Do LLMs Dream of Energy-Efficient Code?

ANTIMO DI BERNARDO*, Independent Researcher, Italy
GIANLUCA CAPOZZI, Sapienza University of Rome, Italy
PASQUALE DE ROSA, University of Neuchâtel, Switzerland
DANIELE CONO D'ELIA, Sapienza University of Rome, Italy
LEONARDO QUERZONI, Sapienza University of Rome, Italy
GIUSEPPE ANTONIO DI LUNA, Sapienza University of Rome, Italy
VALERIO SCHIAVONI, University of Neuchâtel, Switzerland

Large language models (LLMs) are drawing considerable attention as tools for automatic code generation. LLM-generated code is increasingly integrated with human-written code in the so-called vibe-coding process, producing software that is already being used within companies. A recent study by Huang et al. (NeurIPS 2024) investigates the time efficiency of LLM-generated code, showing that it still lags behind human-written code. In this paper, we extend this line of research by focusing on the energy efficiency of code produced by LLMs. We compare the energy consumption of 812 Python programs generated by several LLMs on a large set of benchmark problems. Our experiments show that LLM-generated code is, on average, less energy efficient than human-written code. The best model, GPT-3.5-turbo, generates solutions that consume roughly 50% more energy than human ones. Interestingly, larger (yet closed) models, such as Gemini and GPT-4o, can improve the energy efficiency of their solutions when provided with profiler feedback, although still lagging behind human-written code.

ACM Reference Format:
Antimo Di Bernardo, Gianluca Capozzi, Pasquale De Rosa, Daniele Cono D'Elia, Leonardo Querzoni, Giuseppe Antonio Di Luna, and Valerio Schiavoni. 2026. Do LLMs Dream of Energy-Efficient Code?. In *Proceedings of The Third International Workshop on Large Language Models for Code (LLM4Code'26).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large language models (LLMs) are increasingly used to support developers in code generation and related programming tasks. Recent studies examine how well these models perform at coding [7, 32], and further extend the investigation also to more subtle respects such their biases in adhering to software design patterns [25] or their effectiveness on algorithmic and competitive programming challenges [15]. One can envision a future where LLMs will be able to tackle coding tasks of medium to high complexity. Thus, LLMs will take the lion's share of development effort, with human developers tasked to create specific, sensitive code that AI cannot handle or to patch what AI generates. The speed with which LLMs progress [14] makes this scenario more likely each day.

The economic aspect—that is, the entire cost of running a system, comprising its development, deployment, operation, maintenance, *etc.*—will be a paramount push in the substitution of a large number of human developers with neural

*This research was conducted while the author was affiliated with Sapienza University of Rome.

Authors' Contact Information: Antimo Di Bernardo, dibernardo.2057634@studenti.uniroma1.it, Independent Researcher, Italy; Gianluca Capozzi, capozzi@diag.uniroma1.it, Sapienza University of Rome, Rome, Italy; Pasquale De Rosa, pasquale.derosa@unine.ch, University of Neuchâtel, Neuchâtel, Switzerland; Daniele Cono D'Elia, delia@diag.uniroma1.it, Sapienza University of Rome, Rome, Italy; Leonardo Querzoni, querzoni@diag.uniroma1.it, Sapienza University of Rome, Rome, Italy; Giuseppe Antonio Di Luna, diluna@diag.uniroma1.it, Sapienza University of Rome, Rome, Italy; Valerio Schiavoni, valerio.schiavoni@unine.ch, University of Neuchâtel, Neuchâtel, Switzerland.

2026. Manuscript submitted to ACM

models. The direct costs of using models will obviously be much lower than the costs needed to train and retain human experts. However, the development cost is only a fraction of the expenses required to run complex software. One has to factor in the costs needed to run and maintain the software.

Energy consumption is a key consideration in the overall cost of running software. In the long run, an inefficient algorithm may amount to a significant amount of consumed energy. Theoretically, the economic gain obtained using AI models to develop code may be eventually eroded by the increased energy costs of a "power-wasting" program. Saving energy is also a moral imperative for environmental reasons, and a long term objective also highlighted by the UNESCO Sustainable Development goals[1].

In this paper, we investigate the energy efficiency gap between code generated by state-of-the-art LLMs and human-developed counterparts. We aim to answer the following research questions:

- *Is code generated by LLMs more or less energy-efficient than code written by humans?*
- *Are LLMs able to produce energy efficient code when asked to do so?*
- *Are LLMs able to improve the energy efficiency of the code when using the feedback of a profiler?*

Contribution. We perform an analysis of the energy efficiency of Python code generated by several classes of LLMs on 812 programming challenges and problems from the EffiBench dataset [16], a state-of-the-art benchmark for LLMs. We chose Python as the programming language because it is currently among the most widely used languages [17].

