

# Balancing Green and Clean Code: Prompting for LLM-Based Refactoring

Feray Gulu-zada 

University of Tartu, Estonia

Hina Anwar 

University of Tartu, Estonia

---

## Abstract

---

**Background.** LLMs are increasingly used for code refactoring, but their effects on multiple software quality dimensions remain unclear. Green coding targets energy efficiency, while clean coding emphasizes readability and maintainability. These objectives may interact or conflict, making LLM-based refactoring a multi-objective problem.

**Aims.** This study investigates whether prompting can guide LLM-based refactoring toward balanced green and clean code outcomes. We examine how different prompting strategies affect energy consumption, runtime performance, and maintainability.

**Method.** We conduct a controlled empirical study on 15 human-written Python scripts using four LLMs and five prompting strategies: clean, green, multi-objective, clean-to-green staged, and green-to-clean staged prompting. Generated refactorings are validated for correctness before analysis. Valid outputs are evaluated using CPU energy consumption, runtime, maintainability index, cyclomatic complexity, source lines of code, and Halstead volume. We analyze results using descriptive statistics, statistical tests, correlation analysis, bootstrap confidence intervals, and Pareto-efficiency analysis.

**Results.** Of 300 generated refactorings, 267 passed validation. Among valid outputs, 53.18% (142 of 267) reduced energy consumption, but the overall energy effect was small, variable, and not statistically significant across prompting strategies. Energy and runtime changes showed a moderate positive correlation, while maintainability showed weaker relationships with efficiency metrics. Prompting strategies significantly affected maintainability but not energy or runtime. Clean and multi-objective prompting produced the most balanced Pareto outcomes, though these differences were mainly driven by maintainability.

**Conclusions.** LLM-based refactoring can sometimes reduce energy consumption without necessarily degrading maintainability, but its effects are inconsistent and depend on model choice, prompt design, and correctness validation. Prompting is useful for steering LLM-based refactoring, but current evidence supports cautious, validated use rather than assuming reliable multi-objective optimization.

**2012 ACM Subject Classification** Software and its engineering → Software evolution

**Keywords and phrases** Large language models, code refactoring, energy efficiency, maintainability, green software, prompt engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.29

**Category** Technical Track Paper

**Supplementary Material**

*Software (Source Code and Data):* <https://figshare.com/s/274a16ea23303aacfdcf>

**Acknowledgements** This work was carried out as part of the first author's thesis research.

## 1 Introduction

Large Language Models (LLMs) are increasingly used in software engineering tasks such as code generation, refactoring, and optimization. Their growing adoption positions them not only as productivity tools, but also as automated agents that can influence software



© Feray Gulu-zada and Hina Anwar;  
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 29; pp. 29:1–29:20



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

quality attributes such as maintainability, performance, and energy efficiency. As developers increasingly rely on LLM-generated code transformations, it becomes important to understand not only whether these transformations improve code, but also which quality attributes they improve and which they may degrade.

Software quality has traditionally been associated with *clean code*, which emphasizes readability, simplicity, and maintainability [13]. More recently, *green software* has emerged as a complementary concern, focusing on reducing energy consumption as one contributor to the environmental impact of software systems [33]. As software systems scale, even small inefficiencies in frequently executed code can accumulate into substantial energy costs. Sustainability, therefore, represents an increasingly important dimension of software quality, particularly for code that is executed repeatedly or deployed at scale.

However, clean and green code objectives are not always aligned. Optimizations that improve runtime or energy efficiency may reduce readability or maintainability, while refactorings that improve clarity may introduce computational overhead [5, 26, 23]. This creates a multi-objective refactoring problem in which improving one quality attribute may come at the expense of another. Such trade-offs are especially relevant for LLM-based refactoring, where developers may accept generated changes without fully understanding their effects across multiple quality dimensions.

Existing research has investigated LLMs for code generation, repair, refactoring, performance optimization, and maintainability improvement. In parallel, empirical software engineering research has studied energy-aware programming and trade-offs among performance, readability, and maintainability. However, these lines of work remain only partially connected. Existing studies often evaluate clean-code or green-code objectives separately, leaving limited empirical evidence on whether LLM-based refactoring can be guided to balance these objectives simultaneously. In particular, it remains unclear how prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability.

Prompting provides a lightweight mechanism for steering LLM behavior. Prior work shows that prompt structure, context, and stated objectives can substantially influence generated code [36, 20, 23]. Unlike fine-tuning or specialized optimization tools, prompting can be applied directly across models without modifying their internals, making it a practical intervention for studying multi-objective LLM-based refactoring. Nevertheless, LLM-generated transformations are useful only if they preserve the behavior of the original program. Therefore, functional correctness must be treated as a prerequisite before assessing non-functional quality changes.

The goal of this paper is to empirically evaluate how LLMs behave when asked to refactor existing Python programs under different optimization objectives. We use the term *refactoring* to refer to LLM-generated code transformations intended to preserve input-output behavior while improving non-functional properties. We conduct a controlled study using 15 human-written Python scripts, four LLMs, and five prompting strategies: clean, green, multi-objective, clean-to-green staged prompting, and green-to-clean staged prompting. The generated outputs are first validated for correctness and then evaluated using CPU energy consumption, runtime, maintainability index, cyclomatic complexity, source lines of code, and Halstead volume metrics. We analyze the resulting trade-offs using descriptive statistics, statistical tests, correlation analysis, bootstrap confidence intervals, and Pareto-efficiency analysis. This study makes the following contributions:

- an empirical evaluation of LLM-generated refactorings across energy consumption, runtime performance, and maintainability;
- a comparison of single-objective, multi-objective, and staged prompting strategies for steering LLM-based refactoring;

- an analysis of trade-offs among clean and green code objectives, showing how model choice, prompt design, and correctness validation shape sustainable and maintainable LLM-based refactoring.

The remainder of this paper is organized as follows. Section 2 reviews related work on LLM-based refactoring, green-code optimization, and prompting. Section 3 describes the study design, including research questions, dataset, models, prompts, metrics, validation, and analysis. Section 4 presents the results, Section 5 discusses the findings, Section 6 addresses threats to validity, and Section 7 concludes the paper.

## 2 Related Work

This section summarizes prior work on LLM-based refactoring, energy-aware code generation, and prompt engineering for code improvement. Prior studies were identified through database search and backward/forward snowballing.

### 2.1 LLM-Based Clean-Code Refactoring

LLMs have shown promise for automated refactoring and maintainability improvement. Prior studies report that Copilot resolved 87.1% of Python code smells [36], GPT improved code quality by 20% [15], produced effective refactorings for most Java code segments [9], and matched human recommendations in 63.6% of cases [14]. Structured and multi-agent LLM systems further improve reliability: CORE achieved up to 76.8% fix rates while reducing unintended changes by 25.8% [34], a Haskell-based multi-agent system reduced complexity by 11% and improved quality by 22% [30], and RefAgent achieved a 90% unit-test pass rate while reducing code smells by 52.5% [19]. These studies show that LLMs can support clean-code refactoring, but their effectiveness varies across models, tasks, and evaluation settings.

### 2.2 LLM-Based Green-Code Optimization

Research on LLMs for energy-efficient code reports more mixed results. Studies evaluating Copilot, GPT-4o, o1-mini, Code Llama, DeepSeek-v3, Gemini-1.5-Pro, and related models show that LLM-generated code often underperforms human-written baselines in runtime or energy consumption, although occasional parity or marginal gains appear under specific conditions [31, 32, 10, 33, 7, 1]. Prompt-based energy optimization is similarly inconsistent: general efficiency prompts often yield negligible improvements [33, 3, 7, 1], while structured, tagged, or iterative prompting can improve efficiency in some cases [27, 2, 23, 12, 11]. However, iterative energy optimization may also reduce correctness when not guided by execution feedback [35]. These findings suggest that LLMs do not reliably optimize energy efficiency without explicit guidance and validation.

