Towards Energy-Aware Software Evolution: A Rule-Based Approach to Source-Code Transformation with an Initial Comparison to LLMs

Carlos Pulido[1,*,†], Mª Ángeles Moraga[1,†], Félix García[1,†] and Fernando Castor[2,†]

[1]Instituto de Tecnologías y Sistemas de Información, Universidad de Castilla-La Mancha, Camino de Moledores, s/n, 13005 Ciudad Real,Castilla-La Mancha, Spain

[2]University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands

Abstract

Empirical studies have shown that source-code transformations can have a significant impact on software energy consumption. However, identifying transformations that are beneficial for energy efficiency is only the first step. This knowledge must also be made actionable so that developers can apply these transformations during software evolution. Recent advances in Artificial Intelligence (AI), particularly Large Language Models (LLMs), offer new possibilities for automating source-code transformations. However, many energy-oriented transformations are relatively simple and well-defined, which raises the question of whether the use of generative models is justified given their additional computational and energy costs. Moreover, since LLMs generate code based on probabilistic predictions, their outputs may vary across executions, making it difficult to ensure controlled, predictable, and reproducible transformations. In this work, we propose a rule-based approach for automatically applying empirically motivated source-code micro-transformations. The approach detects transformation opportunities, applies the corresponding transformations, and verifies the correctness of the resulting code. The current implementation supports two transformations, *Dead Code Elimination* and *Remove Exception Handling*, selected based on their positive energy impact reported in a previous empirical study. We also conduct an initial comparison with an LLM-based approach for applying the same transformations, analyzing transformation correctness, completeness, and the computational resources required by the generative approach. Across 30 independent LLM executions, all produced valid results, although two contained an incomplete *Dead Code Elimination* transformation. The LLM-based approach also required additional token and energy consumption, whereas the rule-based approach applied the transformations deterministically and reproducibly.

Keywords

Energy-Aware Software, Source-Code Transformation, Rule-Based Systems, Large Language Models, Software Evolution

1. Introduction

The increasing demand for computing resources has made energy efficiency an important concern in software development [1]. Software decisions can influence the energy required to execute a program [2], making software evolution an opportunity to consider energy efficiency alongside more traditional software evolution concerns, such as adapting functionality to changing requirements and maintaining the software over time [3]. Source-code transformations provide a means of modifying existing software during its evolution, and can therefore be used to explore how code-level changes may contribute to improving energy efficiency.

Previous work provided empirical evidence on the energy impact of specific source-code micro-transformations [4]. These results provide a basis for identifying transformations that may contribute to improving software energy efficiency. However, obtaining such empirical knowledge is only the first

BENEVOL 2026: The 25th Belgium-Netherlands Software Evolution Workshop, 26-27 November 2026, Madrid, Spain

*Corresponding author.

†These authors contributed equally.

carlos.pulido@uclm.es (C. Pulido); mariaangeles.moraga@uclm.es (M. Á. Moraga); felix.garcia@uclm.es (F. García); f.castor@utwente.nl (F. Castor)

0009-0008-8122-3500 (C. Pulido); 0000-0001-9165-7144 (M. Á. Moraga); 0000-0001-6460-0353 (F. García); 0000-0002-6389-3630 (F. Castor)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

step towards making it useful in practice. Developers still need to identify where these transformations can be applied in their source code and implement the corresponding changes. Thus, a key challenge is to bridge the gap between empirical findings and their practical adoption in software development.

One possible direction is to automate the identification and application of empirically motivated transformations. Given a transformation that has been shown to have a potential positive impact on energy consumption, an automated approach could identify suitable instances in the source code and generate the corresponding changes [5]. Such an approach would reduce the effort required from developers and provide a way to integrate empirical knowledge about energy-efficient transformations into existing software evolution activities.

Different approaches can be considered to automate this process. One possibility is to rely on AI-based techniques, such as Large Language Models (LLMs), which can be guided through instructions and examples to identify potential transformation opportunities and generate the corresponding code changes [6]. Their ability to interpret instructions and adapt to different code contexts provides a high degree of flexibility when applying transformations. However, this flexibility also comes at the cost of control, as the non-deterministic nature of LLMs can make it difficult to ensure that transformations are consistently applied according to the intended specification and may result in incorrect or unintended code changes, commonly referred to as hallucinations [7].

Moreover, using LLMs to perform these transformations introduces an additional energy cost associated with model inference [8]. This raises an important question in the context of energy-aware software evolution: *is the use of an energy-intensive generative model justified when applying small and well-defined source-code micro-transformations whose purpose is precisely to improve software energy efficiency?* The micro-transformations considered in this study were selected based on the empirical results reported by Pulido et al. [4], who evaluated their impact on energy consumption across different contexts. These transformations do not require architectural changes or complex reasoning about the overall structure of the software. Rather, they involve specific and well-defined modifications to existing source-code patterns.

