When Faster Isn't Greener: The Hidden Costs of LLM-Based Code Optimization

Tristan Coignion
Univ. Lille, CNRS, Inria
Lille, France
tristan.coignion@inria.fr
0009-0003-2525-3637

Clément Quinton
Univ. Lille, CNRS, Inria
Lille, France
clement.quinton@inria.fr
0000-0003-3203-6107

Romain Rouvoy
Univ. Lille, CNRS, Inria
Lille, France
romain.rouvoy@inria.fr
0000-0003-1771-8791

***Abstract*—*Large Language Models* (LLMs) are increasingly adopted to optimize source code, offering the promise of faster, more efficient programs without manual tuning. This capability is particularly appealing in the context of sustainable computing, where enhanced performance is often assumed to correspond to reduced energy consumption. However, LLMs themselves are energy- and resource-intensive, raising critical questions about whether their use for code optimization is energetically justified. Prior work mainly focused on runtime performance gains, leaving a gap in our understanding of the broader energy implications of LLM-based code optimization.**

In this paper, we report on a systematic, energy-focused evaluation of LLM-based code optimization methods. Relying on 118 tasks from the EvalPerf benchmark, we assess the trade-offs between code performance, correctness, and energy consumption of multiple optimization methods across multiple families of LLMs. We introduce the *Break-Even Point* (BEP) as a key metric to quantify the number of executions required for an optimized program to outweigh the energy consumed when generating the optimization itself.

Our results show that, while certain configurations achieve substantial speedups and energy reductions, these benefits often demand from hundreds to hundreds of thousands of executions to become energetically profitable. Moreover, the optimization process often yields incorrect or less efficient code. Importantly, we identify a weak negative correlation between performance gains and actual energy savings, challenging assumptions that faster code automatically equates to a smaller energy footprint. This work underscores the necessity of energy-aware optimization strategies. Practitioners should carefully target LLM-based optimization efforts to high-frequency, high-impact workloads, while monitoring energy consumption across the entire life-cycle of development and deployment.

***Index Terms*—large language model, code efficiency, code optimization, energy consumption**

I. Introduction

Large Language Models (LLMs) have revolutionized many aspects of software development, from code completion [1], [2] and bug detection [3] to code customization [4] and automatic code documentation [5]. Among these emerging capabilities, one particularly promising application is *code optimization*: automatically rewriting programs' code to improve performance. Recent advances in code-oriented LLMs have made it increasingly feasible to generate optimized versions of existing code, potentially reducing execution time, energy consumption, and resource usage without requiring deep expertise in performance engineering [6]–[8]. Most of these methods rely on iterative refinement [9] which enables them to adapt and improve based on the outcomes of each optimization step.

However, the growing popularity of these models coincides with increasing concerns about the environmental sustainability of computing. As digital infrastructure scales and AI systems proliferate, both the energy demands of running software and the energy required to train and operate these optimization models are becoming non-negligible. In their latest report, the World Bank and ITU estimated that the ICT sector accounts for a growing 1.7% of global greenhouse gas emissions [10], while the ADEME (French National Agency for the Ecological Transition) estimated that 11% of France's energy consumption stemmed from the ICT sector [11].

In this context, it is tempting to view LLM-based optimization as a straightforward win for sustainability, faster code often resulting in greener code. Unfortunately, the reality is more nuanced. While GPUs have become increasingly efficient, their overall energy consumption and environmental impact keep exponentially increasing [12]. LLMs, which depend heavily on GPU-based computation, are known to be very resource-intensive [13]–[15], requiring powerful inference servers for deployment and operation [16]. Thus, any potential savings achieved by optimizing code must be weighed against the substantial energy cost of using LLMs for the optimization itself. This concern raises a critical question for the community: *What is the real energy cost of using LLM-powered code optimizers? Is the performance and energy gain of optimized code sufficient to justify the energy cost of optimization?*

In this paper, we address this question by assessing the energy cost of LLM-based code optimizations and determining when it is energetically beneficial to optimize a given program. Additionally, our evaluation setup allows us to also compare the effectiveness of different LLM-based optimization methods in terms of both performance and correctness. To the best of our knowledge, this is the first study to examine not only the benefits of LLM-based code optimization but also its environmental trade-offs.

Specifically, we investigate the following research questions:

- **RQ1:** *What is the best LLM-based method to optimize code?*

- **RQ2:** *What is the energy necessary to produce one optimized result?*
- **RQ3:** *In terms of energy, at what point is it profitable to optimize a given code?*

To answer these questions, we evaluated 8 code optimization methods using the EvalPerf [17] benchmark. For each experiment, we measured both the energy consumption caused by the optimization process and the energy difference between unoptimized and optimized code. To quantify the energy trade-offs, we introduce a new metric that measures the number of optimized code executions required to compensate for the energy cost of the optimization. Our methodology is designed to be LLM-agnostic and easily extensible. As long as a model can be deployed via an inference server compatible with the OpenAI API, it can be integrated into our evaluation pipeline with minimal adjustments.

Our results show that, while most LLM-based optimization strategies are capable of reducing the energy consumption of code, the energy savings they deliver do not always offset their own energy costs. In some cases, the optimized code must be executed tens of thousands of times before the energy spent on optimization is compensated by the energy saved. For example, a large model like DeepSeek-R1-14B combined with a powerful optimization method can reduce energy usage by up to 49% on average, but only becomes energetically beneficial after approximately 200,000 executions. In contrast, smaller models like Qwen2.5-Coder-7B can achieve worthwhile savings with far fewer executions, making them more suitable for lightly used code samples or one-off scripts. Notably, in some high-consumption scenarios, we observed energy reductions from 318 J to just 2.6 J, which offset the optimization cost in under 100 executions. These findings demonstrate that while LLM-based optimization can yield substantial energy savings, its actual profitability is highly context-dependent and must be carefully evaluated based on how frequently the optimized code is expected to run.

