UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Fidan Süleymanova

Code Smarter Not Harder: Measuring Energy Efficiency of LLM-Generated Code

Master's Thesis (30 ECTS)

Supervisor:
Hina Anwar, PhD

Tartu 2025

Code Smarter Not Harder: Measuring Energy Efficiency of LLM-Generated Code

Abstract:

As large language models (LLMs) become more integrated into software development, questions arise not only about correctness but also about the energy behavior of the code they generate. This thesis investigates how LLM-generated code compares to human-written solutions in terms of runtime and energy efficiency, and if minimal prompting can improve the energy profile of the output. Using four real-world programming tasks which is implemented in Python, JavaScript, and Rust, we benchmarked code produced by three LLMs, GPT-4o, LLaMA-3.3-Instruct, and Qwen 2.5-Coder against curated human-written submissions. Energy and runtime were measured using low-level CPU profiling tools in a tightly controlled environment. Results show that LLMs can often approach the performance of human-written code, but rarely exceeds it, with GPT-4o delivering the most consistent outcomes. Prompting for energy efficiency had a measurable but inconsistent effect, improving results in some cases while degrading them in others. Statistical analysis found no significant group-level differences between LLM-generated code and human-written code, but small task-level variations were also observed. These findings highlight both the potential and the current limits of prompt-based energy optimization and they provide a reproducible framework for future research in energy-aware code generation.

Keywords: Large Language Models (LLMs), Energy Efficiency, Code Generation, Prompt Engineering, Runtime Performance, Sustainable Computing, Human vs Machine Code, Empirical Software Engineering, Green Software

CERCS: P170 - Computer science, numerical analysis, systems, control

Koodi targemini, mitte raskemini: LLM-idega genereeritud koodi energiatõhususe mõõtmine

Lühikokkuvõte:

Kuna suured keelemudelid (LLM-id) muutuvad tarkvaraarenduses üha enam integreerituks, kerkivad esile küsimused mitte ainult loodud koodi korrektsuse, vaid ka selle energiatõhususe kohta. Käesolev lõputöö uurib, kuidas LLM-ide poolt genereeritud kood võrreldes inimeste kirjutatud lahendustega käitub täitmisaja ja energiatarbimise osas, ning kas minimaalne juhendamine (promptimine) võib parandada väljundi energiaprofiili. Kasutades nelja reaalse maailma programmeerimisülesannet, mis on implementeeritud Pythonis, JavaScriptis ja Rustis, võrdlesime kolme LLM-i - GPT-4o, LLaMA-3.3-Instruct ja Qwen 2.5-Coder - poolt genereeritud koodi valitud inimeste kirjutatud lahendustega. Energiatarbimist ja täitmisaega mõõdeti madala taseme protsessori profileerimise tööriistade abil kontrollitud keskkonnas. Tulemused näitavad, et LLM-id suudavad sageli läheneda inimeste kirjutatud koodi jõudlusele, kuid harva ületavad seda; kõige järjepidevamaid tulemusi andis GPT-4o. Energiatõhususele suunatud juhendamine avaldas mõõdetavat, kuid ebaühtlast mõju - mõnel juhul paranesid tulemused, teistel juhtudel halvenesid. Statistiline analüüs ei näidanud olulisi erinevusi LLM-ide ja inimeste loodud koodi vahel grupi tasandil, kuid ülesande tasandil ilmnesid väikesed erinevused. Tulemused toovad esile nii promptimise potentsiaali kui ka praegused piirangud energiatõhusa koodi genereerimisel ning pakuvad reprodutseeritavat raamistikku edasiseks uurimistööks selles valdkonnas.

Võtmesõnad: Suured keelemudelid (LLM-id), energiatõhusus, koodi genereerimine, promptimise tehnika, täitmisaja jõudlus, jätkusuutlik arvutus, inimese ja masina loodud koodi võrdlus, empiiriline tarkvaratehnika, roheline tarkvara

CERCS: P170, Computer science, numerical analysis, systems, control

Contents

1. Introduction 6
1.1 Problem Statement 7
1.2 Research Goal and Research Questions 8
1.3 Significance and Impact of the Study 8
1.4 Overview of Results and Contributions 9
1.5 Structure of the Thesis 9
2. Related Work 10
2.1 The Role of Energy Efficiency in Software and Code-Level Development 10
2.2 LLM-Generated Code: Characteristics, Efficiency, and the Role of Prompting 11
3. Methodology 14
3.1 Experiment Definition 14
3.2 Experiment Planning 14
3.2.1 Task Selection (Human-written Code) 14
3.2.2 Measurement Tools and Setup 16
3.2.3 Experiment Design 17
3.3 Experiment Execution 21
3.3.1 Experimental Setting 21
3.3.2 Data Collection 23
3.3.3 Statistical Analysis 24
4. Results 25
4.1 Comparison Between Human-Written Code and LLM-Generated Code (RQ1) 26
4.1.1 GPT-4o generated code vs Human-written code 26
4.1.2 LLaMA-3.3-Instruct generated code vs Human-written code 30
4.1.3 Qwen2.5-Code generated code vs Human-written code 33
4.2 Prompt Optimization Effects (RQ2) 36
4.2.1 GPT-4o 37
4.2.2 LLaMA-3.3-Instruct 39
4.2.3 Qwen2.5-Code 43
4.3 Correlation Analysis 46
5. Discussion 48
6. Limitations and Threats to Validity 50
7. Conclusion 52

References 53
License 56

1. Introduction

Software consumes energy. As systems grow, energy consumption becomes harder to ignore. In today's world, applications run across local machines and global cloud platforms. This makes software efficiency concerns not only for the performance but also for sustainability. Manner reported that abstraction-heavy code has led to, what he calls "black software", inefficient systems with rising carbon impact [1].

Meanwhile, large language models (LLMs) are shaping how code is written [2]. Different LLM tools generate valid code from the plain language prompts. Those models are now helping software developers in real projects. They support rapid prototyping and solve common problems automatically. But, as their usage grows, some other questions, beyond correctness, arise. We still know very little about the energy usage of LLM-produced code. Hou et al. [3] have shown that LLM-generated code can match human-written code in terms of accuracy and runtime, but its energy performance remains largely unclear.

Energy-aware coding should no longer be optional. Green Algorithms frameworks quantify the carbon output of computation and link this to software design choices [4]. The programming language choice is also part of this. Pereira et al. found that some slower languages are more energy-efficient, due to memory behavior and compilation effects [5].

In parallel with language-level design, LLMs are rapidly integrating into software workflows. Models like GPT-4o, Llama, and Qwen are now used for generating functions, translating between languages, and even debugging. Recent studies show that LLMs can generate code that is both correct and performant. For example, GPT-4 produced code faster than 85% of human submissions in a competitive setting. However, most of these evaluations focus on accuracy and speed, not energy usage, leaving a gap in understanding their sustainability impact [3, 6].

Even small optimizations in software development can reduce the power draw. Refactoring, batching I/O, and using efficient libraries are known to help. Şanlıalp et al. show that combining simple refactoring can lower energy usage in Java and C programs, which highlights that energy use can be influenced by the way the solution is written [7].

Also, several tools now help to estimate carbon impact. Green Algorithms, for example, quantify emissions from computation based on hardware type and electricity grid data [4]. These methods support a growing field of carbon-aware computing. Yet few are applied to generated code. Programming language choice can also affect energy use directly. Pereira et al. benchmarked 27 languages and found large differences in runtime, memory use, and energy demand [5]. Compiled languages like C and Go performed better than interpreted ones like Python and Ruby.

However, the relationship was not always linear. In some cases, slower code consumed less energy due to more efficient memory access patterns.
Rust is often cited as a highly efficient language, especially for system-level tasks. Python and JavaScript, in contrast, are slower and more power-hungry but remain popular due to ease of use. For this reason, our study does not compare languages directly, but instead it evaluates LLM output and human-written code within each language separately.
Some researchers question whether language alone matters. A recent study showed that when execution time is controlled, energy usage across languages becomes similar. This suggests optimization at the algorithmic and structural level may have an impact more than language choice alone [8].
Beyond programming languages and libraries, prompt design has emerged as another factor influencing energy behavior in generated code. They affect what LLMs generate. Prior work has shown that changing the wording of the prompt can affect not just correctness but also runtime behavior. Cappendijk et al. tested energy optimization prompts, and they found that they sometimes helped to reduce energy usage in some cases, but not consistently across all models and tasks [9].
Ilager et al. proposed a model-level method (GREEN CODE) that reduces energy by controlling when generation stops. This technique decreased LLM inference energy use by up to 50% [10]. While this is not about the structure of generated code, it confirms that efficiency can be optimized during the generation phase. Still, in most settings, users do not deal with modifying inference pipelines. Instead, they rely on prompting. In this thesis, we focus on the effect of a lightweight prompting strategy, asking the model to make code more energy-efficient without giving it specific instructions or examples. The goal is to test whether such basic prompts have a measurable impact on runtime and power consumption.

1.1 Problem Statement

Code quality has many dimensions. In LLM research, most attention has been given to correctness and syntax. But energy use is rarely measured. Benchmarking LLM-generated code is not straightforward. Most existing studies focus on functional accuracy. A recent work also examines runtime, but energy usage mostly remains unreported [6]. Even fewer papers compare model output to human-written code in energy terms [11–15].
Another challenge lies in standardization. There is no common framework for evaluating the energy consumption of small code samples across models and languages. Pereira et al. compare

programming languages in isolation [5]. But little is known about how generated code behaves in realistic use. As a result, it is unclear whether LLMs can match or even outperform human-written code in energy-sensitive environments.

Prompt engineering adds further complexity. It is well known that prompts influence the output. Some studies suggest that simple optimizations can reduce energy usage [9]. But these effects are not consistent, and their impact on performance remains poorly understood.

1.2 Research Goal and Research Questions

In this thesis, we explore the energy profile of LLM-generated code. It compares that output to human-written solutions, by using real programming tasks. It also measures how models react to the energy efficiency prompts. In this thesis, our main goal is to assess whether LLMs can be reliably used to produce energy-efficient code and to explore how minimal prompting affects code performance.

We have 2 research questions that guide this thesis:

- **RQ1:** To what extent does the energy efficiency and runtime of the code generated by LLMs differ from human-written implementations across various programming languages and problems?
- **RQ2:** How does prompting LLMs specifically for energy efficiency affect the runtime and energy usage of the generated code?

1.3 Significance and Impact of the Study

This research is important for two main reasons, First, it addresses a gap in how code quality is measured. Energy use is a concern for both cloud services and power-constrained devices. Developers and researchers need tools that not only produce correct code but also efficient code. Second, this work examines whether prompt-based guidance can help LLMs improve in this area, without changing the underlying model. The results have practical relevance. If prompting leads to better energy behaviour, it may offer a simple, model-agnostic way to reduce energy costs. If not, it may highlight deeper limitations of current LLM generations. For both cases, the outcome will provide insight into how LLMs should be used and improved when efficiency matters. This thesis aims to fill these gaps. It compares LLM-generated and human-written code using the same problems, in the same languages, under the identical test conditions. We measure both runtime and energy consumption. We also evaluate whether a basic optimization prompt can steer LLMs toward more efficient solutions. Our work contributes new empirical evidence on how LLMs perform under energy-aware constraints.

1.4 Overview of Results and Contributions

The study found that LLM-generated code can approximate the energy and runtime performance of human-written code in many cases. GPT-4o produced stable results and responded to the prompt optimization predictably. Qwen2.5-Code, while being the only LLM that was able to generate correct output for Rust, its outputs showed higher energy usage overall. LLaMA-3.3-Instruct displayed inconsistent responses, with optimization prompts mostly increasing energy consumption. Prompting for energy efficiency led to moderate improvements for GPT-4o, but had limited negative effects for the other models. Across all LLMs, the impact of prompting varied by task and language. Statistical analysis showed no significant group-level differences in energy or runtime between LLMs and human-written code. The effect sizes were small. For all implementations, there was a noticeable correlation between time and energy usage. This supports the use of runtime as a proxy for energy consumption in short-running code under stable conditions. This thesis contributes:

- A cross-model, multi-language comparison of energy and runtime performance between LLM-generated and human-written code.
- An empirical test of prompt-based optimization for energy efficiency.
- A replication package is provided to enable replication and extension of this study.

1.5 Structure of the Thesis

This thesis consists of six chapters. Chapter 1 introduces the topic, outlines the motivation, and presents the problem statement, research objectives, and research questions. Chapter 2 provides background and reviews related work on energy-efficient software, code generation by large language models, programming language energy behavior, and prompt-based code optimization. Chapter 3 describes the experimental setup, including the selection of programming tasks, the models used, the prompting strategies, and the tools for measuring runtime and energy consumption. Chapter 4 presents the empirical results, comparing human-written and LLM-generated code across different languages and prompt conditions. Chapter 5 discusses the findings in relation to the research questions. Chapter 6 identifies limitations and discusses the associated threats to validity. Chapter 7 concludes the thesis with a summary of the contributions, outlines directions for future research, and reflects on the broader implications for energy-aware software development using LLMs.

2. Related Work

2.1 The Role of Energy Efficiency in Software and Code-Level Development

Energy efficiency is becoming an important topic in software engineering. As global computing infrastructure expands, software plays a growing role in energy consumption. This includes how software is designed, how its code is written, and how it is executed. Researchers now see software as a part of sustainable computing, not just hardware.

Hardware efficiency has improved steadily [16]. On the other hand, the energy cost of software continues to rise. Mancebo et al. [17] argue that software contributes to ICT-related energy use both directly and indirectly. This contribution includes energy consumed during the execution phase of software and energy triggered by software behavior, such as excessive use of external services.

Anwar et al. [18] support this view and define three types of software energy impact: direct, indirect, and rebound effects. Direct impact refers to the energy used by the software itself; indirect effects come from the energy used by supporting infrastructure. Rebound effects appear when gains in energy efficiency lead to more usage. This framing shows that software is not passive, it actively shapes energy consumption.

At the code level, small decisions can lead to large energy differences. Pereira et al. [5] evaluated 27 programming languages on the common set of tasks. They found significant variation in energy usage. While C and Rust showed the best performance, Python and Ruby consumed more energy to solve the same problems. An interesting fact is that faster code was not always more energy efficient. Runtime and memory do not directly predict energy use. This means energy must be measured on its own, not inferred from other metrics.

Because of these findings, code-level energy profiling has become a priority. For this, Mancebo et al. proposed a structured process [17]. It includes modeling, measurement, and interpretation. They highlight the importance of granularity. Some measurements work at the application level, while others must go deeper down to functions or classes. The key challenge is to make the results reproducible and meaningful.

Anwar et al. offer practical insights and review tools that help developers see how their code uses energy [18]. These include system profilers, mobile platform APIs, and analytics frameworks. The findings show that tool support is often limited, metrics are inconsistent, and developers

lack clear feedback. Although the focus was on mobile apps, the challenges apply across all platforms.

These studies mainly focused on human-written code, but a new challenge is emerging: Large Language Models (LLMs) are now producing significant amounts of code. These models are trained on large codebases and they can produce working solutions from natural language prompts. However, little is known about the energy performance of code generated by LLMs at the time of writing.

Peng et al. highlight this gap [19]. They note that code produced by LLMs is usually evaluated for correctness, but energy usage is rarely measured. They propose training models with multiple objectives, not just accuracy but also efficiency. They also call for new benchmarks that include energy and performance metrics. Without these benchmarks, it is difficult to know whether LLM-generated code is sustainable.

Current literature makes two points clear: First, how code is written directly affects how much energy it uses. Second, current tools can measure this, but results vary depending on the tool, platform, or method used.

The uncertainty lies in how LLM-generated code compares. We don't yet know if it is more or less energy efficient than human-written code, or whether there is any consistency in the results at all. We also don't know if prompting can guide models to generate greener code.

2.2 LLM-Generated Code: Characteristics, Efficiency, and the Role of Prompting

LLMs help with coding by giving support to developers through tools like GitHub Copilot and Amazon CodeWhisperer. These models are able to generate entire code blocks from natural language prompts. It is clear that they are useful, but their efficiency remains under debate.

