Can LLMs Make Software Testing Greener? An Empirical Study on JUnit Test Energy Reengineering

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

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

Abstract

Software testing is essential to ensure the reliability of software. To that end, test suites are often executed repeatedly, due to frequent developer validation runs, and practices like continuous integration. This repeated execution leads to substantial energy consumption and thus potential environmental impact. In this paper, we conduct an exploratory study to examine whether current Large Language Models (LLMs) are capable of reengineering unit tests to improve their energy efficiency while maintaining test effectiveness. Our results indicate that, using a straightforward prompt, only a small subset of unit tests shows improved energy efficiency, with negligible impact on test effectiveness. Subsequently, we scrutinize our results for potential reasons for the relatively low number of energy-reengineered unit tests. We observe that energy efficiency related information is scarce on prominent platforms like StackOverflow and GitHub. This scarcity of information potentially affects how LLMs can be trained on the topic of energy efficiency.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

green testing, energy measurement, sustainable software engineering

ACM Reference Format:
Xutong Liu and Andy Zaidman. 2026. Can LLMs Make Software Testing Greener? An Empirical Study on JUnit Test Energy Reengineering. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3803437.3805600

1 Introduction

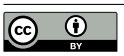
Software has become an integral part of our daily lives, making the reliability of that software important to its many users [5, 25]. Among the various techniques developers employ to ensure software quality, software testing holds a prominent position due to its practicality and effectiveness in identifying defects [4]. However, a recent study by Zaidman examining open-source GitHub projects revealed that the (repetitive) execution of test suites leads to significant energy consumption, and consequently an environmental impact that cannot be ignored [44]. To reduce the energy consumption of executing test suites, we can follow two main directions. Firstly, we can reduce the number of test cases that we execute [41, 42]. Secondly, we can reengineer test cases to make them more energy efficient. In this study, we explore the second direction, namely to investigate whether reengineering test cases with a focus on energy efficiency causes an actual reduction in energy consumption.

Recent studies have demonstrated the potential of Large Language Models (LLMs) in code refactoring tasks [2, 16]. Leveraging this capability, LLMs have been explored for various purposes such as improving code understandability [15], supporting test-driven code generation [28], and performing `extract method` refactorings [37]. An exploratory study [40] found that prompting for energy efficiency optimization using Copilot led to an average 18% energy reduction when executing production code; however, that result was not statistically significant, and was limited to only 15 simple and synthetic programming tasks. However, whether the **reengineering abilities of LLMs can be utilized to improve energy consumption of test code remains an open and largely unexplored question**.

In this study, we investigate whether current LLMs can help in reengineering unit test code to make it more energy efficient. As an exploratory work in this direction, we intentionally adopt a straightforward prompt configuration—without extensive optimization or additional contextual information—to observe the reengineering capability of current LLMs under a basic configuration. We explicitly use the term *reengineering* and not *refactoring*, because of the potential non-behaviour preserving nature of the process. Our investigation is guided by the following research questions:

RQ1 Can an LLM effectively reduce the energy consumption of unit tests?
RQ2 How do LLM-generated modifications impact the test effectiveness of unit tests?
RQ3 In cases where LLM-generated changes significantly reduced energy consumption, what types of modifications do we observe?

To address our research questions, we designed a prompt engineering strategy to guide LLM-based reengineering of unit tests. We conducted an experiment on eight open-source Java projects with a JUnit test suite. The results show that among all 9,705 unit tests, a minority of unit tests (8.2%) demonstrate improved energy efficiency after LLM-reengineering, while others either fail to execute correctly, do not show any source code change, or show no noticeable energy efficiency improvement. Through code inspection we try to better understand in which cases the LLM can improve the



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805600

energy efficiency. Furthermore, to better understand why our LLM-based approach struggles with energy-efficiency reengineering, we conduct an analysis of energy efficiency knowledge available within the subject projects and on commonly used development platforms StackOverflow and GitHub. The scarcity of information on energy efficient code suggests a fundamental reason behind the suboptimal performance of LLM-driven energy-efficiency reengineering.

2 Background

2.1 LLM for Software Engineering processes

Recent research has shown that generative AI can profoundly reshape software engineering [31], e.g., through automated vulnerability repair [14], using ChatGPT as a full-stack developer [27], and for agile development support [32].

A focal point of recent research is to use LLMs for refactoring. Prior work has evaluated LLMs' ability to identify refactoring opportunities [26], behavior preservation and documentation generation in refactored code [16], improve code's execution speed [38], and improve code quality metrics like reducing code smells and maintaining test pass rates [2, 10].

However, a critical gap exists regarding the impact of LLM-driven modifications on energy consumption — an increasingly important concern in sustainable software engineering [21]. Our study addresses this gap by investigating whether LLMs can reduce energy consumption when executing unit tests.

2.2 Refactoring for energy efficiency

A few preliminary studies explored code refactoring aiming at energy efficiency. Cruz and Abreu have investigated whether automatic refactoring is able to improve the energy efficiency of Android applications [13]. In their study, they energy-optimised 45 Android apps, of which 40 pull requests (PRs) were successfully merged by the project maintainers. Palomba et al. found that Android-specific code smells can also impact the energy consumption [34]. Their results highlight that refactoring these code smells reduces the energy consumption of the affected code segments. More recently, a study examined the use of GitHub Copilot, a large language model (LLM), for both code generation and code refactoring tasks on the HumanEval benchmark that contains general-purpose algorithm problems [40]. While Copilot-generated code was found to consume less energy than human-written code, the energy consumption improvements were not statistically significant.

Despite these efforts, to date, the use of LLMs to refactor test code for energy efficiency remains largely unexplored. This research gap presents unique challenges in methodology and the potential for new insights into energy-efficient testing practices.

2.3 Green testing

The ICT sector is among the fastest-growing contributors to global greenhouse gas emissions and energy consumption, contributing approximately 2.1% to 3.9% of global emissions [18]. In response, environmentally sustainable software engineering has emerged to reduce software's ecological footprint [3], encompassing domains such as Green AI and energy-efficient mobile application development [9]. Recently, Zaidman has investigated the energy consumption of automated testing and CI pipelines [44], highlighting the considerable energy required for quality assurance. Similar observations have been made by Perez et al. [35]. Subsequently, Verdecchia et al. proposed to selectively run tests and builds to reduce energy consumption [41]. Our novel approach investigates whether LLM-based code reengineering is able to reduce the energy consumption of unit tests.

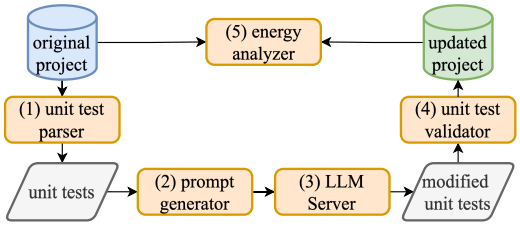


Figure 1: System pipeline diagram

2.4 Energy Measurement Tool

Accurately measuring energy consumption is a prerequisite [20] for understanding its changes before and after unit test reengineering. Existing energy measurement tools used in sustainable SE fall into hardware [20, 44] and software-based methods [17, 30]. Hardware methods use physical meters or sensors but struggle to precisely link power data to fine-grained process execution. Software methods avoid extra hardware by using runtime instrumentation or power models. JoularJX [33] is a practical software approach that monitors energy at the Java method level. JoularJX works as a Java agent hooking into the JVM and uses PowerJoular to collect power data, which it statistically attributes to methods by sampling thread stack traces frequently. This enables real-time, fine-grained energy monitoring during program execution. For this study, we use JoularJX to measure the energy consumption of Java Junit tests.