### 2.3 Prompt Engineering for Code Improvement

Prompt engineering is central to LLM-based code improvement because small changes in task framing, constraints, role specification, examples, and feedback can substantially affect generated code. In refactoring studies, iterative, role-based, contextual, constrained, few-shot, and task-specific prompts have been shown to reduce complexity, lines of code, code smells, and maintainability issues, while also improving correctness, runtime efficiency, and memory usage in some settings [29, 6, 30, 14, 18, 36]. Prior work also shows that prompt formulation and task framing strongly affect LLM refactoring performance [20, 22].

For energy-oriented tasks, prompting evidence is less stable. Generic requests for efficient or green code often produce negligible gains, whereas more structured prompts, prompt tags, iterative refinement, and explicit optimization cues can improve energy or runtime in specific contexts [33, 3, 7, 1, 27, 2, 23, 12, 11]. However, these gains may come with correctness risks when outputs are not validated [35]. This motivates treating prompt design and functional validation as linked concerns in empirical studies of LLM-based refactoring.

Prior work shows that LLMs can improve maintainability and, in some settings, runtime or energy efficiency. However, these objectives are typically studied separately, despite potential trade-offs between clean and green code. Although prompting, constraints, feedback, and multi-agent coordination affect LLM-based code improvement [14, 18, 36, 6, 29, 4, 30, 34, 8, 28, 21], limited evidence exists on how prompting strategies shape trade-offs among maintainability, runtime, and energy efficiency in refactoring. This study addresses that gap by comparing single-objective, multi-objective, and staged prompting strategies across clean and green code metrics.

### **3 Methodology**

This section describes the empirical study used to evaluate whether LLM-based refactoring can improve energy efficiency, runtime performance, and maintainability of existing Python programs. We present the research questions, study design, dataset, models, prompts, validation procedure, metrics, measurement setup, and analysis methods.

#### **3.1 Research Questions**

The following research questions guide this study:

- **RQ1:** To what extent can LLM-based refactoring improve the energy efficiency of existing Python code?
- **RQ2:** How do prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability?
- **RQ3:** Which prompting strategies yield the most balanced outcomes across green and clean code metrics?

RQ1 focuses on energy changes, RQ2 on interactions among efficiency and maintainability metrics, and RQ3 on identifying prompting strategies with favorable multi-objective trade-offs.

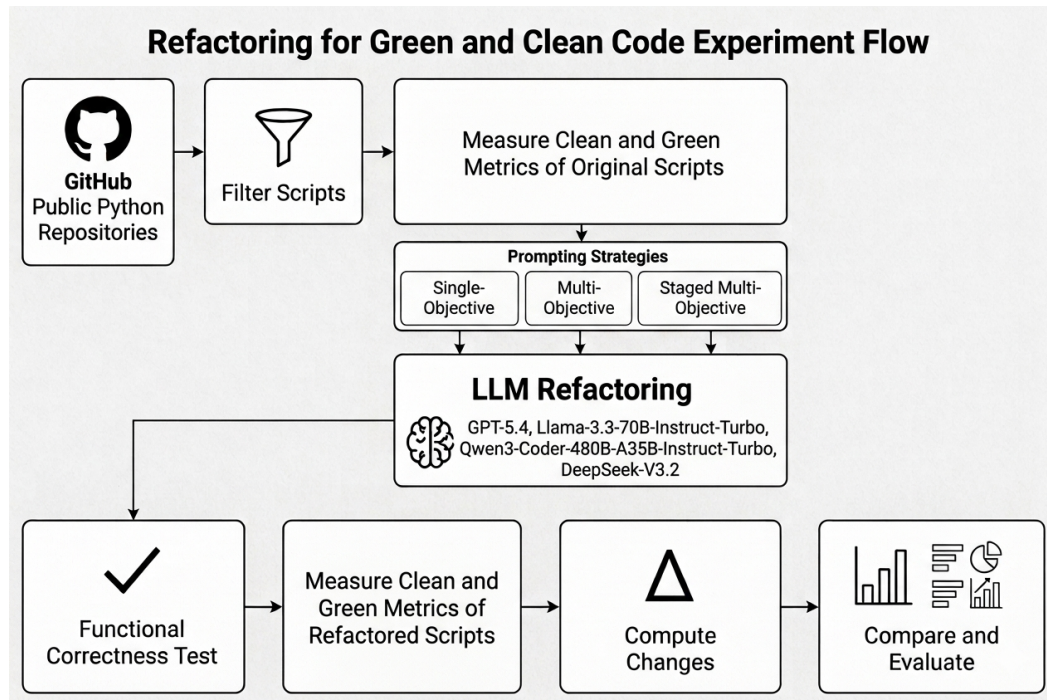
#### **3.2 Study Design**

We conduct a controlled empirical study in which existing Python programs are refactored using multiple LLMs under different prompting strategies. Each original program serves as its own baseline. The independent variables are the LLM and prompting strategy; the dependent variables are energy consumption, runtime performance, and maintainability-related metrics. Functional correctness is treated as a prerequisite for metric comparison.

Figure 1 summarizes the experimental pipeline. For each script, we collect baseline measurements, generate refactored versions using each model and prompt, validate correctness, measure valid outputs, and compare the resulting metrics against the baseline.

#### **3.3 Dataset and Task Definition**

The dataset consists of 15 human-written Python scripts manually selected from public GitHub repositories containing standalone implementations of classical algorithms and data structures. Repositories were manually inspected, and scripts satisfying the predefined



■ **Figure 1** Overview of the experimental pipeline.

inclusion criteria were selected for the study. The scripts implement classical data structures and algorithms (e.g., trees, graph algorithms, dynamic programming, and search algorithms), providing deterministic computational workloads with varying control flow and algorithmic complexity. These characteristics make them suitable for controlled comparisons of runtime, CPU energy consumption, and maintainability. Scripts were selected to satisfy the following inclusion criteria: they contained 50–200 lines of code, three to eight user-defined functions, at least one conditional branch and one iterative construct, implemented computationally significant tasks (e.g., graph algorithms, data structure manipulation, dynamic programming, recursive algorithms, or search algorithms), and exhibited at least one maintainability issue, such as elevated cyclomatic complexity, a low Maintainability Index, or code smell warnings reported by Pylint. Scripts that require external dependencies, interactive input, external services, or complex runtime environments were excluded to reduce measurement noise. The dataset size was kept modest to support controlled, repeated energy measurements and correctness validation.

For scripts without a fixed entry point or predefined inputs, we added a standardized `main()` function and deterministic input values. These execution harnesses enabled automated execution, correctness validation, and consistent runtime and energy measurement.

We use *refactoring* to refer to LLM-generated code transformations intended to preserve input-output behavior while improving non-functional properties. Since LLMs may alter behavior, all generated outputs are validated before metric analysis.

### 3.4 LLMs

We evaluate four transformer-based LLMs representing instruction-tuned and code-oriented models:

- **Instruction-tuned models:** GPT-5.4 and Llama-3.3-70B-Instruct-Turbo.
- **Code-oriented models:** Qwen3-Coder-480B-A35B-Instruct-Turbo and DeepSeek-V3.2.