Recent studies show that LLM-generated code behaves differently from human-written code. The researchers tested 18 LLMs on Leetcode problems in one large-scale comparison [6]. Some models produced code that had similar or even better performance than human submissions. However, the results varied. In larger problems, model accuracy was reduced, and correctness remained inconsistent.

In another study, seven LLMs were tested on standard programming tasks. These models included GPT-3.5, GPT-4, Copilot, and Code Llama [15]. In some cases, LLM-generated code matched human solutions in speed or memory use. But in most cases, human-written

code showed better performance. As complexity increased, the execution time and correctness declined.

LLMs are probabilistic systems [20]. They do not always generate the same code for the same problem. This randomness affects both correctness and performance. To guide the models, prompts are often used. Researchers have tested whether prompts can help produce more efficient or sustainable code.

One study introduced the green capacity metric [14]. It includes correctness, runtime, memory, energy, and FLOPs into a single score. The authors used this to compare outputs from GitHub Copilot, ChatGPT-3, and CodeWhisperer. They tested both default and optimized prompts. Results showed that prompts improved efficiency in some cases. However, results were inconsistent. The models did not have a clear awareness of sustainability. Prompts produced different code, but not always greener code.

Prompting strategies were also tested to compare LLMs and classical compilers. For this purpose, two methods were used in one study: instruction prompting and chain-of-thought prompting [21]. The models were GPT-4 and CodeLlama-70B. No prompting helped to achieve major optimization. LLMs could not optimize the code in most cases. As code complexity increased, the correctness dropped. LLMs succeeded in only 14% of the cases. The study concluded that LLMs are unreliable optimizers, but they may help on simpler tasks.

Other researchers proposed more structured approaches. One prototype used two LLMs interacting with each other for optimization. One model produced the code, and another reviewed its energy usage. This system created a feedback loop. The authors argued that prompting can support energy efficiency, but only if models receive proper evaluation during the generation process. Their method is still in the experimental phase, but it points to systems that may combine energy optimization and code generation in the future [14].

In another controlled experiment, researchers used CodeLlama to test the optimization prompts [12]. JavaScript, Python, and C++ were used in this experiment. Code was generated with and without optimization instructions. In most cases, performance got even worse. Temperature settings also had little effect. The study showed that prompting affects code structure but not energy behavior in a predictable way.

Other researchers aimed to reduce energy usage at the model level [10]. The GREEN-CODE framework introduced a method for early exit during inference. The model used a Reinforcement Learning (RL) agent to stop generation when there was enough information available. This helped to reduce inference energy by 23–50% while keeping code correctness. Although it

does not change the generated code, this helps with improving energy usage during the code generation phase itself.

Overall, current research shows that prompting can influence LLM output. However, it has not yet produced consistent gains in energy efficiency. Some prompts help with simple functions, while others do not improve or even lead to worse performance. None of the tested models is trained with energy-aware objectives. This creates a gap between user intent and model behavior. At the same time, this variation also shows LLMs' flexibility. With the right guidance or additional feedback, models can become more energy-aware. Existing studies suggest potential, not certainty. Further controlled testing is needed. The studies so far use small benchmarks or synthetic tasks. There is a need for research using real-world problems and reliable energy measurement.

This thesis addresses that need. It compares human-written and LLM-generated code using both time and energy metrics. It also analyzes the effect of optimized prompts for different LLMs. The goal is to analyze energy data for the differences among different LLMs and programming languages and to test whether prompting can improve energy efficiency in practice.

3. Methodology

3.1 Experiment Definition

In this experiment, we aim to compare the energy efficiency and runtime performance of human-written code and the code generated by different LLMs, with a specific focus on how prompting can influence these outcomes. This investigation is grounded in practical programming tasks across real-world languages and benchmarks. The study aims to answer the following research questions:

- **RQ1:** To what extent does the energy efficiency and runtime of the code generated by LLMs differ from human-written implementations across various programming languages and problems?
- **RQ2:** How does prompting LLMs specifically for energy efficiency affect the runtime and energy usage of the generated code?

3.2 Experiment Planning

To systematically evaluate the energy efficiency of LLM-generated code, a detailed experiment planning phase was conducted involving task selection, tool setup, and prompt design across three programming languages.

3.2.1 Task Selection (Human-written Code)

Four programming tasks were selected from the Codewars [22] platform to serve as benchmarks in this study. We aimed to make sure that the selection has both algorithmic diversity and practical relevance, which allows us to have a comprehensive evaluation of energy consumption across multiple languages. Each task represents a distinct category of computational problem, such as sorting, tree traversal, etc., thereby enabling assessment across a range of algorithmic and structural complexities.

To get the validity and comparability of the evaluation, problems were filtered according to several criteria : 1) having a high number of community submissions, which suggests optimized and widely accepted human solutions, 2) the presence of platform-verified test cases, ensuring the correctness across all implementations. The selected tasks are frequently used in real-world applications such as web development, system-level programming, and data processing domains where performance and energy efficiency are critical concerns. They vary from simple to moderate difficulty, and they mirror typical coding problems that developers face in real-world settings. Codewars ranks difficulty from 8kyu (easiest) to 1kyu (hardest). Tasks between 4kyu

to 8kyu were selected because the submissions at these levels are high, which makes it possible to select quality submissions from user contributions. We also verified that each selected task had an implementation in all three chosen programming languages. For each task-language pair, the first available solution labeled as "Best Practice" was selected from Codewars without additional manual filtering or review. An overview of the selected tasks, including their category, difficulty level, language availability, and source, is presented in Table 1.

Table 1. Overview of Benchmark Problems

Problem	Category	Difficulty	Languages	Source
Merge Arrays	Array Manipulation	8kyu	JavaScript, Python, Rust	Codewars
Sort Binary Tree by Levels	Tree Traversal	4kyu	JavaScript, Python, Rust	Codewars
Supermarket Queue	Queue Simulation	6kyu	JavaScript, Python, Rust	Codewars
SJF Scheduling	Scheduling Algorithm	6kyu	JavaScript, Python, Rust	Codewars

Programming languages

The assessment focused on three programming languages: JavaScript[1], Python[2], and Rust[3]. The versions used for each programming language were: Python 3.12.2, Node.js 20.11.1 (for JavaScript), and Rust 1.77.2.

To begin with, these languages were chosen to illustrate different points of the performance spectrum, such as interpreted versus compiled and dynamic versus static typing. In addition to this, these languages were picked because of their widespread usage and community support, as it is demonstrated by the 2024 Stack Overflow Developer Survey, where all three languages are highly ranked in terms of usage or developer preference [23]. Finally, we faced a practical constraint concerning the availability of all benchmark problems in the same three languages. This ensured a direct comparison of logic and implementation patterns.

In addition to their technical relevance, the selected languages differ in terms of their energy efficiency. Prior research has shown that Python is among the least energy-efficient languages [5]. This is consistent with other interpreted and dynamically typed languages. JavaScript, while also interpreted, tends to perform better than Python in energy consumption. However, it still falls behind compiled languages. Rust has been identified as one of the most energy-efficient

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[2] https://www.python.org
[3] https://www.rust-lang.org

languages [5]. For this reason, in this study, we do not compare energy consumption across languages. Each language is evaluated on its own. The goal is to compare LLM-generated code with human-written code within the same language. Including different languages helps to cover a wider range of use cases and programming styles.

3.2.2 Measurement Tools and Setup

The dependent variables in this study are energy consumption (J) and runtime (s). The independent variables are programming language, model used, and prompt type (default vs. energy-optimized).

Tool Overview

Energy and time were measured with a tool called perf, which also allows for performance profiling and is available on Linux systems [24]. This tool collects low-level data directly from the CPU by accessing hardware counters exposed by Intel processors.

Energy Measurement Setup

In this experiment, the energy-pkg event was used. This event reports the total energy used by the CPU package, which contains essential components used while running code. That includes processor cores, shared caches, the memory controller, and other internal components. Among the available RAPL domains, energy-pkg is the most stable and widely supported. It has also been used in several studies for energy profiling. Energy readings came from Intel's RAPL interface (Running Average Power Limit). This system tracks energy use inside the processor and stores the data in special registers called MSRs (Model Specific Registers). These values are accurate and are updated in real time. Energy values were collected in joules. For each test, the script recorded the energy used just before and after the code ran. The difference between those two values gave the total energy used. This method adds very little overhead and does not require any external hardware.

Execution Time Measurement

Execution time was also measured using perf. CPU cycles were tracked, and the task-clock counter was used to measure how long the CPU was actively running the code. This gave a clear view of performance during execution. This setup follows standard practice from earlier work. For example, Vartziotis et al. used perf to compare human-written code with LLM-generated code [14] . Ilager et al. used similar hardware counters in their GREEN-CODE framework [10]. Intel states that RAPL readings are accurate within 1–2% compared to external power meters. Because of this, it has become a reliable method for measuring energy use in software experiments.

3.2.3 Experiment Design

LLMs and Their Generated Code

We utilized three LLMs: GPT-4o, LLama-3.3-Instruct, and Qwen 2.5-Coder. These models were selected based on three main criteria.

Firstly, all three selected models are instruction-tuned. This tuning helps to ensure they respond to structured prompts consistently. Since this thesis also evaluates how prompting can affect energy efficiency, models that interpret instructions were essential.

The second criterion was code generation specialization. Qwen2.5-Coder-32B-Instruct and Llama-3.3-70B-Instruct are optimized for programming tasks [25, 26]. Qwen 2.5-Coder has been fine-tuned on benchmarks such as HumanEval, MBPP, and competitive coding datasets, which makes it strong at producing efficient code. Llama-3.3-Instruct is part of Meta's code generation series and supports multiple programming languages. GPT-4o, while being a general-purpose model, is competitive on coding tasks. Its effectiveness is demonstrated through community benchmarks such as CodeWars [27].

Lastly, the selected models represent different ecosystems. GPT-4o is a commercial and closed-source model by OpenAI. LLaMA 3.3-Instruct and Qwen 2.5-Coder are open-source models and were available via Hugging Face as of April 2025. This variety allowed the study to explore whether energy efficiency differences were similar across platforms or if they were model-specific.

All models were accessed between April 10 and April 18, 2025, using publicly available interfaces: GPT-4o via ChatGPT (chat.openai.com), and both LLaMA 3.3-Instruct and Qwen 2.5-Coder through Hugging Face [4]. Default generation settings were used without modification. No local inference or fine-tuning was applied, to make sure the reproducibility of results under similar conditions.

Also, their public availability and relevance to everyday programming tasks are taken into account. All generations were carried out via browser-based interfaces.

The temperature and token settings were not modified. Default configurations were kept to reflect real-world usage. No API calls or custom inference pipelines were involved. Each prompt was entered manually into the respective interface. Each sample was only generated once and filtered out if it failed the correctness test.

[4] https://huggingface.co/chat/

Prompting Strategy and Input Format

This thesis used a lightweight prompting approach. The goal was to test whether small prompt changes could help lower the energy usage of LLM-generated code. Prompt design can influence what kind of code the model produces. It has been shown that prompts can affect not only accuracy but also how much energy is consumed during generation.

Several studies have looked into this. Cappendijk et al. [9] found that asking for energy-efficient code helped in some tasks. But the improvements were not consistent. The effects varied depending on the model and the problem. Ilager et al. [10] and Rubei et al. [28] also studied similar strategies. They showed that prompt wording, and also generation stop conditions and can affect energy consumption.

In this study, each model received the full task description and sample test cases. These were copied directly from Codewars [22]. They were not changed. The default prompt used for every task was:

"Solve the problem in [language]."

To generate the energy-optimized version, the corresponding code sample and the following energy-efficiency prompt were used:

"Give me an energy-efficient version of this code. Avoid unnecessary loops or conditionals."

This prompt was simple by design. The first part asked for an efficient version without giving technical details. The second part gave a specific instruction to guide the structure.

The phrase "Avoid unnecessary loops or conditionals" is based on findings from recent LLM research. Cappendijk et al. [9] showed that prompts that mention control structures can change how models write code. In their study, asking the model to optimize energy sometimes led to fewer loops and branches. Peng et al. [19] also found that simplification of control flow helped reduce memory traffic and energy use in certain tasks.

These results suggest that loops and conditionals, if poorly structured, can increase energy use. They can lead to more CPU activity and memory operations. Avoiding redundant or inefficient control flow may reduce this impact.

We chose the word "unnecessary" carefully. It leaves room for essential logic while discouraging excess. Our aim was not to fully optimize the code but to test if this kind of prompt could influence energy behavior in a way that is measurable.

No explanation was requested from the model. Only the generated code was collected. If the output did not run correctly, it was excluded. All samples were manually reviewed.

Code versions

Each of the four selected problems (Table 1) was implemented in the three selected programming languages: JavaScript, Python, and Rust. For each problem-language pair, a human-written solution was selected as a baseline. Each human-written version was taken directly from the highest-rated community submissions on Codewars, labeled as “Best Practice.” No manual filtering or review was applied; the first available best-practice solution was used.

Three selected LLMs were used to generate code for the selected problems in the same languages. The LLM-generated code is compared against the baseline of human-generated code in that language . This produced 36 initial LLM-generated code samples (4 problems × 3 languages × 3 models).

An energy-optimized version was then generated for each of the 36 LLM generated codes. In total, the dataset consisted of 12 human-written code versions (baseline), 36 LLM-generated default-prompt variants, and 36 LLM generated energy-optimized variants, which resulted in 84 distinct samples.

All versions were saved as individual files with consistent naming conventions. This ensured traceability and allowed for automated scripting during measurement. The complete dataset of code samples is publicly available in a GitHub repository.[5]

Correctness Validation

Before energy and performance measurements, each LLM-generated code sample was checked for correctness. This step ensured that the LLM-generated code produced valid results and solved the intended task.

Correctness was tested using test cases from Codewars [22], which provides a set of public and hidden tests for each problem. The LLM-generated code was directly pasted into the online editor on the Codewars website and executed against the available tests. A version was accepted only if it passed all the tests. The Codewars system marks a solution as “passed” when all test assertions succeed.

This process was applied to all LLM-generated versions. If a model returned incorrect or incomplete code, such as hallucinations [29] or syntax errors, that version was excluded from the evaluation. No replacement was used. Only the outputs that were correct and runnable were passed to the energy measurement step. Manual review helped to catch common issues such as

[5] https://github.com/fidangithub/perf-energy-measurements/tree/main/samples

missing return statements or invalid logic. However, if the model produced runnable code that failed due to logic errors, it was still rejected.

Each version of the code was tested using the same input. These inputs were placed in the script after the solution, so that the program ran automatically. The test cases were based on Codewars examples. However, they were expanded to allow more accurate energy measurements.

For example, large arrays were used in array-related problems. These included sorted, reversed, and mixed sequences. This was done to reflect a wider range of execution patterns. It also made the results more stable across runs.

The same samples were used for both LLM-generated and human-written code. This ensured comparability. The inputs were not changed between versions. They were written manually and kept consistent.

The input samples appear immediately after each code solution in the script files.[6]

We did not generate energy-optimized versions for samples where the corresponding default LLM output failed. Optimization was only applied to code that was correct and passed all test cases. As shown in Table 2, 46 LLM-generated samples passed the correctness checks. When the 12 human-written baselines are included, the total dataset consists of 58 valid code samples.

[6] https://github.com/fidangithub/perf-energy-measurements/tree/main/samples

Table 2. Correctness Outcomes Across Models and Languages

Problem	GPT[1]	GPT'[2]	LLaMA[3]	LLaMA'[4]	Qwen[5]	Qwen'[6]
Merge Arrays/ Javascript	Passed	Passed	Passed	Passed	Passed	Failed
Merge Arrays/ Python	Passed	Failed	Passed	Passed	Passed	Failed
Merge Arrays/ Rust	Failed	-	Failed	-	Failed	-
Supermarket Queue/ Javascript	Passed	Passed	Passed	Passed	Passed	Failed
Supermarket Queue/ Python	Passed	Passed	Passed	Passed	Passed	Passed
Supermarket Queue/ Rust	Failed	-	Failed	-	Failed	-
SJF Scheduling/ Javascript	Passed	Passed	Passed	Passed	Passed	Passed
SJF Scheduling/ Python	Passed	-	Passed	Passed	Passed	Failed
SJF Scheduling/ Rust	Failed	-	Failed	-	Passed	Passed
Sort Binary tree by levels/ Javascript	Passed	Passed	Passed	Passed	Passed	Passed
Sort Binary tree by levels/Python	Passed	Passed	Passed	Passed	Passed	Passed
Sort Binary tree by levels/ Rust	Failed	-	Failed	-	Passed	Passed

[1] Code generated by GPT-4o using the default prompt.