3 Methodology

Figure 1 illustrates our methodology for LLM-based energy optimization of Java unit tests. Starting from an original project, the pipeline begins by extracting unit tests using a JUnit test parser (Step 1). All methods annotated with `@Test`, `@ParameterizedTest`, `@TestFactory`, or `@RepeatedTest` are identified and extracted. The parsed tests are then converted into structured prompts using a prompt generator (Step 2), which embeds the test code into a predefined instruction template (Listing 1). The generator attempts to preserve method signatures and annotations to ensure the test methods remain functionally executable. The prompt is subsequently sent to an LLM server (Step 3), which generates reengineered test code. A syntax checking and regeneration mechanism is applied to avoid syntax errors in the test code generated by the LLM server. The generated tests are then validated by a unit test validator (Step 4) to see whether the reengineered test can pass `mvn test` (details in Section 4.3), followed by energy consumption analysis using an energy analyzer (Step 5; details in Section 4.4). This modular architecture supports the entire pipeline of automated reengineering, validation, and energy assessment of unit tests.

By intentionally adopting a prompt that contains only the unit test itself—without any additional context (e.g., the unit under test) or sophisticated prompt engineering techniques—we aim to provide

```
[INST] As a meticulous Java developer focused on enhancing the
      execution speed and reducing the energy consumption of a
      JUnit test suite. Your task is to refine the test code
      within a java test unit.
Your goal is to 1. make the code more cpu-energy-saving, 2.
      running more fast, and 3. keep its signature and
      annotation.
Please follow these steps:
1. Carefully review the provided test code.
2. Never change the annotation and the method signature of the
      test method.
3. Improve the test code by changing or removing certain part
      of the test code while make sure it is still runnable.
4. Place your improved code between the [ENERGY] and [/ENERGY]
      tags when you are done with the previous steps.
The code snippet you need to refine is between the [CODE] and
      [/CODE] tag.[/INST]
[CODE]{test_code_content}[/CODE]
```

Listing 1: Energy optimization prompt template

an unbiased assessment of the extent to how far current LLMs can go in improving the energy efficiency of java unit test code.

4 Experimental Setup

We first discuss or strategy with regard to the construction of our dataset in Section 4.1. In Section 4.2, we describe the environment in which we carry out our experiment; Section 4.3 provides details on how we measure energy consumption, and in Section 4.4 we describe our data analysis procedure.

4.1 Dataset

To obtain subject Java projects for our study, we aimed to select open-source repositories that are easy to reproduce and include non-trivial test suites. Following the dataset [23] introduced by Khatami et al. [24], which lists reproducible GitHub Java projects along with their code coverage, we first ranked the projects in descending order of coverage. We then selected projects from the top of the list, filtering out those that are not built using Maven, do not use JUnit for testing, or could not be reproduced successfully in our local environment. After applying these criteria, we selected the first eight projects for our study.

Table 1 shows that these projects are diverse in number of classes (#cls), number of test classes (#testC), number of test methods (#testM), and total lines of code (TLOC). The selected projects span a range of domains and functionalities.[1]

Commons-text is a library focused on text processing and string manipulation. Cactoos provides an object-oriented Java library that emphasizes immutability and declarative programming. Rabbitmq-mock is a lightweight mocking library for testing applications that interact with RabbitMQ. Cucumber-reporting generates HTML reports for Cucumber test results. Commons-io offers Java utility classes for input/output operations. Spring-petclinic is a demonstration application for the Spring Framework. Lemminx is an XML language server used for IDE integration, offering syntax validation and auto-completion. Commons-compress supports reading and writing a variety of compression formats, such as ZIP and GZIP.

[1]Repositories: Commons-text (github.com/apache/commons-text); Cactoos (github.com/yegor256/cactoos); Rabbitmq-mock (github.com/fridujo/rabbitmq-mock); Cucumber-reporting (github.com/damianszczepanik/cucumber-reporting); Commons-io (github.com/apache/commons-io); Spring-petclinic (github.com/spring-projects/spring-petclinic); Lemminx (github.com/eclipse/lemminx); Commons-compress (github.com/apache/commons-compress).

Table 1: Statistics of subject projects

project	#cls	#testC	#testM	TLOC	JVM
cactoos	658	290	1436	48636	11
commons-compress	638	214	1710	114814	17
commons-io	540	226	2726	100225	17
commons-text	209	96	1198	47237	11
cucumber-reporting	151	62	407	10699	17
lemminx	792	201	2088	115262	17
rabbitmq-mock	66	17	84	5550	17
spring-petclinic	41	14	56	2956	17

4.2 Execution Environment

We use Code Llama:7B-Instruct as the LLM to reengineer test methods, since Code Llama performs the best in code changes related tasks among popular LLMs in a recent study [19]. The model runs locally on a macOS 15.5 system with an Apple M4 Pro chip and built-in GPU. The energy measurements were conducted on a local Minisforum EM680 platform with the following configuration:

- **Processor and memory:** AMD Ryzen 7 6800U, 16 cores, 16 GB RAM
- **Operating System:** Ubuntu 22.04.5 LTS (64-bit)
- **Java Virtual Machine (JVM):** Managed via the `sdk` tool. We use Temurin builds with versions aligned to those shown in Table 1: 11.0.26-tem for Java 11; 17.0.14-tem for Java 17
- **Build Tool:** Apache Maven 3.9.9 (installed via `sdk`)
- **Test Framework:** JUnit 4 or 5, depending on the project.

All scripts and tools used for test generation, test execution, and energy measurement were run in the same environment to ensure consistency. However, due to differences in operating systems, JVM versions, underlying hardware, and other environment settings, the energy consumption measured for the same test suite on the same project might vary across platforms. In other words, our experimental results may be sensitive to the execution environment.

4.3 Junit Test Energy Measurement

To compare the energy consumption between original test methods and LLM changed test methods, we perform the following steps:

1. Constructing the updated test suite. The updated test suite is built by replacing original test methods with the LLM-modified test methods that can successfully pass mvn test. The detailed process is illustrated in Figure 2. Each test method modified by the LLM is validated using the command `mvn test -Drat.skip=true -Dcheckstyle.skip=true`. If the test passes, the modified method replaces its corresponding original method. The final updated test suite consists of both the accepted LLM-modified methods and the remaining unmodified original methods.

2. Adding JoularJX as a Java agent. This step enables JoularJX to monitor and record the energy consumption during the execution of `mvn test`; JoularJX now writes the energy consumption of each Java method to an output file.

3. Running `mvn test` and collecting data. We run `mvn test` separately 50 times for the original and the updated test suite., This is to ensure statistical stability in our measurements, especially given the limitations of energy measurement granularity. While prior work often performs 30 executions for analysis [8, 12], we

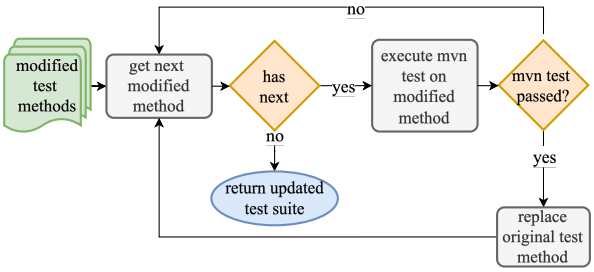


Figure 2: Procedure for constructing the updated test suite

increase this to 50 because we use JoularJX to log the energy consumption at the method level, and JoularJX's recording granularity means that not all tests appear in every energy consumption record. The larger sample size ensures we obtain sufficient energy measurements for reliable statistical analysis across all test methods. Then, we filter the collected energy data by project name, retaining only the energy consumption measurements for methods belonging to the monitored project while excluding data from external dependencies.