■ **Table 1** Prompting strategies evaluated in the study.

| Strategy | Type             | Primary objective                |
|----------|------------------|----------------------------------|
| clean    | Single-objective | Maintainability/readability      |
| green    | Single-objective | Runtime/energy efficiency        |
| multi    | Multi-objective  | Maintainability, runtime, energy |
| stagedcg | Staged           | Clean $\rightarrow$ green        |
| stagedgc | Staged           | Green $\rightarrow$ clean        |

GPT-5.4 was accessed through the OpenAI API, while the remaining models were accessed through DeepInfra. All generations were conducted between April 15 and May 8, with the temperature set to 0.0 to reduce randomness in generation.

### 3.5 Prompting Strategies

Prompting is the main experimental treatment. We evaluate five strategies representing single-objective, multi-objective, and staged optimization. Each strategy asks the model to preserve input-output behavior, avoid external dependencies, and return only refactored code. The prompting strategies evaluated in this study are summarized in Table 1.

The **clean** strategy targets structural quality by reducing complexity, unnecessary lines of code, and Halstead Volume. The **green** strategy targets runtime and CPU energy consumption. The **multi** strategy combines both objective sets into a single prompt. The staged strategies apply the objectives in a sequential order: **stagedcg** applies clean then green optimization, while **stagedgc** applies the reverse order. Only the final second-stage output is included in the main analysis; intermediate outputs are retained for traceability.

All prompts share a common structure: role definition, task description, optimization goals, behavior-preservation and dependency constraints, and an output-format instruction. The complete set of prompt templates is available in the replication package.

```
You are a software engineer specializing in [objective].
Task: Refactor the code to improve [target qualities].
Constraints:
- Do not add external dependencies.
- Preserve input-output behavior.
- Limit changes to internal computation logic.
Optimization goals: [strategy-specific goals]
Output: Return ONLY the refactored code.
Code to refactor: [code]
```

### 3.6 Correctness Validation

Before measuring non-functional properties, each refactored program is validated against the corresponding original implementation. Both versions are executed using the same deterministic inputs, which were fixed before model generation and reused across all model-prompt combinations for a given script. A refactoring is valid only if it executes successfully and produces output equivalent to the baseline.

Generated outputs are excluded if they fail correctness validation. The number of generated, excluded, and valid refactorings is reported in Section 4.

■ **Table 2** Metrics used to evaluate original and refactored programs.

| Category | Metric                | Tool                             | Unit          | Improvement Direction |
|----------|-----------------------|----------------------------------|---------------|-----------------------|
| Green    | Runtime               | <code>time.perf_counter()</code> | seconds       | Lower                 |
| Green    | CPU energy            | Intel RAPL/pyRAPL                | $\mu\text{J}$ | Lower                 |
| Clean    | Maintainability Index | Custom Calculation               | score (0–100) | Higher                |
| Clean    | Cyclomatic Complexity | Lizard                           | count         | Lower                 |
| Clean    | SLOC                  | Radon                            | lines         | Lower                 |
| Clean    | Halstead Volume       | Radon                            | –             | Lower                 |

### 3.7 Metrics

We evaluate each original and refactored program using green and clean code metrics. Green metrics capture execution efficiency, while clean metrics capture structural maintainability. Table 2 summarizes the metrics, tools, and optimization direction.

Maintainability Index is computed as follows:

$$MI = \max \left( 0, \frac{171 - 5.2 \ln(HV) - 0.23 \cdot CC_{\text{total}} - 16.2 \ln(SLOC)}{171} \times 100 \right) \quad (1)$$

where  $HV$  is Halstead Volume,  $CC_{\text{total}}$  is the total Cyclomatic Complexity, calculated as the sum of the per-function complexity values reported by Lizard, and  $SLOC$  is Source Lines of Code. We use the normalized Maintainability Index formulation adopted by Microsoft Visual Studio, which yields values on a 0–100 scale, with higher values indicating better maintainability [17].

### 3.8 Static Analysis Tools

We use three static analysis tools to extract code quality metrics from the original and refactored programs. **Radon** is a Python code analysis tool used to compute Source Lines of Code (SLOC) and Halstead Volume [25]. **Lizard** is a complexity analysis tool used to measure Cyclomatic Complexity of the analyzed programs [16]. **PyLint** is a Python static analysis tool used to identify maintainability-related warnings [24].

### 3.9 Execution Environment

All experiments were executed on a single HP EliteBook 8 G1i 14-inch notebook running Ubuntu 24.04.4 LTS, with an Intel(R) Core(TM) Ultra 5 228V processor, 8 logical cores, 30 GB RAM, 8 GB swap, and Intel integrated graphics. To reduce measurement noise, the machine was connected to power, non-essential applications and network-intensive tasks were closed, and automatic updates were disabled.

### 3.10 Runtime and Energy Measurement

Runtime and energy consumption were measured for both original and refactored programs. Each execution was pinned to a single physical CPU core using Linux CPU affinity through `taskset`. Each script was executed 30 times, and the complete measurement procedure was repeated five times per script. A 60-second cooldown interval was enforced between measurement repetitions. Final runtime and energy values were computed using the mean across repeated runs.

### 3.11 Analysis Method

For each metric, percentage change is computed relative to the original implementation:

$$\Delta M = \frac{M_{\text{refactored}} - M_{\text{original}}}{M_{\text{original}}} \times 100 \quad (2)$$

For energy consumption, runtime, Cyclomatic Complexity, SLOC, and Halstead Volume, negative values indicate improvement. For the Maintainability Index, positive values indicate improvement.

We summarize the results using the mean, median, and standard deviation, grouped by model and prompting strategy. Spearman’s rank correlation coefficients were used to analyze relationships among changes in energy, runtime, and maintainability. Spearman’s method was selected because the percentage-change variables exhibited skewed distributions and contained outliers, making a rank-based correlation more appropriate. We use the Kruskal–Wallis test to compare prompting strategies. When significant effects are observed, we perform post-hoc pairwise comparisons using Dunn’s test with Holm correction to adjust for multiple testing. Uncertainty in the estimates is assessed using bootstrap confidence intervals for mean percentage changes across strategies.

To identify balanced solutions, we perform Pareto-efficiency analysis. A refactoring is Pareto-efficient if no other refactoring improves at least one objective without worsening another.

All analyses were conducted in Python using NumPy, Pandas, SciPy, Matplotlib, and Seaborn.

## 4 Results

This section reports the results for 300 generated refactorings. We first summarize validation outcomes and failure patterns, then present the results for each research question. Only refactorings that passed correctness validation and completed measurement are included in the energy, runtime, maintainability, and Pareto analyses, consistent with the validation procedure described in Section 3.6.

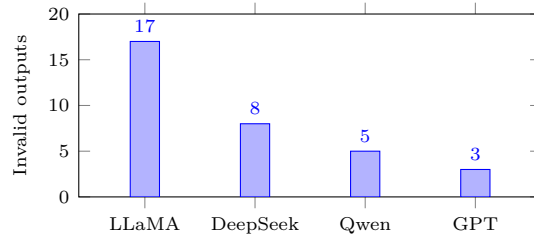
Table 3 summarizes the validation outcomes. Out of 300 generated refactorings, 267 passed validation and were included in the final analysis. The remaining 33 outputs were excluded: 31 failed correctness validation, one timed out during correctness evaluation, and one timed out during energy measurement.