[2] Code generated by GPT-4o using the energy-optimized prompt.

[3] Code generated by LLaMA 3.3-Instruct using the default prompt.

[4] Code generated by LLaMA 3.3-Instruct using the energy-optimized prompt.

[5] Code generated by Qwen 2.5-Coder using the default prompt

[6] Code generated by Qwen 2.5-Code using the energy-optimized prompt

3.3 Experiment Execution

With the experimental setup finalized, the execution phase involved running each code sample under tightly controlled hardware conditions, ensuring consistent data collection for runtime and energy consumption.

3.3.1 Experimental Setting

All tests were run on a single device to ensure consistency. The machine used was an HP EliteBook 840 G10 running Ubuntu 24.04.1 LTS. It has a 13th Gen Intel Core i5-1335U processor with 12 logical cores and 16 GB of RAM. The version of perf used was 6.8.12.

To reduce variation between runs, some system configurations were changed. Firstly, the CPU performance governor was changed to "performance" mode using the command `sudo cpupower frequency-set -g performance`.

Modern CPUs adjust their speed based on system load [30]. This feature helps save energy in daily use, but it makes performance profiling harder. A slower CPU can take longer to finish a task and use more energy, a faster CPU, on the other hand, might use less. These shifts can create noise in the results. Setting the governor to performance mode locks the CPU at its highest frequency. This makes the system behave the same during every test and removes an important source of randomness.

Another important setting was hyperthreading. This lets one CPU core handle two threads at once. While this helps with multitasking, it can affect how energy is used. When two processes share the same core, they also share power and cache, which makes energy readings less reliable. To avoid this, each process was pinned to a specific core using `taskset -c 2` at the start of the command. This restricted execution to a single core, minimizing interference and improving measurement consistency.

Other sources of interference were also disabled. The network connection was turned off, and all background services were stopped. Auto-brightness and screen dimming were also turned off to avoid inconsistent power draw from the display subsystem. No desktop environment or additional services were running during the tests. These steps followed best practices from Cruz's [31] energy measurement guide.

Each code sample was tested separately. To allow a cooldown, a short pause of 1 minute was used between runs [31]. This helped keep the CPU temperature stable and avoided throttling. By locking down the system like this, the experiment kept test conditions consistent and made the comparisons between code versions fair and reliable.

To ensure reliable measurements, each code sample was executed repeatedly. They were called 10000 times in a loop. This helped to smooth out random variation and produce more stable energy readings. Inputs were not created dynamically (e.g., array.fill, array.map). Instead, all inputs were hardcoded manually. This avoided variation between languages and reduced noise. It also helped ensure that measurements reflected execution only, not input setup.

A separate run was used to measure system idle energy. For this, a 60-second idle script was executed under the same conditions. Idle power was averaged from these runs. The average idle power measured across these runs was 0.6559 Watts.

It was then subtracted from each measurement to isolate the actual code energy using the formula

$$\textit{Adjusted Energy} = \textit{Measured Energy} - (\textit{Idle Power} \times \textit{Measured Execution Time})$$

JavaScript needed extra preparation. Due to JIT compilation and runtime optimizations, a warm-up phase was added. Each JavaScript file ran for 5,000 extra iterations before measurement. This

allowed the engine time to stabilize. Python and Rust did not need this step, as their runtimes are consistent from the beginning.

Energy readings came from the energy-pkg domain. This captures only CPU package energy. It does not include external components like the screen or peripherals. Still, features such as auto-brightness were turned off to avoid side effects.

The full experiment took about 30 hours. The device stayed plugged in for all tests. This avoided battery-related variation in energy or performance.

3.3.2 Data Collection

All measurements were carried out using a dedicated Bash script. This script was designed to automate execution, profiling, and logging tasks [7]. It handled all benchmarks across implementations and languages. Each version of the code was executed independently. Every variant was run thirty times, always under the same system conditions.

Internal repetition features of the profiling tools were not used. Instead, each execution was launched as its own process. This allowed better control over how the system behaved. It also helped reduce any artifacts that might occur due to repeated execution in batches. To avoid introducing systematic bias, the complete list of [script, run] combinations was randomized beforehand. This shuffling aimed to balance possible variation in system state or background load.

After each run, the machine remained idle for 60 seconds. This pause helped reduce leftover heat or power effects. It was added to allow the system to return to baseline conditions. These short cooling periods were considered important for keeping energy data consistent across runs.

The script selected the right compiler or interpreter automatically. This was based on file extension. Rust files were compiled before running. Python and JavaScript files were executed directly with the corresponding interpreter. All output, including runtime and energy values, was appended to a shared log. Each log entry included metadata like file name, language, run ID, and measurement values.

Results were stored in CSV format [8]. Each row represented one execution. It contained the raw energy (in joules) and time (in seconds). After collecting all data, a separate Python script was

[7] https://github.com/fidangithub/perf-energy-measurements/blob/main/energy_measurement_runner.sh

[8] https://github.com/fidangithub/perf-energy-measurements/blob/main/energy_results_wide.csv

used. This script computed the average values for each script version [9].The final summary was saved to another file, which was then used for comparison.

3.3.3 Statistical Analysis

All statistical tests were performed using the SciPy library [32]. The Shapiro–Wilk test was used to assess normality. In most cases, the data were not normally distributed ($p < 0.05$), so non-parametric methods were applied. The Mann–Whitney U test was used to compare energy and runtime between groups. This included comparisons between human-written and LLM-generated code, and between default and optimized LLM outputs. Effect sizes were reported using Cliff's delta. This helped interpret the practical relevance of observed differences. To assess the relationship between execution time and energy use, Spearman's rank correlation was used. This method is suitable for non-linear, monotonic associations. All tests used a significance level of 0.05.

[9] https://github.com/fidangithub/perf-energy-measurements/blob/main/calculate_mean_results.py

4. Results

The results are presented in line with the research questions stated earlier. We start with descriptive summaries for energy usage and execution time. These values are grouped by programming task, language, and model type. Tables are used to display the mean values clearly. After that, statistical comparisons are made between human-written and model-generated code. Each model is discussed in a separate subsection to keep the structure consistent.

Next, the effect of optimization prompting is examined. The analysis compares default and optimized outputs from each model. The data are presented again by task and language. Tables are supported by figures to help with interpretation. The subsections follow the same format for easier reading. At the end, a correlation analysis is included. It is used to check if there is a clear association between execution time and energy usage.

Tables 3 to 6 show a summary of average energy use and execution time for each model and prompt type. These tables cover four benchmark problems used in this thesis: Merge Arrays (Table 3), Supermarket Queue (Table 5), and SJF Scheduling (Table 4), Sort Binary Tree by Levels (Table 6). Values are grouped by language and model. For each group mean energy(in joules) and time (in seconds) are reported. Human-written code[10] is used as the reference for comparison.

Table 3. Mean Energy and Execution Time for the Merge Arrays problem Across Programming Languages and LLM Prompt Variants

Metrics	Mean Energy Usage (J)							Mean Execution Time (s)						
LLMs	HW	GPT	GPT’	LLaMA	LLaMA’	Qwen	Qwen’	HW	GPT	GPT’	LLaMA	LLaMA’	Qwen	Qwen’
JS	3.26	3.39	3.39	3.44	3.37	3.23	-	0.22	0.23	0.23	0.24	0.23	0.22	-
PY	1.5	1.49	-	1.6	1.5	6.97	-	0.11	0.1	-	0.11	0.11	0.43	-
Rust	0.82	-	-	-	-	-	-	0.06	-	-	-	-	-	-

[10]Hereafter, “human-written code” is abbreviated as HW in all tables.

Table 4. Mean Energy and Execution Time for the Shortest Job First problem Across Programming Languages and LLM Prompt Variants

Metrics	Mean Energy Usage (J)							Mean Execution Time (s)						
LLMs	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'
JS	0.88	1.77	0.82	1.52	1.52	1.77	1.82	0.06	0.12	0.06	0.11	0.11	0.12	0.13
PY	1.21	3.31	-	3.51	22.13	1.82	-	0.08	0.21	-	0.22	1.37	0.12	-
Rust	0.25	-	-	-	-	1.49	1.49	0.02	-	-	-	-	0.1	0.1

Table 5. Mean Energy and Execution Time for the Supermarket Queue problem Across Programming Languages and LLM Prompt Variants

Metrics	Mean Energy Usage (J)							Mean Execution Time (s)						
LLMs	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'
JS	1.97	1.83	1.34	1.94	13.9	1.35	-	0.14	0.14	0.1	0.14	0.89	0.1	-
PY	5.35	2.2	1.23	5.41	2.26	5.38	2.24	0.34	0.14	0.08	0.35	0.14	0.35	0.14
Rust	2.21	-	-	-	-	-	-	0.16	-	-	-	-	-	-

Table 6. Mean Energy and Execution Time for the Sort Binary Tree by Levels problem Across Programming Languages and LLM Prompt Variants

Metrics	Mean Energy Usage (J)							Mean Execution Time (s)						
LLMs	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'	HW	GPT	GPT'	LLaMA	LLaMA'	Qwen	Qwen'
JS	0.92	0.92	0.9	0.91	0.93	0.92	0.92	0.07	0.07	0.07	0.07	0.07	0.07	0.07
PY	0.9	0.9	0.89	0.9	0.9	0.62	1.92	0.06	0.06	0.06	0.06	0.06	0.04	0.12
Rust	0.2	-	-	-	-	0.2	0.2	0.01	-	-	-	-	0.01	0.01

4.1 Comparison Between Human-Written Code and LLM-Generated Code (RQ1)

4.1.1 GPT-4o generated code vs Human-written code

Comparisons of GPT-4o generated and human-written code in terms of energy use and execution time are shown in Table 7. The largest energy difference appeared in the Python Shortest Job First problem, where GPT-4o used 3.31 J, compared to 1.21 J for the human version. In contrast, the Supermarket Queue problem showed the lowest energy for GPT-4o with 2.2 J, which is way lower than the relative human baseline with 5.5 J. JavaScript results were more stable across all problems, most tasks differed by less than ±0.15 J.

Execution time showed a similar trend. The Supermarket Queue problem in Python was faster (0.14 s vs. 0.34 s) but slower in Shortest Job First (0.21 s vs. 0.08 s). JavaScript times were

mostly aligned, with a small delay in Shortest Job First (0.12 s vs. 0.06 s), and near-identical values were reported in other problems.

Figure 1 illustrate energy and time per task. Table 8 reports statistical test results. The Mann-Whitney U test showed no significant difference in energy (p = 0.57) or execution time (p = 0.71). Cliff's delta [33] values were −0.19 for energy and −0.13 for time. Both indicate small effect sizes, suggesting that overall differences between GPT-4o and human-written code were minimal.

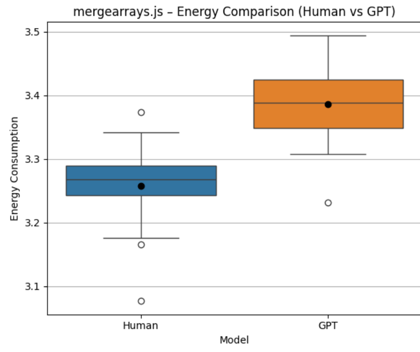
Rust is excluded from this comparison because of the lack of valid GPT-4o outputs in this language.

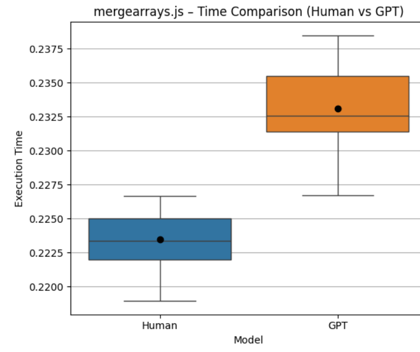
Table 7. Benchmark Results for GPT-4o generated and Human-written Code on All Tasks

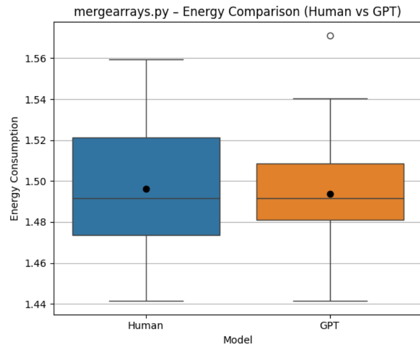
Metrics	Mean Energy Usage (J)								Mean Execution Time (s)							
Problems	Merge Arrays		Supermarket Queue		SJF		Tree by Levels		Merge Arrays		Supermarket Queue		SJF		Tree by Levels	
LLMs	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**	**HW**	**GPT**
JS	3.26	3.39	1.97	1.83	0.88	1.77	0.92	0.92	0.22	0.23	0.14	0.14	0.06	0.12	0.07	0.07
PY	1.5	1.49	5.35	2.2	1.21	3.31	0.9	0.9	0.11	0.1	0.34	0.14	0.08	0.21	0.06	0.06
Rust	0.82	-	2.21	-	0.25	-	0.2	-	0.06	-	0.16	-	0.02	-	0.01	-

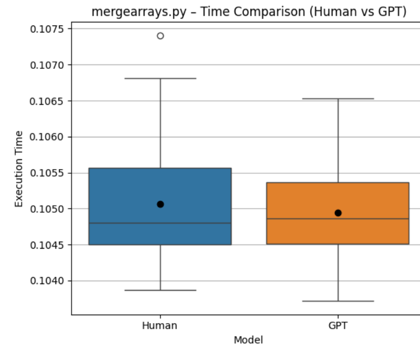
Table 8. Statistical Comparison of Energy and Runtime Between GPT-4o generated and Human-Written Code

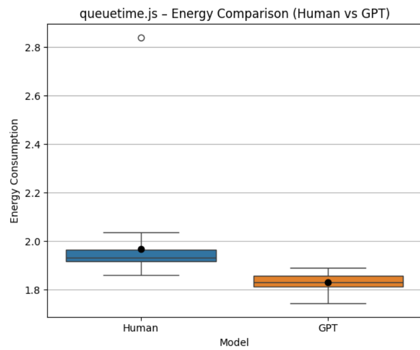
Metric	U-Statistic	p-Value	Cliffs Delta
mean_energy	26.0	0.5737373737373737	-0.1875
mean_time	28.0	0.7209013209013208	-0.125

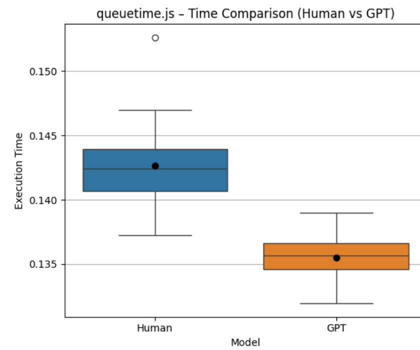

mergearrays.js – Energy Comparison (Human vs GPT)
Energy Consumption
Human
GPT
Model