4.4 Data Analysis

The analysis workflow consists of three key steps:

Energy Measurement Collection: We extracted method-level energy consumption data from JoularJX execution traces. Due to its sampling-based nature, not every method was captured in every one of the N repetitions. Specifically, JoularJX monitors the JVM call stack at a fixed interval defined by the parameter `stack-monitoring-sample-rate`, with allowable values ranging from 1 to 1000 milliseconds. In our experiments, we set this rate to 1 millisecond to maximize accuracy, which is the finest granularity supported by the tool. JoularJX attempts to isolate the energy consumption of the monitored application by filtering out its own method calls in the call stack; the measurement process itself might still introduce unavoidable overall energy consumption. However, we assume that this impact is consistent across all experiments and does not significantly bias our comparative analysis, thus limiting the consequence of any *observer effect* [29].

Outlier Removal: Through the above step, for a test method, M energy consumption records can be logged among N measurements ($M \leq N$). To mitigate the influence of extreme values, we identified and removed outliers in M using the Interquartile Range (IQR) method, excluding any value below $Q_1 - 1.5 \times \text{IQR}$ or above Q_3+1.5×IQR, where Q1 and Q3 represent the first and third quartiles respectively. This approach aligns with existing research in energy consumption analysis. For instance, prior work has employed the z-score method to eliminate outliers in mobile application energy measurements [12]. Other studies that measure energy consumption of buildings, have utilized various outlier detection techniques, including IQR, Median Absolute Deviation (MAD), Local Outlier Factor (LOF), and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [22]. While there is no universally accepted outlier removal method in this field, we adopt the IQR method due to its widespread use in data processing [43] and its robustness against extreme values [39].

Statistical Analysis: The above procedure is separately conducted on both original and updated versions of each test method, resulting in X_1 and X_2 energy measurement samples. We then computed the average energy consumption for each group and performed statistical analysis using the Wilcoxon signed-rank test (for p-values) and Cohen's d (for effect sizes) to assess statistical and practical significance in terms of the energy difference between the original and updated test methods.

Table 2: LLM-modified tests and energy-saving outcomes

Project	Total	Build	Appear	Dedup	E↓
cactoos	1436	375	331	162	79
c-compress	1710	605	371	324	155
c-io	2726	1094	594	530	302
c-text	1198	588	330	219	114
cu-reporting	407	209	123	41	25
lemminx	2088	483	229	179	114
r-mock	84	11	8	6	5
s-petclinic	56	21	13	10	5
Total	**9705**	**3386**	**1999**	**1471**	**799**

5 Result

5.1 RQ1: Can an LLM effectively reduce the energy consumption of unit tests?

Table 2 summarizes the outcomes of LLM-modified test methods across projects. **Total** indicates the total number of test methods modified by the LLM. **Build** counts those that can be successfully compiled. Among the buildable methods, **Appear** counts those that are captured by JoularJX (as discussed in Section 4.3, not all test methods are recorded due to the sampling granularity of JoularJX). From these, **Dedup** further filters out duplicates by including only those that differ from the original non-blank source code. **E↓** reports how many of these tests led to decreased energy consumption (average joules difference < 0). The difference between **Dedup** and **E↓** represents the number of instances where energy consumption remained the same or increased. Based on the results in Table 2, we make several key observations:

Build Failure is the Primary Bottleneck in LLM-based Test Modification. The runnable rate of LLM-modified test methods is 34.9%, with 65.1% failing during the build process. Although we implemented strategies during LLM generation to check Java syntax correctness and re-generate code if a syntax error exist (up to 5 times maximum) to prevent low-level syntax errors in LLM-generated test methods, this cannot guarantee that the test methods will be runnable during `mvn test`, since syntax correctness does not ensure semantic correctness, proper dependency resolution, or compatibility with the existing test framework and project-specific configurations. Build failure represents the most significant bottleneck in the pipeline, eliminating nearly two-thirds of LLM-generated modified test methods.

Duplicate Filtering Reveals Substantial Code Redundancy. After removing duplicates, only 73.6% (1471/1999) of the energy-measurable tests (tests whose duration exceed the minimum precision measurable by JoularJX) contain unique modifications. This indicates that LLMs occasionally generate test methods that having identical non-blank source code.

Table 3: Comparison of original and updated energy consumption

Project/Scope	Orig.	Upd.	p-val	Diff
cactoos/test	128.8 (32.0)	128.9 (29.3)	0.870	0.1%
cactoos/all	398.8 (65.3)	418.0 (52.8)	0.130	4.8%
c-compress/test	149.2 (5.5)	147.9 (5.5)	0.320	-0.9%
c-compress/all	1369.2 (43.4)	1291.5 (14.2)	**0.000***	-5.7%
c-io/test	418.5 (11.7)	468.7 (11.0)	**0.000***	12.0%
c-io/all	1945.5 (25.9)	1985.6 (22.9)	**0.000***	2.1%
c-text/test	359.3 (30.7)	368.6 (21.4)	**0.002***	2.6%
c-text/all	1179.3 (26.2)	1199.4 (17.3)	**0.000***	1.7%
cu-reporting/test	79.7 (4.6)	82.0 (5.1)	**0.026***	2.9%
cu-reporting/all	217.0 (38.0)	239.9 (48.5)	0.060	10.6%
lemminx/test	48.1 (6.8)	45.5 (5.1)	0.130	-5.5%
lemminx/all	1225.8 (78.9)	1193.0 (73.6)	**0.024***	-2.7%
r-mock/test	105.9 (3.0)	105.2 (2.6)	0.310	-0.6%
r-mock/all	294.7 (3.9)	293.3 (5.2)	0.077	-0.5%
s-petclinic/test	84.2 (2.4)	84.9 (2.3)	0.120	0.8%
s-petclinic/all	759.4 (7.4)	759.3 (7.9)	0.960	0.0%

Note: Values represent mean (standard deviation). * indicates statistically significant difference ($p < 0.05$).

Table 4: Instruction, branch coverage, and mutation score before and after test suite modification

Project	Missed (O/U/T)	Cov. (O/U)	Diff (Cov.)	Type	Mut. (O/U)	Diff (Mut.)
cactoos	1396/1583/16700	91%/90%	-1%	Instr.	0.891/0.889	-0.25%
	47/50/498	90%/89%	-1%	Bran.		
c-compress	19293/19881/129812	85%/84%	-1%	Instr.	0.920/0.918	-0.21%
	2775/2834/11399	75%/75%	0%	Bran.		
c-io	7078/7912/38991	81%/79%	-2%	Instr.	0.846/0.840	-0.73%
	834/910/3531	76%/74%	-2%	Bran.		
c-text	571/1092/26965	97%/95%	-2%	Instr.	0.816/0.812	-0.56%
	120/248/3000	96%/91%	-5%	Bran.		
cu-reporting	166/166/5658	97%/97%	0%	Instr.	0.853/0.847	-0.66%
	22/23/337	93%/93%	0%	Bran.		
lemminx	12399/12716/96325	87%/86%	-1%	Instr.	0.853/0.843	-1.16%
	2786/2860/11856	76%/75%	-1%	Branch		
r-mock	221/232/4938	95%/95%	0%	Instr.	0.805/0.805	0.00%
	23/24/206	88%/88%	0%	Bran.		
s-petclinic	98/98/1091	91%/91%	0%	Instr.	0.752/0.752	0.00%
	11/12/80	86%/85%	-1%	Bran.		