■ **Table 3** Validation outcomes for generated refactorings.

| Outcome              | N (%)       |
|----------------------|-------------|
| Valid refactorings   | 267 (89.0)  |
| Correctness failures | 31 (10.3)   |
| Correctness timeout  | 1 (0.3)     |
| Energy timeout       | 1 (0.3)     |
| Total                | 300 (100.0) |

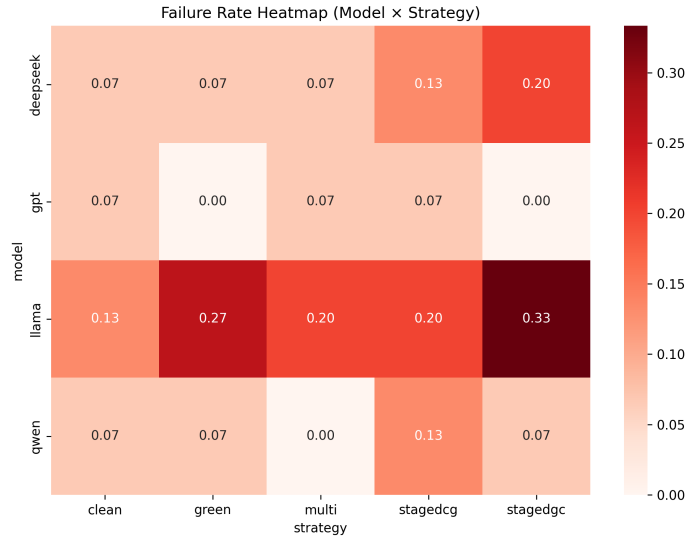
### 4.1 Correctness Failures

Before analyzing non-functional quality changes, we examine invalid outputs by model, prompting strategy, and base script to identify where correctness-preserving refactoring was most difficult. As shown in Figure 2, invalid outputs are unevenly distributed across models. LLaMA accounts for the largest share, followed by DeepSeek, Qwen, and GPT, indicating higher instability in LLaMA-generated refactorings.



■ **Figure 2** Invalid outputs across models, including correctness failures and timeouts.

Figure 3 shows that failures are relatively balanced across prompting strategies, although staged strategies, especially `stagedgc` and `stagedcg`, account for more failures. LLaMA shows consistently higher failure rates across prompting strategies, whereas GPT, DeepSeek, and Qwen remain more stable with generally low error rates.



■ **Figure 3** Failure-rate heatmap across models and prompting strategies.

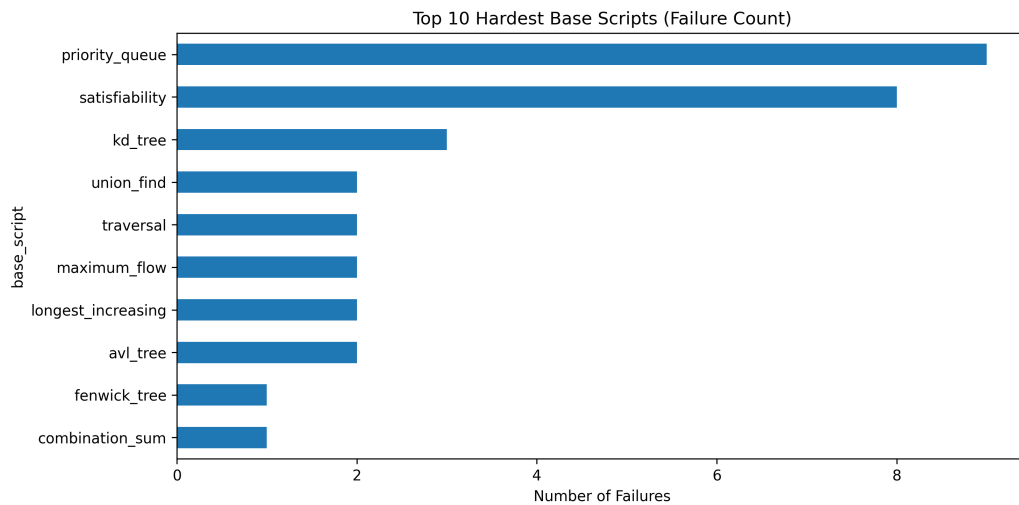
At the base-script level, Figure 4 shows that failures are concentrated in computationally complex tasks such as `priority_queue` and `satisfiability`. This suggests that algorithmic complexity affects the likelihood of correctness-preserving refactoring.

Overall, the validation results indicate that model choice and task complexity contribute more strongly to failure patterns than prompting strategy alone. This supports treating functional correctness as a prerequisite for non-functional metric analysis.

## 4.2 RQ1: Energy Efficiency Improvements

RQ1 asks to what extent LLM-based refactoring improves the energy efficiency of existing code. We analyze percentage changes in energy consumption relative to the original implementations across the 267 valid refactorings.

Overall, 53.18% (142 of 267) of valid refactorings reduced energy consumption. However, the effect size is small and highly variable: the mean energy change is 0.50%, the median is -0.11%, and the standard deviation is 11.59 percentage points. Thus, slightly more than half of the valid refactorings reduce energy consumption, but the overall effect remains close to zero.



■ **Figure 4** Top 10 most failure-prone base scripts.

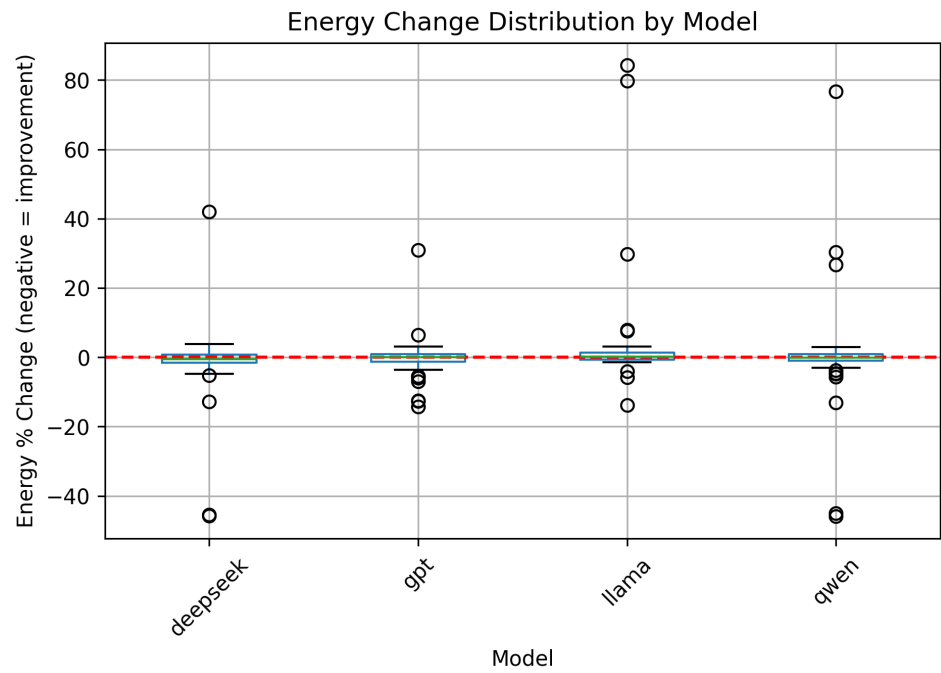
Figure 5 shows the distribution of energy change by model. DeepSeek achieves the strongest average reduction (-1.46%), followed by GPT (-0.36%). Qwen is approximately neutral (0.28%), while LLaMA shows the largest average increase in energy consumption (4.23%), with several extreme positive outliers.