From our analysis, we observe that:

- The solutions generated by LLMs, including large models such as GPT-4o, are on average less energy efficient than those implemented by human developers.
- When explicitly prompted to generate energy-efficient code, all examined LLMs fail to do so and instead markedly increase the energy consumption of their generated solutions.
- Large LLMs such as GPT and Gemini are able to produce more energy-efficient code if provided with a (correct) solution they previously generated and an energy profile from executing it, and asked to improve its efficiency. However, this comes at the expense of a reduced number of correctly solved problems. Almost all the smaller models we examined fail instead to generate more efficient solutions under the same conditions.

2 Related Work

The capability of LLMs to generate functioning code has been investigated in several studies [2, 4, 7, 15, 16]. The majority of these studies focus on evaluating correctness aspects [1, 3].

A recent research trend examines not only the correctness but also the efficiency of LLM-generated code in terms of running time and memory consumption. In this line of work, correctness is typically verified by evaluating the program's output on a set of known inputs, effectively performing testing by example.

The first paper in this line of research [21] evaluates the runtime efficiency of solutions generated by LLMs versus humans on two benchmarks (HumanEval [4] and MBPP [2]) and a set of problems sampled from LeetCode [19], a web-based platform offering a repository of programming and algorithmic problems used for education, skill assessment, and interview preparation. The study's main findings reveal that there is no correlation between a model's ability to generate correct code and its efficiency. In fact, the model generating the most correct solutions does not necessarily produce the most efficient ones. Additionally, a larger model parameter count does not impact the time efficiency of generated code.

[1]https://sdgs.un.org/goals

Dataset sample problem: ‘longest substring‘

“problem_idx": 3,
“task_name": "Longest Substring Without Repeating Characters",
“description": "Given a string s, find the length of the longest substring without repeating characters. [...] s consists of English letters, digits, symbols and spaces.",
“human_solution": "class Solution: def lengthOfLongestSubstring(self, s: str) [...] return ans",
“test_case_generator": "[...]",
“test_case": "assert
solution.lengthOfLongestSubstring('krLKl6F') == 7 [...]",
“small_test_cases": "[...]"

The authors of the EffiBench dataset [16] compare the efficiency of solutions generated by LLMs and humans, along with an extensive experimental evaluation of the most common LLMs (GPT-4, CodeLlama, CodeGemma, and others) on the proposed benchmark. Their main findings show that time and memory efficiency of LLM-generated solution is usually much worse than the human solutions.

We remark that the two above papers focus on efficiency from a runtime and memory consumption perspective, without explicitly considering the energy efficiency of the generated code. To the best of our knowledge, ours is the first study to examine this dimension.

While there is a certain correlation between computational efficiency and energy efficiency [28], they are not directly equivalent. Energy consumption can be influenced by additional factors, such as hardware characteristics and specific energy overheads associated with memory access and I/O operations. Authors in [5, 26] report that the fastest programming languages are not always the most energy-efficient. Similarly, [27] reports as a misconception the idea of a direct relationship between time and energy efficiency. Moreover, both the efficiency studies above [16, 21] did not consider the impact of different prompts on the generated code's efficiency.

Finally, loosely related to our work but worth mentioning, we are aware of another line of research pertaining the energy efficiency of LLMs models during training and inference [30].

3 Methodology

In this section, we describe the experimental methodology behind our study and experiments. We leverage a dataset (described in §3.1) of 812 algorithmic problems, each accompanied by reference human-written solutions in Python. Then, for each problem, we first ask LLMs to generate a solution with the algorithm implemented in Python. This first prompt we employ is focused solely on the correctness of the solution we ask a model to generate. We then prompted the same model to generate a solution with an emphasis on energy efficiency. Next, we took the solution generated with the correctness-focused prompt, ran it through a profiler on a range of inputs, and collected various performance-related metrics. We then provided this log to the model along with the previously generated solution, requesting optimization for energy efficiency. We further detail these code-generation and prompt-engineering operations in §3.3. Finally, as we explain in §3.4, each of the generated solutions and their human-developed counterparts is executed in an isolated environment where we monitor their energy consumption.

3.1 Dataset

We used for the study the EffiBench dataset from [16], designed to assess the code efficiency of LLM-generated solutions.

Table 1. Characteristics of LLMs evaluated.

Model	Parameters	Fine-tuned for code?	Context window
gpt-3.5-turbo	175B	✗	16K tokens
gpt-4o	175B	✗	128K tokens
gemini-1.5-flash	–	✗	1M tokens
llama3-8b	8B	✓	8K tokens
codellama-7b	7B	✓	16K tokens
mistral-7b	7.3B	✓	8K tokens
gemma-7b	7B	✗	8K tokens
wizardcoder-7b	7B	✓	2K tokens

EffiBench contains 1000 algorithmic coding problems, each with a reference solution implemented by a verified human. These problems are gathered from LeetCode [19]. For each problem, the authors of EffiBench chose as reference human solution the most upvoted one among those provided by the LeetCode user base.

We pick for the study a subset of 812 problems from the 1000 in EffiBench. This choice is due to (1) errors produced by the human solutions when employed outside of the LeetCode environment, or (2) the provided implementations do not pass some of the automatic tests provided by the Leetcode platform.