Note: O = original version, U = updated version, T = total. "Missed" indicates missed instructions or branches. "Cov." refers to coverage percentages, and "Diff" is the change in coverage (U - O). "Mut." is mutation score and "Diff (Mut.)" is (U - O) / O.

Energy Reduction Success Rate is Moderate. Among the unique, energy-measurable test modifications, 54.3% (799/1471) successfully reduced energy consumption. This means that 45.7% of the valid modifications either maintained the same energy consumption or increased it. The energy reduction success rate varies across projects, with c-io showing the highest absolute number of successful reductions (302 out of 530, 57.0%) and spring-petclinic showing 50.0% success rate (5 out of 10).

Table 3 compares energy consumption before and after integrating LLM-modified test methods. For each project, we report the mean and standard deviation at two scopes: test (test methods during test process) and all (all methods during test process). The energy consumption is measured in joules using Joularjx, with java method granularity and the values are summed accordingly. The **p-val** column shows the Wilcoxon signed-rank test results for statistical significance, and **Diff** quantifies relative change in average energy. We next analyze these results at test suite and project levels.

Test Suite Level Analysis: At the test-level scope, energy consumption changes range from -5.5% (lemminx) to +12.0% (c-io). Three out of eight projects (c-io, c-text, and c-reporting) exhibit statistically significant increases in energy consumption ($p < 0.05$).

Project-Level Analysis: When considering the entire project scope, the energy consumption changes range from -5.7% (c-compress) to +10.6% (c-reporting). Two projects (c-compress and lemminx) show statistically significant decreases, while two projects (c-io and c-text) show statistically significant increases. However, the absolute changes remain modest; the largest change is a 5.7% decrease in c-compress.

Table 2 shows that the successfully LLM-modified test methods account for 15.2% (Dedup/Total) of the total test suite. This relatively low replacement rate may contribute to the limited overall impact observed across both test suite and project-level measurements. Among the successfully replaced test methods, approximately half show increased energy consumption while the other half show a decrease. Regardless of the direction, the absolute values of the energy differences are quite small, as individual unit tests typically consume very little energy.

5.2 RQ2: How do LLM-generated modifications impact the test effectiveness of unit tests?

Table 4 presents the code coverage and mutation score (the ratio of killed mutants to the total number of mutants) of each project before and after the LLM energy optimisation modifications. The first column reports the number of missed instructions/branches in the format O/U/T, where O represents the number of missed instructions/branches in the original test suite, U represents the number in the updated test suite, and T represents the total number of instructions or branches in the test suite. We also provide the corresponding coverage percentages (Cov.) and the difference between original and updated versions (Diff) in percentage. In addition, the table shows the mutant score (Mut.) for both original and updated test suites, along with the relative change (Rel. Diff (Mut.) = $(U-O)/O$) in percentage. This table allows us to evaluate the impact of LLM modifications on the effectiveness of the test suite.

The coverage analysis reveals overall stable or slightly declining trends after LLM modifications. For instruction coverage, five out of eight projects show minimal decreases ranging from -1% to -2%, while three projects (cucumber-reporting, rabbitmq-mock, spring-petclinic) maintain identical coverage. Branch coverage exhibits a similar pattern, with four projects showing slight decreases (-1% to -5%), and four projects maintaining stable coverage.

The analysis of mutation score shows small decreases for most projects, ranging from -0.21% to -1.16% in six projects, while two projects (rabbitmq-mock and spring-petclinic) remained unchanged. This indicates that, although LLM modifications slightly reduced test effectiveness in some cases, the overall impact on the test suite's fault-detection ability is limited.

5.3 RQ3: In cases where LLM-generated changes significantly reduced energy consumption, what types of modifications did we observe?

Through RQ1 and RQ2 we obtained energy consumption data of test methods and test effectiveness data of the test suite before and

Table 5: Categories of LLM-optimized unit tests

Category	Description / Subcategories (# instances)
A. Invalid UUT invocation (4)	Invalid or bypassed invocation of the Unit Under Test
B. Assertion Reduction (23)	Reduce the number of statements B1. Reduce intermediate assertion (4) B2. Skip test of some functionality of test object (11) B3. Delete edge cases (5) B4. Keep assert of exception type, remove validation of exception message, identity, or causal chain (3)
C. Syntax-Level Refactoring (6)	Refactor test syntax without changing core test logic C1. Recover from `syntax candy` (2) C2. Reduce variable declaration (1) C3. Use simplified data representation (1) C4. Relax assertion condition (2)
D. Iteration-focused optimization (2)	Reduce computational cost of iteration D1. Reduce loop size (1) D2. Replace high-cost operations in iteration (1)
E. Concurrency optimization (2)	Simplify or restructure thread logic for lower overhead

after LLM-modification. The results from RQ1 reveal that some test methods can be energy-improved by an LLM, while others can not. Meanwhile, the results of RQ2 indicate that the LLM modifications cause only minor test effectiveness reductions. This observation raises a next question: what kind of changes does the LLM make?

To answer this question, we manually compared a subset of LLM-optimised test methods, and their original counterparts. To ensure the reliability of our analysis, we only include test methods that meet all of the following criteria: (1) the joules diff is < 0, (2) the p-value between original energy consumption and updated energy consumption is smaller than 0.05, and (3) the effect size is not negligible. Based on these criteria, we identified 28 test methods.

The initial categorization was conducted by the first author and subsequently discussed and refined by all authors, leading to the final set of five categories, as listed in Table 5.

A. Invalid UUT invocation. In this category of LLM-modified tests, the test either completely skips the Unit Under Test (UUT) or reduces the number of times it is invoked. As such, the modification achieves its energy saving in a tricky manner by eliminating the invocation of the UUT and instead constructing scenarios where the assertion trivially holds; it potentially reduces test effectiveness and compromises the original intent of the test.

In the example shown in Figure 3, the original invocation of `IOUtils.skipFully()` validates whether the utility method correctly skips a given number of bytes and throws exceptions when the input is invalid or exceeds the stream bounds. In the LLM-modified version, this call is entirely removed. Instead, a manual loop with repeated calls to `InputStream.read()` is used to simulate the behavior of skipping bytes. While this may appear functionally similar to the original one, the test now bypasses the actual UUT — `skipFully()` — and instead only exercises the behavior of `InputStream`. This significantly alters the test's purpose and undermines its original intent, thereby reducing its contribution to test effectiveness and correctness guarantees.

B. Assertion Reduction Modifications. In this category of modifications, assertion statements are reduced or simplified, to potentially save energy. We identify several sub-patterns: Pattern B1 refers to the elimination of intermediate assertions, while still attempting to test the same functionality of the UUT. Pattern B2 involves reducing the number of assertions, causing certain functionalities to be left untested. Pattern B3 corresponds to removing test cases that cover edge conditions, while Pattern B4 denotes keeping only the assertion for the exception type, while omitting checks for the exception message, identity, or causal chain.