In such cases, a rule-based approach may provide a more lightweight alternative, while offering greater control and reproducibility over how transformations are applied. Rather than relying on an LLM to generate code modifications, this approach encodes the transformation logic explicitly as a set of predefined rules [9]. These rules specify both the conditions that must be satisfied for a transformation to be applicable and the corresponding modifications to the source code. As a result, the same input and rules lead to the same transformation outcome, making the process deterministic, transparent, and reproducible. We therefore adopt a rule-based approach to automate the application and evaluation of empirically motivated source-code micro-transformations.

In this work, we propose a pipeline for the automated application and evaluation of empirically motivated source-code micro-transformations, comprising five stages, as illustrated in Figure 1: (i) detecting transformation opportunities, (ii) applying the corresponding transformations, (iii) verifying the correctness of the resulting code, (iv) evaluating their energy consumption, and (v) recommending the most promising transformations based on the obtained results. Energy consumption is evaluated using the FEETINGS framework [10], which provides a methodology for reliable and reproducible software energy measurement.

The initial evaluation compares the proposed rule-based approach with an LLM-based approach for applying the selected source-code micro-transformations. The results show that the rule-based approach applies these transformations deterministically and reproducibly. The LLM-based approach successfully completed both transformations in 28 out of 30 executions. In the remaining two executions, the *Dead Code Elimination* transformation was incomplete, whereas the *Remove Exception Handling* transformation was successfully applied in all executions. Furthermore, each LLM execution consumed an average of 10.14K input and output tokens, adding to the computational cost of using LLMs. Overall, these results suggest that LLMs offer no clear advantage over rule-based automation for the micro-transformations considered in this study.

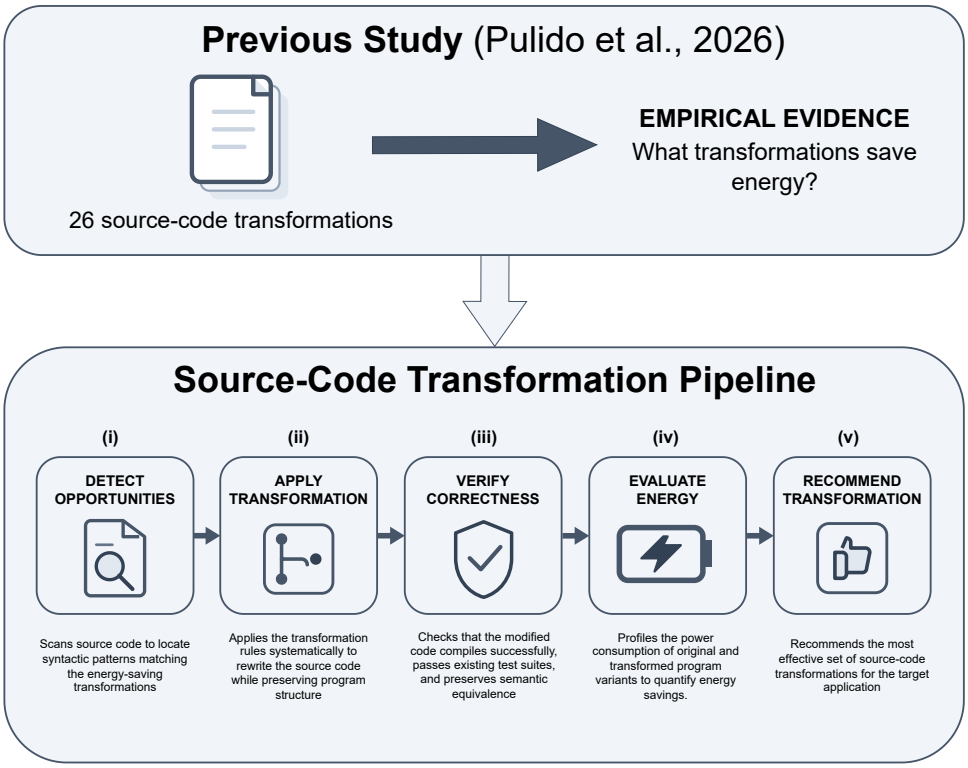


Figure 1: Schematic flow of the energy-aware source-code transformation pipeline

2. Related work

Applying source-code transformations solely because they are known to improve software quality does not necessarily result in lower energy consumption. This highlights the need to first establish their potential energy impact before using them as energy-oriented transformations. In this context, Imran et al. [11] investigated the energy impact of refactoring code smells in cloud software. Their study analyzed 16 code smells and other coding practices across 31 real-world open-source Java and Python applications and showed that refactoring a code smell does not necessarily lead to lower energy consumption. In some cases, refactorings commonly used to address code smells were found to increase energy consumption. Conversely, the selective application of refactorings with a positive energy impact resulted in reductions of up to 13.1% in energy consumption and 5.1% in carbon emissions per workload on average.