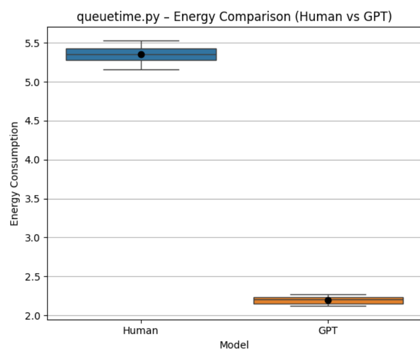

mergearrays.js – Time Comparison (Human vs GPT)
Execution Time
Human
GPT
Model

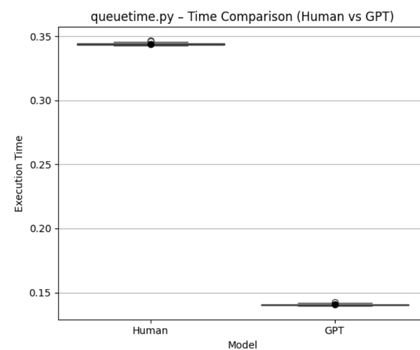

mergearrays.py – Energy Comparison (Human vs GPT)
Energy Consumption
Human
GPT
Model


mergearrays.py – Time Comparison (Human vs GPT)
Execution Time
Human
GPT
Model


queuetime.js – Energy Comparison (Human vs GPT)
Energy Consumption
Human
GPT
Model


queuetime.js – Time Comparison (Human vs GPT)
Execution Time
Human
GPT
Model


queuetime.py – Energy Comparison (Human vs GPT)
Energy Consumption
Human
GPT
Model


queuetime.py – Time Comparison (Human vs GPT)
Execution Time
Human
GPT
Model

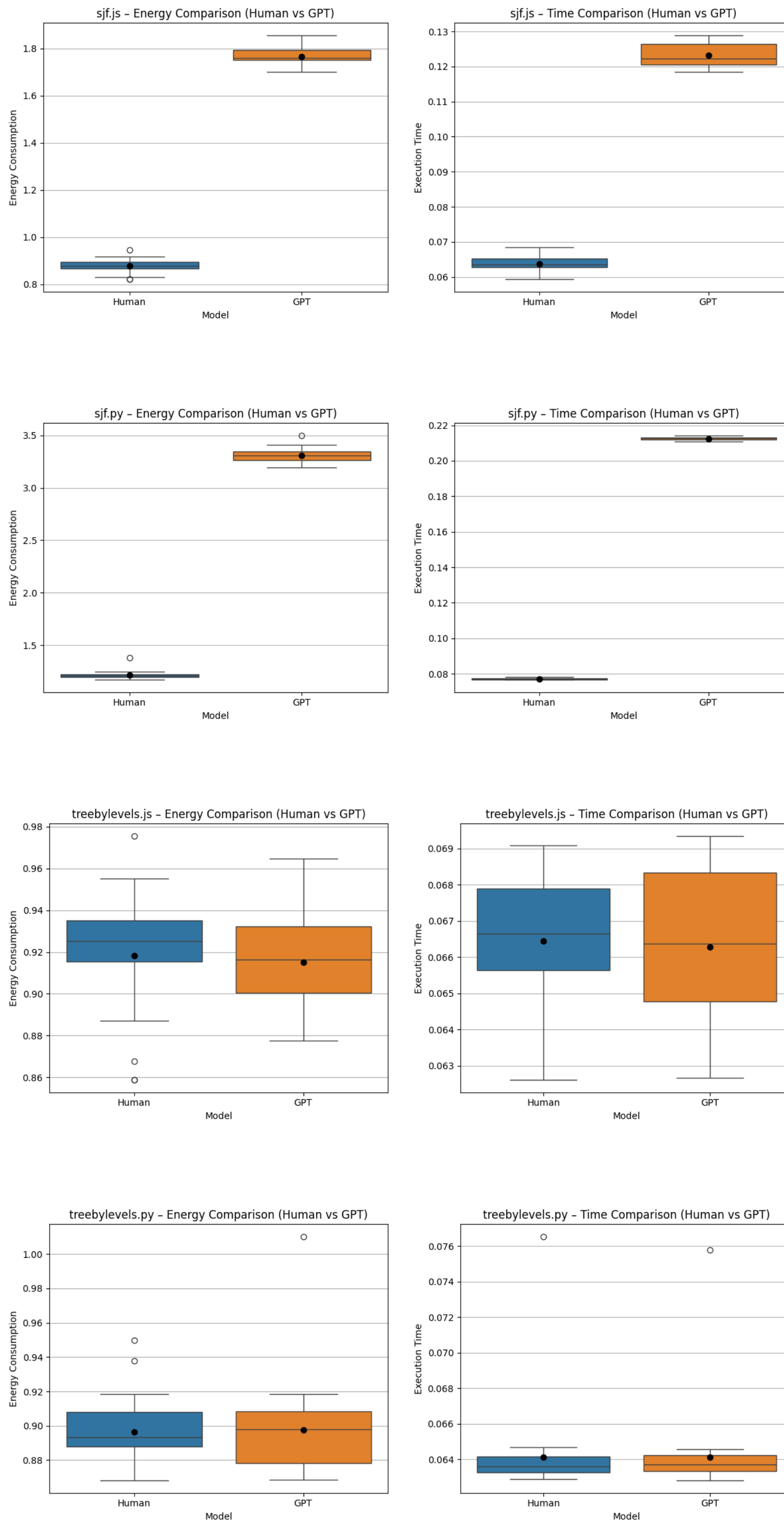


Figure 1. Visual Comparison of Runtime and Energy Usage Between GPT-4o generated and Human-Written Code Across All Tasks and Languages

4.1.2 LLaMA-3.3-Instruct generated code vs Human-written code

Energy and time results for LLaMA-3.3-Instruct are summarized in Table 9. Rust was excluded again due to invalid outputs for this model as well. The largest energy difference is shown in the Python, Shortest Job First problem, where LLaMA consumed 3.51 J compared to human-written code, which was 1.21 J. In contrast, the smallest difference was in the Sort Tree by Levels problem, where both showed 0.9 J. Other tasks showed moderate variation, generally within ±0.2 J of the baseline.

Execution time followed a similar pattern. In most tasks, LLaMA's execution times were too close to human-written code. But, in the Python Shortest Job First problem, it took 0.22s, nearly three times longer than human versions at 0.08s. Javascript times were more stable for this model as well, with differences typically under 0.02 s.

Figure 2 visualize these patterns. The most visible gap appears in Shortest Job First for both energy and time, especially in Python. Other tasks show minimal visual differences.

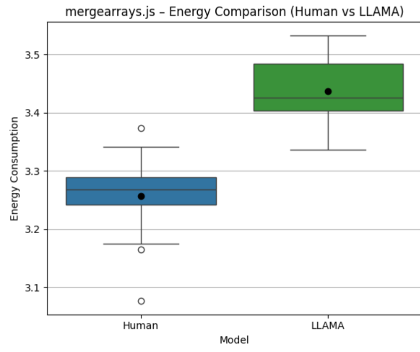
Table 10 illustrates the statistical results. The Mann-Whitney U test showed no significant difference between groups for either metric (p = 0.38). Cliff's delta was −0.28 for both energy and time, indicating a small to moderate effect size.

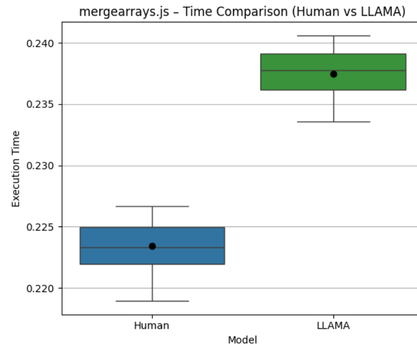
Table 9. Benchmark Results for LLaMA-3.3-Instruct generated and Human-written Code on All Tasks

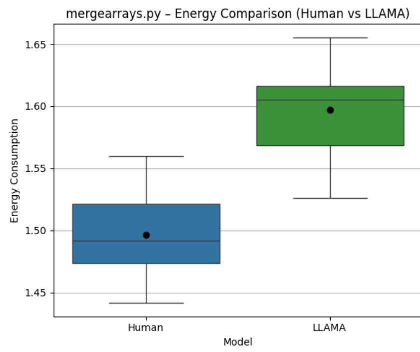
Metrics	Mean Energy Usage (J)								Mean Execution Time (s)							
Problems	Merge Arrays		Supermarket Queue		SJF		Tree by Levels		Merge Arrays		Supermarket Queue		SJF		Tree by Levels	
LLMs	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**	**HW**	**LLaMA**
JS	3.26	3.44	1.97	1.94	0.88	1.52	0.92	0.91	0.22	0.24	0.14	0.14	0.06	0.11	0.07	0.07
PY	1.5	1.6	5.35	5.41	1.21	3.51	0.9	0.9	0.11	0.11	0.34	0.35	0.08	0.22	0.06	0.06
Rust	0.82	-	2.21	-	0.25	-	0.2	-	0.06	-	0.16	-	0.02	-	0.01	-

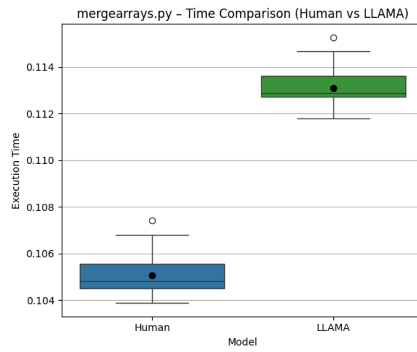
Table 10. Statistical Comparison of Energy and Runtime Between LLaMA-3.3-Instruct generated and Human-Written Code

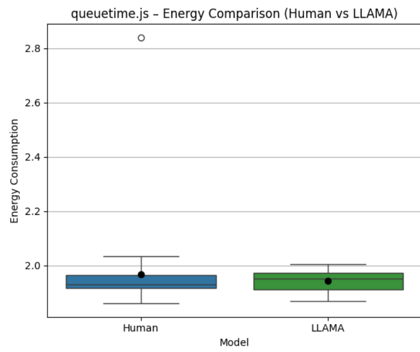
Metric	U-Statistic	p-Value	Cliffs Delta
mean_energy	23.0	0.3822843822843822	-0.28125
mean_time	23.0	0.3822843822843822	-0.28125

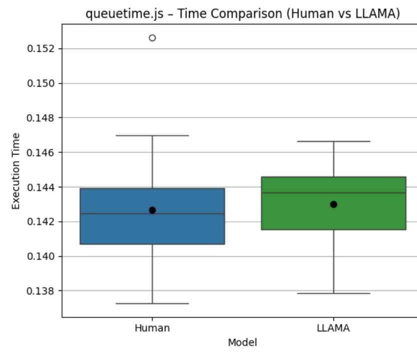

mergearrays.js – Energy Comparison (Human vs LLAMA)
Energy Consumption
3.1
3.2
3.3
3.4
3.5
Human
LLAMA
Model

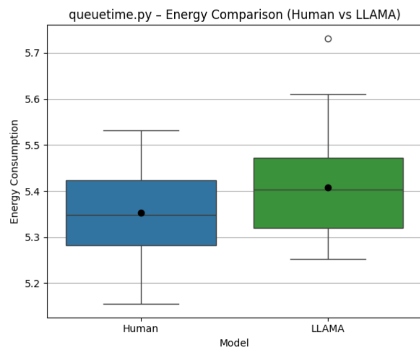

mergearrays.js – Time Comparison (Human vs LLAMA)
Execution Time
0.220
0.225
0.230
0.235
0.240
Human
LLAMA
Model

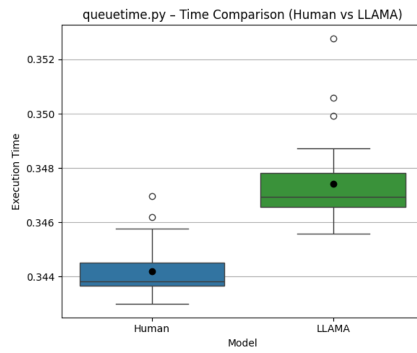

mergearrays.py – Energy Comparison (Human vs LLAMA)
Energy Consumption
1.45
1.50
1.55
1.60
1.65
Human
LLAMA
Model


mergearrays.py – Time Comparison (Human vs LLAMA)
Execution Time
0.104
0.106
0.108
0.110
0.112
0.114
Human
LLAMA
Model


queuetime.js – Energy Comparison (Human vs LLAMA)
Energy Consumption
2.0
2.2
2.4
2.6
2.8
Human
LLAMA
Model


queuetime.js – Time Comparison (Human vs LLAMA)
Execution Time
0.138
0.140
0.142
0.144
0.146
0.148
0.150
0.152
Human
LLAMA
Model


queuetime.py – Energy Comparison (Human vs LLAMA)
Energy Consumption
5.2
5.3
5.4
5.5
5.6
5.7
Human
LLAMA
Model


queuetime.py – Time Comparison (Human vs LLAMA)
Execution Time
0.344
0.346
0.348
0.350
0.352
Human
LLAMA
Model

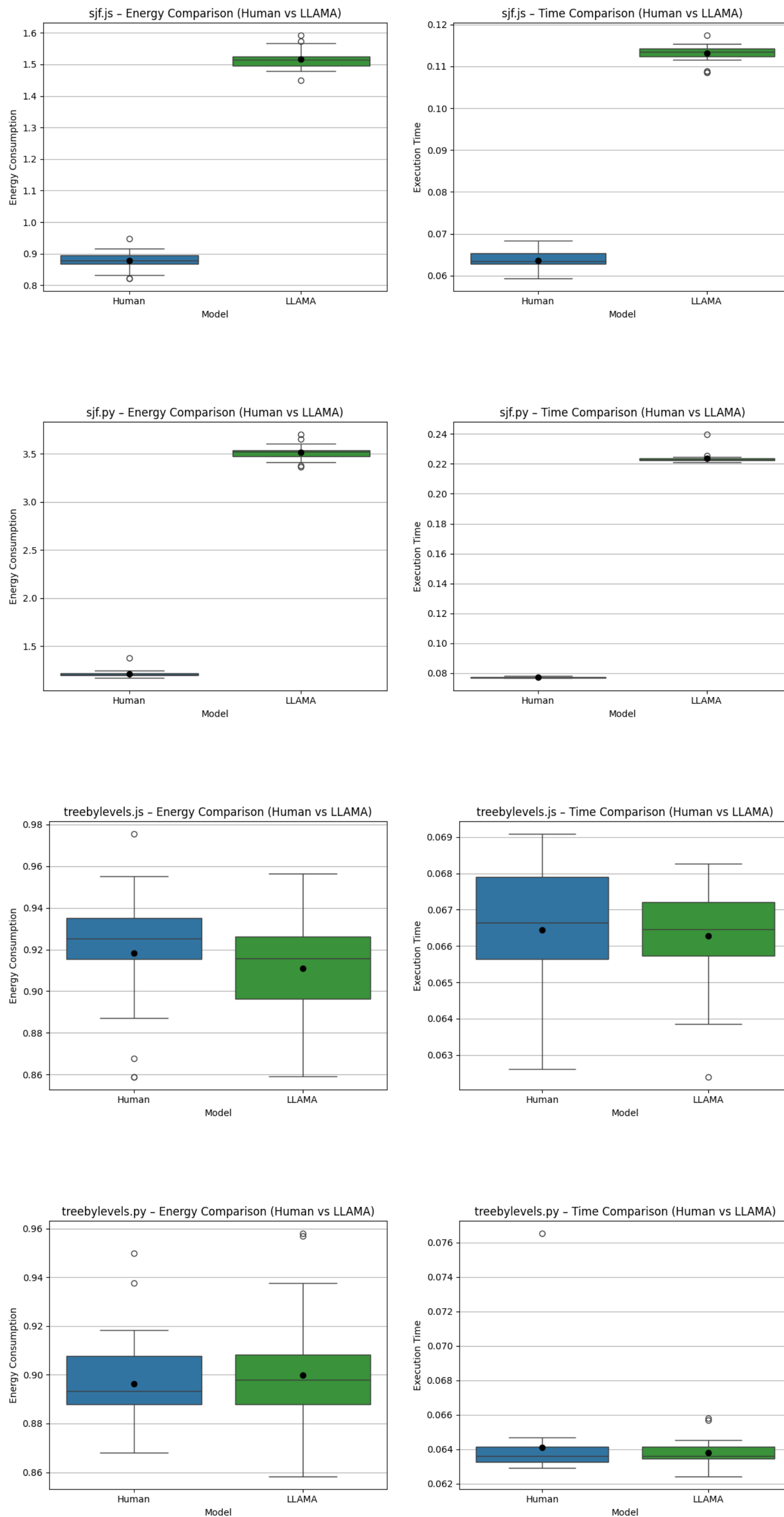


Figure 2. Visual Comparison of Runtime and Energy Usage Between LLaMA-3.3-Instruct generated and Human-Written Code Across All Tasks and Languages

4.1.3 Qwen2.5-Code generated code vs Human-written code

Energy and time results for Qwen2.5-Code are summarized in Table 11. Qwen2.5-Code was the only model that produced correct outputs for Rust across all selected problems. This sets it apart from other LLMs, which failed to return correct and runnable code in that language. Despite this, Qwen's performance in Rust was consistently less efficient than the human-written baseline, both in energy and execution time. In the Shortest Job First task, for example, it used 6 times more energy (1.49 J vs. 0.25 J) and took five times longer to execute (0.1 s vs. 0.02 s). Similar patterns have been seen for the Sort Tree by Levels problem.