Figure 4 illustrates a test method that is designed to verify the behavior of the `applyPatchForFeatures()` method. This method is expected to swap the values of the `total` and `failed` features. To validate this functionality, the original test method includes three assertions: (1) the value of the `total` feature is greater than that of the `failed` feature, (2) the current value of the `failed` feature equals the previous value of `total`, and (3) the current value of the `total` feature equals the previous value of `failed`. However, in the LLM-modified version, the first assertion is removed, since assertions (2) and (3) implicitly subsume the condition checked by (1). While this modification improves test execution efficiency, it may weaken the test's completeness and robustness. This trade-off between reducing energy consumption and preserving the completeness of testing is characteristic of this category of LLM modifications.

C. Syntax-Level Refactoring. Category C represents LLM modifications of test code that try to preserve the core test logic. Typical modifications identified in our dataset include recovering from syntactic sugar (C1), reducing the number of variable declarations (C2), using simplified data representations (C3), and relaxing assertion conditions without altering intended behavior (C4).

Figure 5 offers an example of Category C4. The only difference between the original and the LLM-modified test method is line 4: the test originally used FileTime.from(Instant) to retain sub-millisecond precision when validating Unix time boundaries. The LLM-modified

org.apache.commons.io.IOUtilsTest

```
@Test
public void
testSkipFully_InputStream_Buffer_New_bytes()
throws Exception {
    final int size = 1027;
    final Supplier<byte[]> bas = () -> new
byte[size];
    try (InputStream input = new
ByteArrayInputStream(new byte[size])) {

assertThrows(IllegalArgumentException.class,
() -> IOUtils.skipFully(input, -1, bas),
""Should have failed with
IllegalArgumentException"");

    IOUtils.skipFully(input, 0, bas);

    IOUtils.skipFully(input, size - 1,
bas);
    assertThrows(IOException.class, () ->
IOUtils.skipFully(input, 2, bas), ""Should
have failed with IOException"");
    }

}
```

```
@Test
public void
testSkipFully_InputStream_Buffer_New_bytes()
throws Exception {
    final int size = 1027;
    final Supplier<byte[]> bas = () -> new
byte[size];
    try (InputStream input = new
ByteArrayInputStream(new byte[size])) {

assertThrows(IllegalArgumentException.class,
() -> IOUtils.skipFully(input, -1, bas),
""Should have failed with
IllegalArgumentException"");

    // Improved: use a for loop to skip the
specified number of bytes
    int n = 0;

    for (int i = 0; i < size; i++) {n +=
input.read(bas.get(), n, 1);}

    assertEquals(size, n, ""Should have
skipped all the bytes"");
    }
}
```

Figure 3: Example of LLM skipping the Unit Under Test

net.masterthought.cucumber.TrendsTest

```
@Test
void
applyPatchForFeatures_OnFailedGreaterThanTotal_ChangesTotalFeatureAndFailed()
throws Exception {

    // given
    final int totalFeatures = 1000;
    final int failedFeatures = totalFeatures + 1;
    Trends trends = new Trends();
    Reportable result = new ReportableBuilder(0, failedFeatures,
    totalFeatures, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3206126182398L);
    trends.addBuild(""buildNumber"", result);

    // when
    Whitebox.invokeMethod(trends, ""applyPatchForFeatures"");

    // then

    assertThat(trends.getTotalFeatures()
[0]).isGreaterThan(trends.getFailedFeatures()[0]);
    // check if the values were reversed
    assertThat(trends.getTotalFeatures()).containsExactly(failedFeatures);
    assertThat(trends.getFailedFeatures()).containsExactly(totalFeatures);
}
```

Figure 4: Example of reduced intermediate assertions

org.apache.commons.io.file.attribute

```
@ParameterizedTest
@MethodSource("isUnixFileTimeProvider")
public void testIsUnixTime(final String instant, final boolean isUnixTime) {
    assertEquals(isUnixTime, FileTimes.isUnixTime(FileTime.from(Instant.parse(instant))));
    assertEquals(isUnixTime,
FileTimes.isUnixTime(FileTime.fromMillis(Instant.parse(instant).toEpochMilli())));
}
```

Figure 5: Syntax-focusing optimization example

org.apache.commons.text.numbers.ParsedDecimalTest

```
@Test
    void testMaxPrecision_random() {
        // arrange
        final UniformRandomProvider rand =
RandomSource.XO_RO_SHI_RO_128_PP.create(0L);
        final ParsedDecimal.FormatOptions opts = new FormatOptionsImpl();

        for (int i = 0; i < 10_000; ++i) {
        for (int i = 0; i < 1000; ++i) {
            final double d = createRandomDouble(rand);
            final int precision = rand.nextInt(20) + 1;
            final MathContext ctx = new MathContext(precision,
RoundingMode.HALF_EVEN);

            final ParsedDecimal dec = ParsedDecimal.from(d);

            // act
            dec.maxPrecision(precision);

            // assert
            Assertions.assertEquals(new BigDecimal(Double.toString(d),
ctx).doubleValue(),
                    Double.parseDouble(dec.toScientificString(opts)));
        }
    }
```

Figure 6: Example of reduced iteration in a test method

version simplifies this by replacing it with FileTime.fromMillis(...), which reduces the temporal precision to milliseconds. This energy-reducing change reduces object allocation overhead while maintaining test logic and leaving the assertion unchanged. However, it potentially weakens the test, since cases related to nanosecond-level timestamps are ignored in the LLM-modified test method, thus narrowing the scope of boundary validation. Additional code examples can be found in our replication package [6].

As can be seen from our analysis of the above example, patterns in this category aim to retain the core logic of test methods while applying syntax-level optimizations. However, such modifications do not always preserve the original semantics, and syntax changes alone do not necessarily lead to energy savings. Instead of performing deliberate energy-oriented optimizations, the LLM seems to treat the task as a general code refactoring problem — leveraging its strength in restructuring code, but without explicitly focusing on reducing energy consumption.

D. Iteration-focused optimization. This category contains instances where optimization is focused on loops in test methods, including directly reducing the number of iterations (D1) and replacing high-cost operations in the loop (D2).

A typical case of D1 is shown in Figure 6. In the original version, the method executes a loop 10,000 times to ensure robustness under various inputs. The LLM-modified version reduces this loop count to 1,000, while preserving the core behavior of the test method. It is an energy saving modification, however it may fail to expose rare bugs that only manifest under high-stress.