Figure 6 shows the corresponding distribution by prompting strategy. The **stagedgc** strategy achieves the strongest average reduction (-1.45%), followed by **green** (-0.96%) and **multi** (-0.49%). In contrast, **clean** (1.92%) and **stagedcg** (3.56%) increase energy consumption on average. This suggests that prompts with explicit efficiency objectives are more likely to reduce energy consumption than maintainability-only prompts.

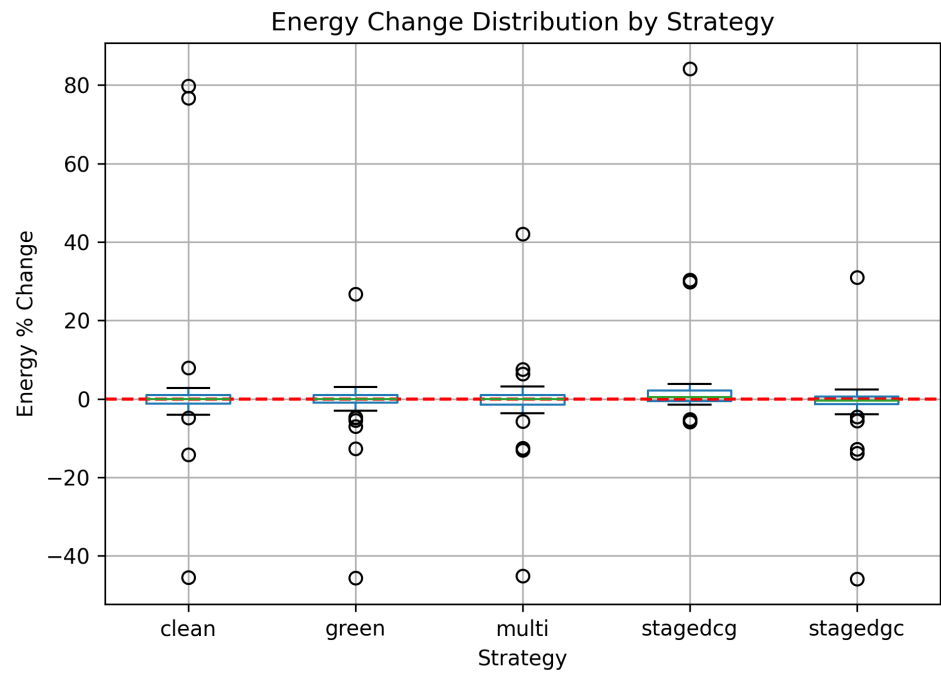
Figure 7 shows the interaction between the model and the prompting strategy. The best-performing combinations include Qwen with **multi** prompting (-5.11%) and **stagedgc** prompting (-4.59%), and DeepSeek with **clean** prompting (-4.17%) and **green** prompting (-3.80%). However, the widespread combinations indicate that energy outcomes depend on both model choice and prompt design. The boxplots also show several outliers, highlighting substantial variability within combinations, indicating that energy outcomes are highly sensitive to both model choice and prompting strategy, even when they passed correctness validation.

To assess whether energy differences across prompting strategies are statistically significant, we apply a Kruskal-Wallis test. The result is not significant ( $p = 0.52$ ), indicating that energy variations across strategies are not statistically distinguishable. Bootstrap confidence intervals further indicate high variability, with most intervals crossing zero.

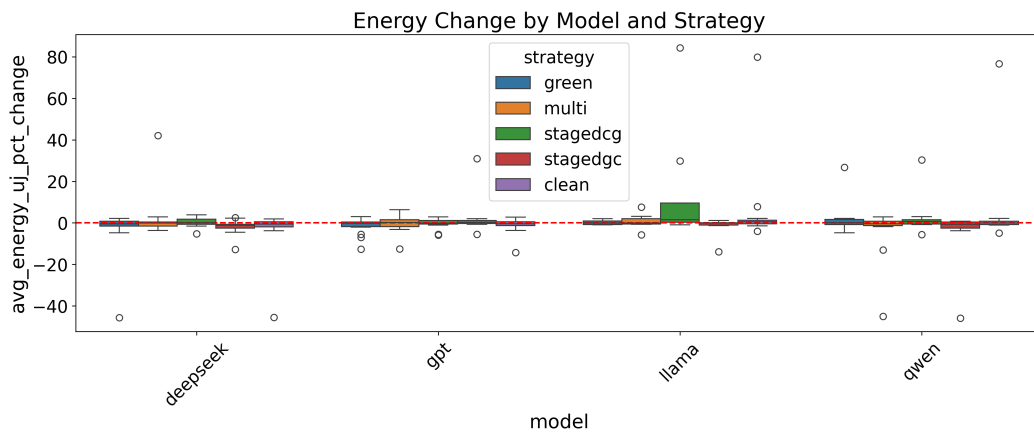
**Summary of RQ1.** LLM-based refactoring reduces energy consumption in a slight majority of valid cases, but the overall effect is small, highly variable, and not statistically significant across prompting strategies. Model choice matters descriptively, with DeepSeek and GPT showing lower mean energy consumption than Qwen and LLaMA.



**Figure 5** Distribution of energy change by model. Negative values indicate reduced energy consumption relative to the original implementation.



**Figure 6** Distribution of energy change by prompting strategy. Negative values indicate reduced energy consumption relative to the original implementation.



■ **Figure 7** Distribution of energy change by model and prompting strategy. Negative values indicate reduced energy consumption relative to the original implementation.

### 4.3 RQ2: Trade-offs Between Metrics

RQ2 examines how prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability. Table 4 compares the median values of the strategies across the three dimensions. Following the metric directions defined in Section 3.7, lower values indicate improvement for energy and runtime, while higher values indicate improvement for maintainability.

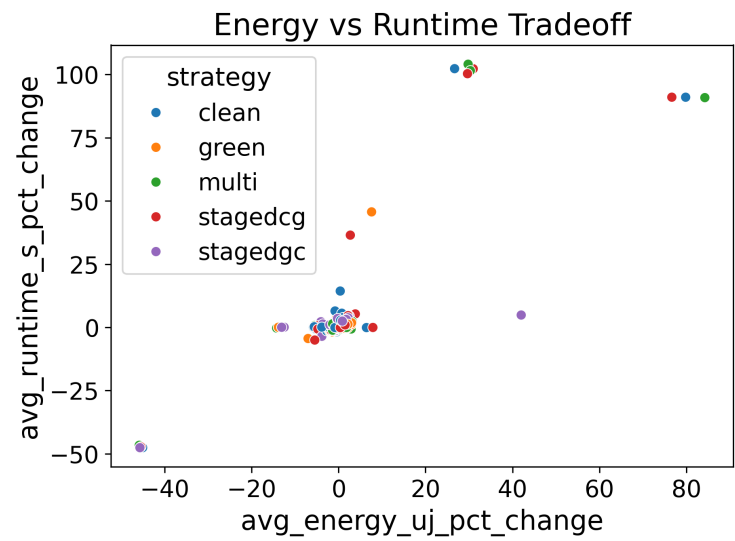
■ **Table 4** Median Energy, Runtime, and Maintainability Changes by Prompting Strategy.

| Strategy | Energy ↓ | Runtime ↓ | Maintainability Index ↑ |
|----------|----------|-----------|-------------------------|
| clean    | -0.10    | 0.14      | 0.37                    |
| multi    | -0.51    | -0.03     | 0.27                    |
| green    | -0.085   | 0.355     | -2.82                   |
| stagedcg | 0.19     | 0.37      | -2.295                  |
| stagedgc | -0.05    | 0.41      | -2.22                   |