Building on the idea of using knowledge about the energy impact of transformations to guide their application, Morales et al. [5] proposed EARMO, an energy-aware refactoring approach for mobile applications. Their study first analyzed the energy impact of eight types of antipatterns in 20 Android applications and found that the effect of refactoring on energy consumption varies across antipattern types. Based on this empirical knowledge, EARMO uses a multi-objective search-based strategy to explore sequences of refactorings that simultaneously improve design quality and energy consumption. Rather than directly measuring the energy consumption of every candidate during the search, the approach estimates the energy impact of individual refactorings using coefficients derived from the previous empirical study. This reduces the cost of repeatedly measuring candidate solutions while allowing energy consumption to be considered during the search. EARMO also checks refactoring preconditions and postconditions to avoid invalid transformation sequences.

A complementary approach is provided by Cruz et al. [12] with Leafactor, a tool for applying energy-oriented refactorings to Android applications. Leafactor relies on predefined refactoring patterns associated with energy efficiency and applies the corresponding transformations when the required patterns are identified in the source code. While EARMO explores combinations of refactorings

according to multiple objectives, Leafactor focuses on the direct application of predefined energy-oriented transformations. Both approaches are specifically designed for Android applications, leaving room for approaches applicable to other application domains.

Previous approaches have relied on specialized tools to automate energy-oriented source-code transformations. More recently, Large Language Models (LLMs) have emerged as an alternative for automating software refactoring. Liu et al. [6] conducted an empirical study of the ability of general-purpose LLMs to identify refactoring opportunities and recommend corresponding refactoring solutions. Using a dataset of 180 real-world refactorings from 20 Java projects, they found that the ability of LLMs to identify refactoring opportunities was initially limited, although providing more specific information about the expected refactoring categories substantially improved their performance. However, some of the generated transformations were unsafe, either changing the functionality of the original code or introducing syntax errors. These findings highlight the potential of LLMs to automate source-code transformations, while also illustrating the challenges of relying on generative models when transformations must be applied reliably and correctly.

Beyond the challenges of automating source-code transformations, another important consideration is whether the expected energy benefits of a transformation remain consistent across different applications and execution contexts. Wang et al. [13] studied the energy impact of software refactorings under different workloads, considering both controlled examples and real-world commits. Their results show that the energy impact of a refactoring can vary depending on the workload and execution context, highlighting the importance of grounding energy-related transformations in empirical evidence. In our work, we build on empirically validated transformations and focus on automating their identification and application in source code. This allows existing empirical knowledge to guide the generation of transformed program variants, rather than applying transformations blindly without prior evidence of their potential energy impact. However, since the effect of a transformation may depend on the specific software and execution context, its energy impact still needs to be evaluated when applied to a particular program. Such evaluation constitutes a subsequent stage of the proposed approach.

3. Automated Source-Code Transformation Pipeline

This section describes the proposed energy-aware source-code transformation pipeline. The pipeline is designed to automate the identification, application, and validation of source-code transformations with previously established potential energy benefits, while generating program variants that can subsequently be evaluated under specific execution conditions. The following subsections first describe the overall transformation pipeline, then present the sample repository used to illustrate and evaluate the current set of supported transformations, and finally compare the proposed approach with an LLM-based alternative for source-code transformation.

3.1. Transformation Pipeline

The proposed approach is organized as a pipeline that progressively transforms the original source code into validated program variants that can be evaluated for their energy consumption. As illustrated in Figure 1, the pipeline comprises five stages:

1. **Transformation opportunity detection.** The system analyzes the source code to identify locations where one or more of the supported transformations can be safely applied. Each transformation defines a set of structural conditions that must be satisfied for it to be applicable. The system traverses the source-code representation and identifies the program elements that match these conditions, producing a set of candidate transformation opportunities.
2. **Transformation application.** The identified transformations are applied to the source code. For each applicable opportunity, the corresponding transformation rules modify the program representation to produce a transformed program variant. When multiple transformations are applicable, different variants can be generated by considering individual transformations and

compatible combinations. This allows alternative implementations of the same program to be generated rather than committing to a single transformation sequence.