Qwen's results were also relatively stable for JavaScript. Energy and time values stayed close to the human-written reference in all problems. Merge Arrays showed almost equal values (3.23 J vs. 3.26 J), and execution time was effectively identical. This trend continued in Supermarket Queue and Sort Tree by Levels where differences remained within small margins.

Python results were more erratic. In the Merge Arrays problem, Qwen consumed 6.97 J, over four times the human baseline. Execution time in this task was also much higher (0.43 s vs. 0.11 s). However, this was not consistent, for the Queue problem, Qwen showed almost the same numbers as the human version (5.38 J vs. 5.35 J). For the Sort Tree by Levels problem, Qwen even used less energy (0.62 J vs. 0.9 J). These shifts show unpredictable behaviour across different problem types, rather than consistent advantage or disadvantage.

Figure 3 visualize this variability. Python Merge Arrays and Rust SJF clearly stand out with higher energy use for Qwen in both energy and time. Other figures show very little difference between the two implementations.

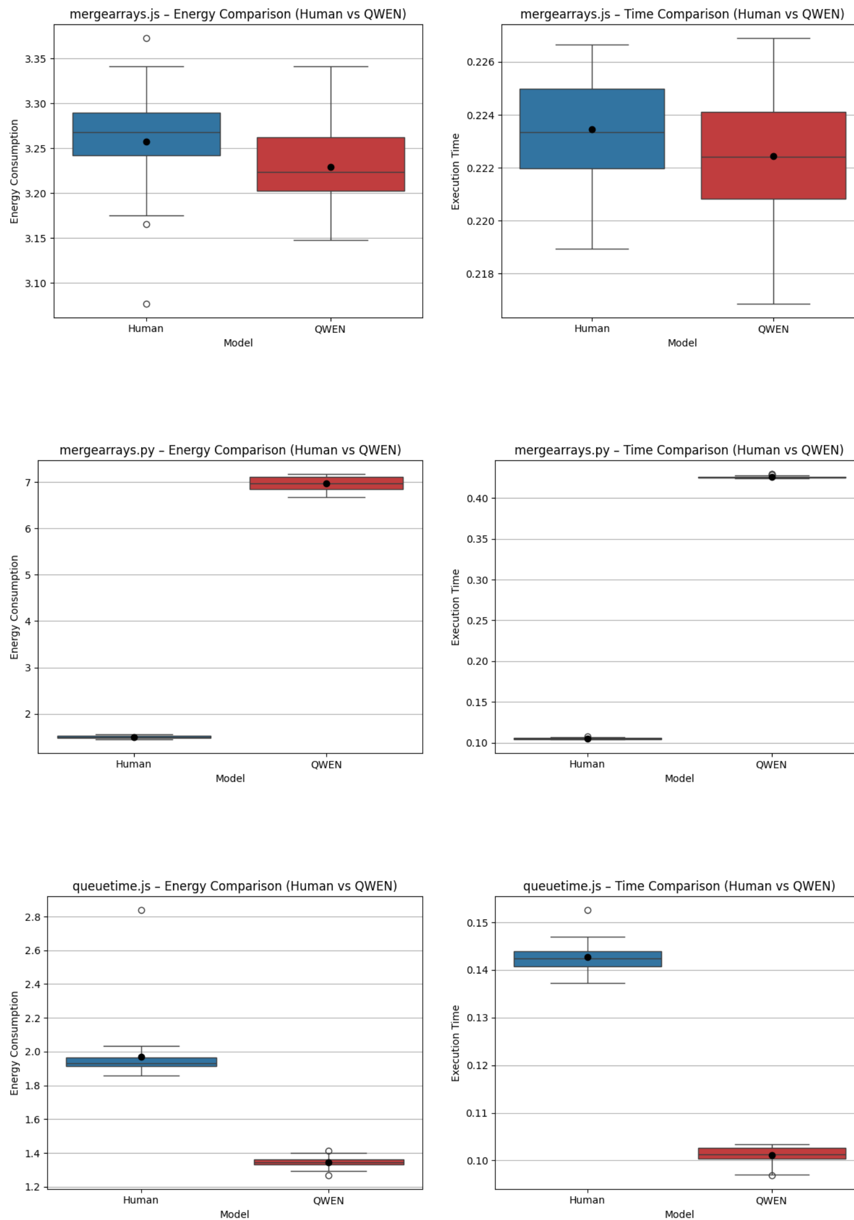
Statistical test results are shown in Table 12. The Mann-Whitney U test returned non-significant p-values for both energy ($p = 0.38$) and time ($p = 0.43$), which shows no reliable difference at the group level. Cliff's delta values were −0.24 for energy and −0.22 for execution time. These values suggest a small effect for the favor of the human-written code, with Qwen tending to perform slightly worse, though not consistently or significantly.

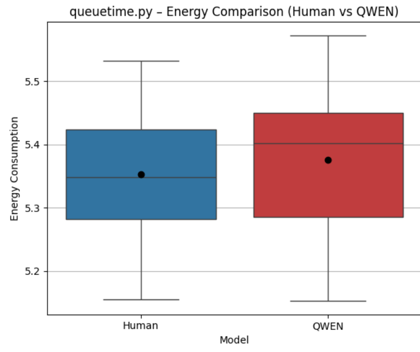
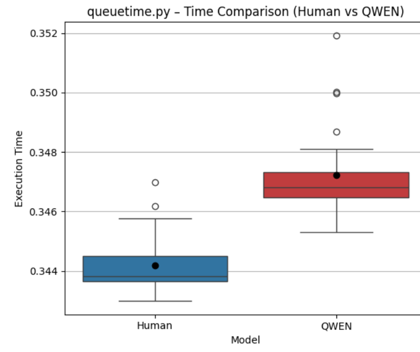
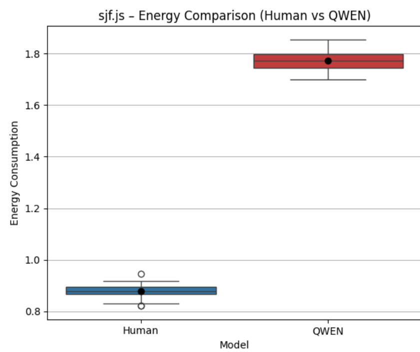
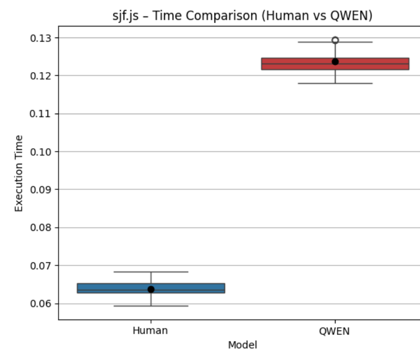
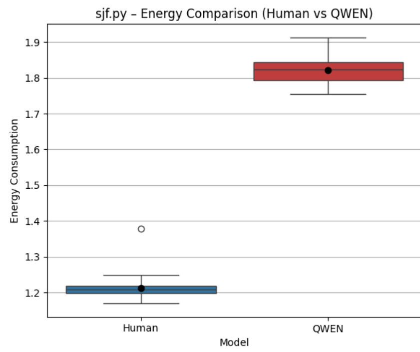
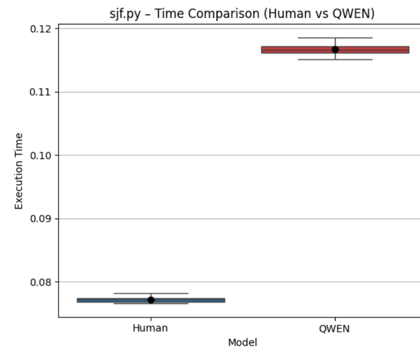
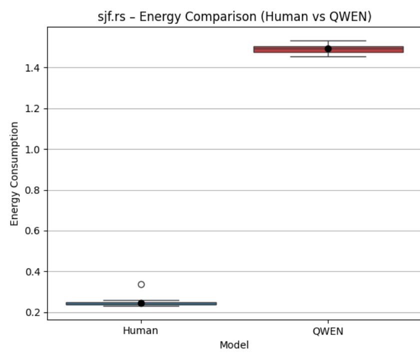
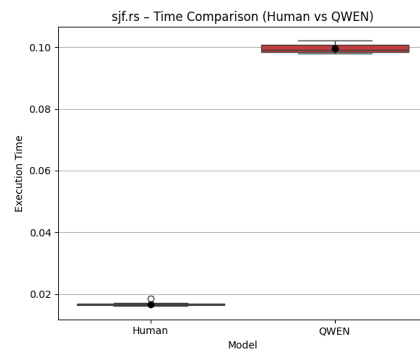
Table 11. Benchmark Results for Qwen2.5-Code generated and Human-written Code on All Tasks

Metrics	Mean Energy Usage (J)								Mean Execution Time (s)							
Problems	Merge Arrays		Supermarket Queue		SJF		Tree by Levels		Merge Arrays		Supermarket Queue		SJF		Tree by Levels	
LLMs	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**	**HW**	**Qwen**
JS	3.26	3.23	1.97	1.35	0.88	1.77	0.92	0.92	0.22	0.22	0.14	0.1	0.06	0.12	0.07	0.07
PY	1.5	6.97	5.35	5.38	1.21	1.82	0.9	0.62	0.11	0.43	0.34	0.35	0.08	0.12	0.06	0.04
Rust	0.82	-	2.21	-	0.25	1.49	0.2	0.2	0.06	-	0.16	-	0.02	0.1	0.01	0.01

Table 12. Statistical Comparison of Energy and Runtime Between Qwen2.5-Code generated and Human-Written Code

Metric	U-Statistic	p-Value	Cliffs Delta
mean_energy	38.0	0.3846730627355087	-0.24
mean_time	39.0	0.4273553138978077	-0.22




queuetime.py – Energy Comparison (Human vs QWEN)
Energy Consumption
Model
Human
QWEN
5.2
5.3
5.4
5.5

queuetime.py – Time Comparison (Human vs QWEN)
Execution Time
Model
Human
QWEN
0.344
0.346
0.348
0.350
0.352

sjf.js – Energy Comparison (Human vs QWEN)
Energy Consumption
Model
Human
QWEN
0.8
1.0
1.2
1.4
1.6
1.8

sjf.js – Time Comparison (Human vs QWEN)
Execution Time
Model
Human
QWEN
0.06
0.07
0.08
0.09
0.10
0.11
0.12
0.13

sjf.py – Energy Comparison (Human vs QWEN)
Energy Consumption
Model
Human
QWEN
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9

sjf.py – Time Comparison (Human vs QWEN)
Execution Time
Model
Human
QWEN
0.08
0.09
0.10
0.11
0.12

sjf.rs – Energy Comparison (Human vs QWEN)
Energy Consumption
Model
Human
QWEN
0.2
0.4
0.6
0.8
1.0
1.2
1.4

sjf.rs – Time Comparison (Human vs QWEN)
Execution Time
Model
Human
QWEN
0.02
0.04
0.06
0.08
0.10

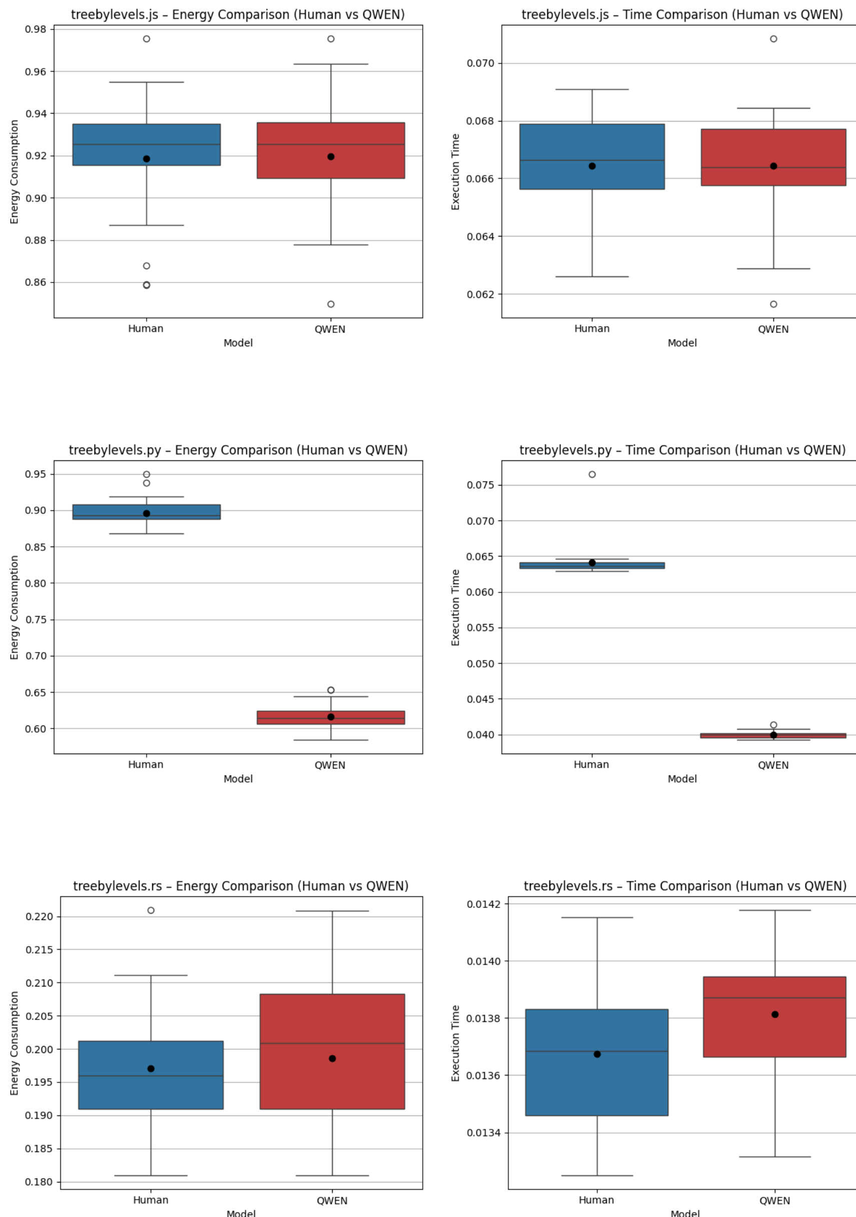


Figure 3. Visual Comparison of Runtime and Energy Usage Between Qwen2.5-Code generated and Human-Written Code Across All Tasks and Languages

4.2 Prompt Optimization Effects (RQ2)

This section investigates how prompting LLMs for energy efficiency influenced both energy consumption and execution time. The optimization prompts used here were relatively simple, we asked the models to avoid unnecessary loops or conditionals, for example. However, we did not provide any constraints or explicit examples. The outcomes varied across models. The

differences shown in Tables 11–13 represent the change between the default and optimized variants, with positive values meaning improvements.

4.2.1 GPT-4o

GPT-4o demonstrated small but consistent gains in response to energy-efficiency prompting. As seen in Table 13, positive delta values were recorded for both energy and time in three different tasks. The largest energy reduction was observed in the Python version of Supermarket Queue (0.97 J), followed by JavaScript Shortest Job First (0.95 J). Both also showed faster runtimes (−0.06 s). In the Sort Tree by Levels and Merge Arrays problems, the difference between default and optimized outputs was negligible.