The EffiBench dataset also includes a set of 100 test cases per problem, automatically generated by LLMs. In addition to these, some problems also contain extra test cases written by humans. We use the 100 LLM-generated test cases (and, when available, the human-written ones) to evaluate the energy efficiency of both LLM-generated and human-written code.

The code snippet provided in the above figure is an example of the details available for each of the samples from the dataset.

3.2 Experimental testbed

Our hardware testbed allowed us to test models with up to around 7 billions of parameters. In the case of open models, we trained and tested the models on a server-grade machine equipped with a Intel Xeon Gold 6326 CPU with 32 cores clocked at 2.9Ghz, 64 GB of RAM, and 512 GB of SSD storage. We use this machine also for the power consumption measures, which we will cover in §3.4. As for the closed models, their operation is fully remote.

3.3 LLMs and Code Generation

Table 1 reports the LLM models for our experimental evaluation, selected according to their capability to solve coding problems and their commonality. We used both closed and open models. Specifically: GPT-3.5 [23], GPT-4o[24] and Gemini-1.5-flash [12] for closed models, and llama3-8b [9], codellama-7b [29], mistral-7b [18], gemma-7b [13] and wizardcode-7b [20] for open models.

To interact with closed models, we rely on the public APIs exposed for these services. We rely on the Ollama framework [22] (v0.12.3) to deploy and interface with the open-source models, leveraging its native support for handling the full deployment and execution lifecycles of such open LLMs directly on-premises. Inference for open models is always executed on our experimental testbed (§3.2).

For code generation, we used three types of prompts. The first (exemplified in the box `Prompt 1`) is focused solely on code correctness. The second (box `Prompt 2`) prompts the model to prioritize energy efficiency. The third is longer and richer (box `Prompt 3`), as we provide the code generated from `Prompt 1` and an execution energy profile of this code obtained with `cProfiler` [6], asking the model to generate a new solution with improved energy efficiency. In

`Prompt 1` (focus on correctness)

Please, based on the task description, write a solution to pass the provided test cases. You must follow these rules:

(1) The code should be in a `python [Code]` block.
(2) Do not add the provided test cases into your `python [Code]` block.
(3) You do not need to write the test cases; they will be provided for you.
(4) Ensure that the provided test cases can pass your solution.

Task description: `{class_desc}`.
Test cases: `{class_test_case_small}`.
Remember to account for all possible cases to ensure a correct solution.

`Prompt 2` (optimize for energy efficiency)

Please, based on the task description, write a solution to pass the provided test cases. **You must focus on energy efficiency**: this includes optimizing data structures and loops, reducing the number of calls and accesses, caching functions, etc.

(1) The code should be in a `python [Code]` block.
(2) Do not add the provided test cases into your `python [Code]` block.
(3) You do not need to write the test cases; they will be provided for you.
(4) Ensure that the provided test cases can pass your solution.

Task description: `{class_desc}`.
Test cases: `{class_test_case_small}`.

Remember to account for all possible cases to ensure a correct solution. **Focus on correctness and energy efficiency!**

`Prompt 3` (example of solution and `cProfile` trace)

I will provide you with a Python solution to a problem, along with its description, the test cases, and an analysis made with cProfiler regarding the code's time efficiency. Your task is to try, if possible, to provide an optimized version of the solution while maintaining the correctness of the original code.
Task description: `{class_desc}`.
Original version of the solution: `{orig_vers}`
Result of the analysis made by the Profiler: `{profile}`

all prompts, we restrict the solutions to pure Python code—and we defer to §5 a discussion of the limitations of this approach.

We check the correctness of the solutions generated by each prompt by running the code against the 100 LLM generated test cases from EffiBench, as described in §3.1. We discard from further consideration all solutions failing against such testcases, removing them from the test set used for energy-consumption measuring purposes (§3.4).

3.4 Power Consumption Measurement

To measure power consumption, we rely on PowerMeter[2], a tool that incorporates components from the PowerAPI framework [10]. PowerMeter enables monitoring of power consumption at various levels, including individual processes. It comprises two main components: *(1)* the Hardware Performance Counter (HWPC) Sensor, which monitors Intel CPU performance and power consumption via RAPL (Running Average Power Limit) [8], and *(2)* SmartWatts Formula (based on [11]), which estimates the power consumption of monitored software based on the data collected by the sensor.

Each program solution executes inside a Docker container. Since HWPC requires the PID of the monitored process, we initially pause the container running the solution to retrieve the PID of the process, and then resume execution until completion.

[2]https://powerapi.org

Operationally, this is done by issuing the command `docker pause` to pause the container. The PID of the solution program is then retrieved and added to `perf_event`. Once this setup is complete, the container is restarted using `docker unpause`. During the pause period, since the program is not running, the measured energy consumption is zero.