3. **Correctness verification.** The generated variants are checked to determine whether they preserve the intended behavior of the original program. The transformed source code is compiled and then executed against the available test cases to identify compilation errors or behavioral differences. Variants that do not satisfy these correctness requirements are discarded, ensuring that only valid program variants proceed to the subsequent stages.
4. **Energy evaluation.** The energy consumption of the valid transformed variants is measured under the execution conditions of interest. Each generated variant is executed under a defined workload and measurement configuration, allowing its energy consumption to be compared with that of the original program and with other transformed variants. Since the energy impact of a transformation may depend on the execution context, these measurements provide the evidence required to determine its actual effect for a particular program and workload.
5. **Transformation recommendation.** The energy measurements obtained for the generated variants are used to identify the transformations that provide the most promising energy benefits for the given context. The measured results can be compared across variants to determine which transformations, or combinations of transformations, achieve the lowest energy consumption while preserving program correctness.

The proposed approach comprises five stages. The first three—transformation opportunity detection, transformation application, and correctness verification—are carried out by the rule-based system and together make up the refactoring process. The resulting variants are then evaluated for energy consumption, and the obtained measurements are used to identify and recommend the transformations that provide the greatest energy benefits for a given context.

The transformations supported by the approach are motivated by empirical evidence from our previous work [4], which identified transformations with a potential positive impact on energy consumption. However, their actual benefits may vary depending on the execution context. Furthermore, several transformations may be applicable to the same code, and applying one transformation may affect whether others can still be applied. For this reason, the system does not simply apply all identified transformations independently. Instead, it explores different transformation sequences, checking the applicability of the remaining transformations after each application. This allows it to generate variants containing individual transformations as well as valid combinations of transformations. Finally, these variants are evaluated under the target execution conditions to determine which transformations provide the greatest energy benefits.

Currently, the system implements two transformations [4] from the candidate catalog (see Figure 2): *Dead Code Elimination* and *Remove Exception Handling*. Both transformations were selected based on the positive energy impact observed across all evaluated contexts in previous empirical study [4]. The current implementation supports these transformation rules for both Java and Python, two of the most popular programming languages according to the PYPL PopularitY of Programming Language Index[1]. Their application is defined through explicit rules that specify the conditions under which each transformation can be safely applied and the corresponding modifications to the source code.

- The first transformation, *Dead Code Elimination*, targets code that does not contribute to the program's observable behavior, including unused local variables[2] and dead stores[3]. When such code can be safely identified, the corresponding declarations or assignments are removed from the source code. This transformation therefore eliminates unnecessary computations and memory operations while preserving the program's behavior. The transformation is applied only when the analysis can determine that removing the identified code does not affect subsequent computations or other observable effects.

[1] https://pypl.github.io/PYPL.html
[2] https://rules.sonarsource.com/java/RSPEC-1481/
[3] https://rules.sonarsource.com/java/RSPEC-1854/

- The second transformation, *Remove Exception Handling*, targets cases where exceptions are used to handle expected or otherwise manageable situations that can be addressed without actually throwing an exception. Throwing and handling an exception incurs runtime and energy costs associated with identifying the appropriate handler, unwinding the stack, and transferring control, even when the exception and its handling logic are simple. The transformation replaces this exception-based control flow with an alternative that preserves the intended behavior without requiring the exception to be thrown. As illustrated in Figure 2, when a `raise` statement within a `try` block can be associated with a specific `except` branch, the system replaces the `raise` statement with the statements executed by the corresponding handler. Depending on the context, `break` or `continue` may also be used to replace the exception-based control flow.

greencoderefactor-python/output/operations.py

```
@@ -1,19 +1,10 @@
def test():
    a = 5
-   b = 2
-   b = 3
    b = 10
    c = 20
    d = 2
-   y = a + 50
-   z = y * 2
-   message = "Hello world"
-   pi = 3.14159
    sum_result = a + b
    subtraction = c - b
    multiplication = sum_result * d
-   division = c // d  # Integer division as in Java
-   temp = multiplication + 10
-   flag = False
-   rate = 1.5
+   division = c // d
    return multiplication + subtraction + division
```

(a) Example of the *Dead Code Elimination* transformation

greencoderefactor-python/output/process_data.py

```
@@ -1,18 +1,11 @@
from operations import operate

- class MyException(Exception):
-     def __init__(self, message):
-         super().__init__(message)
-
def process_data(x, data):
    result = [0] * len(data)
    for i in range(len(data)):
-         try:
-             if x == 1:
-                 result[i] = data[i] * 2
-             else:
-                 raise MyException("x is not 1")
-         except MyException:
+         if x == 1:
+             result[i] = data[i] * 2
+         else:
              result[i] = data[i] * 3
    operate()
    return result
```

(b) Example of the *Remove Exception Handling* transformation

Figure 2: Examples of implemented transformations in the energy-aware source-code transformation pipeline

The transformations are implemented through static source-code analysis based on Abstract Syntax Trees (ASTs). The system first parses each source file into an AST, where the program is represented as a structured hierarchy of syntactic elements, including functions, control-flow constructs, statements, expressions, and assignments. Each transformation traverses this representation to identify nodes that satisfy its transformation rules and modifies, removes, or replaces the corresponding nodes when the required conditions are met. Once the transformation has been applied, the modified AST is converted back into source code, producing a transformed program variant.