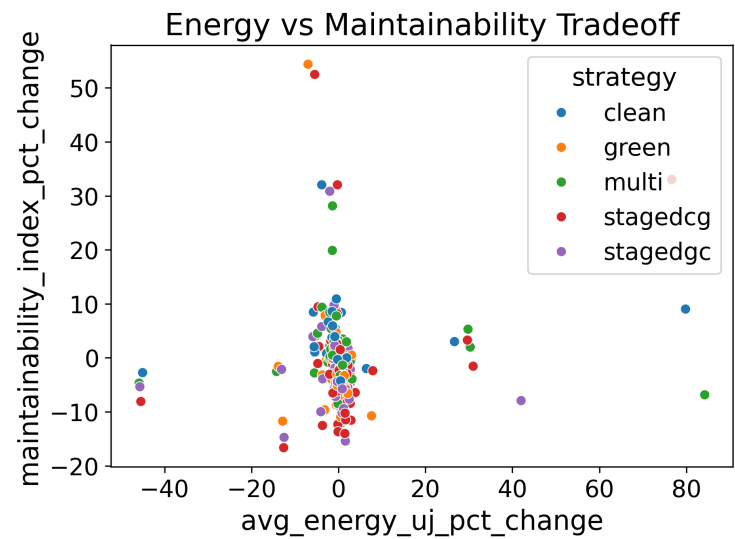
Table 4 shows that `multi` prompting reduces both energy consumption (approximately -0.51) and runtime (approximately -0.03), while achieving moderate maintainability improvement (approximately 0.27). Clean-only prompting yields the highest maintainability gain (0.37), suggesting a trade-off between efficiency and structural quality.

Figure 8 illustrates the relationship between changes in energy consumption and runtime. Spearman’s rank correlation indicates a moderate positive association ( $\rho = 0.50$ ). Because reductions in both energy consumption and runtime are represented by negative percentage changes, this result suggests that refactorings that reduce runtime also tend to reduce energy consumption. In contrast, Figure 9 shows a weak negative association between changes in energy consumption and maintainability ( $\rho = -0.27$ ). Since lower energy consumption and higher Maintainability Index values both represent improvements, the negative correlation indicates a weak tendency for energy-efficient refactorings to improve maintainability as well. However, the small magnitude of the correlation suggests that this relationship is limited and that improvements in the two dimensions are not strongly coupled.

Overall, multi-objective prompting tends to yield comparatively balanced outcomes while preserving a moderate level of structural quality. Clean prompting primarily improves maintainability, whereas green and staged strategies more often reduce maintainability.



■ **Figure 8** Trade-off between energy and runtime. Negative values indicate improvements relative to the original implementation.

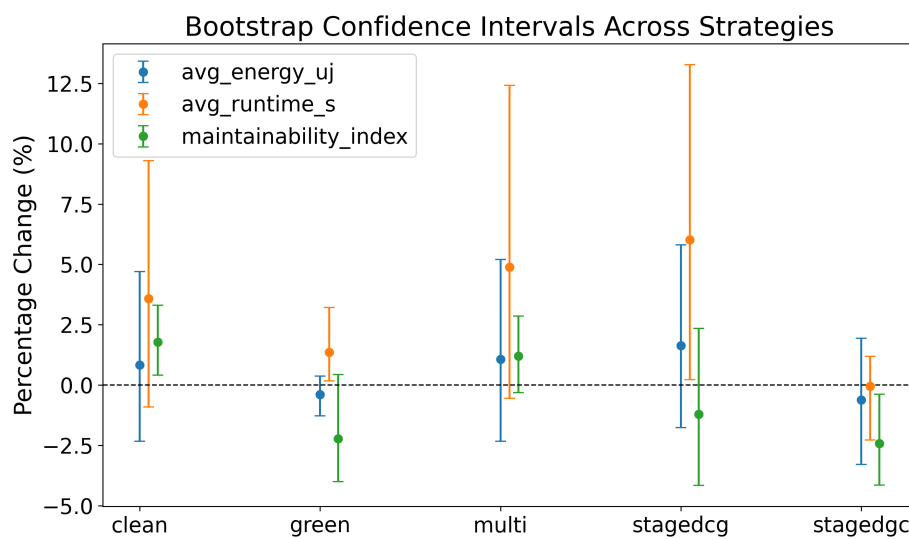


■ **Figure 9** Trade-off between energy and maintainability. Negative energy values indicate reduced energy consumption, while positive maintainability values indicate improved maintainability.

These results suggest that energy/runtime optimization and maintainability improvement are only weakly related, and that explicitly balancing objectives may help avoid extreme degradation in either dimension.

Kruskal–Wallis tests show no significant differences across prompting strategies for energy consumption ( $p = 0.52$ ) or runtime ( $p = 0.68$ ), but maintainability differs significantly ( $p < 0.001$ ). Post hoc pairwise comparisons using Dunn’s test with Holm correction indicate that significant differences in maintainability are primarily observed for **clean** and **multi** prompting compared with the remaining prompting strategies.

Figure 10 shows bootstrap confidence intervals across prompting strategies. The intervals indicate high uncertainty in energy and runtime estimates, with multiple intervals crossing zero. Runtime exhibits particularly wide intervals, while maintainability shows more consistent directional shifts.



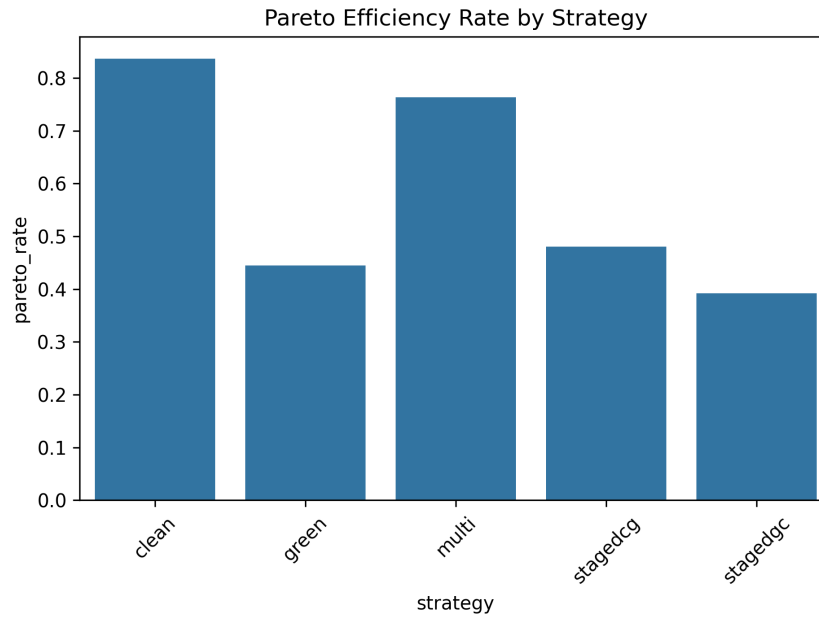
■ **Figure 10** Bootstrap confidence intervals across prompting strategies. Intervals crossing zero indicate uncertainty about the direction of the effect.

**Summary of RQ2.** Energy and runtime improvements tend to occur together, showing a moderate association, while maintainability is only weakly related to energy outcomes. Prompting strategies significantly affect maintainability, while their effects on energy and runtime are descriptive rather than statistically significant.

#### 4.4 RQ3: Balanced Outcomes (Pareto Analysis)

RQ3 examines which prompting strategies yield the most balanced outcomes across green and clean code metrics. We evaluate balance using Pareto efficiency across energy, runtime, and maintainability. Pareto efficiency is used as a descriptive measure of trade-offs between objectives.