The HWPC Sensor operates with a minimum sampling period of 500 ms. Hence, we run both the LLM and human solutions in continuous loops on our test cases to ensure a minimum execution time of 2 seconds and to gather at least 4 samples. To achieve a fair comparison between human and LLM solutions, we enforce the same number of executed loops on both: doing so, the execution time of the faster solution remains above 2 seconds.

With the aforementioned process, we are able to measure the total joules consumed by each solution while executing the test cases. We validated the measurements obtained by HWPC with a restricted set of experiments using a PDU-based hardware sensor that measures the consumption of the entire system. We observe that these hardware measurements are noisy, as it is not possible to measure the consumption of a single process. However, we were able to observe consumption trends that are compatible with our software-based measurements based HWPC. We discuss the details of these measurements in the reminder of the next section.

4 Experimental Results

This section presents our extensive experimental evaluation. Our evaluation intends to answer the following research questions:

- **RQ1:** How does LLM-generated code compare to human-written code in terms of energy consumption when solving the same programming problem?
- **RQ2:** Can LLMs make their generated code more efficient if they are given execution data, *i.e.,* the output of an energy profiler?

We considered four evaluation metrics: Time/Energy WDL, Normalized Execution Time, Normalized Consumed Energy, and the percentage of Correct solutions from the LLM, as detailed next.

Time/Energy W(in)–D(raw)–L(ose). We report the number of problems on which the LLM outperforms humans, respectively draws or underperforms. An LLM is considered to win on a problem when it is least 10% more efficient (in terms of time or energy) than the human solution. Similarly, we consider it to lose against a human solution when it is at least 10% less efficient[3].

Normalized Execution Time (NET). We report the average ratio between the total execution time of the LLM-generated solutions (*i.e.*, the time required to execute each generated solution over all test inputs) and the total execution time of the human-made solutions, computed over all correct solutions produced by the LLM.

Formally, let S be the set of correct solutions generated by the LLM. We define:

$$NET(S) = \frac{1}{|S|} \sum_{s \in S} \frac{Time(s)}{Time(Human(s))},$$

where $Time(s)$ denotes the execution time of the LLM-generated solution s, and $Time(Human(s))$ is the execution time of the corresponding human-written solution for the same problem.

Normalized Consumed Energy (NCE). Analogous to NET, NCE tracks total energy consumption. Formally:

$$NCE(S) = \frac{1}{|S|} \sum_{s \in S} \frac{Energy(s)}{Energy(Human(s))},$$

[3]This specific margin was chosen to avoid possible oscillations due to measurement errors.

Table 2. `Prompt 1` *vs.* Human.

LLM Model	Time: W-D-L	Energy: W-D-L	NET	NCE	Correct
Gpt-3.5-turbo	180 - 110 - 155	185 - 95 - 165	1.55	1.48	54.8%
Gpt-4o	262 - 139 - 236	262 - 120 - 255	1.75	1.68	78.5%
Gemini-1.5-flash	137 - 80 - 163	133 - 82 - 165	2.56	2.23	46.8%
Codellama:7b	13 - 7 - 28	17 - 4 - 27	6.42	5.83	5.9%
Llama3:7b	58 - 38 - 72	63 - 31 - 74	3.96	3.29	20.8%
Wizardcoder:7b	29 - 16 - 54	34 - 9 - 56	7.14	6.20	12.2%
Gemma:7b	28 - 9 - 45	28 - 8 - 46	8.84	7.07	10.1%
Mistral:7b	14 - 14 - 52	14 - 14 - 52	10.77	9.45	9.85%

Table 3. `Prompt 2` *vs.* Human.

LLM Model	Time: W-D-L	Energy: W-D-L	NET	NCE	Correct
Gpt-3.5-turbo	153 - 99 - 163	166 - 81 - 168	2.15	2.05	51.1%
Gpt-4o	247 - 132 - 254	261 - 114 - 258	2.30	2.06	77.96%
Gemini-1.5-flash	181 - 73 - 130	184 - 66 - 134	3.03	2.29	47.4%
Codellama:7b	20 - 16 - 48	22 - 12 - 50	7.03	5.60	10.35%
Llama3:7b	36 - 16 - 52	37 - 15 - 52	8.05	7.22	12.8%
Wizardcoder:7b	12 - 7 - 47	14 - 3 - 49	10.47	7.82	8.13%
Gemma:7b	0 - 5 - 12	1 - 3 - 13	17.49	14.08	2.1%
Mistral:7b	12 - 7 - 33	14 - 5 - 33	10.20	9.23	6.4%

Table 4. `Prompt 2` *vs.* `Prompt 1`.

LLM Model	Time: W-D-L	Energy: W-D-L	NET	NCE
Gpt-3.5-turbo	65 - 161 - 133	79 - 121 - 159	1.58	1.53
Gpt-4o	130 - 271 - 186	167 - 218 - 202	1.21	1.17
Gemini-1.5-flash	74 - 156 - 75	91 - 125 - 89	1.58	1.51
Codellama:7b	2 - 13 - 6	5 - 7 - 9	1.54	1.55
Llama3:7b	13 - 33 - 21	17 - 28 - 22	1.42	1.32
Wizardcoder:7b	4 - 10 - 11	7 - 5 - 13	2.10	2.18
Gemma:7b	-	-	-	-
Mistral:7b	2 - 4 - 9	3 - 3 - 9	1.50	1.47

where $Energy(s)$ denotes the energy consumed by the LLM-generated solution, and $Energy(Human(s))$ is the energy consumed by the human solution for the corresponding problem.