3.2. Illustrative Case Study

To illustrate and evaluate the proposed transformation pipeline, we use a custom-made Python repository specifically created for this evaluation (see the experimental package [14] for further details). The repository contains code patterns that allow both *Dead Code Elimination* and *Remove Exception Handling* to be applied together, producing a single transformed variant. This variant is verified to ensure that program behavior is preserved before comparing its energy consumption with that of the original implementation. The experiments were conducted on the Device Under Test (DUT) described in Table 1.

Table 1
Specifications of the Device Under Test (DUT)

Component	Specification
Motherboard	Gigabyte H410M S2H V2
Processor	Intel Core i7-10700, 2.9 GHz
RAM	2 × 8 GB DDR4 Crucial, 2666 MHz, CL19
Graphics Card	ATI Sapphire Radeon X1950 GT
Storage	Western Digital Blue SSD, 500 GB
Power Supply	3Go PS580S, 580 W
Operating System	Ubuntu 20.04.2 LTS

The evaluation uses a synthetic benchmark based on a custom-made Python repository rather than a real-world software system. Following the Green Software Measurement Process (GSMP) principles described by Feetings et al. [10], the original implementation and the transformed variant were executed under the same experimental conditions, with 30 independent runs per version. Energy consumption was measured using `perf` and the `energy-pkg` event, which records CPU package energy consumption through the RAPL interface. In addition, the energy cost of applying the rule-based transformation approach was measured across 30 runs, resulting in a total energy consumption of 23.31 J and an average consumption of 0.777 J per run. Figure 3 presents the energy consumption results for the original and transformed implementations, showing an average energy reduction of 2,193.16 J (77.72%).

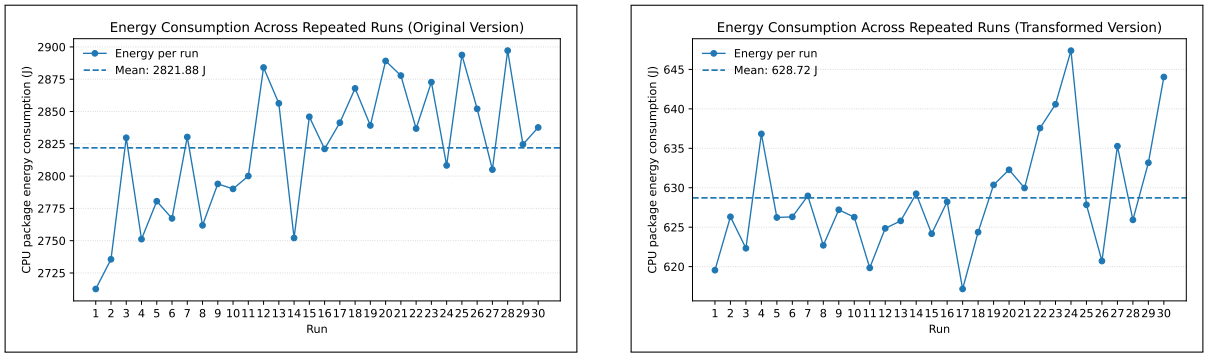


Figure 3: Energy consumption across 30 measurement runs for the original and transformed versions of the sample repository.

3.3. Comparison to LLM-based approach

To compare the proposed transformation pipeline with an LLM-based approach, we evaluated the ability of an LLM to perform the targeted source-code transformations on the sample repository. The experiments were performed using OpenCode[4] with the Big Pickle model[5], with 30 independent executions. The Big Pickle model provides a context window of 200,000 tokens and supports up to 32,000 output tokens, with reasoning enabled. It is a stealth model, meaning that its underlying architecture, training details, and number of parameters are not publicly disclosed. OpenCode was used in its agentic workflow, allowing the model to interact with the source repository through tool calls rather than simply generating a response to a single prompt. This corresponds to a full-fledged agentic approach, in which the model can iteratively inspect the repository, modify source files, and execute commands as part of a control loop. In each execution, the agent received a structured prompt (see Table 2) defining the two considered transformations and the conditions for their application. The complete prompt is available in the experimental package [14]. Before each execution, the repository was reset to its original

[4]https://opencode.ai/
[5]https://pi.dev/models/opencode/big-pickle

state, and the resulting repository was subsequently inspected to determine whether the requested transformations had been successfully applied.