These patterns are reflected in the statistical data in Table 14. The Mann-Whitney U test did not indicate significant differences ($p = 0.228$ for energy; $p = 0.372$ for time). Still, Cliff's delta values were moderate (0.44 and 0.33), suggesting some meaningful, although not strong, effects. Figure 4 confirm that the optimized versions often performed equally well or slightly better than the default ones.

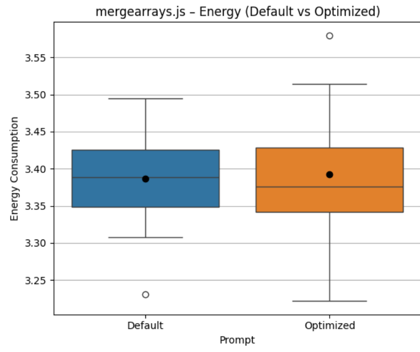
In general, GPT-4o reacted to the optimized prompt in a predictable manner. Improvements were modest but visible, and results were consistent across multiple problem types.

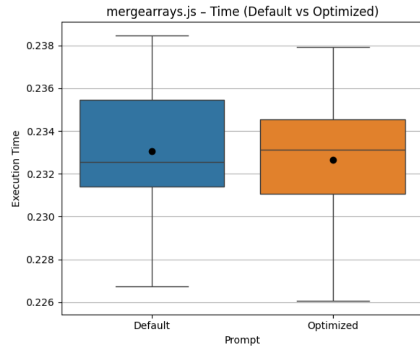
Table 13. Differences Between Default and Optimized GPT-4o Generated Code in Energy Usage and Execution Time

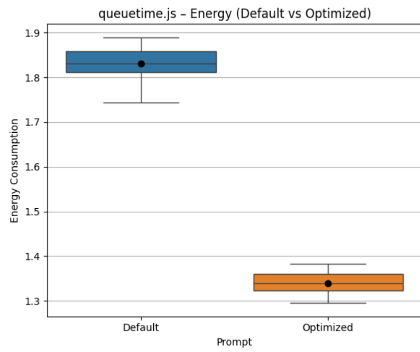
Problem	∆ Energy	∆ Execution Time
Merge Arrays/ JS	0.0	0.0
Supermarket Queue/ JS	-0.49	-0.04
Supermarket Queue/ PY	-0.97	-0.06
SJF/ JS	-0.95	-0.06
Sort Binary Tree by Levels/ JS	-0.02	0.0
Sort Binary Tree by Levels/ PY	-0.01	0.0

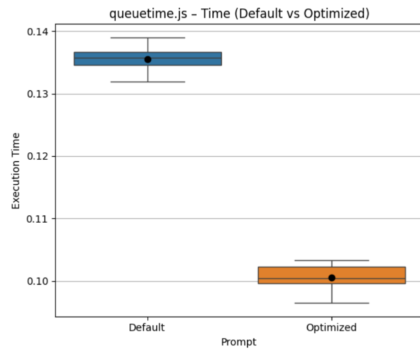
Table 14. Statistical Analysis of Energy and Runtime Differences in GPT-4o Code with and without Optimization Prompt

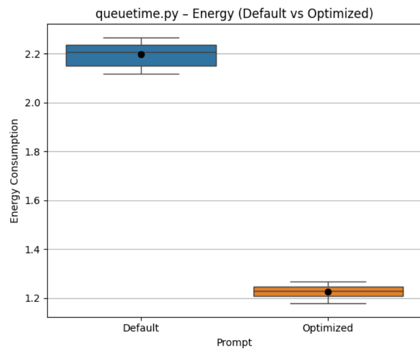
Model	Metric	U-statistic	p-value	Cliff's delta	Effect size
GPT	Energy	26.0	0.2281	0.4444	medium
GPT	Time	24.0	0.3726	0.3333	medium

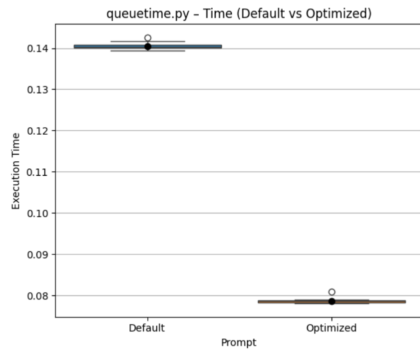

mergearrays.js – Energy (Default vs Optimized)
Energy Consumption
3.55
3.50
3.45
3.40
3.35
3.30
3.25
Default
Optimized
Prompt

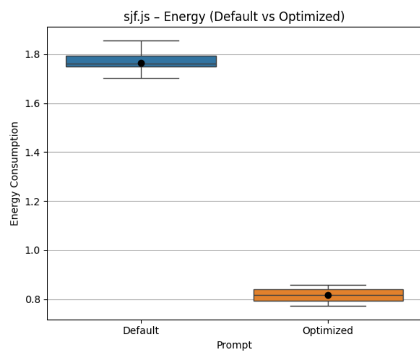

mergearrays.js – Time (Default vs Optimized)
Execution Time
0.238
0.236
0.234
0.232
0.230
0.228
0.226
Default
Optimized
Prompt

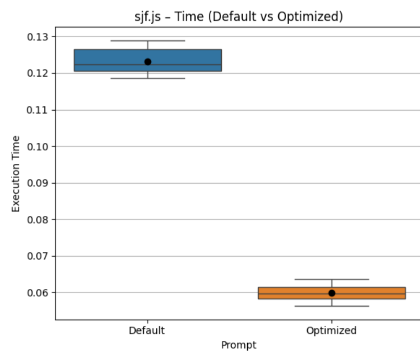

queuetime.js – Energy (Default vs Optimized)
Energy Consumption
1.9
1.8
1.7
1.6
1.5
1.4
1.3
Default
Optimized
Prompt


queuetime.js – Time (Default vs Optimized)
Execution Time
0.14
0.13
0.12
0.11
0.10
Default
Optimized
Prompt


queuetime.py – Energy (Default vs Optimized)
Energy Consumption
2.2
2.0
1.8
1.6
1.4
1.2
Default
Optimized
Prompt


queuetime.py – Time (Default vs Optimized)
Execution Time
0.14
0.13
0.12
0.11
0.10
0.09
0.08
Default
Optimized
Prompt


sjf.js – Energy (Default vs Optimized)
Energy Consumption
1.8
1.6
1.4
1.2
1.0
0.8
Default
Optimized
Prompt


sjf.js – Time (Default vs Optimized)
Execution Time
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.06
Default
Optimized
Prompt

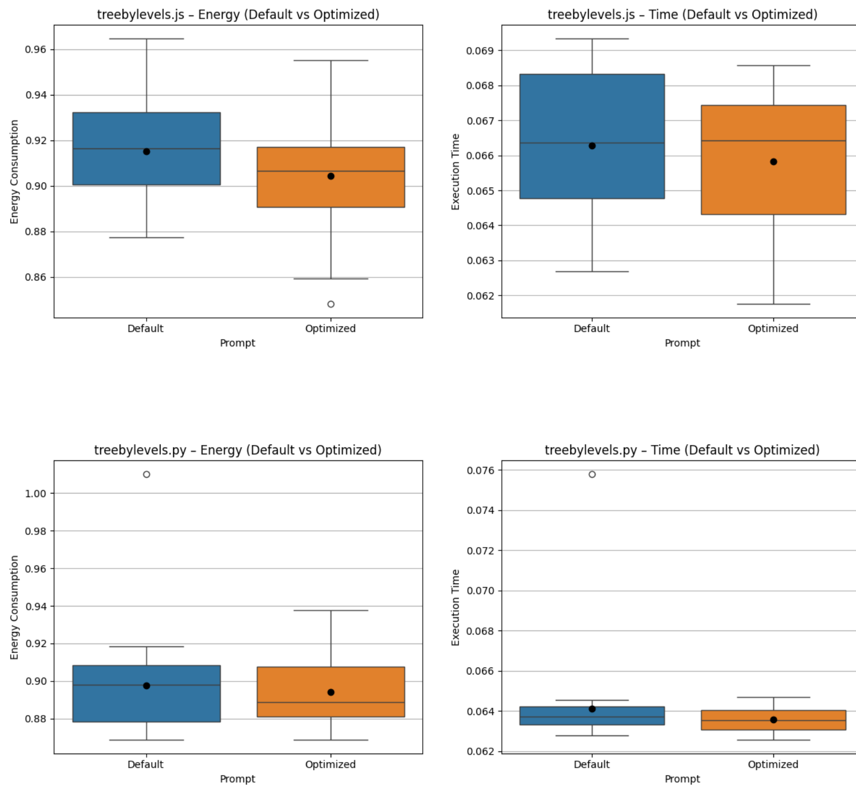


Figure 4. Effect of Prompt Optimization on Energy and Execution time for GPT-4o-Generated Code

4.2.2 LLaMA-3.3-Instruct

LLaMA's behavior was less stable. In the Python version of Shortest Job First, prompting for efficiency actually caused energy usage to increase by 18.62 J, and execution time increased by 1.15 s (Table 15). A similar result was seen in JavaScript Supermarket Queue, where energy and time also rose substantially (11.96 J and 0.75 s, respectively). These were among the highest deltas observed in the dataset, but unfortunately, in the wrong direction. Some tasks, like Python Supermarket Queue, showed slight improvement (−3.15 J, −0.21 s), but those were more the exception.

Table 16 shows that these changes were not statistically significant ($p = 0.958$ for energy, $p = 0.832$ for time), and Cliff's delta values were very low (−0.03 and −0.08).

Figure 5 illustrate this inconsistency-sometimes the optimized code improved, but often it did worse.

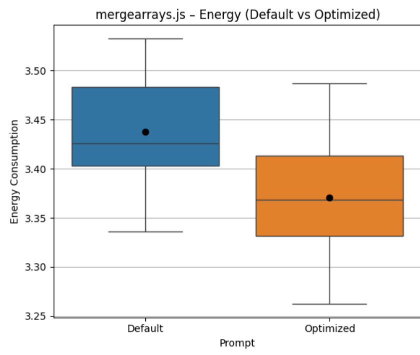
Overall, the optimization prompt introduced considerable instability in LLaMA's output. Instead of simplifying the code, in many cases it increased its complexity, resulting in more energy use and longer execution time.

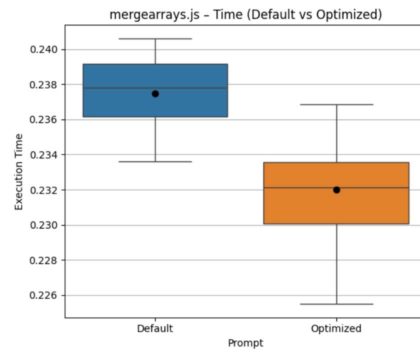
Table 15. Differences Between Default and Optimized LLaMA-3.3-Instruct-Generated Code in Energy Usage and Execution Time

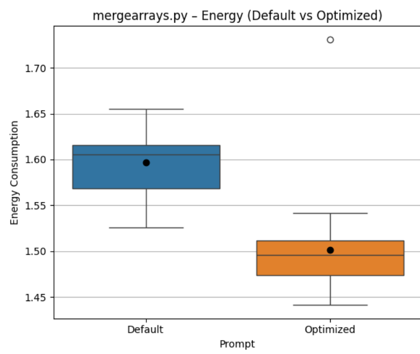
Problem	Δ Energy	Δ Execution Time
Merge Arrays/ JS	-0.07	-0.01
Merge Arrays/ PY	-0.1	0.0
Supermarket Queue/ JS	11.96	0.75
Supermarket Queue/ PY	-3.15	-0.21
SJF/ JS	0.0	0.0
SJF/ PY	18.62	1.15
Tree by Levels/ JS	0.02	0.0
Tree by Levels/ PY	0.0	0.0

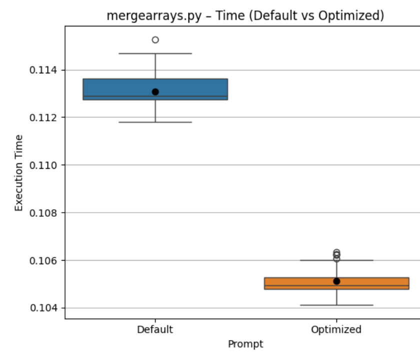
Table 16. Statistical Analysis of Energy and Execution Time Differences in LLaMA-3.3-Instruct Generated Code with and without Optimization Prompt

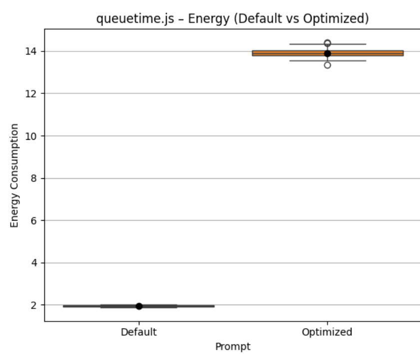
Model	Metric	U-statistic	p-value	Cliff's delta	Effect size
LLaMA	Energy	31.0	0.9581	-0.0312	Negligible
LLaMA	Time	29.5	832	-0.0781	Small

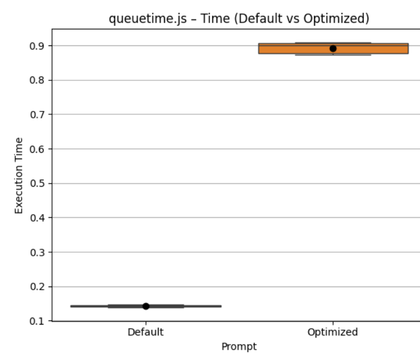

mergearrays.js – Energy (Default vs Optimized)
Energy Consumption
Default
Optimized
Prompt

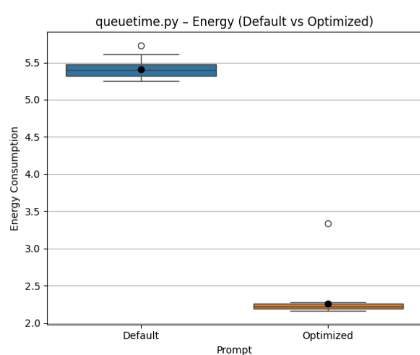

mergearrays.js – Time (Default vs Optimized)
Execution Time
Default
Optimized
Prompt

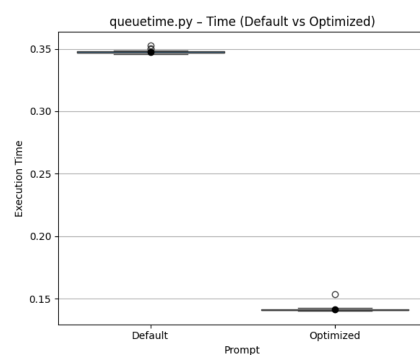

mergearrays.py – Energy (Default vs Optimized)
Energy Consumption
Default
Optimized
Prompt


mergearrays.py – Time (Default vs Optimized)
Execution Time
Default
Optimized
Prompt


queuetime.js – Energy (Default vs Optimized)
Energy Consumption
Default
Optimized
Prompt


queuetime.js – Time (Default vs Optimized)
Execution Time
Default
Optimized
Prompt


queuetime.py – Energy (Default vs Optimized)
Energy Consumption
Default
Optimized
Prompt


queuetime.py – Time (Default vs Optimized)
Execution Time
Default
Optimized
Prompt