The remainder of this paper is structured as follows. Section II presents the related works. In Section III, we explain the methodology of our experiment. We report our results in Section IV and discuss them in Section V. Finally, we discuss our limitations in Section VI and conclude in Section VII.

II. Related Works

Research at the intersection of LLMs and code efficiency has developed along three main directions: (i) optimizing code with LLMs, (ii) studying the efficiency of LLM-generated code, and (iii) measuring the energy consumption of LLM inference itself.

Optimizing code with LLMs. A growing body of work develops LLM-based methods to optimize existing code, most often with the goal of improving runtime performance or resource efficiency. Many of these approaches rely on iterative refinement and prompt engineering, such as LLM4EFFI [6], PerfCodeGen [8], Self-Refine [9], RAPGen [18], Effi-Learner [7], and SBLLM [19]. Other approaches apply fine-tuning to create a model that is able to optimize code, such as with the PIE dataset [20], the SwiftCoder model [21], or Google's ECO tool [22], which aims at improving the performance of all of Google's production code. Complementing these, evaluation benchmarks have also emerged to assess the efficiency of LLM-based optimizers, including COFFEE [23], EvalPerf and the DPE method [17], EffiBench [24], ENAMEL [25], and Mercury [26], along with comparative studies by Coignion *et al.* [27] and Li *et al.* [28].

Efficiency of LLM-generated code. Parallel to research on optimization methods, other studies evaluate whether LLM-generated code is intrinsically more or less efficient in terms of runtime or energy. Vartziotis *et al.* [29], Tuttle *et al.* [30], and Cursaru *et al.* [31] analyzed the energy consumption of LLM-generated solutions across various problem datasets. Focusing on energy-aware optimization, Rani *et al.* [32] compared Matlab code optimized by an LLM to that of a senior developer, reporting mixed trade-offs between energy efficiency and memory usage. Similarly, Peng *et al.* [33] showed that LLMs could improve energy efficiency in 3 of their 6 tested programs, in some cases surpassing compiler optimizations alone.

Energy consumption of LLM inference. Finally, a complementary research line investigates the energy and resource costs of running LLMs themselves. Samsi *et al.* [13] were among the first to measure inference energy use across LLaMa model variants. Luccioni and Strubell [14] studied inference costs across multiple models and tasks, while Berthelot *et al.* [15] extended this to a full life-cycle analysis of Stable Diffusion. Specific to software development, Coignion *et al.* [16] measured the energy usage of GitHub Copilot in developer workflows. More recent work by Jegham *et al.* [34] broadened the scope to estimate energy, water, and carbon costs of both closed and open-source models, including GPT-4o and DeepSeek-R1.

Positioning of our work. Most of the above studies focus on either (i) developing LLM-based code optimizers, (ii) evaluating the runtime or energy efficiency of LLM-generated code, or (iii) measuring the energy and environmental costs of LLM inference. To the best of our knowledge, our study is the first to jointly measure both the efficiency gains of LLM-based code optimization and the energy costs of performing the optimization itself.

III. Methodology

In this section, we explain how we set up and led our experiment. Namely, we list the various optimization methods we chose and how we implemented them, as well as the LLMs chosen to be used alongside the optimization methods. We also present the metrics considered to analyze our results.

A. Dataset

We evaluated the optimization methods on the 118 tasks from the EvalPerf dataset [17], which is based on HumanEval+ and MBPP+. This dataset is well-suited for our evaluation purposes as it contains both performance-exercising programming tasks and tests, as well as reference solutions with varied

performances. EvalPerf's tasks were selected because they are computationally non-trivial, which enable a higher runtime variation in the possible solutions of the tasks. This higher variation, coupled with the performance exercising tests make it easier to differentiate good and bad optimizations.

B. Methods and their implementation

We selected 8 methods to optimize code, which can be classified in two distinct categories: i) single-code optimizers and ii) batching optimizers. Single-code optimizers take one sample as input and output one code sample, while batching optimizers may take multiple inputs and output multiple samples. The key consideration about batching optimizers is that they produce batches of code samples, in which only the sample with the best performance, in terms of number of CPU instructions, is selected as a result for analysis.

During the selection of optimization methods, our goal was to capture a diverse set of techniques representative of the current landscape of LLM-based code optimization. From the literature, we identified methods that (i) could be realistically implemented within our experimental setup and time constraints, and (ii) targeted improvements in runtime or energy efficiency, either through simple prompting strategies or more structured generation techniques. We excluded approaches requiring fine-tuning (*e.g.*, PIE [20]) or those with integration costs beyond our available resources (*e.g.*, EffiLearner and RapGen [7], [18]). Within these limits, our selection aims to balance practical feasibility with methodological diversity, covering a representative range of optimization paradigms.

The single-code optimization methods considered in this study are:

- **Simple prompting.** This method consists of simply asking the LLM to optimize the program.
- ***Chain-of-Thought* prompting (CoT).** This method [35] is similar to simple prompting, but asks the LLM to think step by step before generating its answer.
- ***Chain-of-Draft* prompting (CoD).** This method [36], while being inspired by CoT prompting, aims at reducing the number of tokens produced by the LLM by prompting it with the instruction "*Keep a minimum draft for each thinking step, with 5 words at most*".
- **Self-refinement with Natural Language Feedback.** This method and the associated prompts come from the works of Madaan *et al.* [9] and Waghjale *et al.* [37], respectively. For every code sample, the LLM is prompted to give feedback on the solution, then prompted a second time to optimize/fix the code sample.
- **Self-refinement with Execution Feedback.** This method and the associated prompts come from Waghjale *et al.* [37]. Instead of LLM-generated natural language feedback, the LLM is enhanced with execution feedback, which includes either runtime information or test execution results, depending on whether the sample passes the unit tests.