Table 2
Structure of the prompt used for the LLM-based transformation experiment

Section	Description
Transformations	*Dead Code Elimination* and *Remove Exception Handling*
Examples	Representative before-and-after examples for each transformation
Constraints	Apply only transformations and preserve program behavior and coding style
Scope	Analyze all Python files in the specified repository directory
Modification rules	Modify only files in the specified directory
Output	Summary of modified files and applied transformations

Each resulting version was stored independently together with the model output and token statistics. The transformation results were then validated to determine whether each of the requested transformations had been fully applied. Table 3 summarizes the resulting validation outcomes.

Table 3
Validation of transformations generated by OpenCode

Aspect	Result
Iterations	30
Valid	30 (100%)
Fully transformed	28 (93.3%)
Incomplete *Dead Code Elimination* transformations	2 (6.7%)
Incomplete *Remove Exception Handling* transformations	0 (0%)

All 30 executions produced valid repositories. In 28 executions (93.3%), no additional applicable transformation was identified. Two executions (6.7%), iterations 23 and 27, contained an incomplete *Dead Code Elimination* transformation. In both cases, the same unused local variable, `y`, remained. No incomplete *Remove Exception Handling* transformations was detected. The token consumption of each OpenCode execution was also recorded. Across the 30 executions, OpenCode consumed 159.60K input tokens and 144.70K output tokens.

Table 4
LLM token consumption across the 30 executions

Metric	Value
Mean input tokens / iteration	5.32K
Mean output tokens / iteration	4.82K
Mean input + output / iteration	10.14K
Total input tokens	159.60K
Total output tokens	144.70K
Cache read tokens	1.98M

The mean consumption was 10.14K input and output tokens per execution, with 5.32K input tokens and 4.82K output tokens on average. OpenCode also reported 1.98M cache-read tokens. These cached tokens are reported separately because they represent reused context rather than newly generated input or output tokens. The experiments used the Big Pickle model through OpenCode, which was available free of charge during the experiments. Therefore, the actual monetary cost of the 30 executions was $0. For comparison, we also calculated a market-reference cost using the median blended LLM API price reported by the LLM API Price Index.[6] As shown in Table 5, the 304.30K input and output tokens consumed in our experiment correspond to an equivalent reference cost of approximately $0.51. We also estimated the associated LLM energy consumption using a published energy-per-token reference

[6] https://llmcostlab.com/price-index/

of 0.15 J/token.[7] Based on the 304.30K tokens consumed, this corresponds to approximately 45,645 J, or 12.68 Wh.

Table 5
Actual cost and reference estimates for LLM usage

Metric	Value
Actual monetary cost	$0.00
Total input + output tokens	304.30K
Median market reference price	$1.69 / 1M tokens
Equivalent reference cost	≈ $0.51
Reference energy per token	0.15 J/token
Estimated LLM energy	≈ 45,648 J

4. Threats to Validity

Several factors may affect the validity and generalizability of the results presented in this work. Regarding *external validity*, the evaluation was conducted on a single, relatively small and simple Python repository and considered only two source-code transformations. Although the repository provides a controlled setting for evaluating their application, its characteristics may not be representative of larger or more diverse real-world software systems. The limited complexity of the repository may also have reduced the difficulty of the transformation task, potentially making incomplete or incorrect transformations less likely than in more complex systems. Larger systems may require more extensive code analysis and reasoning, which could also result in higher token consumption and energy usage. The comparison with an LLM-based approach was further limited to a single model, Big Pickle, accessed through OpenCode, and to a specific prompt. Different models or prompts may produce different transformation outcomes and incur different token and energy costs. Consequently, these results should be interpreted as an initial comparison between the proposed rule-based approach and an LLM-based alternative rather than as a general assessment of LLM-based source-code transformation.

Regarding *internal validity*, the energy measurements were obtained under a fixed hardware, software, and workload configuration. Since energy consumption can be influenced by these factors, the absolute values may differ under other execution conditions. However, both the original and transformed versions were evaluated under the same conditions, allowing their relative energy consumption to be compared consistently. To account for measurement variability, each version was executed 30 times independently, and the mean energy consumption was calculated from these measurements. This reduces the impact of transient factors, e.g., OS scheduler or garbage collector, on the measurement results. The use of repeated observations also supports the reliability of the sample mean according to the Central Limit Theorem, as the sampling distribution of the mean becomes increasingly stable as the number of independent observations increases [15].

5. Discussion

For the micro-transformations evaluated in this study, the rule-based approach provides a straightforward and reproducible way of defining the conditions under which each transformation can be applied and the corresponding code modifications. This makes the transformation process predictable and avoids variability between executions [9]. The comparison with the LLM-based approach shows that LLMs can also perform these transformations, but do not offer a clear advantage over the rule-based approach in this scenario. The LLM-based approach successfully completed both transformations in 28 out of 30 executions. In the remaining two executions, *Dead Code Elimination* was not fully applied, whereas *Remove Exception Handling* was successfully applied in all executions. This variability

[7] https://ojs.aaai.org/index.php/AAAI/article/view/40535

is consistent with previous studies reporting that LLM-generated refactorings may be incomplete or unsafe in some cases [6].

