	0.176	$3.30e-01$
6	SolidTest.cachesScalarResults	0.2222	0.165	$3.14e-01$
7	SolidBiFuncTest.testThatFuncResultCacheIsLimited	0.1968	0.196	$1.11e-01$
8	InputOfTest.readsRealUrl	0.1255	0.204	$1.20e-07$
9	BytesOfTest.readsInputIntoBytesWithSmallBuffer	0.0767	0.096	$3.24e-02$
10	SyncedTest.worksInThreads	0.0664	0.069	$2.05e-08$
11	SlicedTest.sliceTheHead	0.0649	0.032	$8.33e-01$
12	PagedTest.throwsNoSuchElement	0.0639	0.053	$4.46e-01$
13	ReaderOfTest.readsCharArrayWithCharsetByName	0.0637	0.061	$3.87e-01$
14	TempFolderTest.deletesNonEmptyDirectory	0.0633	0.081	$2.57e-04$
15	SolidBiFuncTest.testThatFuncIsSynchronized	0.0581	0.07	$4.60e-04$

TABLE II: Detailed energy usage for cactoos

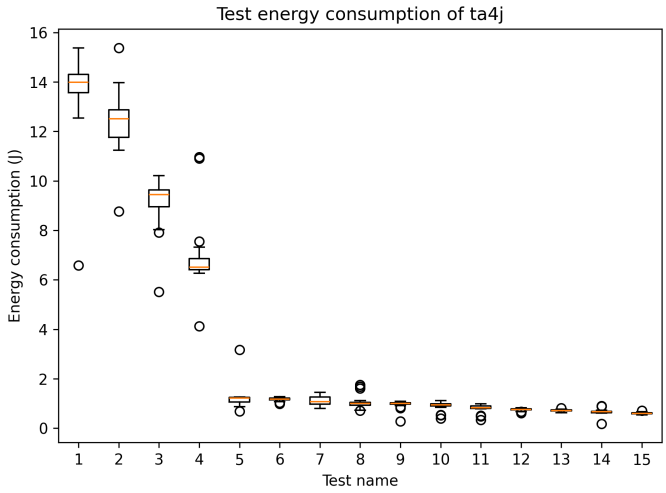

Fig. 4: Energy consumption for ta4j test suite. Corresponding tests can be found in table III

Index	Test name	Mean J	Std. dev.	p-value
1	RWIHighIndicatorTest.randomWalkIndexHigh	13.7994	1.501	$8.95e-08$
2	NumTest.sqrtLudicrousPrecision	12.3894	1.137	$5.07e-02$
3	RWILowIndicatorTest.randomWalkIndexHigh	9.1645	0.897	$1.15e-05$
4	CashFlowTest.reallyLongCashFlow	6.8357	1.231	$1.89e-07$
5	DecimalNumTest.decimalNumTest	1.3508	0.717	$9.62e-04$
6	RSIIndicatorTest.xlsTest	1.171	0.072	$5.26e-02$
7	LSMAIndicatorTest.lsmaIndicatorTest1	1.1067	0.197	$1.12e-01$
8	MMAIndicatorTest.testAgainstExternalData	1.037	0.239	$1.52e-05$
9	ADXIndicatorTest.externalData	0.9695	0.14	$1.08e-08$
10	LowestValueIndicatorTest.onlyNaNValues	0.9078	0.155	$1.08e-05$
11	HighestValueIndicatorTest.onlyNaNValues	0.8199	0.137	$4.78e-06$
12	ATRIndicatorTest.testXls	0.7571	0.047	$6.95e-02$
13	SMAIndicatorTest.externalData	0.7161	0.037	$6.63e-01$
14	MinusDIIndicatorTest.xlsTest	0.667	0.11	$2.75e-07$
15	EMAIndicatorTest.externalData	0.6069	0.042	$2.91e-01$