Correct: the percentage of correct solutions generated by the LLM for the 812 test problems. Since we measure power efficiency only for correct solutions, this metric helps compare results across different models.

For each solution, we report the average over four distinct executions. At each run, we log the elapsed time and the associated energy consumption. We enforce a timeout of five minutes for each run: if a solution exceeds this limit on a test case, its execution is aborted. This approach prevents bias in favor of models that produce highly inefficient solutions leading to frequent timeouts. Such an effect would unfairly benefit models that generate a few efficient solutions alongside many that are excessively slow.

4.1 Humans vs LLMs

Table 2 reports the comparison between LLMs and humans against `Prompt 1`. From the point of view of energy efficiency, GPT-3.5 and GPT-4.o have similar performance, with an NCE of 1.48 and 1.68, respectively. Their generated solutions are more efficient than a human's in roughly 41% of the cases (*e.g.*, the ratio of W on W-D-L). We note that GPT-4.o is solving significantly more problems than GPT-3.5 (*i.e.*, 644 problems instead of 449). Thus GPT-4o clearly outperforms GPT-3.5.

Table 5. `Prompt 3` *vs.* Human.

LLM Model	Time: W-D-L	Energy: W-D-L	NET	NCE	Correct
Gpt-3.5-turbo	120 - 79 - 120	127 - 59 - 133	1.53	1.45	39.29%
Gpt-4o	259 - 117 - 208	281 - 101 - 202	1.60	1.32	71.92%
Gemini-1.5-flash	96 - 84 - 105	106 - 66 - 113	1.99	1.94	35.1%
Codellama:7b	-	-	-	-	1.2%
Llama3:7b	21 - 14 - 26	22 - 13 - 26	1.56	1.46	7.5%
Wizardcoder:7b	7 - 4 - 7	7 - 2 - 9	2.62	2.30	2.22%
Gemma:7b	-	-	-	-	0%
Mistral:7b	4 - 8 - 7	5 - 8 - 6	10.96	10.95	2.34%

However, none of the tested models manage to be more energy efficient than an human. The NET mirrors the consumed energy, showing the same trend. The fact that, on average, solutions generated by LLMs are not on par, from the time efficiency point of view, with human solutions is consistent with the results of [16].

Interestingly, energy efficiency markedly worsens when using Prompt 2, which specifically asks to optimize the solution to minimize the energy footprint (see Table 3). For reference, GPT-4o produces solutions with an NCE of 2.06, compared to 1.68 with Prompt 1, corresponding to roughly a 22% degradation in performance. This behavior is consistent across all models. In general, the number of correct solutions also decreases when using Prompt 2. We observe exceptions, such as Gemini-1.5-flash, for which the number of correct solutions slightly increases, and CodeLlama:7b, for which the number of correct solutions doubles.

In Table 4 we report the comparison between solutions generated with Prompt 2 and Prompt 1. In this scenario, Prompt 1 serves as the reference for human-level solutions in the metrics of Time/Energy W–D–L, NET, and NCE. The table compares the solutions generated by Prompt 2 with those obtained using Prompt 1. For instance, the NCE value represents the average ratio between the energy consumed by solutions generated with Prompt 2 and that consumed by solutions generated with Prompt 1.

From the NCE values, it can be observed that all solutions perform worse than those generated with Prompt 1, with the best-performing model being GPT-4o, which achieves an NCE of 1.17. Results for Gemma-7b are omitted, as the number of correct solutions is negligible.

Overall, the results of these experiments indicate that LLMs struggle to understand and correctly use energy-efficient coding techniques. Interestingly, the situation changes in our test with Prompt 3, where the model is provided with a correct solution it previously generated using Prompt 1 and with an energy profiler report. Results are listed in Table 5. In this case, the larger models are able to correctly modify their solutions to improve energy efficiency.

For instance, in GPT-4o the NCE decreases from 1.68 with Prompt 1 to 1.32 with Prompt 3; in GPT-3.5 it drops from 1.48 to 1.45; and in Gemini-1.5-Flash from 2.23 to 1.94. This suggests that, once provided with both a correct solution and a profiler report, the models can identify and correctly optimize portions of code to enhance their energy efficiency. A similar trend is observed for the NET metric, which also improves under Prompt 3.

However, this comes at the cost of a reduced number of correct solutions. GPT-3.5 decreases from 54% with Prompt 1 to 39.29% with Prompt 3; Gemini shows a similar behavior (46.8% to 35.1%), while GPT-4o exhibits the smallest drop (78.5% to 71.92%).