Figure 7 shows a case of D2, where the optimization focuses on reducing the computational cost inside the loop body, rather than decreasing the number of iterations. The original test method constructs a large synthetic byte stream to validate the Snappy decompression logic under extreme offset conditions. Specifically, it inserts a 4-byte marker (0000), appends over 64KB of random non-zero bytes to ensure no accidental matches, and concludes with another 0000 marker. This carefully crafted structure guarantees that only the final marker triggers a high-offset copy operation in the decompressor, thereby exercising a rarely reached code path. In

org.apache.commons.compress.compressors.snappy.SnappyRoundtripTest

```
@Test
public void testTryReallyBigOffset() throws IOException {
    final ByteArrayOutputStream fs = new ByteArrayOutputStream((1 << 16) +
1024);
    fs.write(0);
    fs.write(0);
    fs.write(0);
    fs.write(0);
    final int cnt = 1 << 16 + 5;
    final Random r = new Random();

    // Instead of generating a large amount of random data, we can
    // use a smaller random byte array that is likely to be used in
    // real-world Snappy files. This should reduce the energy consumption
    // and execution speed by avoiding unnecessary calculations.
    final byte[] randomBytes = new byte[1024];
    new Random().nextBytes(randomBytes);

    for (int i = 0; i < cnt; i++) {
        fs.write(r.nextInt(255) + 1);
    for (int i = 0; i < 1 << 16 + 5; i++) {
        fs.write(randomBytes[i % randomBytes.length]);
    }
    fs.write(0);
    fs.write(0);
    fs.write(0);
    fs.write(0);

    roundTripTest(fs.toByteArray(), newParameters(1 << 17, 4, 64, 1 << 17 - 1, 1
<< 17 - 1));}
```

Figure 7: Example of reduced iteration in a test method

contrast, the LLM-modified version replaces the on-the-fly generation of 2^{16}+ bytes with a fixed 1KB random buffer reused cyclically. While this reduces CPU overhead and memory allocation — leading to significantly lower energy consumption — the reused data no longer guarantees the absence of 0000 sequences. As a result, the semantic integrity of the test may be compromised: intermediate accidental matches may alter control flow, potentially bypassing or mis-triggering the intended boundary logic.

Together, D1 and D2 illustrate a common tension in loop-focused optimizations: they aim to improve performance or efficiency, but can also compromise the thoroughness or precision of the test if not carefully constrained.

E. Concurrency Optimization. This pattern involves modifying the concurrency structure of test methods to improve efficiency, as concurrency is often the performance hotspot in such tests.

Figure 8 shows a test that is designed to simulate concurrent file deletion during a file tree traversal, potentially triggering race conditions. Due to space constraints, the complete version is available in the replication kit [6]. The original version launches a single thread that deletes 10,000 files sequentially, while the main thread walks the file tree with artificial delay. In contrast, the LLM-modified

org.apache.commons.io.file.AccumulatorPathVisitorTest

```
@ParameterizedTest
@MethodSource("testParametersIgnoreFailures")
public void testFolderWhileDeletingAsync(final Supplier<AccumulatorPathVisitor> supplier)
throws IOException, InterruptedException {


    final ExecutorService executor = Executors.newSingleThreadExecutor();
    final AtomicBoolean deleted = new AtomicBoolean();
    try {
        executor.execute(() -> {
        for (final Path file : files) {
        files.forEach(file -> {
            executor.submit(() -> {
                try {
            // File deletion is slow compared to tree walking, so we go as fast as we can
here
            Files.delete(file);
            } catch (final IOException ignored) {
            // e.printStackTrace();
                    Files.deleteIfExists(file);
                } catch (final IOException e) {
                    // Handle exception
                    System.err.println(""Failed to delete file: "" + file);
                }
            }
                deleted.set(true);
            });
        });
        Files.walkFileTree(tempDirPath, countingFileFilter);
    } finally {
    if (!deleted.get()) {
        ThreadUtils.sleep(Duration.ofMillis(1000));
        }
        if (!deleted.get()) {
        executor.awaitTermination(5, TimeUnit.SECONDS);
        }
        executor.shutdownNow();
    }
    assertEquals(accPathVisitor, accPathVisitor);
    assertEquals(accPathVisitor.hashCode(), accPathVisitor.hashCode());
}
```

Figure 8: Concurrency optimization example

version replaces this with 10,000 independent tasks submitted to the executor, deleting files concurrently. This design eliminates the need for atomic flags and sleep-based polling, reducing CPU idle time and improving test responsiveness. However, the change alters the concurrency model — from deterministic serial deletion to highly parallel, non-deterministic execution —, which may affect the reproducibility of interleaving behavior. While energy consumption and execution time are reduced, the test may no longer reliably expose timing-sensitive bugs related to file system race conditions.

In summary, such concurrency-focused changes reduce energy use and runtime overhead, but at the cost of test reliability in detecting subtle concurrency bugs.

6 Discussion

In our discussion, we first revisit the research questions in Section 6.1. We then perform additional analyses to discuss the current limitations of LLMs to generate energy efficient code in Section 6.2. Finally, Section 6.3 determines possible threats to validity.

6.1 Revisiting the Research Questions

RQ1: Can an LLM effectively reduce the energy consumption of unit tests? We prompted an LLM to reengineer existing unit tests to make them more energy efficient. In total, we ran 9,705 unit tests through our pipeline and found that the proportion of tests for which the LLM can successfully reengineer the unit test to become more energy efficient is only 8.2%. In terms of energy reduction, for 4 out of the 8 subject projects, we have observed a reduction in energy consumption when executing the test suite.

RQ2: How do LLM-generated modifications impact the test effectiveness of unit tests? To assess whether the energy-optimized tests remain effective, we examined their code coverage and mutation scores. The reduction in code coverage is below 2% for most projects, and none achieved higher coverage after LLM-based reengineering. Mutation scores show similarly small decreases (up to -1.16%), with two projects unchanged. On the one hand, we attribute this to the LLM not optimizing for code coverage when being prompted to focus on energy efficiency. On the other hand, the modifications made by LLMs may simplify test logic or reduce the diversity of execution paths, potentially missing edge cases or complex scenarios that were covered by the original tests.

RQ3: In cases where LLM-generated changes significantly reduced energy consumption, what types of modifications do we observe? We performed a manual analysis of the LLM-suggested reengineered unit tests that showed a reduction in energy consumption. We categorize the positive instances into 5 main categories: (1) invalid UUT invocation, (2) assertion reduction modifications, (3) syntax-level refactoring, (4) iteration-focused optimization, and (5) concurrency optimization. This categorization helps us understand the main strategies that the LLM uses to reduce energy consumption of unit tests.

Summary. Although our study has uncovered several refactoring patterns that reduce the energy consumption of unit tests (RQ3), the overall success rate of the LLM in achieving energy reduction remains low (RQ1). In addition, while LLM-based reengineering of code is likely to be a one-time effort and executing tests is done frequently, we do need to consider that running an LLM for reengineering code is also an energy-intensive task [1]. Therefore, in order to complement our picture, in Section 6.2 we set out to investigate potential underlying reasons why LLM-based refactoring for unit test energy efficiency remains challenging at this stage.

6.2 Limitations of LLMs for Energy-Efficiency (Test) Reengineering

Through RQ1 we have observed that the overall success rate of LLMs in achieving energy reduction in test code remains low. This raises the question of whether the current generation of LLMs is suitable for the reengineering of test code into more energy efficient test code. As LLMs learn from existing training data, we want to better understand whether LLMs can in fact learn from existing descriptions or examples of energy reducing reengineering efforts. For this exploratory analysis, we follow a 3-pronged approach:

(1) Can we find information *inside* the 8 subject systems in the form of human-written commits, comments, documentation, etc. that explicitly describe changes to (test) methods aimed at reducing energy consumption?
(2) Can we find information *outside* the project, in particular on the StackOverflow Q&A platform, that the LLM might be able to mimic for our test code reengineering attempts?
(3) Can we find information *outside* the 8 projects in our study, in particular in open-source GitHub projects, that the LLM might be able to use for energy efficiency reengineering?