TABLE III: Detailed energy usage for ta4j

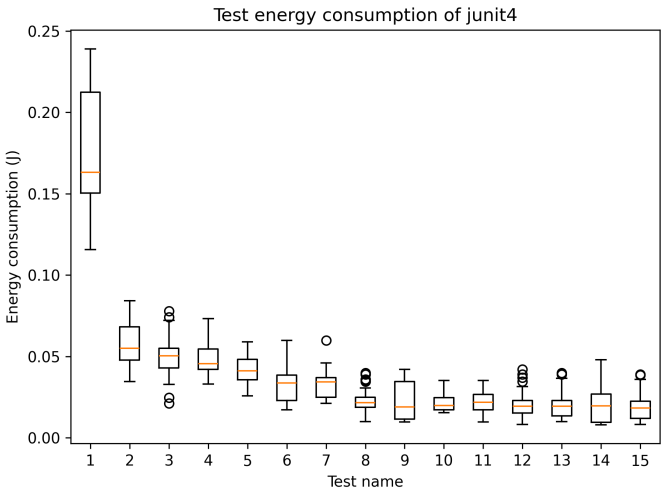

Fig. 5: Energy consumption for junit4 test suite. Corresponding tests can be found in table IV

Index	Test name	Mean J	Std. dev.	p-value
1	MaxStarterTest.testCountsStandUpToFiltration	0.1764	0.035	$3.11e-02$
2	MaxStarterTest.rememberOldRuns	0.0571	0.013	$4.92e-01$
3	CategoriesAndParameterizedTest.runsTestMethodWithCategory	0.0495	0.013	$6.86e-01$
4	JUnit38SortingTest.preferRecentlyFailed38Test	0.0479	0.009	$2.53e-01$
5	TempFolderRuleTest.randomSubFoldersAreDeleted	0.0425	0.009	$8.63e-01$
6	FailOnTimeoutTest.throwTimeoutExceptionOnSecondCallAlthoughFirstCallThrowsException	0.034	0.012	$7.61e-02$
7	FailingDataPointMethods.shouldIgnoreSingleDataPointMethodExceptionsOnRequest	0.0332	0.009	$3.06e-02$
8	ArrayComparisonFailureTest.classShouldAccept411Version	0.0233	0.008	$2.11e-02$
9	FailOnTimeoutTest.statementThatCanBeInterruptedIsStoppedAfterTimeout	0.0223	0.011	$4.93e-03$
10	ExpectedExceptionTest.runTestAndVerifyResult	0.022	0.006	$2.20e-02$
11	ErrorCollectorTest.runTestClassAndVerifyEvents	0.022	0.007	$8.14e-01$
12	MaxStarterTest.preferFast	0.021	0.009	$1.48e-02$
13	SpecificDataPointsSupplierTest.shouldReturnNothingIfTheNamedDataPointsAreMissing	0.0207	0.009	$5.18e-03$
14	TemporaryFolderUsageTest.tearDown	0.0203	0.011	$5.12e-02$
15	FailOnTimeoutTest.throwsTestTimedOutExceptionWithMeaningfulMessage	0.0201	0.009	$1.27e-02$

TABLE IV: Detailed energy usage for junit4

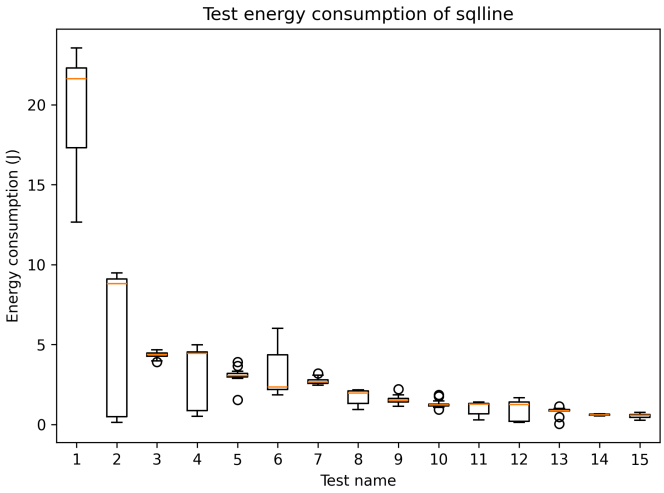

Fig. 6: Energy consumption for sqlline test suite. Corresponding tests can be found in table V