We believe this behavior is related to the effect observed in [31], where it was shown that ChatGPT, when challenged about correct beliefs, often changes its mind, although incorrectly. A similar phenomenon may be at play here, where the models alter previously correct programming constructs in an attempt to improve efficiency.

The smaller open-source models (codellama, llama3, wizardcoder, gemma and mistral, each with 7 billion parameters) produce a negligible number of correct solutions (below 3%). Therefore, their efficiency and energy measurements are not meaningful in this context. The only exception is Llama 3, which, despite almost halving its number of correct solutions reaching 7.5%, reduces its NCE from 3.29 to 1.46.

In this case, `Prompt 1` takes the role of Human solutions in the measures of Time/Energy W-D-L, NET and NCE.

To summarize, larger models demonstrate the ability to enhance the energy efficiency of their solutions *only* when given profiler information; however, on average, they do not yet achieve human-level performance.

Answer to RQ1: *The results show that the tested LLMs lag behind human developers in both time efficiency (confirming previous studies) and energy efficiency of the generated programs. Interestingly, the model that generates the most energy-efficient solutions is not the one that achieves the highest rate of correct solutions and is, on average, roughly 50% less efficient than human solutions.*

Answer to RQ2: *Closed models such as GPT and Gemini are able to improve the energy and time efficiency of their generated solutions when provided with an execution trace from a profiler. Interestingly, in a non-negligible number of cases, the generated solutions are no longer correct, with the percentage of incorrect solutions ranging between 6% and 15%.*

4.2 Comparison with EffiBench

Our analysis (Table 2) indicates that, when considering `Prompt 1`, GPT-4.o is the best model at generating correct solutions, correctly solving 78.5% of the proposed problems. Instead, Gemini solves only 46.8%, and GPT-3.5 solves 54.8%. The open-source models have much worse performance: Llama3 can only produce 20.8% correct solutions. GPT-4o is the best model considering energy and time expenditure, and it achieves the best ratio of correctly solved problems. Interestingly, all models performs markedly worse against `Prompt 2`, achieving a comparable NCE among GPT models.

Our absolute time measurements differ from those reported in the EffiBench study [16]. However, they exhibit the same overall trend, and in particular LLMs tend to generate less time-efficient solutions. We speculate that this discrepancy stems from including additional test cases in our evaluation, as well as only considering a (large) subset (Section 3.1) of their problems. We leave to future work a more extensive comparison against EffiBench.

4.3 Qualitative Evaluation

To gain a qualitative understanding of the energy-wasting patterns introduced by LLMs, we manually analyzed the top-10 most energy-eager solutions generated by GPT-4o, GPT-3.5-turbo, and Gemini-1.5-flash using Prompt 1, comparing them against their counterparts produced by other models. The rationale is that examining the cases with the largest energy gap allows us to identify the factors contributing to energy inefficiency in LLM-generated code.

Our analysis reveals two main reasons behind the largest sources of energy waste:

- the use of unsuitable data structures, which failed to optimize memory usage and processing efficiency;
- improper loop design and lack of logic checks designed to minimize the number of loop iterations, significantly increasing computational load.

In the remainder, we analyze in further detail two specific problems, *relative sort* and *integer break*, as well as the produced solutions by GPT-4o, GPT-3.5-turbo, and Gemini-1.5-flash.

Relative Sort - GPT-3.5-turbo solution to Prompt 1

```
class Solution:
    def relativeSortArray(self, arr1, arr2):
        cnt = [0] * 1001
        result = []
        for num in arr1:
            cnt[num] += 1
        for num in arr2:
            result.extend([num] * cnt[num])
            cnt[num] = 0
        for i in range(1001): //flawed if ctn[i]>>1000
            if cnt[i] > 0:
                result.extend([i] * cnt[i])
        return result
```

Relative Sort - Gemini-1.5-flash solution to Prompt 1

```
class Solution:
  def relativeSortArray(self, arr1: List[int], arr2: List[int]) -> List[int]:
        count = {}
        for num in arr1:
            if num in count:
                count[num] += 1
            else:
                count[num] = 1
        res = []
        for num in arr2:
            res.extend([num] * count[num])
            del count[num]

        remaining = sorted(count.keys())
        for num in remaining:
            res.extend([num] * count[num])
        return res
```

Relative sort. The problem is: “Given two arrays $arr1$ and $arr2$, the elements of $arr2$ are distinct, and all elements in $arr2$ are also in $arr1$. Sort the elements of $arr1$ such that the relative ordering of items in $arr1$ are the same as in $arr2$. Elements that do not appear in $arr2$ should be placed at the end of $arr1$ in ascending order.”

The worst solution in this case is generated by GPT-3.5-turbo. In this approach, the solution uses an array of size 1001 named $count$ as a lookup table. This lookup table counts the occurrences of elements in $arr1$, where each element of $arr1$ serves as an index in $count$. First, the table is used to arrange the elements of $arr1$ in the same relative order as $arr2$. Then, it is used to append the remaining elements in ascending order by simply scanning $count$. This solution is flawed in certain cases; for example, if an element in $arr1$ is greater than 1000, the algorithm will fail. More importantly, it is generally inefficient since it scans the entire lookup table.