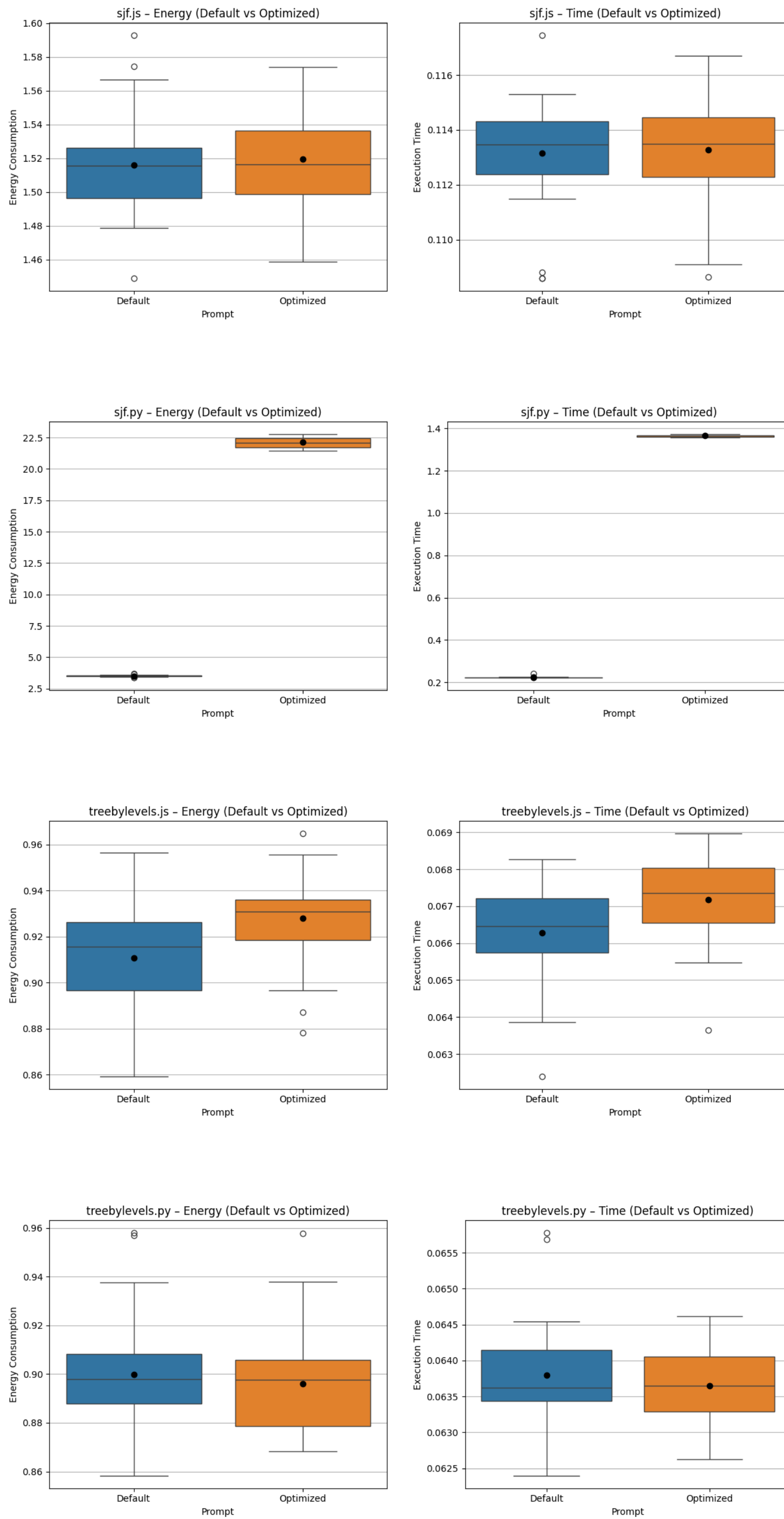


Figure 5. Effect of Prompt Optimization on Energy and Execution time for LLaMA-3.3-Instruct-Generated Code

4.2.3 Qwen2.5-Code

Qwen's response to prompting was mostly neutral. For the majority of tasks, there was no observed change, delta values for energy and time remained at 0.0, as reported in Table 17. However, there were a few exceptions. The most notable was in Python Supermarket Queue, where the optimized version used 3.14 J less energy and ran 0.21 s faster. On the other hand, in Sort Tree by Levels (Python), prompting resulted in a slight energy increase (1.3 J) and slower execution (by 0.08 s).

As summarized in Table 18, none of these changes reached statistical significance ($p = 0.629$ for energy; $p = 0.686$ for time), and the effect sizes were small (Cliff's $\delta = -0.19$ for energy, -0.17 for time). Figure 6 show minimal visual difference between prompt types across most tasks.

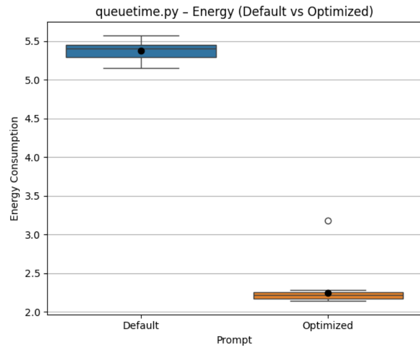
In conclusion, Qwen was the least responsive to energy-optimization prompts. Code structure remained mostly unchanged, and performance improvements were rare or minimal.

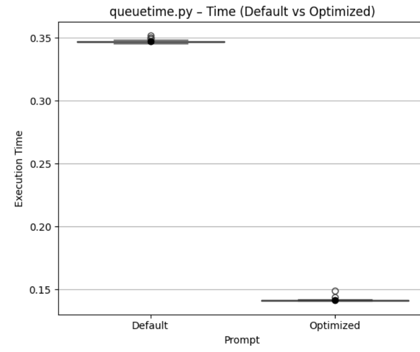
Table 17. Differences Between Default and Optimized Qwen2.5-Code-Generated Code in Energy Usage and Execution Time

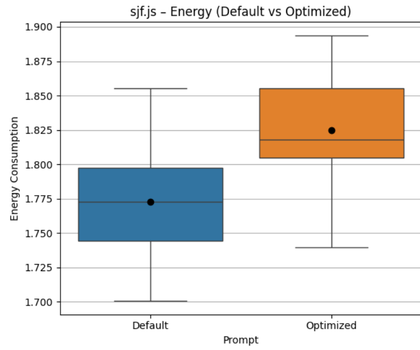
Problem	Δ Energy	Δ Execution Time
Supermarket Queue/ PY	-3.14	-0.21
SJF/ JS	0.05	0.01
SJF/ Rust	0.0	0.0
Tree by Levels/ JS	0.0	0.0
Tree by Levels/ PY	1.3	0.08
Tree by Levels/ Rust	0.0	0.0

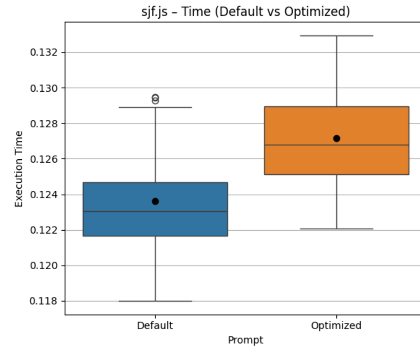
Table 18. Statistical Analysis of Energy and Execution Time Differences in Qwen2.5-Code Generated Code with and without Optimization Prompt

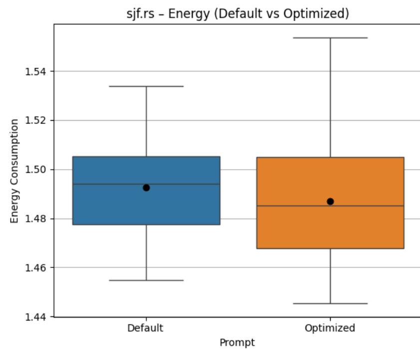
Metric	Model	U-statistic	p-value	Cliff's delta	Effect size
QWEN	Energy	14.5	0.6291	-0.1944	small
QWEN	Time	15.0	0.6868	-0.1667	small

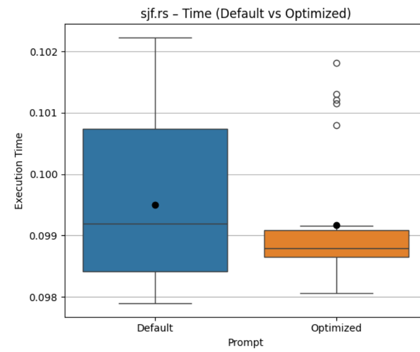

queuetime.py – Energy (Default vs Optimized)
Energy Consumption
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Default
Optimized
Prompt

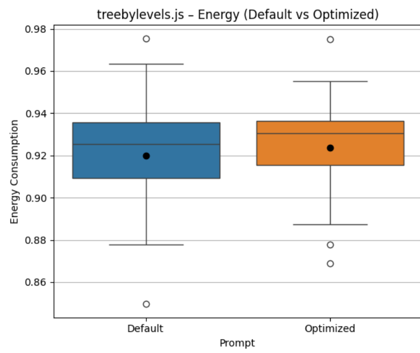

queuetime.py – Time (Default vs Optimized)
Execution Time
0.35
0.30
0.25
0.20
0.15
Default
Optimized
Prompt

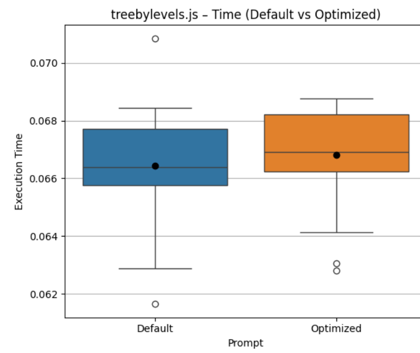

sjf.js – Energy (Default vs Optimized)
Energy Consumption
1.900
1.875
1.850
1.825
1.800
1.775
1.750
1.725
1.700
Default
Optimized
Prompt


sjf.js – Time (Default vs Optimized)
Execution Time
0.132
0.130
0.128
0.126
0.124
0.122
0.120
0.118
Default
Optimized
Prompt


sjf.rs – Energy (Default vs Optimized)
Energy Consumption
1.54
1.52
1.50
1.48
1.46
1.44
Default
Optimized
Prompt


sjf.rs – Time (Default vs Optimized)
Execution Time
0.102
0.101
0.100
0.099
0.098
Default
Optimized
Prompt


treebylevels.js – Energy (Default vs Optimized)
Energy Consumption
0.98
0.96
0.94
0.92
0.90
0.88
0.86
Default
Optimized
Prompt


treebylevels.js – Time (Default vs Optimized)
Execution Time
0.070
0.068
0.066
0.064
0.062
Default
Optimized
Prompt

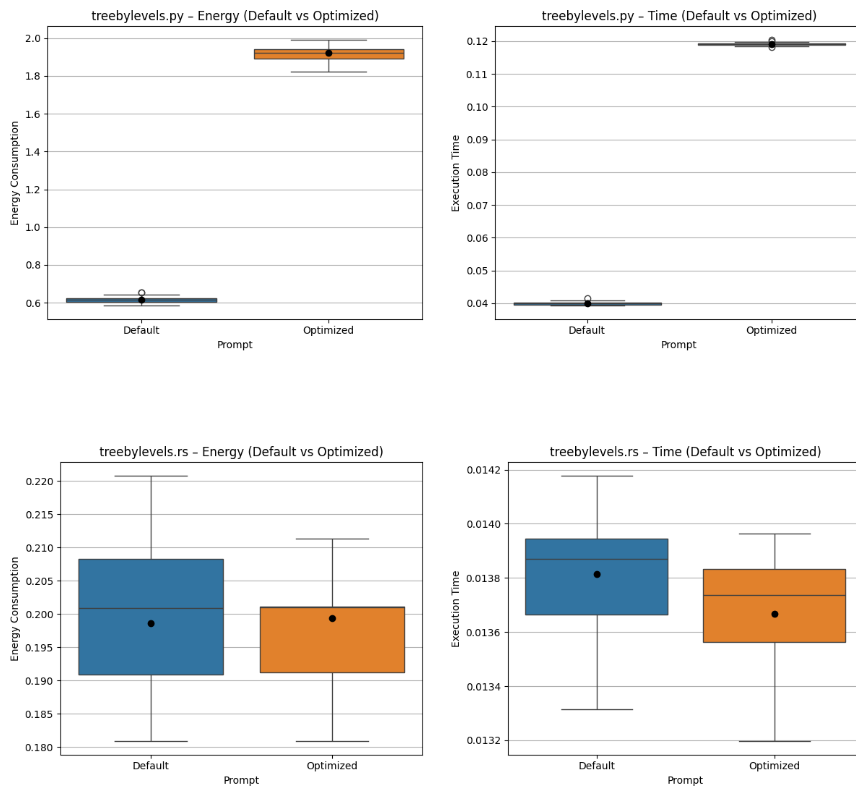


Figure 6. Effect of Prompt Optimization on Energy and Execution time for Qwen2.5-Code-Generated Code

4.3 Correlation Analysis

Is execution time related to energy consumption?

This part investigates the monotonic relationship between execution time and energy consumption. Spearman's rank correlation was applied to analyze this trend across all tested models and prompt variants. Results are grouped by source: human-written code, GPT-4o, LLaMA-3.3-Instruct, and Qwen2.5-Code.

In the case of human-written code, a strong correlation was observed. As can be seen in Figure 7-Human, energy consumption increased consistently as execution time became longer. The values followed a clear rising trend. Spearman's ρ was 0.95 ($p < 0.01$), which confirms a very strong positive relationship. (Table 19, row 1)

GPT-4o presented a similar pattern. In Figure 7- GPT, energy usage generally increased with runtime. Although the data showed slightly more dispersion compared to the human-written results, the direction of the relationship remained consistent. The correlation coefficient was 0.82 ($p = 0.03$), which also indicates a statistically significant result. (Table 19, rows 2 and 3)

For LLaMA-3.3-Instruct, the results showed a comparable upward trend. In Figure 7- LLaMA, both the default and optimized prompt outputs followed a rising pattern. While some data points indicated higher energy than expected, they still aligned with longer execution times. The Spearman's correlation was 0.78 ($p = 0.04$), suggesting a relatively stable relationship. (Table 19, rows 4 and 5)

Qwen2.5-Code showed a similar trend. Figure 7-Qwen demonstrates that most samples clustered along an increasing slope. Although a few deviations were seen, the overall trend remained upward. The computed Spearman coefficient was 0.81 ($p = 0.02$), which supports the presence of a strong correlation. (Table 19, rows 6 and 7)

In summary, execution time and energy consumption were positively correlated across all groups. The correlation was strongest for human-written code, but the pattern also held consistently for the generated code. This implies that, despite differences in output structure or prompt formulation, longer runtime generally results in higher energy usage.

Table 19. Spearman Correlation Between Execution Time and Energy Consumption

LLM	Spearmans rho (ρ)	p-value
HUMAN	0.9930	<0.000001
GPT	0.9940	<0.000001
GPT’	0.9856	0.000309
LLAMA	0.9701	0.000065
LLAMA’	0.9940	<0.000001
QWEN	0.9939	0.0
QWEN’	0.9429	0.004805

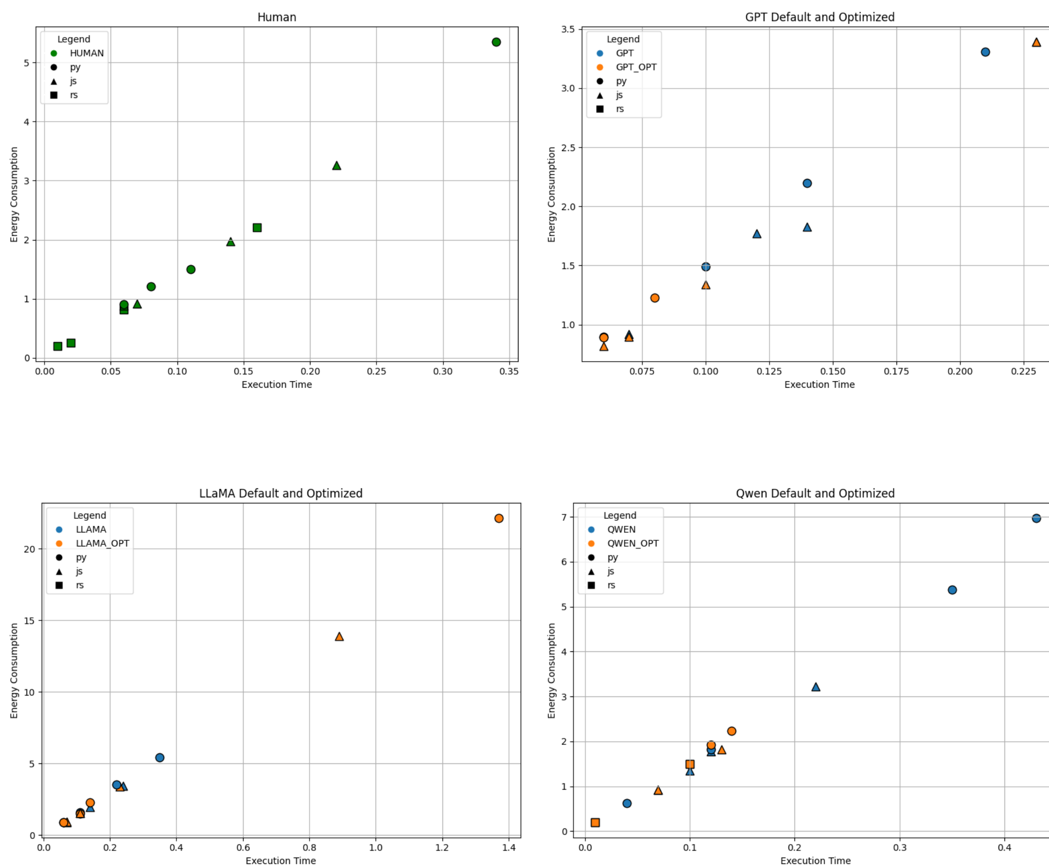


Figure 7. Correlation Between Execution Time and Energy Consumption

5. Discussion

Across different tasks and programming languages, none of the models showed consistent alignment with the efficiency levels of human-written code. GPT-4o was the one that came closest in this regard. On the other hand, both LLaMA and Qwen displayed more variation. It seems that task type and programming language both had some effect on these differences. Certain tasks, especially the ones involving more memory operations or nested structures, showed larger gaps. This was more visible in Python. It is possible that the interpreter overhead made the difference more clear in these cases. On the other hand, simpler tasks with flatter logic produced smaller differences. This was especially seen in JavaScript implementations. In such cases, model-generated code came closer to the human versions. So, the effect of task and language seems related to how the code runs on the platform and how complex the logic is. These two factors together appear to shape the energy and time behavior.