Unlike classical code transformations, energy-aware transformations require evaluating not only their correctness and completeness, but also their actual impact on execution energy. A transformation may preserve program behavior and be applied correctly without necessarily reducing energy consumption. Furthermore, when LLMs are used to generate transformations, their computational cost must also be considered, as the energy consumed during inference may offset some of the potential savings achieved during program execution. Therefore, comparing rule-based and LLM-based approaches requires considering both the energy impact of the resulting code and the additional cost of generating the transformations with an LLM. This difference is particularly relevant in our experiment, where each LLM-based execution required an average of 10.14K input and output tokens. The energy consumption associated with LLM inference was estimated using an external energy-per-token value rather than measured directly, and should therefore be interpreted only as an indication of the additional computational cost of the LLM-based approach. A more complete comparison would require measuring the energy consumed during transformation generation and program execution under equivalent experimental conditions.

Although previous work has investigated the energy impact of these specific transformations [4], their environmental benefits should be considered in the broader context of the factors that influence software energy consumption. This study does not compare the impact of these transformations with that of broader changes, such as switching the programming language [16] or changing the runtime environment [17]. Assessing the relative energy impact of these alternatives is beyond the scope of this study and remains an important direction for future research. Consequently, our findings should be interpreted in terms of the automation and reliability of applying the investigated transformations, as well as the computational cost associated with the LLM-based approach, rather than as evidence of their environmental benefits relative to other software changes.

The findings of this study do not suggest that LLM-based approaches are preferable to rule-based approaches for the micro-transformations investigated. However, LLMs may offer benefits for more complex transformations, such as macro-transformations, that are difficult to formalize as predefined rules. While rule-based techniques provide control and reproducibility when transformation logic can be explicitly specified, LLMs may offer greater flexibility for transformations that require reasoning across different parts of a codebase. Recent work has explored combining both paradigms. Dilhara et al. [18] combine LLMs with "Transformation by Example", using LLM-generated variants to extend the examples from which transformation rules are inferred. Similarly, Zhao et al. [19] combine LLMs with rule-based static analysis to guide code optimization. These studies illustrate potential ways of integrating both approaches. Future work should investigate whether LLM-based or hybrid approaches provide advantages for macro-transformations compared with rule-based techniques.

6. Conclusions

This work presented a rule-based approach for automating the application of empirically motivated source-code transformations with the goal of supporting energy-aware software evolution. The approach organizes the transformation process into five stages: detecting transformation opportunities, applying the corresponding transformations, verifying the correctness of the resulting code, evaluating their energy consumption, and recommending the most promising transformations. The current implementation automates the first three stages, providing a controlled and reproducible mechanism for generating and validating transformed program variants.

The approach was demonstrated through two transformations, *Dead Code Elimination* and *Remove Exception Handling*, selected based on their positive energy impact in our previous empirical study [4]. By encoding the transformation logic as explicit rules, the approach provides control over when and how transformations are applied and avoids the generative inference required by LLM-based approaches for these well-defined transformations. The resulting variants can then be used for energy evaluation

under the desired execution conditions.

The energy impact of a transformation cannot be assumed to remain constant across applications, workloads, or execution environments. Therefore, the positive effects reported in previous empirical studies should be treated as evidence of potential benefits rather than as guarantees for every application context. Integrating energy measurement into the pipeline would allow transformed variants to be evaluated under their actual execution conditions and would provide the basis for selecting transformations according to their observed energy impact. Such integration would also strengthen the connection between transformation generation and energy-aware decision making.

Building on this direction, another priority is to extend the transformation catalog beyond the two transformations currently supported. Incorporating additional empirically motivated transformations would increase the range of opportunities that the system can identify. The comparison with LLM-based transformation approaches should also be extended beyond the current sample repository by considering different repositories, models, and prompts. This would provide a more comprehensive assessment of the trade-offs between explicit rule-based transformations and more flexible generative approaches in terms of transformation correctness, applicability, reproducibility, computational cost, and energy consumption. Finally, the recommendation stage could be extended to make transformation selection context-aware. Rather than relying only on energy benefits reported in previous studies, the system could use the results obtained for a specific application and workload to determine which transformations are most promising in that context. The approach should also be evaluated on a broader set of real-world software systems and programming languages to assess its generalizability beyond the current evaluation.