As shown in Figure 11, clean prompting achieves the highest Pareto-efficiency rate (0.84), followed by multi-objective prompting (0.76). In contrast, **stagedcg** (0.48), **green** (0.44), and **stagedgc** (0.39) show lower rates of non-dominated solutions.



■ **Figure 11** Pareto efficiency across prompting strategies. Higher values indicate a larger proportion of non-dominated refactorings.

Although Pareto differences are observable, the statistical tests in RQ2 indicate that these differences are mainly driven by maintainability rather than consistent improvements in energy or runtime. Therefore, Pareto rankings should be interpreted as descriptive summaries of multi-objective behavior rather than definitive performance hierarchies.

Across the observed model-level results, clean prompting appears comparatively stable, while multi-objective prompting performs competitively but with higher variance, particularly in maintainability outcomes. Green and staged strategies generally underperform across models. No single prompting strategy dominates all objectives, but clean and multi-objective prompting provide the most consistently balanced trade-offs.

**Summary of RQ3.** Clean and multi-objective prompting provide the most balanced Pareto outcomes. However, because the strongest differences are driven by maintainability rather than energy or runtime, these results should be interpreted descriptively rather than as statistically definitive rankings.

## 5 Discussion

This study examined whether prompting strategies can guide LLM-based refactoring toward balanced improvements in energy efficiency, runtime performance, and maintainability. The results show a distinction between structural code quality and execution-level efficiency. Energy consumption and runtime showed limited and statistically non-robust differences across prompting strategies, whereas maintainability was more consistently affected. Although slightly more than half of the valid refactorings reduced energy consumption, the overall effect was small, highly variable, and not statistically significant across prompting strategies. This suggests that LLM-based refactoring can sometimes improve energy efficiency, but such improvements are not reliable enough to be assumed from prompting alone. The

validation outcomes also show that correctness remains a practical barrier: 33 generated outputs were excluded due to correctness failures or timeouts. This reinforces the need to treat LLM-generated refactorings as candidates requiring validation before non-functional improvements are assessed.

The moderate positive association between energy and runtime changes suggests that these two efficiency metrics are related in our experimental setting. Refactorings that reduced execution time also tended to reduce CPU energy consumption. However, both metrics showed high variability across models, prompts, and scripts. A likely explanation is that energy and runtime are shaped primarily by algorithmic structure, input behavior, and execution-level effects rather than by surface-level code restructuring. Small syntactic or structural changes introduced by LLMs may therefore have inconsistent performance consequences, especially when they do not alter the underlying algorithm.

In contrast, maintainability metrics responded more clearly to prompting. This is expected because LLMs are trained on large corpora of human-written code where naming conventions, modular structure, idiomatic style, and common refactoring patterns are strongly represented. Maintainability-oriented prompts, therefore, align more closely with the models' learned capabilities than energy-oriented prompts. The significant differences in maintainability across strategies suggest that prompt design is useful for steering structural code improvements, even when it is less effective for controlling runtime or energy behavior.

Model-specific differences were also observed. DeepSeek and GPT produced lower average energy consumption than Qwen and LLaMA, while LLaMA generated the highest number of invalid refactorings and the largest average increase in energy consumption. These observations suggest that model selection is an important factor alongside prompt design when applying LLM-based refactoring in practice.

These findings have important implications for multi-objective LLM-based refactoring. Prompt engineering alone appears insufficient for reliably optimizing energy efficiency, especially when no execution feedback is provided during generation. Multi-objective and staged prompts can encourage more balanced outputs, but they do not guarantee improvements across all objectives. In practice, developers should treat LLM-generated refactorings as candidates that require validation, measurement, and review rather than as automatically improved versions of the original code. Correctness validation is particularly important, since invalid outputs cannot meaningfully be evaluated for non-functional quality.

The results are consistent with prior work showing that LLMs are effective at improving readability and maintainability-related aspects of code, but less reliable for performance- and energy-oriented optimization. Our study extends this evidence by showing that even structured prompting does not fully resolve the challenge of jointly optimizing clean and green code objectives. More reliable green refactoring may require approaches that combine prompting with execution feedback, profiling information, search-based optimization, or tool-assisted validation. In our setting, LLMs appear better suited for maintainability-oriented refactoring than for standalone energy optimization.

## **6 Threats to Validity**

Several threats may affect the interpretation and generalizability of our findings. First, LLM outputs can be non-deterministic. We mitigated this by setting the temperature to 0.0, the lowest available setting for each API, but minor variations may still occur. Second, energy measurements based on Intel RAPL can be affected by background activity and hardware state. To reduce this threat, all experiments were run in a controlled environment with

minimal background processes, and each script was executed repeatedly across multiple measurement runs. Third, results may be sensitive to the wording of the prompt. We mitigated this by using standardized prompt templates and varying only the controlled elements required by each prompting strategy.

The external validity of the results is limited by the dataset and model selection. Our dataset consists of curated Python scripts focused on algorithmic tasks, so the findings may not generalize to other programming languages, larger software systems, or industrial codebases with complex dependencies and architectures. We evaluated four current LLMs: GPT-5.4, Llama-3.3-70B-Instruct-Turbo, Qwen3-Coder-480B-A35B-Instruct-Turbo, and DeepSeek-V3.2. Although these models represent relevant current capabilities, newer or differently configured models may produce different results.

Construct validity is mainly affected by the metrics used to approximate maintainability and energy efficiency. Maintainability Index and related static metrics are widely used, but they do not fully capture human-perceived readability or long-term maintainability. Similarly, Intel RAPL measures CPU energy consumption, but not full-system energy use or GPU-related workloads. Therefore, our results should be interpreted as evidence about CPU-level execution energy and static maintainability indicators rather than complete measures of software sustainability or maintainability. Functional correctness was assessed using a predefined test input, which may not capture behavioral differences across all possible inputs. Thus, some refactorings may pass validation while still failing on untested edge cases or boundary conditions.

The statistical conclusions are limited by the relatively small sample size and substantial variability in energy changes, which may reduce the power to detect small differences between prompting strategies. Accordingly, non-significant results should be interpreted as a lack of statistically detectable evidence rather than evidence of no effect.

## 7 Conclusion

This paper investigated whether prompt engineering can guide LLM-based refactoring to improve energy efficiency, runtime performance, and maintainability. We evaluated multiple LLMs under single-objective, multi-objective, and staged prompting strategies and analyzed the resulting refactorings using green- and clean-code metrics.

The results show that LLM-based refactoring does not produce statistically robust or consistent improvements in energy consumption or runtime across prompting strategies. Although some refactorings reduce energy consumption, the effects are small, highly variable, and not statistically significant. In contrast, maintainability is more consistently influenced by prompting, suggesting that prompt design has a stronger impact on structural code quality than on execution-level performance.

Overall, the findings highlight a relative separation between maintainability-oriented and efficiency-oriented optimization in LLM-based refactoring. Prompt engineering can support maintainability improvements, but it is insufficient on its own for reliable energy or runtime optimization. Robust multi-objective LLM-based refactoring will likely require approaches that combine prompting with correctness validation, runtime feedback, profiling, or other tool-supported optimization mechanisms. Future work should evaluate these approaches across larger, more diverse codebases, additional programming languages, and more complex software systems.

## Data Availability Statement

To ensure verifiability and replicability, we provide the replication package of this study, available at <https://figshare.com/s/274a16ea23303aacfdcf>.