The batching optimization methods considered in this study are:

- **LLM4EFFI.** This method [6] revolves around the idea of first producing a correct and efficient algorithm, and then producing the corresponding code. In this method, iteration 0, which is normally used to generate the baseline, is tasked with generating efficient code, and the subsequent iterations, which are normally used to optimize the samples, are only focused on improving the correctness of non-correct samples. Using this method, 10 algorithms and their associated code samples are generated to produce a single result, which is the fastest of the code samples. As we use our own test cases, we did not implement the test case generation feature of LLM4EFFI. LLM4EFFI generates batches containing 10 samples.
- ***Evolution of Heuristics* (EoH).** This method from Liu *et al.* [38] uses an evolutionary algorithm to make LLMs produce better code. While this method was used on optimization problems, such as the traveling salesman problem, we thought it would be interesting to apply it to code optimization. Instead of evolving the code samples directly, this method uses the LLM to evolve ideas (heuristics) through various mutations. Each idea is also associated with a code sample generated with the idea. We set the population size parameter to 10 (meaning, after every generation of 50 heuristics, only the 10 best heuristics are selected), and the parents parameter (for exploratory mutations) to 3. EoH generates batches of 10 or 50 samples, depending on the amount of correct samples in the input. If at least one correct samples is found, then the batch size is 50, as 10 samples are sampled for each of the five mutation, otherwise, the batch size is 10, as 10 samples are selected and a fix is attempted on them.
- **Simple10 (custom method).** As LLM4EFFI and EoH both generate batches of optimizations and take the best one out of the batch, they are set apart from the other methods which generate single code optimizations ("batches" of one optimization). To study whether the differences between LLM4EFFI and EoH and the single-code optimizers are due to their higher batch size, or due to other aspects of the methods, we implemented Simple10, a variant of the simple method which generates batches of 10 optimizations using the prompt from the simple method, and considers the best sample of the batch as the result.

C. LLMs under study

During our study, we tested the optimization methods using 5 different LLMs. The models we used were Starcoder2-15B (instruct), Qwen2.5-Coder-{3,7,14}B (instruct) and DeepSeek-R1-Distill-Qwen-14B.

We selected models based on their open-source availability and on their diversity, in both architecture and size. Specifically: Qwen2.5-Coder and DeepSeek-R1 were selected due to strong performance in public code generation benchmarks (*e.g.*,, BigCodeBench). StarCoder2 was chosen for its open

training data and reproducibility focus. Multiple Qwen variants were included to assess the effect of model size on energy/performance trade-offs.

D. Experimental setup

Our experiment aims to optimize code samples using LLMs in multiple iterations while assessing the energy consumption of the optimization process as well as the energy consumption of the programs before and after optimization. Thus, we split our experiment into two distinct phases: a) the optimization phase and b) the program energy consumption evaluation phase. Separating the experiment into these two phases helped us manage our server budget, as evaluating the energy consumption of the programs is not required by the optimization process and does not need GPUs. Our experiment was conducted on the 118 tasks from the EvalPerf dataset with 8 optimization methods, using 5 models. For a given configuration (*i.e.*, method + model combination) and for the single-code optimization methods, 40 independent samples were generated by task. Simple10 generated 40 independent batches by task, EoH generated 3 independent batches by task and LLM4EFFI generated 5 independent batches by task.

Optimization phase. The optimization phase consists of one base iteration (iteration 0) and four optimization iterations (iterations 1–4). Each iteration is split into three steps: i) code generation ii) correctness evaluation and iii) performance evaluation. In the base iteration (iteration 0), during the code generation step and depending on the optimization method, we either generate a solution to a problem, or use the optimization method to generate a solution if the method allows it (LLM4EFFI and EoH are the only two methods concerned). The code samples that were generated in iteration 0 with just the problem's base prompt are then used as baseline for comparison with the optimized programs. In subsequent iterations, in the code generation step, we generate optimizations of samples from the previous iteration using the optimization methods' techniques.

In the correctness evaluation step and performance evaluation step, we evaluate each code sample that was produced during this iteration using the correctness and performance tests from EvalPerf. Only samples that passed all the correctness tests are evaluated on performance. During each step of the optimization process, we measured the energy consumed by the machine. While the CPU was always taken into account, the GPU was only recorded during the code generation phase, as it is the only one using the GPU. Additionally, at the start of each iteration, we performed a calibration for 30 seconds where we recorded the energy consumption of the machine at rest.

Program energy consumption evaluation phase. During this phase, we measured the marginal CPU energy consumption of every valid sample that was generated in the optimization phase. The samples were executed with the performance-exercising tests from EvalPerf. Before evaluating each configuration, a calibration phase of 30 seconds was realized, in order to get the idle consumption of the machine. This idle consumption as well as the energy measured during the evaluation were used to calculate the marginal energy consumption of the samples [39]. When measuring the energy consumption of the samples, our idle measures had a coefficient of variation of 0.014, which indicates a great stability of the measures across the runs.

To increase the accuracy of our results, we repeated each sample's execution until at least one second was elapsed. As we had to evaluate $1{,}147{,}198$ samples, this minimum duration seemed like a good trade-off between measure accuracy and feasibility of the experiment. When analyzing the data, to aggregate the energy consumption at the task level and then at the configuration level, we either averaged the energy consumption of all generated samples for single-code optimizers or averaged the energy consumption of the best samples from each batch for batching optimizers. Many parts of our experiment were built upon the EvalPerf and DPE experiment framework from Liu *et al.* [17]. Namely, we adapted their generation and evaluation framework to fit our needs.