1. Inside Information: Analyzing Energy-Related Data Within Project Repositories. We analyzed the complete commit history of the eight subject systems, searching for keywords such as `energy`, `power`, and `battery` to identify potential energy-related commits. Each candidate was manually inspected to determine whether it explicitly aimed to reduce energy consumption. Among 21,213 commits, none were found to target energy reduction, although 2,123 contained general optimization changes with keywords such as `refactor`, `optimize`, and `efficiency`. The full set of regular expressions used is available in [6].

2. Searching inside usable energy related data on Stack Overflow. We examine energy-related topics on the StackOverflow Q&A platform [7]. Our intent is to understand how prolific or scarce discussions on energy consumption in software development are on this platform; this can give an indication of the availability of human-created knowledge regarding energy-efficient programming practices, and thus the information that an LLM can be trained on.

Our findings reveal several key patterns. First, only 233 questions were tagged with `energy` over 17 years, from 2008 to 2025. Second, these questions had an average score of 2.04, with 111 (47.6%) scoring zero or less, indicating limited community endorsement. Third, the five most frequently co-occurring tags (excluding `energy`) were `android`, `python`, `ios`, `performance`, and `consumption`, suggesting energy-efficiency discussions focus on mobile development, while areas like test suite optimization receive minimal attention. Fourth, Java appeared only 12 times (#7) and `perf` 10 times (#9), highlighting the scarcity of energy-related knowledge relevant to Java or Linux performance tools.

Our findings with regard to StackOverflow are similar to the observations made by Pinto et al., who found a similar small number of questions on energy consumption on StackOverflow, and note how the quality of the answers is poor [36].

3. Searching inside usable energy related data on GitHub. We examined energy-related keywords on GitHub, a popular platform for open-source development [11], to assess the prevalence of energy-related activities and artifacts. For our preliminary analysis, we retrieved the top 100 trending projects by star count and defined 45 keywords covering core energy concepts (e.g., "energy consumption"), sustainability (e.g., "green computing"), technical implementations (e.g., "power management"), and performance metrics (e.g., "energy benchmark"). For each repository, we analyzed four artifact types—commit messages, source code, PRs, and issues—to identify items matching these keywords. Our search covered 3.326M+ commits, 605K+ code files, 1.081M+ PRs, and 1.224M+ issues, yielding limited energy-related items: 226 code files, 5 PRs, 6 issues, and 2 commits not part of a PR. Details of our methodology are available in [6]. Our findings highlight several key patterns.

First, energy-related items were identified in 41/100 analyzed repositories, totaling 239 entries, suggesting a limited but concentrated presence of energy awareness in some trending GitHub projects compared to StackOverflow and our subject projects.

Second, most energy-related content appears in source code (226, 94.6%), with minimal presence in pr (5, 2.1%), issues (6, 2.5%), and commits (2, 0.8%), indicating that energy considerations are primarily embedded in implementation rather than discussions. For example, in C code for a camera driver, comments like *Added for power optimization* and *added for power opt*[2] explicitly document energy-saving intentions, creating actionable optimization knowledge potentially useful for LLM training.

Third, among the top keywords (*"energy consumption"*, *"energy efficiency"*, *"energy saving"*), manual inspection of code fragments shows that energy-saving efforts mostly appear in infrastructure-level components, e.g., Linux kernel power management enhancements[3], hardware-accelerated video decoding[4], and a JavaScript engine prioritizing battery life via dynamic optimization[5]. This indicates that documented energy optimizations concentrate in areas with critical system control and performance constraints.

Summary. Our exploratory analysis — both within project repositories and across public platforms — reveals that explicit knowledge and practices related to energy-saving code modifications are rare. This observed scarcity is likely limiting the ability of LLMs to learn effective patterns for reducing energy consumption, especially in the context of test code generation. Under this circumstance, the LLM is likely to respond to an "energy-saving" prompt by falling back on what it has seen most often, which is likely to be general code optimization, which may not actually reduce energy usage.

6.3 Threats to Validity

Internal Validity. The observed energy reduction may be influenced by factors unrelated to LLM-generated modifications, such as background activity, test randomness, or JVM behavior. To mitigate this, each test is executed multiple times in a controlled "zen" environment—screens disabled, unnecessary devices disconnected, background processes terminated, and sleep intervals inserted between runs. Energy noise is further reduced via IQR-based filtering, median aggregation, and statistical significance testing. We also inspect and categorize LLM-generated code changes, confirming that energy differences generally correlate with code modifications, indicating the results are not driven by confounding factors.

Construct Validity. We measure energy consumption using JoularJX, assuming it accurately reflects the energy consumed by each test method. However, JoularJX's minimum time resolution is 1 millisecond, meaning that test methods executing in less than this time frame cannot be reliably recorded or analyzed. We assume such short-running methods naturally consume little energy and are unlikely to contribute meaningfully to energy-saving opportunities. Nonetheless, this highlights a limitation of the measurement tool itself: JoularJX does not offer perfect precision, and its constraints may affect the accuracy of our energy consumption estimates at the test-method level. However, we assume that this impact is consistent across all experiments and does not significantly bias our comparative analysis.

External Validity. Our study is conducted on a limited number of open-source Java projects using JUnit. Therefore, the generalizability of our findings to other programming languages, testing frameworks (e.g., PyTest, Mocha), or non-Java systems may be limited. Additionally, the results may not hold for proprietary or industrial projects with different test structures or constraints. Future replication studies should verify our findings.

7 Conclusion and future work

As the software engineering community increasingly seeks to minimize the environmental impact of software development practices, in this study we attempt to address the repeated execution of large test suites, for example in continuous integration (CI) environments. Our assumption being that even small improvements in the energy efficiency of test code can lead to substantial cumulative savings. As such, in this exploratory study, we investigated the potential of an LLM to reengineer unit tests to improve their energy efficiency. Through a study involving 8 open-source GitHub projects, we have observed that 8.2% of the 9,705 test cases exhibited lower energy consumption after LLM-based reengineering (RQ1) and the corresponding reduction in test effectiveness is limited (RQ2). To deepen our understanding of the reengineered test cases, we further investigated what patterns the LLM applies to make the test code more energy efficient (RQ3).

Ultimately, our study shows that reengineering test code for energy efficiency through an LLM is currently not delivering major gains in terms of energy consumption. This is why we have carried out an additional investigation into what knowledge an LLM can potentially have when it comes to energy efficiency. Through an analysis of StackOverflow posts and GitHub development artifacts we found that energy efficiency related information on those two major development platforms is scarce, giving a potential indication as to why LLMs might not have enough training data to be effective at reengineering for energy efficiency.

[2] https://github.com/torvalds/linux/blob/d7b8f8e2/drivers/staging/media/atomisp/i2c/ov2722.h
[3] https://github.com/torvalds/linux/commit/a79a588fc
[4] https://github.com/Genymobile/scrcpy/issues/6208
[5] https://github.com/nodejs/node/blob/1effb26/deps/v8/include/v8-isolate.h

Our conjecture that optimizing repeatedly executed test code can save energy remains. For future work, we propose integrating runtime energy feedback into the LLM workflow during code optimization. This approach offers two benefits: first, by incorporating energy differences directly into the prompt, LLMs can iteratively refine their outputs, analogous to gradient descent; second, over time, this interaction generates a rich dataset of code versions and their energy profiles, providing valuable material for future training to improve LLM performance in energy-aware software engineering.

Data Availability

The datasets and research artifacts are available at [6].

References

[1] Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. 2025. Language Models in Software Development Tasks: An Experimental Analysis of Energy and Accuracy. In *Int'l Conf. on Mining Software Repositories (MSR)*. IEEE, 725–736.