---

## References

- 1 R. Apsan, V. Stoico, M. Albonico, R. Dhar, K. Vaidhyanathan, and I. Malavolta. Generating energy-efficient code via large-language models – where are we now? In *ICSE 2026*, 2026. doi:10.1145/3744916.3764522.
- 2 H. Ashraf, S. M. Danish, A. Leivadeas, Y. Otoum, and Z. Sattar. Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable ai programming. *arXiv preprint arXiv:2508.08332*, 2025. doi:10.48550/arXiv.2508.08332.
- 3 T. Cappendijk, P. de Reus, and A. Oprescu. An exploration of prompting llms to generate energy-efficient code. In *GREENS 2025*, pages 31–38, 2025. doi:10.1109/GREENS66463.2025.00010.
- 4 J. Choi, G. An, and S. Yoo. Iterative refactoring of real-world open-source programs with large language models. In *Search-Based Software Engineering*, pages 49–55. Springer, 2024. doi:10.1007/978-3-031-64573-0\_4.
- 5 Tristan Coignon, Clément Quinton, and Romain Rouvoy. When faster isn’t greener: The hidden costs of llm-based code optimization. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1655–1666, 2025. doi:10.1109/ASE63991.2025.00139.
- 6 Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology*, 2026. doi:10.1145/3801158.
- 7 V.-A. Cursaru, L. Duits, J. Milligan, D. Ural, B. R. Sanchez, V. Stoico, and I. Malavolta. A controlled experiment on the energy efficiency of the source code generated by code llama. *arXiv preprint arXiv:2405.03616*, 2024. doi:10.48550/arXiv.2405.03616.
- 8 M. T. Dearing, Y. Tao, X. Wu, Z. Lan, and V. Taylor. Leveraging llms to automate energy-aware refactoring of parallel scientific codes. *arXiv preprint arXiv:2505.02184*, 2025. doi:10.48550/arXiv.2505.02184.
- 9 K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar. Exploring chatgpt’s code refactoring capabilities: An empirical study. *Expert Systems with Applications*, 249:123602, 2024. doi:10.1016/j.eswa.2024.123602.
- 10 M. A. Islam et al. Evaluating the energy-efficiency of the code generated by llms. *arXiv preprint arXiv:2505.20324*, 2025. doi:10.48550/arXiv.2505.20324.
- 11 S. Garg, R. Zilouchian Moghaddam, and N. Sundaresan. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077*, 2025. doi:10.48550/arXiv.2306.17077.
- 12 R. Jonnala, J. Yang, Y. Lee, G. Liang, and Z. Cao. Measuring and improving the efficiency of python code generated by llms using cot prompting and fine-tuning. *IEEE Access*, 13:1–10, 2025. doi:10.1109/ACCESS.2025.3585742.
- 13 S. H. Kannangara and W. M. J. I. Wijayanake. An empirical evaluation of impact of refactoring on internal and external measures of code quality. *International Journal of Software Engineering and Applications*, 6(1):1–12, January 2015.
- 14 B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu. Exploring the potential of general-purpose llms in automated software refactoring: An empirical study. *Automated Software Engineering*, 32:26, 2025. doi:10.1007/s10515-025-00500-0.
- 15 Y. Liu, T. Le-Cong, R. Widayarsi, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM*, 2023.

- 16 Lizard Developers. Lizard: A cyclomatic complexity analyzer. <https://pypi.org/project/lizard/>, 2026.
- 17 Microsoft. Code metrics - maintainability index range and meaning. Microsoft Learn, 2025. URL: <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=visualstudio>.
- 18 H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves. Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects. In *IEEE SANER*, pages 1–10, 2025. doi:10.1109/SANER64311.2025.00069.
- 19 K. Oueslati, M. Lamothe, and F. Khomh. Refagent: A multi-agent llm-based framework for automatic software refactoring. *arXiv preprint arXiv:2511.03153*, 2025. doi:10.48550/arXiv.2511.03153.
- 20 G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana. An exploratory study on architectural smell refactoring using large language models. In *IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 1–10, 2025. doi:10.1109/ICSA-C65153.2025.00070.
- 21 H. Peng, A. Gupte, N. J. Eliopoulos, C. C. Ho, R. Mantri, L. Deng, W. Jiang, and Y.-H. Lu. Large language models for energy-efficient code: Emerging results and future directions. *arXiv preprint arXiv:2410.09241*, 2024. doi:10.48550/arXiv.2410.09241.
- 22 Y. C. K. Piao, J. C. Paul, L. D. Silva, A. M. Dakhel, M. Hamdaqa, and F. Khomh. Refactoring with llms: Bridging human expertise and machine understanding. *arXiv preprint arXiv:2510.03914*, 2025. doi:10.48550/arXiv.2510.03914.
- 23 S. Podder, H. Date, and S. Murthy. An empirical study: Leveraging prompt engineering with ai coding assistants to develop energy-efficient code, 2025.
- 24 Pylint Developers. Pylint documentation. <https://pylint.readthedocs.io/>, 2026.
- 25 Radon Developers. Radon documentation. <https://radon.readthedocs.io/>, 2026.
- 26 P. Rani, J.-A. Bard, J. Sallou, A. Boll, T. Kehrler, and A. Bacchelli. Can we make code green? understanding trade-offs in llms vs. human code optimizations. *arXiv preprint arXiv:2503.20126*, 2025. doi:10.48550/arXiv.2503.20126.
- 27 R. Rubel, A. Moussaid, C. Di Sipio, and D. Di Ruscio. Prompt engineering and its implications on the energy consumption of large language models. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1–10, 2025. URL: <https://ieeexplore.ieee.org/document/11039291>.
- 28 J. Scheffler, M. Westing, and V. Wohlgemuth. Eneragent: Energy-aware code refinement with llm-based agents. In *EnviroInfo*, pages 111–118, 2025.
- 29 A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe. Refactoring programs using large language models with few-shot examples. *arXiv preprint arXiv:2311.11690*, 2023. doi:10.48550/arXiv.2311.11690.
- 30 S. Siddeeq, M. Waseem, Z. Rasheed, M. M. Hasan, J. Rasku, M. Saari, H. Terho, K. Mäkelä, K.-K. Kemell, and P. Abrahamsson. Llm-based multi-agent system for intelligent refactoring of haskell code. *arXiv preprint arXiv:2506.19481*, 2025. doi:10.48550/arXiv.2506.19481.
- 31 L. Solovyeva, S. Weidmann, and F. Castor. Ai-powered, but power-hungry? energy efficiency of llm-generated code. *arXiv preprint arXiv:2502.02412*, 2025. doi:10.48550/arXiv.2502.02412.
- 32 J. F. Tuttle, D. Chen, A. Nasrin, N. Soto, and Z. Zong. Can llms generate green code: A comprehensive study through leetcode. In *IEEE Green and Sustainable Computing Conference (IGSC)*, pages 1–10, 2024. doi:10.1109/IGSC64514.2024.00017.
- 33 T. Vartziotis, I. Dellatolas, G. Dasoulas, M. Schmidt, F. Schneider, T. Hoffmann, S. Kotsoopoulos, and M. Keckeisen. Learn to code sustainably: An empirical study on llm-based green code generation. *arXiv preprint arXiv:2403.03344*, 2024. doi:10.48550/arXiv.2403.03344.
- 34 N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani. Core: Resolving code quality issues using llms. *Proceedings of ACM Software Engineering*, 1(FSE), 2024. doi:10.1145/3643762.

- 35 S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024. doi:10.48550/arXiv.2407.14044.
- 36 B. Zhang, P. Liang, Q. Feng, Y. Fu, and Z. Li. Copilot-in-the-loop: Fixing code smells in copilot-generated python code using copilot. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, pages 1–5, 2024.