Index	Test name	Mean J	Std. dev.	p-value
1	SqlLineArgsTest.begin	19.5945	3.873	$2.12e-05$
2	CompletionTest.getDummyLineReader	6.3111	3.976	$3.34e-07$
3	CompletionTest.testSchemaTableColumnCompletions	4.3537	0.194	$3.38e-01$
4	CompletionTest.testSchemaTableColumnCompletionsForSquareBracketsDialect	3.4142	1.722	$8.07e-07$
5	SqlLineHighlighterTest.testHighlightWithException	3.0691	0.36	$3.18e-06$
6	SqlLineArgsTest.testOutputWithFailingColumnDisplaySize	3.0246	1.226	$1.43e-05$
7	CompletionTest.testSchemaTableColumnCompletionsForMySqlDialect	2.7121	0.195	$1.95e-01$
8	BufferedRowsTest.nextListTest	1.7327	0.462	$2.09e-05$
9	SqlLineParserTest.testSqlLineParserForWrongLinesWithEmptyPrompt	1.5206	0.215	$1.20e-01$
10	SqlLineParserTest.testSqlLineParserWithException	1.2417	0.18	$6.50e-05$
11	CompletionTest.testReadAsDialectSpecificName	1.0862	0.326	$1.32e-05$
12	CompletionTest.getLineReaderCompletedList	0.9744	0.548	$4.13e-05$
13	PromptTest.testPromptWithSchema	0.7944	0.285	$4.44e-03$
14	CompletionTest.testGetSchemaTableColumn	0.6079	0.039	$1.18e-01$
15	SqlLineArgsTest.init	0.5379	0.132	$2.71e-01$

TABLE V: Detailed energy usage for sqlline

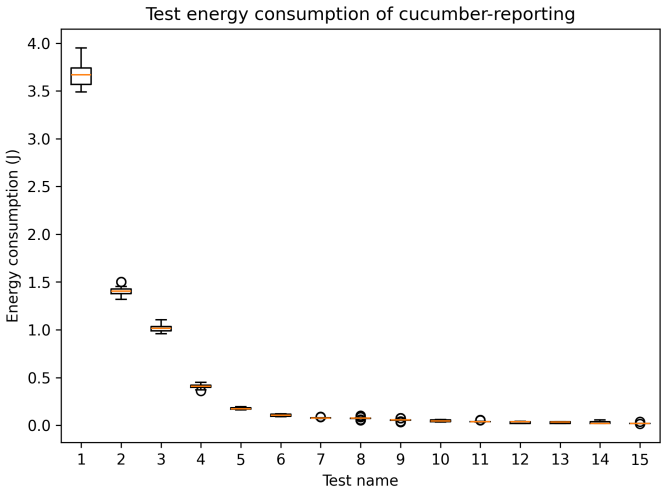

Fig. 7: Energy consumption for cucumber-reporting test suite. Corresponding tests can be found in table VI

Index	Test name	Mean J	Std. dev.	p-value
1	EmbeddingDeserializerTest.deserialize_OnEncodedData_returnsEmbeddingWithEncodedData	3.6669	0.111	$4.30e-01$
2	ReportResultMergeTest.checkPartTwoIsValidReport	1.4042	0.045	$5.64e-01$
3	EmbeddingWithNameTest.getFileName_ReturnsFileName	1.0172	0.035	$3.82e-01$
4	ReportResultMergeTest.unsupportedReportFormat	0.4084	0.019	$9.25e-01$
5	ResultTest.getFormattedDuration_ReturnsDurationAsString	0.1766	0.009	$2.14e-01$
6	ReportResultMergeTest.merge_PartOneTwo_WithPassedRerun_Equals_AllInOnePassed	0.105	0.011	$2.23e-04$
7	ReportResultMergeTest.parsePartOneTwo_WithFailedRerun	0.0771	0.004	$8.03e-05$
8	ReportResultMergeTest.merge_PartOneTwo_WithFailedRerun_Equals_AllInOneFailed	0.0727	0.01	$3.93e-04$
9	StepNameFormatterTest.format_shouldEscape	0.0563	0.011	$9.46e-04$
10	ReportResultMergeTest.checkPartOneIsValidReport	0.0474	0.01	$2.54e-05$
11	ReportResultMergeTest.checkAllFailedFileIsValidReport	0.0398	0.005	$2.09e-08$
12	TagsDeserializerTest.deserialize_returnsTags	0.0316	0.009	$1.53e-06$
13	FeatureTest.calculateReportFileName_ReturnsFileName	0.0288	0.009	$3.08e-06$
14	EmbeddingWithNameTest.getDecodedData_ReturnsDecodedContent	0.0258	0.011	$1.80e-05$
15	TagsDeserializerTest.deserialize_OnExcludedTags_returnsTags	0.0196	0.004	$5.64e-10$