We qualitatively compare the solution produced by Gemini-1.5-flash. It uses a dictionary as a lookup table. This not only makes the algorithm correct in all cases, but also makes it much more efficient, particularly when the number of unique elements in $arr1$ is significantly less than a thousand.

Integer break. The problem is “Given an integer n, break it into the sum of k positive integers, where $k >= 2$, and maximize the product of those integers. Return the maximum product you can get”. The least efficient solution in this case is generated by Gemini-1.5-flash, which uses a dynamic programming strategy to solve the problem. This approach results in quadratic complexity.

In contrast, GPT-3.5-turbo and GPT-4.0 produce a more optimal solution by maximizing the product of factors. These two models observe that the maximum number of factors is reached when these factors are equals and of the minimum acceptable quantity, that is three for the problem definition.

Integer break - Gemini-1.5-flash solution to Prompt 1

```
class Solution:
    def integerBreak(self, n: int) -> int:
        if n <= 3:
            return n - 1
        dp = [0] * (n + 1)
        dp[2] = 2
        dp[3] = 3
        for i in range(4, n + 1):
            for j in range(1, i):
                dp[i] = max(dp[i], dp[j] * dp[i - j], j * (i - j))
        return dp[n]
```

Integer Break - GPT-3.5-turbo solution to Prompt 1

```
class Solution:
    def integerBreak(self, n: int) -> int:
        if n == 2:
            return 1
        if n == 3:
            return 2
        product = 1
        while n > 4:
            product *= 3
            n -= 3
        return product * n
```

Unique Paths - GPT-4 solution to Prompt 1

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[1] * n for _ in range(m)]
        for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i-1][j] + dp[i][j-1]
        return dp[m-1][n-1]
```

Unique Paths - GPT-4 solution to Prompt 3

```
from math import comb
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        return comb(m + n - 2, m - 1)
```

4.3.1 Prompt 3 Optimizations. In this section, we describe some of the notable optimizations applied by the LLMs when executing Prompt 3. For example, in the Unique Paths problem, the Prompt 1 solution of GPT-4 uses a dynamic programming strategy to compute the number of unique paths in a grid that reach the bottom-left corner. In contrast, the Prompt 3 solution of GPT-4 observes that this number can be obtained using a combinatorial formula and leverages Python's combinatorial functions, which provide a more efficient implementation. Another example is the Jump Game problem. In this task, an array of integers is given, where each element at position i represents the maximum number of steps that can be jumped forward from that position. The goal is to determine whether it is possible to reach the last index of the array starting from the first one. In this case, the Gemini-1.5-Flash solution for Prompt 1 iterates over all positions in the array when the answer is positive, whereas the Prompt 3 solution employs early termination, returning true as soon as it determines that the last position is reachable.

Jump Game - Gemini-1.5-flash solution to Prompt 1

```
class Solution:
    def canJump(self, nums: list[int]) -> bool:
        reachable = 0
        for i, num in enumerate(nums):
            if i > reachable:
                return False
            reachable = max(reachable, i + num)
        return True
```

Jump Game - Gemini-1.5-flash solution to Prompt 3