E. Hardware and software setup

The optimization-related phases of our experiment ran on devices with 1 AMD EPYC 7513 (Zen 3), as well as 4 Nvidia A100-SMX4-40GB. The energy profiling of the individual samples was executed on devices with 2 AMD EPYC 7301 (Zen). We used VLLM [40] to run our LLMs. It was executed using Docker, with tensor parallelism set to 4 (the number of GPUs). The generations made by the LLMs were made using temperature of 0.2, which is the temperature that was chosen by most of the previous studies on LLM-based optimization. To maximize GPU usage and get more realistic energy data, generation requests were sent concurrently, and then batched internally by VLLM.

We measured energy consumption using software-based tools. For the CPU, we used `perf`, which uses the RAPL interface, a standard and widely used method to estimate power usage. For the GPU, we used `nvidia-smi`, which reports power draw with $\sim 5\%$ error margin [41]. This approach is consistent with recent studies on AI energy profiling [13], [29], [42]. Hardware power analyzers would have been ideal, but were not available in our experimental environment.

In total, this experiment consumed 651.24 KWh which amounts to 36.47 kg of CO2eq using France's energy mix's carbon intensity, which is where the servers hosting the experiment were located. The replication package of our experiment, with all of the data from our experiment as well as a companion notebook with additional figures and tables is available at https://doi.org/10.5281/zenodo.15526179. The code and data in our replication package are designed to be extensible, allowing practitioners to integrate and evaluate their own optimization methods or models. Moreover, the framework can also be used to analyze code optimizations made externally.

F. Evaluation metrics

To evaluate the quality of the optimization methods and models, we used multiple metrics to quantify the performance of the samples generated, their correctness as well as their energy cost. Each metric was chosen to capture a distinct aspect of the optimization process

DPS_{norm}. The *Differential Performance Score* (DPS) is a metric developed by Liu *et al.* [17]. DPS uses clustering to more accurately assess the runtime performance of a given program on a lot of different tasks, compared to traditional metrics such as speedup. This metric is based on CPU performance counters, which is more reliable than CPU time. Given a set of reference solutions to a problem and a sample solution, the DPS is the cumulative ratio of the reference that is immediately slower than the sample solution. For example, a DPS of 80% implies that overall the LLM generates code whose runtime efficiency can match or improve 80% of all LLM-generated solutions. We use a variant of DPS, named DPS_{norm}, a normalized version of DPS that ignores the volume of solutions at each level of runtime efficiency. This metric is more suitable for our study as it is better at highlighting differences in runtime efficiency levels among a large variety of tasks. We also define the $\text{DPS}_{\text{norm}}\Delta$ as the difference in the DPS_{norm} of the baseline and the DPS_{norm} obtained in the final optimization iteration.

Pass@k. We evaluate the correctness of the code produced by a given method using the Pass@1 developed by Chen *et al.* [43]. This metric represents the proportion of generated samples that are functionally correct, that is they pass all the unit tests of the problem. For optimization methods that produce batches of samples (e.g., EoH, LLM4EFFI, Simple10), we compute Pass@1 over all individual samples in the batch, regardless of whether they were ultimately selected as the final result. However, since this does not necessarily reflect the correctness of the selected output, we additionally introduce a metric called *Pass@result*. This metric evaluates the correctness of the chosen result per optimization attempt: for single-code optimizers, Pass@result is equivalent to Pass@1, as only one sample is generated per attempt; for batch-based optimizers, Pass@result measures the proportion of batches whose selected result passes all tests. In this sense, Pass@result acts as a dynamic variant of Pass@k, where k corresponds to the batch size used by the method.

Efficient@k. We adapt the Efficient@k metric from Peng *et al.* [23]. While pass@k is the probability of generating a correct solution, Efficient@k is the probability that at least one of the top-k correct code samples for a problem is more efficient than our baseline. For example, an Efficient@1 of 0.7 means that 70% of the generations that are correct are also better than the baseline. Our version of the metric is slightly different from the original one, because incorrect samples are excluded from its calculation whereas, in the original, incorrect samples would count as "not more efficient" samples and lower the score. We chose to alter the metric in order to really focus on the probability of a successful optimization to actually be faster than the baseline.

Energy reduction. While the Efficient@k gives us a good idea of the probability of a successful optimization, the Energy reduction is designed to give insights on how much more efficient the optimized code is compared to the baseline. The Energy reduction is simply defined as the ratio of the energy consumed by the optimized sample over the energy consumed by the baseline. For instance, an Energy reduction score of 0.60 means that the optimized sample's energy consumption is 40% lower than that of the baseline.

Break-Even Point (BEP). Let e_{orig} be the energy consumed per execution of the original (non-optimized) program, e_{opt} be the energy consumed per execution of the optimized program, and E_{optim} be the one-time energy cost of the optimization process. The BEP is defined as:

$$\text{BEP} = \frac{E_{\text{optim}}}{e_{\text{orig}} - e_{\text{opt}}}$$

This metric captures the number of executions required for the cumulative energy savings of the optimized program to outweigh the energy cost of performing the optimization. In the case where the "optimized" version of the program consumes more energy than its original version, then the BEP is considered as undefined and is not taken into account in further calculations.

To obtain more representative averages for BEP and Energy reduction, we removed the lowest and highest 5% of values across tasks before computing the mean. This approach is standard in robust statistical analysis [44] and was chosen due to the heavy-tailed nature of these metrics, which are especially sensitive to rare but extreme outliers. We confirmed that this trimming did not alter the overall trends observed across configurations.