TABLE VI: Detailed energy usage for cucumber-reporting

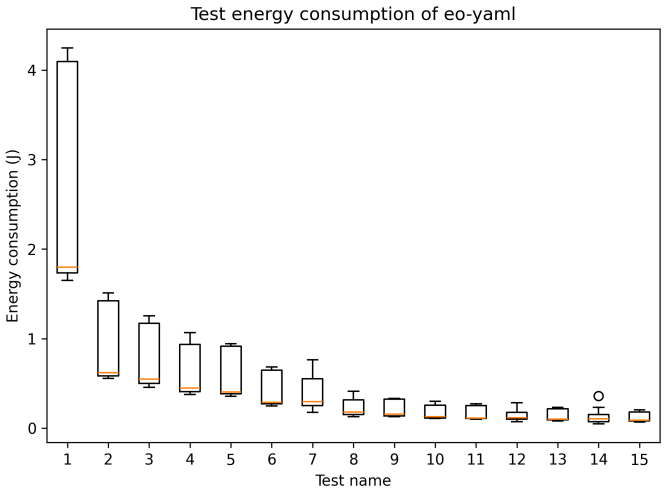

Fig. 8: Energy consumption for eo-yaml test suite. Corresponding tests can be found in table VII

Index	Test name	Mean J	Std. dev.	p-value
1	MutableYamlMappingBuilderTest.complainsOnEmptyKey	2.6786	1.178	$4.07e-05$
2	RtYamlInputTest.readsTypicalDockerComposeFile	0.9212	0.413	$7.07e-05$
3	RtYamlInputTest.readsTypicalAzurePipelineYaml	0.7808	0.336	$1.43e-04$
4	RtYamlInputTest.readsTypicalDeploymentYaml	0.6371	0.278	$3.37e-04$
5	RtYamlInputTest.readsTypicalTravisFile	0.5933	0.26	$4.85e-05$
6	BuiltCommentsTest.iteratesOverComments	0.4247	0.187	$5.43e-05$
7	YamlSequencePrintTest.printsReadYamlSequenceWithAllNodes	0.3858	0.179	$6.77e-03$
8	YamlMappingCommentsPrintTest.printsReadYamlMappingWithScalarComments	0.2299	0.093	$6.98e-03$
9	RtYamlInputTest.readsTypicalSpringApplicationProperties	0.2147	0.091	$1.06e-04$
10	BuiltYamlStreamTest.greaterThanAMapping	0.1758	0.076	$2.91e-04$
11	EmptyYamlSequenceTest.returnsCommentFromDecoratedObject	0.1649	0.073	$7.51e-05$
12	YamlMappingCommentsPrintTest.printsReadYamlMappingWithComments	0.1483	0.067	$1.04e-02$
13	YamlMappingCommentsPrintTest.readsComments	0.1428	0.062	$2.91e-04$
14	ReadYamlMappingTest.returnsValuesOfStringAndComplexKeys	0.1235	0.075	$2.90e-03$
15	RtYamlInputTest.shouldReadSequenceOfFoldedBlockScalars	0.1234	0.055	$3.52e-04$

TABLE VII: Detailed energy usage for eo-yaml

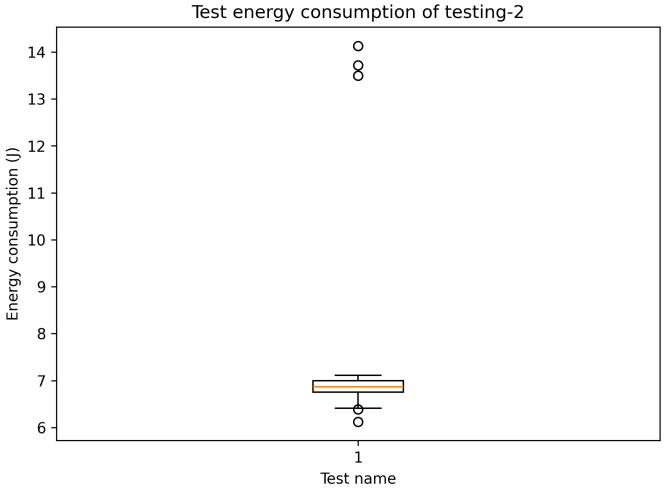

Fig. 9: Energy consumption for testing-2 test suite. Corresponding tests can be found in table VIII

Index	Test name	Mean J	Std. dev.	p-value
1	MainTest.calculateTest	7.492	2.11	$1.71e-09$

TABLE VIII: Detailed energy usage for testing-2