These findings are in line with earlier studies. For example, Vartziotis et al. compared ChatGPT, Copilot, and CodeWhisperer with human-written LeetCode submissions [14]. Their results indicated that AI-generated code, in most cases, consumed more energy and also ran slower. This remained the case even when the models were asked to generate "green" solutions. One possible reason they noted was that LLMs tended to generate general-purpose code rather than logic specific to the task, which may have caused additional overhead. This pattern was also present in the results of this study. For example, Qwen's Python solution for the Merge Ararys task used sorting, multiple list operations, and additional conditionals that were not strictly required. This caused higher energy usage compared to the human-written version (6.97 J vs. 1.50 J). For the SJF task, LLaMA's outputs in both JavaScript and Rust included simulation-style logic. The implementations had branching, counters, and structure that made the logic more layered. On the other hand, GPT-4o generated code and the human-written code used a simpler approach. They calculated the waiting time using arithmetic based on job durations, without added simulation. A similar thing was seen in the Tree By Levels task. The Rust version by LLaMA built the tree with recursive calls and handled the queue in a verbose way. This increased the energy consumption, even when the runtime did not change much. These examples reflect the observation by Vartziotis et al [14]. Their study pointed out that LLMs tend to generate more general-purpose code. The code is often functional and safe, but sometimes more complex than what is needed. This extra logic may support flexibility, but it leads to higher cost in terms of computation, especially in small or repeated programs.

Prompting the models for energy efficiency led to small changes in some cases. In several tasks, Qwen and LLaMA showed small reductions in energy use when optimization prompts were applied. This was especially the case in Python. However, in other cases, prompting had no effect or slightly worsened the outcome. GPT-4o appeared to benefit the least from the optimized prompt. This might be due to the fact that its default code was already closer to optimal, both in terms of structure and resource usage. In the Supermarket Queue task, for example, the optimized prompt slightly reduced energy use for Qwen in JavaScript but introduced additional conditionals in Python, increasing both time and energy. LLaMA showed similar patterns in SJF, where prompted versions added branching logic that was absent in the default output. These results show that the model's interpretation of the prompt is critical, and that simple wording does not always translate to energy improvements.

Across all valid samples we measured, energy consumption and execution time were positively correlated. In this scenario, execution time can act as a practical and useful stand-in for energy usage, especially in code that runs for a short time and depends mainly on the CPU. This is important because it allows researchers and developers to estimate energy behavior even when it is not possible to directly measure energy. Time is also easier to record and compare between different systems. Our results add more support for this approach, at least in limited and controlled testing conditions.

Still, one should remember that correlation is not the same as causation. Two different pieces of code may take the same time to run but consume different amounts of energy, depending on how they make use of system-level resources such as cache or memory bandwidth. Even so, in the context of this study, energy and time values scaled close enough to each other that we can say the relationship is meaningful and not just a coincidence.

Overall, the study contributes to an ongoing discussion on the sustainability of LLM-generated code. It underlines the importance of not only correctness but also computational awareness in future systems.

6. Limitations and Threats to Validity

There are several limitations to this study. First, the dataset was small. It included four programming tasks implemented in three different languages. Therefore, the findings may not generalize well to larger software projects or more complex tasks. Only three models were included. GPT-4o, LLaMA 3.3-Instruct, and Qwen2.5-Code were selected based on availability and support for instruction-following. Other models, especially those trained for performance or fine-tuned with extra constraints, might behave differently. Since LLMs evolve fast, these results should be seen more as a snapshot in time.

The study used only three programming languages: Python, JavaScript, and Rust. These were chosen to represent different language categories, but of course they do not cover all programming paradigms. Results could be different in compiled object-oriented languages, functional environments, or low-level system programming.

All measurements were conducted on a single physical device. Because of this, results might differ on other hardware configurations or execution setups.

Only valid outputs were included in the evaluation. Any generated code that failed functional tests was excluded. This was necessary for making a meaningful comparison of energy and execution time. Measuring incorrect or broken code would not give reliable results. However, this decision narrows the dataset and leaves out cases where models failed to solve the task, something that happens in actual usage. While this approach improves comparability, it also removes data that could offer insights into failure cases or model limitations.

Each model was prompted once per task. No retries or sampling variation were applied. This reflects how developers usually use LLMs in practice, generating code in a single pass, but at the same time, it limits the insight into model variability. A second or even third generation might have produced more efficient or slightly better structured code. Results, in that sense, represent a single output rather than a broader average.

Optimization prompts were short, general phrases used. No detailed structure were provided. This reflects how developers often write prompts, but it also limits how much guidance the model receives. The prompts used were simple. They did not include specific structural instructions. More complex prompt designs might result in different model behaviors. The fact that some optimized outputs looked almost identical to the original shows that prompt design plays a big role in whether the model even tries to optimize.

Prompt interpretation is another source of variation. The same instruction may be interpreted differently depending on the model's training data or internal alignment. Some models returned

significantly changed solutions when prompted for energy efficiency. Others made only slight edits. This inconsistency reflects differences inside the models and makes it harder to separate the effect of the prompt itself.

Human-written code was collected from Codewars' "Best Practice" solutions. These are high-quality and usually peer-reviewed. They may not reflect what an average developer might write under time pressure or in daily work. So, the gap between LLM-generated and human-written code could be smaller in less ideal scenarios. Still, these solutions provide a stable and reliable baseline for evaluation.

Many of the model-generated Rust solutions were dropped due to compilation or runtime issues. One could argue that Rust was a bad choice in this context. Model outputs were not manually validated beforehand, which reduced the effective dataset. This was a trade-off to preserve automation and comparability. In future experiments, manually validating or pre-filtering Rust code could allow for richer comparisons.

Statistical conclusions are limited by the number of valid samples. The dataset is small, especially after splitting by task, language, and model. Mann-Whitney U tests and Cliff's delta were used to reduce sensitivity to distribution shape. Still, small sample sizes reduce statistical strength. Effect sizes were included to help interpretation, but they should be treated as indicative, not final.

Despite these limitations, the study reflects a realistic setup. Models were tested under the same conditions, and all outputs were evaluated using consistent tools. The patterns we observed, especially the low impact of prompting and variation between models, are based on reproducible and verifiable measurements.

7. Conclusion

This thesis examined the energy and runtime characteristics of code generated by large language models, comparing them to human-written solutions across Python, JavaScript, and Rust. We obtained outputs from GPT-4o, LLaMA-3.3-Instruct, and Qwen 2.5-Coder by using controlled experiments and low-level CPU profiling.

Our results show that LLM-generated code can match human-written code in both runtime and energy consumption, but rarely exceeds it. Among all three models,GPT-4o produced the most stable outputs. LLaMA exhibited inconsistent behavior, while Qwen was the only model to generate valid Rust code, though with higher energy usage.

Prompting for energy efficiency resulted in mixed outcomes. GPT-4o showed moderate improvements in some cases, but overall, prompting did not consistently improve energy performance. Statistical analysis did not show significant group-level differences between LLM and human code, but small task specific variations were observed. These findings also align with prior research highligthing that LLM-generated code does not guarantee energy efficiency, even when explicitly prompted [12].

This work contributes a reproducible framework for measuring the energy profile of generated code. And provides empirical insights into the current capabilities and limitations of LLMs in producing energy-efficient software. Future research should investigate more robust prompting strategies and model fine-tuning to expand energy-aware code generation.

Future work could address several of the limitations identified here. First, more structured and diverse prompts should be explored. These might include few-shot examples [34], performance constraints, or even code-style instructions. Such prompts may help models to generate output that is more efficient.

Second, outputs in low-level languages, especially Rust, could be filtered or corrected before evaluation. This step could improve dataset coverage and reduce bias caused by parsing or compilation failures. It would also help ensure that more samples are usable in the final analysis.

Third, the current framework might be extended. It could include a broader set of tasks, larger codebases, or more realistic benchmarks. These additions would help to understand model behavior in different settings. They would also allow for more detailed analysis of how energy cost changes with complexity, language, or the size of the model.

Finally, future tools could integrate energy feedback during generation. Real-time measurement might allow developers to adjust code as it is produced. This could support more sustainable software practices.

References

[1] Manner J. Black software—the energy unsustainability of software systems in the 21st century. *Oxford Open Energy* 2 (2023), oiac011.

[2] Ji Z. An Interdisciplinary Exploration of Concept and Application of Large Language Models. *Applied and Computational Engineering* 133 (2025), pp. 8–15.

[3] Hou W. and Ji Z. Comparing large language models and human programmers for generating programming code. *Advanced Science* 12.8 (2025), p. 2412279.

[4] Lannelongue L., Grealey J., and Inouye M. Green algorithms: quantifying the carbon footprint of computation. *Advanced science* 8.12 (2021), p. 2100707.

[5] Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Fernandes J. P., and Saraiva J. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), p. 102609.

[6] Coignion T., Quinton C., and Rouvoy R. A performance study of llm-generated code on leetcode. *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 79–89.

[7] Şanlıalp İ., Öztürk M. M., and Yiğit T. Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices. *Electronics* 11.3 (2022), p. 442.

[8] Kempen N. van, Kwon H.-J., Nguyen D. T., and Berger E. D. It's Not Easy Being Green: On the Energy Efficiency of Programming Languages. *arXiv preprint arXiv:2410.05460* (2024).

[9] Cappendijk T., Reus P. de, and Oprescu A. Generating Energy-efficient code with LLMs. *arXiv preprint arXiv:2411.10599* (2024).

[10] Ilager S., Briem L. F., and Brandic I. GREEN-CODE: Optimizing Energy Efficiency in Large Language Models for Code Generation. *arXiv preprint arXiv:2501.11006* (2025).

[11] Solovyeva L., Weidmann S., and Castor F. AI-Powered, But Power-Hungry? Energy Efficiency of LLM-Generated Code. *arXiv preprint arXiv:2502.02412* (2025).

[12] Cursaru V.-A., Duits L., Milligan J., Ural D., Sanchez B. R., Stoico V., and Malavolta I. A controlled experiment on the energy efficiency of the source code generated by code llama. *International Conference on the Quality of Information and Communications Technology*. Springer. 2024, pp. 161–176.

[13] Huang D., Qing Y., Shang W., Cui H., and Zhang J. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems* 37 (2024), pp. 11506–11544.

[14] Vartziotis T., Dellatolas I., Dasoulas G., Schmidt M., Schneider F., Hoffmann T., Kotsopoulos S., and Keckeisen M. Learn to code sustainably: An empirical study on llm-based green code generation. *arXiv preprint arXiv:2403.03344* (2024).

[15] Idrisov B. and Schlippe T. Program code generation with generative ais. *Algorithms* 17.2 (2024), p. 62.

[16] Masanet E., Shehabi A., Lei N., Smith S., and Koomey J. Recalibrating global data center energy-use estimates. *Science* 367.6481 (2020), pp. 984–986.

[17] Mancebo J., García F., and Calero C. A process for analysing the energy efficiency of software. *Information and Software Technology* 134 (2021), p. 106560.

[18] Anwar H. and Pfahl D. Towards greener software engineering using software analytics: A systematic mapping. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2017, pp. 157–166.

[19] Peng H., Gupte A., Eliopoulos N. J., Ho C. C., Mantri R., Deng L., Jiang W., Lu Y.-H., Läufer K., Thiruvathukal G. K., et al. Large Language Models for Energy-Efficient Code: Emerging Results and Future Directions. *arXiv preprint arXiv:2410.09241* (2024).

[20] Kucharavy A. From Deep Neural Language Models to LLMs. *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*. Springer, 2024, pp. 3–17.

[21] Rosas M. R., Sanchez M. T., and Eigenmann R. Should ai optimize your code? a comparative study of current large language models versus classical optimizing compilers. *arXiv preprint arXiv:2406.12146* (2024).

[22] Codewars. Codewars: Achieve mastery through coding practice and challenges. Accessed: 2025-05-14. 2025. https://www.codewars.com.

[23] Stack Overflow. Stack Overflow Developer Survey 2024. Accessed: 2025-05-14. 2024. https://survey.stackoverflow.co/2024/technology.

[24] PerfWiki Contributors. Perf Tools Support for Intel Processor Trace. Accessed: 2025-05-14. 2024. https://perfwiki.github.io/main/perf-tools-support-for-intel-processor-trace/.

[25] Meta AI. LLaMA 3.3 70B Instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct. Accessed: 2025-05-14. 2024.

[26] Qwen Team. Qwen2.5-Coder-32B-Instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct. Accessed: 2025-05-14. 2024.

[27] Zhang Z., Wen L., Zhang S., Chen D., and Jiang Y. Evaluating gpt's programming capability through codewars' katas. *International Conference on Knowledge Science, Engineering and Management*. Springer. 2024, pp. 17–26.

[28] Rubei R., Moussaid A., Di Sipio C., and Di Ruscio D. Prompt engineering and its implications on the energy consumption of Large Language Models. *arXiv preprint arXiv:2501.05899* (2025).

[29] Liu F., Liu Y., Shi L., Huang H., Wang R., Yang Z., Zhang L., Li Z., and Ma Y. Exploring and evaluating hallucinations in LLM-powered code generation (2024). *URL https://arxiv.org/abs/2404.00971* (2024).

[30] Bansal N., Kimbrel T., and Pruhs K. Speed scaling to manage energy and temperature. *Journal of the ACM (JACM)* 54.1 (2007), pp. 1–39.

[31] Cruz L. Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments. https://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post. 2021.

[32] Virtanen P., Gommers R., Oliphant T. E., Haberland M., Reddy T., Cournapeau D., Burovski E., Peterson P., Weckesser W., Bright J., et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17.3 (2020), pp. 261–272.

[33] Macbeth G., Razumiejczyk E., and Ledesma R. D. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10.2 (2011), pp. 545–555.

[34] Brown T., Mann B., Ryder N., Subbiah M., Kaplan J. D., Dhariwal P., Neelakantan A., Shyam P., Sastry G., Askell A., et al. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Fidan Süleymanova**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Code Smarter Not Harder: Measuring Energy Efficiency of LLM-Generated Code, supervised by Hina Anwar.
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence **CC BY NC ND 3.0**, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Fidan Süleymanova
15/05/2025