IV. Results

In this section, we share our results addressing the three research questions targeting (i) the optimization capacity of the tested methods (subsection IV-A), (ii) the energy consumption of the optimization process (subsection IV-B), and (iii) the cost-effectiveness of code optimization (subsection IV-C).

A. ***RQ1:*** *What is the best LLM-based method to optimize code?*

The optimization methods we evaluated exhibited varying performance in terms of both their capacity to optimize code and their ability to maintain sample correctness.

Optimization capacity. To visualize the optimization capacity of each method, Figure 1 reports on the performance improvements in DPS_{norm} across all method-model combinations.

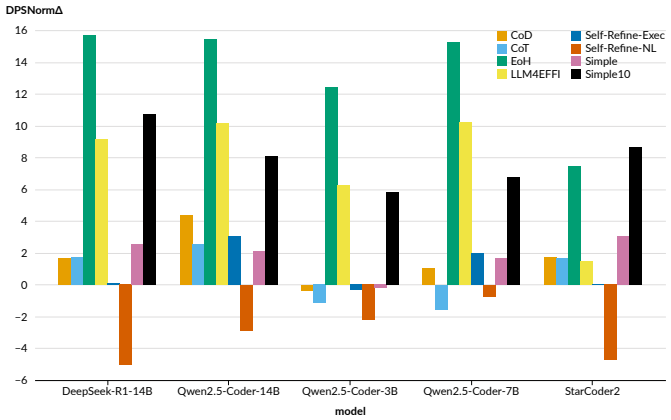

Fig. 1. $\mathrm{DPS}_{\mathrm{norm}}\Delta$ of each method and model combination, at the last iteration. Negative results show decreases in $\mathrm{DPS}_{\mathrm{norm}}$ compared to the baseline. A higher $\mathrm{DPS}_{\mathrm{norm}}\Delta$ is better.

The batching-based methods (EoH, LLM4EFFI and Simple10) exhibit the best optimization capacity and show the highest average gains over the baseline—*i.e.*, 13.24, 7.46 and 7.99 $\mathrm{DPS}_{\mathrm{norm}}$ points by the final iteration, respectively. In contrast, other methods show more modest optimization capacities, with overall $\mathrm{DPS}_{\mathrm{norm}}$ gains ranging from -3.12 (Self-Refine-NL) to 1.83 (Simple). In particular, some methods tend to decrease the overall performance during the optimization process. Namely, Self-Refine-NL significantly reduces the average $\mathrm{DPS}_{\mathrm{norm}}$ of the programs, particularly when combined with the DeepSeek-R1 model, resulting in a drop of 5.03 $\mathrm{DPS}_{\mathrm{norm}}$ points. We hypothesize that this reduction stems from the model's extensive reasoning steps, which may lead to inefficient optimizations.

As shown in Figure 1 and Table II, Simple10 often matches or outperforms EoH and LLM4EFFI in terms of optimization capacity. Although one might assume that LLM4EFFI and EoH outperform traditional optimizers due to their "ideas before code" approach, this assumption is challenged by the effectiveness of a simple batch strategy, like Simple10, which achieves comparable or better results on several models. While LLM4EFFI has a better $\mathrm{DPS}_{\mathrm{norm}}$ at iteration 0, it is eventually matched by Simple10 at iteration 2 (Figure 2). However, as reported in Table II, Simple10 achieves a notably lower Pass@result score (60.41) compared to EoH (91.41) and LLM4EFFI (90.07), indicating a higher likelihood of producing incorrect final outputs. This suggests that, while naive batching can enhance optimization capacity, it may do so at the expense of correctness. The stronger correctness of EoH and LLM4EFFI highlights that their success is likely not due to batching alone, but also to additional factors such as explicit correctness refinement (in LLM4EFFI) or evolutionary selection mechanisms (in EoH).

Figure 2 further illustrates how both the optimization capacity and correctness evolve over iterations. For most methods, $\mathrm{DPS}_{\mathrm{norm}}$ peaks at iteration 2, with further rounds either stabilising or decreasing performance. One exception is EoH, which continues improving until plateauing at iteration 4.

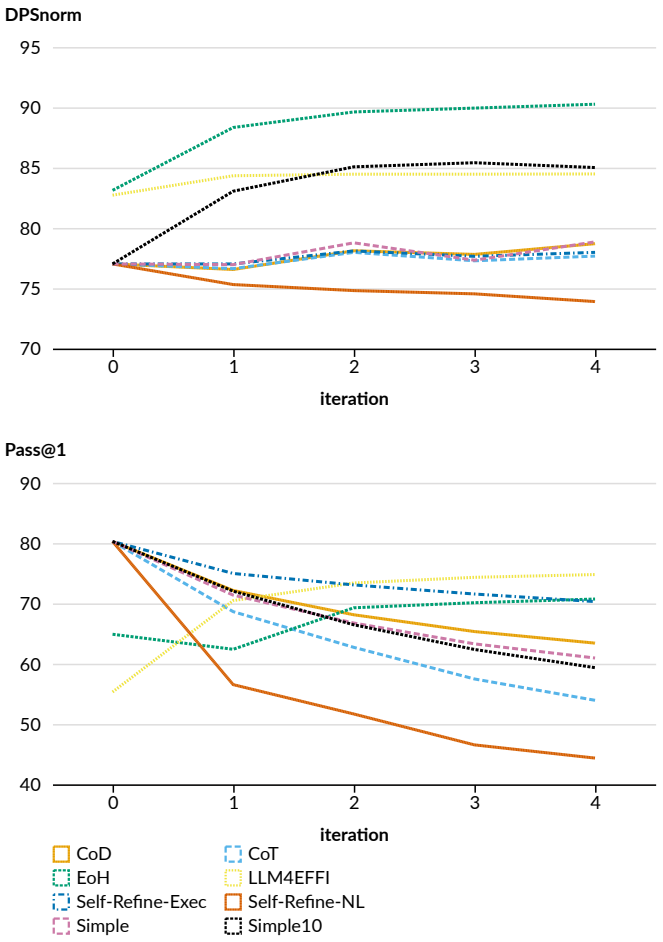

Fig. 2. Pass@1 and $\mathrm{DPS}_{\mathrm{norm}}$ across iterations depending on the method. The point at iteration 0 is the same for all methods except LLM4EFFI and EoH, as it is the baseline. A higher Pass@1 and $\mathrm{DPS}_{\mathrm{norm}}$ is better.