[2] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proc. Int'l Conf. on Mining Software Repositories (MSR)*. ACM, 202–206.

[3] Nadine Amsel, Zaid Ibrahim, Amir Malik, and Bill Tomlinson. 2011. Toward sustainable software engineering (NIER track). In *Proc. of the International Conference on Software Engineering (ICSE)*. ACM, 976–979.

[4] Maurício Aniche. 2022. *Effective Software Testing*. Manning.

[5] Maurício Finavaro Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *Proceedings of the 50th Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 414–420.

[6] Anonymous. 2025. Replication kit for SANER 2026 submission: Can LLMs Make Software Testing Greener? An Empirical Study on JUnit Test Energy Reengineering. doi:10.5281/zenodo.17279250

[7] Sebastian Baltes, Christoph Treude, and Stephan Diehl. 2019. SOTorrent: Studying the Origin, Evolution, and Usage of Stack Overflow Code Snippets. In *Proc. Int'l Conf. on Mining Software Repositories (MSR)*. IEEE / ACM, 191–194.

[8] Enrique Barba Roque, Luis Cruz, and Thomas Durieux. 2025. Unveiling the Energy Vampires: A Methodology for Debugging Software Energy Consumption. In *Int'l Conf. on Software Engineering (ICSE)*. IEEE, 655–655.

[9] Erik Blokland, Luís Cruz, and Arie van Deursen. 2025. EDATA: Energy Debugging And Testing for Android. In *IEEE/ACM 12th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 94–104.

[10] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An Empirical Study on the Code Refactoring Capability of Large Language Models. arXiv:2411.02320 [cs.SE] https://arxiv.org/abs/2411.02320

[11] Valerio Cosentino, Javier Luis, and Jordi Cabot. 2016. Findings from GitHub: methods, datasets and limitations. In *Proc. Int'l Conf. on Mining Software Repositories (MSR)*. ACM, 137–141.

[12] Luis Cruz and Rui Abreu. 2017. Performance-Based Guidelines for Energy Efficient Mobile Applications. In *Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*. 46–57.

[13] Luis Cruz and Rui Abreu. 2019. Improving Energy Efficiency Through Automatic Refactoring. *Journal of Software Engineering Research and Development* 7 (Aug. 2019), 2:1–2:9.

[14] Nguyen Ngoc Hai Dang, Tho Quan Thanh, and Anh Nguyen-Duc. 2024. BERTVRepair: On the Adoption of CodeBERT for Automated Vulnerability Code Repair. In *Generative AI for Effective Software Development*. Springer, 173–196.

[15] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. In *Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1449–1461.

[16] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Systems with Applications* 249 (2024), 123602.

[17] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. PETrA: a software-based tool for estimating the energy profile of Android applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 3–6.

[18] European Commission. 2025. *Rolling Plan for ICT Standardisation 2025*. Technical Report. European Commission. https://interoperable-europe.ec.europa.eu/collection/rolling-plan-ict-standardisation/rolling-plan-2025

[19] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2025. Exploring the Capabilities of LLMs for Code-Change-Related Tasks. *ACM Trans. Softw. Eng. Methodol.* 34, 6, Article 159 (July 2025), 36 pages.

[20] Abram Hindle. 2015. Green mining: a methodology of relating software change and configuration to power consumption. *Empir. Softw. Eng.* 20, 2 (2015), 374–409.

[21] Abram Hindle. 2016. Green Software Engineering: The Curse of Methodology. In *Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 46–55.

[22] Dacian I. Jurj, Levente Czumbil, Bogdan Bârgăuan, Andrei Ceclan, Alexis Polycarpou, and Dan D. Micu. 2021. Custom Outlier Detection for Electrical Energy Consumption Data Applied in Case of Demand Response in Block of Buildings. *Sensors* 21, 9 (2021).

[23] Ali Khatami and Andy Zaidman. 2023. Quality Assurance Awareness in Open Source Software Projects on GitHub Analysis Dataset. doi:10.5281/zenodo.8139381

[24] Ali Khatami and Andy Zaidman. 2024. State-of-the-practice in quality assurance in Java-based open source software development. *Software: Practice and Experience* 54, 8 (2024), 1408–1446.

[25] Amy J. Ko, Bryan Dosono, and Neeraja Duriseti. 2014. Thirty years of software problems in the news. In *Proc. Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 32–39.

[26] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Automated Software Engineering* 32, 1 (2025).

[27] Väinö Liukko, Anna Knappe, Tatu Anttila, Jyri Hakala, Juulia Ketola, Daniel Lahtinen, Timo Poranen, Topi-Matti Ritala, Manu Setälä, Heikki Hämäläinen, et al. 2024. Chatgpt as a full-stack web developer. In *Generative AI for Effective Software Development*. Springer, 197–215.

[28] Moritz Mock, Jorge Melegati, and Barbara Russo. 2025. Generative AI for Test Driven Development: Preliminary Results. In *Agile Processes in Software Engineering and Extreme Programming – Workshops*. Springer, 24–32.

[29] Todd Mytkowicz, Peter F Sweeney, Matthias Hauswirth, and Amer Diwan. 2008. Observer effect and measurement bias in performance analysis. Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder.

[30] Felix Nahrstedt, Mehdi Karmouche, Karolina Bargieł, Pouyeh Banijamali, Apoorva Nalini Pradeep Kumar, and Ivano Malavolta. 2024. An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries. In *Proc. Int'l Conf. on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 58–68.

[31] Anh Nguyen-Duc, Pekka Abrahamsson, and Foutse Khomh. 2024. *Generative AI for Effective Software Development*. Springer.

[32] Anh Nguyen-Duc and Dron Khanna. 2024. Value-Based Adoption of ChatGPT in Agile Software Development: A Survey Study of Nordic Software Experts. In *Generative AI for Effective Software Development*. Springer, 257–273.

[33] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *2022 18th International Conference on Intelligent Environments (IE)*. 1–4.

[34] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (2019), 43–55.

[35] Quentin Perez, Romain Lefeuvre, Thomas Degueule, Olivier Barais, and Benoit Combemale. 2024. Software Frugality in an Accelerating World: the Case of Continuous Integration. https://arxiv.org/abs/2410.15816

[36] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, 22–31.

[37] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *Int'l Conf. on Software Maintenance and Evolution (ICSME)*. 275–287.

[38] Nils Purschke, Sven Kirchner, and Alois Knoll. 2025. SpeedGen: Enhancing Code Efficiency through Large Language Model-Based Performance Optimization. In *Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. 1–12.

[39] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[40] Maria Stivala, Iffat Fatima, and Patricia Lago. 2025. Investigating the Use of GitHub Copilot for Green Software. In *Advances and New Trends in Environmental Informatics*. Springer, 219–235.

[41] Roberto Verdecchia, Emilio Cruciani, Antonia Bertolino, and Breno Miranda. 2025. Energy-Aware Software Testing. In *Int'l Conf. on Software Engineering: New Ideas and Emerging Results, (ICSE — NIER)*. IEEE, 101–105.

[42] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol de Vries. 2021. Green IT and Green Software. *IEEE Software* 38, 6 (2021), 7–15.

[43] HP Vinutha, B Poornima, and BM Sagar. 2018. Detection of outliers using interquartile range technique from intrusion dataset. In *Information and decision sciences: Proceedings of the 6th international conference on ficta*. Springer, 511–518.

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