Acknowledgments

This work has been supported by the following projects: HERMESG (PID2025-170371OB-C31) funded by MICIU/AEI/10.13039/501100011033; and SINERGIA (2025-GRIN-38310), funded within the framework of the UCLM Research Plan and co-financed by the European Regional Development Fund (FEDER).

References

[1] S. U. Lee, N. Fernando, K. Lee, J.-G. Schneider, A survey of energy concerns for software engineering, Journal of Systems and Software 210 (2024) 111944. doi:10.1016/j.jss.2023.111944.

[2] O. Poy, M. Á. Moraga, F. García, C. Calero, Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study, Journal of Systems and Software 222 (2025) 112303. doi:10.1016/j.jss.2024.112303.

[3] K. Bennett, Software evolution: past, present and future, Information and Software Technology 38 (1996) 673–680. doi:10.1016/0950-5849(96)01116-0.

[4] C. Pulido, F. García, M. Á. Moraga, M. Baños-González, J. A. Rico-Gallego, J. Corral-García, Evaluating the Energy Efficiency of Optimization Techniques in C, Python, and Java, Software: Practice and Experience 56 (2026) 1221–1250. doi:10.1002/spe.70096.

[5] R. Morales, R. Saborido, F. Khomh, F. Chicano, G. Antoniol, EARMO: An Energy-Aware Refactoring Approach for Mobile Apps, IEEE Transactions on Software Engineering 44 (2018) 1176–1206. doi:10.1109/TSE.2017.2757486.

[6] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, H. Liu, Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study, Automated Software Engineering 32 (2025). doi:10.1007/s10515-025-00500-0.

[7] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, Z. Zheng, LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation, Proceedings of the ACM on Software Engineering 2 (2025) ISSTA022:481–ISSTA022:503. doi:10.1145/3728894.

[8] M. F. Argerich, M. Patiño-Martínez, Measuring and Improving the Energy Efficiency of Large

Language Models Inference, IEEE Access 12 (2024) 80194–80207. doi:10.1109/ACCESS.2024.3409745.

[9] E. Visser, A survey of strategies in rule-based program transformation systems, Journal of Symbolic Computation 40 (2005) 831–873. doi:10.1016/j.jsc.2004.12.011.

[10] J. Mancebo, C. Calero, F. Garcia, M. A. Moraga, I. Garcia-Rodriguez de Guzman, FEETINGS: Framework for Energy Efficiency Testing to Improve Environmental Goal of the Software, Sustainable Computing: Informatics and Systems 30 (2021) 100558. doi:10.1016/j.suscom.2021.100558.

[11] A. Imran, T. Kosar, J. Zola, M. F. Bulut, Towards Sustainable Cloud Software Systems through Energy-Aware Code Smell Refactoring, in: 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), 2024, pp. 223–234. doi:10.1109/CLOUD62652.2024.00034, iSSN: 2159-6190.

[12] L. Cruz, R. Abreu, J.-N. Rouvignac, Leafactor: improving energy efficiency of Android apps via automatic refactoring, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft '17, IEEE Press, Buenos Aires, Argentina, 2017, pp. 205–206. doi:10.1109/MOBILESoft.2017.21.

[13] H. Wang, H. Li, S. H. Tan, Understanding the Energy Impact of Software Refactoring: A Workload-Aware Study of Controlled Examples and Real-World Commits, 2026.

[14] C. Pulido, refactorem, A rule-based approach to source-code transformation, Zenodo, 2026. doi:10.5281/zenodo.23692369, version 1.1.1.

[15] E. Kern, L. M. Hilty, A. Guldner, Y. V. Maksimov, A. Filler, J. Gröger, S. Naumann, Sustainable software products—Towards assessment criteria for resource and energy efficiency, Future Generation Computer Systems 86 (2018) 199–210. doi:10.1016/j.future.2018.02.044.

[16] A. Gordillo, C. Calero, M. Á. Moraga, F. García, J. P. Fernandes, R. Abreu, J. Saraiva, Programming languages ranking based on energy measurements, Software Quality Journal 32 (2024) 1539–1580. doi:10.1007/s11219-024-09690-4.

[17] M. Stocker, M. Wahler, Energy-aware decision making in software stack upgrades, Journal of Systems and Software 241 (2026) 112963. doi:10.1016/j.jss.2026.112963.

[18] M. Dilhara, A. Bellur, T. Bryksin, D. Dig, Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example, Proceedings of the ACM on Software Engineering 1 (2024) 29:631–29:653. doi:10.1145/3643755.

[19] Y. Zhao, Y.-A. Xiao, Q. Xiao, Z. Zhang, Y. Xiong, SemOpt: LLM-Driven Code Optimization via Rule-Based Analysis, ACM Transactions on Software Engineering and Methodology (2026). doi:10.1145/3820167.