LLM4EFFI, in contrast, peaks early. Indeed, optimization occurs at iteration 0, any subsequent $\mathrm{DPS}_{\mathrm{norm}}$ gains only resulting from converting invalid code into valid code.

Correctness of the optimizations. As shown in Figure 2, all methods but EoH and LLM4EFFI decrease in correctness over iterations. This trend is expected for methods focusing primarily on optimization, such as Simple, CoT and CoD, but is more surprising for methods like Self-Refine-Exec and Self-Refine-NL, which are designed to preserve correctness over iterations. Self-Refine-Exec, nonetheless, is the best single-code optimizer thanks to its use of test failures in prompts, enabling effective error correction. These results are consistent with those of Waghjale *et al.* on similar optimizers [37].

In contrast, LLM4EFFI and EoH improve the average correctness of the code over time, albeit relying on different mechanisms. After the initial generation, LLM4EFFI only focuses on improving code correctness. This avoids re-optimizing valid code, thereby reducing the risk of invalidation. EoH, leveraging evolutionary algorithms to implement its selection phase, retains only valid samples in each generation and attempts to repair non-passing solutions if no valid ones

optimizer	Cumulative energy cost of one optimization (Wh)				
	Iter-0	Iter-1	Iter-2	Iter-3	Iter-4
CoD	0.79	0.86	0.94	1.02	1.09
CoT	0.79	0.94	1.12	1.34	1.53
EoH	0.74	6.15	10.18	14.11	18.19
LLM4EFFI	3.75	5.06	5.73	6.32	6.85
Self-Refine-Exec	0.79	0.96	1.15	1.37	1.62
Self-Refine-NL	0.79	1.04	1.45	1.92	2.39
Simple	0.79	0.94	1.06	1.18	1.30
Simple10	0.79	2.47	3.81	5.10	6.31

TABLE I
TOTAL ENERGY COST TO GET ONE OPTIMIZED RESULT AT THE LAST ITERATION DEPENDING ON THE OPTIMIZATION METHOD OVER ALL MODELS. FOR ALL METHODS BUT EOH AND LLM4EFFI, ITER-0 IS THE BASELINE GENERATION PHASE.

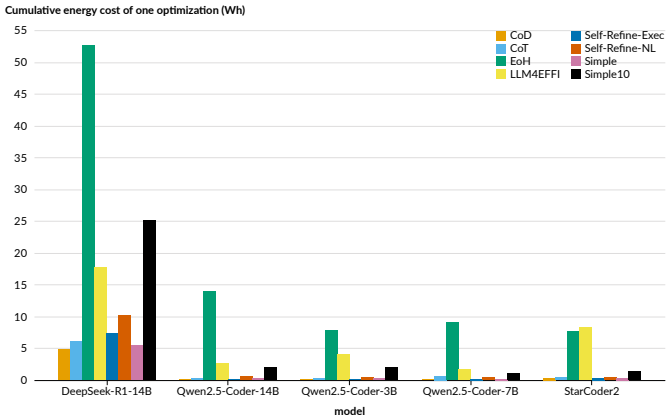

Fig. 3. Total energy consumption required to get one optimized result at the last iteration depending on the model and optimization method. Lower is better.

are found in a batch, as it is not able to produce optimizations without valid code samples.

> **RQ1:** Optimization capacity is strongly influenced by both the method and the LLM used. Most methods failed to reliably outperform the baseline, but batching-based optimization methods demonstrated superior performance. Interestingly, Simple10 often matched or exceeded the optimization capacity of more complex methods, highlighting that performance gains can result from effective sampling rather than complexity alone. These findings emphasize that effective optimization does not solely depend on complex reasoning or additional refinement steps, but also on strategic sample selection and iterative control. Selecting an appropriate optimization strategy thus requires balancing performance gains with the risk of introducing errors, especially in iterative workflows.

*B. **RQ2:** What is the energy necessary to produce one optimized result?*

To answer this question, we first present the distribution of energy consumption across the phases of the optimization process, and we then examine the energy consumption required to produce one optimized result in four iterations of optimization.

Energy usage by phase. The generation phase accounts for between 90.3% (Qwen2.5-Coder-7B) and 99.2% (DeepSeek-R1-14B) of the total energy consumed during the optimization process. On average, about two-thirds of the remaining energy is consumed by the correctness evaluation (from 0.48% to 6.41%), and one-third by the performance evaluation (from 0.34% to 3.29%). This result is not surprising for the generation phase, as it is the only phase involving an LLM and the GPUs. Interestingly, the energy usage is more influenced by the duration of this phase than by GPU power draw. Indeed, during generation, GPU power remained stable, but latencies varied significantly between models. For instance, generating baseline samples took 195 seconds with Qwen2.5-Coder-3B, while DeepSeek-R1-14B required 9,980 seconds for the same task.