```
class Solution:
    def canJump(self, nums: list[int]) -> bool:
        n = len(nums)
        furthest_reach = 0
        for i in range(n):
            if i <= furthest_reach:
                furthest_reach = max(furthest_reach, i + nums[i])
                if furthest_reach >= n - 1:
                    return True
        return False
```

5 Limitations

One limitation of our study is that we measured energy efficiency using a software sensor based on hardware performance counters. This approach allowed us to estimate the power consumption of a single process. In a limited set of experiments, we verified that our measurements were consistent with those provided by a hardware monitor for the entire system. However, a more extensive evaluation is needed to further validate our measurements.

Another limitation is that, due to our computational resources, we restricted our analysis to small open-source models. Extending the evaluation to larger open-source models would be valuable to investigate any potential relationship between model size and energy efficiency.

Lastly, the dataset we used focuses exclusively on algorithmic problems, and we instructed the models to generate solutions in Python, which represents another limitation. A more comprehensive evaluation should include tasks beyond algorithmic problem-solving, such as assessing the energy efficiency of code that primarily interacts with the operating system, web services, networking, or similar domains. It would also be important to investigate whether the observed behavior changes when LLMs generate code in programming languages other than Python. Another interesting direction would be to explore Python implementations that leverage native extensions or just-in-time (JIT) compilation, as these approaches make Python's runtime behavior more comparable to that of compiled languages such as C or C++. However, a more comprehensive analysis should ultimately compare solutions generated in different programming languages to better understand the impact of language choice on energy efficiency.

6 Conclusion

In this paper, we took an initial step toward understanding the energy efficiency of code generated by LLMs. To ease reproducibility, we will release our data and code to the research community.

Our experimental evaluation shows that LLMs are behind human developers in both time efficiency and energy efficiency of the generated programs. However, they are able to improve the efficiency of their solutions when provided with execution information for the consumed energy.

We hope this effort will pave the way for more extensive studies, both in terms of benchmarking and measurement, as well as in designing specific prompts and models that prioritize energy efficiency. As our world faces an impending environmental crisis, we find that the LLM revolution must also address its environmental impact. That is not only the one of its inherent nature (that is, the energy demand of model training and inference) but also the one of its direct products. As the former aspect has been extensively acknowledged and studied, we hope that our research will start the discussion on the latter.

References

[1] B. Athiwaratkun, S. K. Gouda, Z. Wang, et al. 2023. Multilingual Evaluation of Code Generation Models. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR 2023).* Kigali, Rwanda. OpenReview.net.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021). https://arxiv.org/abs/2108.07732 CorpusID: 237142385.

[3] M. Chen, J. Tworek, H. Jun, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

[5] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP '17).* Association for Computing Machinery, Fortaleza, CE, Brazil. https://doi.org/10.1145/3125374.3125382

[6] cProfiler. [n. d.]. cProfiler. https://docs.python.org/3/library/profile.html#module-cProfile.

[7] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. Jiang. 2023. GitHub Copilot AI Pair Programmer: Asset or Liability? *Journal of Systems and Software* 203, C (September 2023), 1–23.

[8] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *ISLPED*. ACM, 189–194.

[9] A. Dubey et al. 2024. The Llama 3 Herd of Models. *arXiv preprint* (2024). arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[10] Guillaume Fieni, Daniel Romero Acero, Pierre Rust, and Romain Rouvoy. 2024. PowerAPI: A Python framework for building software-defined power meters. *J. Open Source Softw.* 9, 98 (2024), 6670.

[11] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. 2020. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *CCGRID*. IEEE, 479–488.

[12] Gemini Team et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint* (2024). arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[13] Gemma Team et al. 2024. Gemma: Open Models Based on Gemini Research and Technology. *arXiv preprint* (2024). arXiv:2403.08295 [cs.CL] https://arxiv.org/abs/2403.08295

[14] A. Ho et al. 2025. Algorithmic Progress in Language Models. In *Proceedings of the 38th International Conference on Neural Information Processing Systems (NeurIPS '24).* Vancouver, BC, Canada, 1–39.

[15] Md. S. Hossain, Anika Tabassum, Md. Fahim Arefin, and Tarannum Shaila Zaman. 2025. LLM-ProS: Analyzing Large Language Models' Performance in Competitive Problem Solving. In *LLM4Code@ICSE*. IEEE, 80–87.

[16] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang. 2024. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. In *Proceedings of the 38th Conference on Neural Information Processing Systems (NeurIPS 2024), Track on Datasets and Benchmarks.*

[17] Paul Jansen. 2025. TIOBE Index for October 2025. https://www.tiobe.com/tiobe-index/. Accessed: 20 Oct 2025.

[18] A. Q. Jiang et al. 2023. Mistral 7B. *arXiv preprint* (2023). arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[19] LeetCode. [n. d.]. LeetCode. https://leetcode.com/.

[20] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *Proceedings of the Twelfth International Conference on Learning Representations.*

[21] C. Niu, T. Zhang, C. Li, B. Luo, and V. Ng. 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (FORGE '24).*

[22] Ollama. [n. d.]. Ollama: Run Large Language Models Locally. https://github.com/ollama/ollama.

[23] OpenAI. 2023. GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5.

[24] OpenAI. 2023. *GPT-4 Technical Report.* Technical Report. https://doi.org/10.48550/arXiv.2303.08774

[25] Zhenyu Pan, Xuefeng Song, Yunkun Wang, Rongyu Cao, Binhua Li, Yongbin Li, and Han Liu. 2025. Do Code LLMs Understand Design Patterns?. In *LLM4Code@ICSE*. IEEE, 209–212.

[26] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE '17)*. Vancouver, BC, Canada, 256–267.

[27] G. Pinto, F. Castor, and Y. D. Liu. 2014. Mining Questions About Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. 22–31. https://doi.org/10.1145/2597073.259711

[28] Isabelly Rocha, Christian Göttel, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. 2019. Heats: Heterogeneity-and Energy-Aware Task-Based Scheduling. In *PDP*. IEEE, 400–405.

[29] B. Rozière et al. 2024. Code Llama: Open Foundation Models for Code. *arXiv preprint* (2024). arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[30] S. Samsi et al. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. In *Proceedings of the 2023 IEEE High Performance Extreme Computing Conference (HPEC)*. Boston, MA, USA, 1–9.

[31] Boshi Wang, Xiang Yue, and Huan Sun. 2023. Can ChatGPT Defend its Belief in Truth? Evaluating LLM Reasoning via Debate. (2023). https://arxiv.org/abs/2305.13160

[32] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. 1–10.