Correctness evaluation consumes twice as much energy as performance evaluation primarily because many generated samples hit the timeout limit during this phase, either due to high time complexity or infinite loops. Since performance is only evaluated on samples that pass the correctness phase, the most computationally expensive ones are already excluded.

Energy usage of the models. Results in Table III show that larger models tend to consume more energy than smaller ones. For instance, Qwen2.5-Coder-14B and Starcoder2 (14B) require respectively 2.48 and 2.37 Wh per optimization, while the 3B and 7B variants of Qwen2.5-Coder only need 1.87 and 1.69 Wh. Interestingly, DeepSeek-R1-14B is the costliest model in terms of energy, with an average optimization cost of 16.15 Wh. While the architecture of the model itself can impact its consumption, the energy overhead of the optimization process in the DeepSeek-R1 model is largely due to its reasoning phase.

Energy usage of the optimization methods. Interestingly, as reported in Figure 3, the ranking of optimization methods by the energy required to produce one optimization varies across models. For most models, EoH is the most energy-intensive, whereas for StarCoder2, LLM4EFFI consumes the most. This discrepancy arises because LLM4EFFI may repeat its task formalization step until a satisfactory prompt is found, leading to additional inference rounds depending on the model's response quality. We observed that the most energy efficient configuration was Qwen2.5-Coder-14B and CoD with an energy per optimization at iteration 4 of 0.10 Wh. On the other hand, the most energy-intensive configuration was DeepSeek-R1-14B and EoH with 52.6 Wh per optimization at iteration 4, which is almost 500 times more expensive. This underscores how much the choice of the configuration can impact the energy consumption of the optimization. In Table I,

Method	Iteration	Efficient@1	Energy reduction	Energy / result (Wh)	BEP	Pass@1	Pass@result	DPS_{norm}	$DPS_{norm}\Delta$
CoD	2	0.819	0.803	0.94	63,086	68.1	68.1	78.1	1.09
CoT	2	0.712	0.912	1.12	95,870	62.7	62.7	78.0	0.96
EoH	2	0.869	0.664	10.18	65,100	69.3	90.2	89.6	12.6
LLM4EFFI	2	0.822	0.759	5.73	84,949	73.4	89.6	84.5	7.44
Self-Refine-Exec	2	0.758	0.822	1.15	109,913	73.1	73.1	78.1	1.08
Self-Refine-NL	2	0.725	0.921	1.45	173,683	51.7	51.7	74.8	-2.21
Simple	2	0.729	0.894	1.06	91,827	66.8	66.8	78.8	1.75
Simple10	2	0.717	0.968	3.81	155,938	66.5	62.0	85.1	8.05
CoD	4	0.795	0.821	1.09	63,586	63.4	63.4	78.7	1.68
CoT	4	0.8	0.808	1.53	102,824	53.9	53.9	77.7	0.65
EoH	4	0.759	0.786	18.19	138,997	70.7	91.4	90.3	13.24
LLM4EFFI	4	0.803	0.789	6.85	104,299	74.8	90.1	84.5	7.46
Self-Refine-Exec	4	0.737	0.877	1.62	153,831	70.3	70.3	78.0	0.96
Self-Refine-NL	4	0.783	0.849	2.39	381,912	44.4	44.4	73.9	-3.12
Simple	4	0.734	0.914	1.30	116,947	61.0	61.0	78.8	1.83
Simple10	4	0.792	0.874	6.31	310,219	59.4	60.4	85.0	7.99

TABLE II
ENERGY PROFITABILITY, CORRECTNESS AND PERFORMANCE OF THE METHODS AT ITERATION 2 AND 4, OVER ALL MODELS. THE OPTIMIZATION COST PER RESULT IS THE CUMULATIVE COST OVER ALL ITERATIONS.

we present the evolution over time of the energy required to produce one optimization depending on the optimization method.

Thanks to the fewer generations they make, single-code methods require much less energy to optimize code than batching methods. As expected, Chain-of-Draft prompts the model to generate fewer tokens, thus reducing the energy use (1.09 Wh). This leads to a threefold reduction in energy per optimization compared the traditional Chain-of-Thought (1.53 Wh). Among single-code methods, Self-Refine-NL is the most energy-intensive (2.39 Wh), mainly because it requires an additional feedback generation round from the LLM which increases the total number of tokens generated.

While LLM4EFFI and Simple10 have similar batch sizes, LLM4EFFI's energy per result increases more slowly between iterations (+0.53 Wh vs +1.21 Wh of additional energy between iteration 3 and 4) as shown in Table I. This is because LLM4EFFI stops optimizing samples once they are correct, whereas Simple10 tries to optimize a sample regardless of correctness or performance.

> **RQ2:** The energy required to produce a single optimized result after four optimization rounds varies significantly depending on the model and optimization method, ranging from 0.10 Wh to 52.6 Wh. Larger models and those that rely on complex reasoning tend to consume more energy. Additionally, optimization methods that involve more inference steps or require longer outputs from the model further increase energy consumption. These findings underscore the need to balance optimization effectiveness with energy efficiency, particularly when selecting models and strategies for large-scale or repeated code optimization tasks.

C. *RQ3: In terms of energy, at what point is it profitable to optimize a given code?*

Having assessed both the optimization capacity and energy consumption of all evaluated models and methods, we now discuss their energy profitability, synthesized in Table II and Table III. Surprisingly, all models achieved relatively consistent Efficient@1 scores, ranging from 0.734 (Simple) to 0.803 (LLM4EFFI). This indicates that all models are generally able to reduce the energy consumption of at least 73.4% of the samples below the consumption of the baseline, provided their solution is correct. In terms of absolute energy savings, EoH and LLM4EFFI were the most effective, with reductions of approximately 21%, in contrast with other methods such as Simple which only reduced energy consumption by 8.6%. Notably, although Simple10 was the second most effective method in terms of optimization capacity, it ranked only sixth in terms of energy savings.

The *break-even point* (BEP)—*i.e.*, the number of executions required for an optimization to offset its energy cost—varied significantly across methods, ranging from 63,586 (CoD) to 381,912 (Self-Refine-NL) executions. Even the most energy-efficient method in terms of BEP still requires, on average, tens of thousands of executions to justify optimization from an energy perspective. Among models, DeepSeek-R1-14B demonstrated the highest Efficient@1 and Energy reduction scores. However, it is also the model for which it takes the most executions to reach the break-even point, suggesting that its substantial energy savings only become profitable in the long run. Interestingly, while Qwen2.5-Coder-14B showed the best optimization capacity in previous analyses, its smaller variant, Qwen2.5-Coder-7B, outperformed it in terms of energy efficiency. The 7B model also had a lower BEP, making it the most cost-effective choice within its family.

The configuration combining DeepSeek-R1-14B with EoH led to the highest reduction in energy consumption, with an Efficient@1 of 0.924, an Energy reduction of 0.493, and a BEP of 220,324. While this configuration has a high BEP,

Model	Efficient@1	Energy reduction	Optimization cost / result (Wh)	BEP	Pass@1	Pass@result	DPS_{norm}	$DPS_{norm}\Delta$
DeepSeek-R1-14B	0.907	0.625	16.15	459,057	63.5	61.4	82.8	4.57
Qwen2.5-Coder-14B	0.716	0.877	2.48	126,042	71.8	77.4	82.4	5.35
Qwen2.5-Coder-3B	0.691	0.941	1.87	110,706	52.0	61.1	80.7	2.53
Qwen2.5-Coder-7B	0.802	0.831	1.69	66,889	66.4	73.7	82.3	4.32
StarCoder2	0.762	0.924	2.37	95,192	57.4	60.6	76.2	2.4

TABLE III
ENERGY PROFITABILITY, CORRECTNESS AND PERFORMANCE OF THE MODELS TESTED AT THE LAST ITERATION OVER ALL METHODS. THE OPTIMIZATION COST PER RESULT IS THE CUMULATIVE COST OVER ALL ITERATIONS.

it can be advantageous for highly energy-intensive code. For instance, one baseline solution consumed an average energy consumption of 318 J, which was reduced to 2.6 J thanks to this configuration, resulting in a BEP of only 74 executions. On the other hand, the configuration using Qwen2.5-Coder-7B with CoD resulted in one of the lowest BEP scores, with an Efficient@1 of 0.805, an Energy reduction of 0.869, and a BEP of 24,016 executions.

Using one "heavyweight" or "lightweight" configuration ultimately depends on the anticipated usage patterns of the optimized code. Heavyweight setups, while energy-intensive during optimization, are suitable for optimizing code that is frequently executed or particularly energy-demanding. Lightweight configurations, on the other hand, offer lower energy costs per optimization and are preferable when optimizing a large number of infrequently executed code samples. In scenarios where the code is unlikely to be executed enough to reach the BEP, not optimizing at all may be the most energy-efficient choice.

Table II shows that running optimizations for two iterations instead of four can be more profitable in terms of energy. Indeed, for most methods, optimizing after the second iteration yields diminishing returns. For instance, EoH Efficient@1 and energy reduction actually worsen at iteration 4, and while Simple10 does get better in terms of Efficient@1 and energy reduction at iteration 4, it almost doubles its BEP. This shows that in order not to waste a great amount of energy and computing power, careful control of the number of optimization rounds is required.

Interestingly, the optimization capacity of a configuration, as measured by $DPS_{norm}\Delta$, and its energy reduction have a Pearson correlation coefficient of only -0.29, indicating that they are weakly correlated. Whereas at the sample level, there is a strong correlation (0.81) between the number of CPU instructions and the energy consumed by the sample. These results outline the need for the development of more energy-aware optimization methods that specifically include energy-related metrics.

RQ3: While most optimization methods and models consistently reduce energy consumption, their energy profitability strongly depends on usage context. High-performance configurations, such as DeepSeek-R1-14B with EoH, offer substantial energy reductions but require a high number of executions to break even, making them suitable for energy-intensive or frequently executed code. In contrast, lightweight setups, like Qwen2.5-Coder-7B with CoD, yield competitive energy gains at a much lower cost, offering a better trade-off for less intensive scenarios. Additionally, running multiple rounds of optimization leads to diminishing returns in terms of energy consumption. These results highlight the importance of selecting optimization strategies not only based on raw performance but also on their contextual profitability, emphasizing that energy-aware optimization should be tailored to the usage patterns of the target code. A careful planning of the number of optimization rounds is also required in order not to waste computing resources.

V. DISCUSSION

A. Practical implications

Our findings offer concrete guidance for software engineers seeking to integrate LLM-based optimizers into real-world development workflows. First, the decision to invoke optimization should depend on the expected execution frequency and energy profile of the target code. Code that powers latency-sensitive hot paths or runs in large batch jobs thousands of times per day can amortize even "heavyweight" optimization costs. Conversely, for one-off scripts or lightly used utilities, using a "lightweight" setup or even skipping optimization may be the more energy-efficient choice.

Second, model and method selection should follow simple heuristics: choose smaller, faster models with token-efficient prompting when the anticipated run count is low, and reserve larger, reasoning-heavy configurations for code that will be executed at scale. Additionally, the number of optimization rounds should be carefully considered, depending on the method used.

Finally, integrating energy-aware optimization into existing CI/CD pipelines (like Google did with their ECO tool [22]) requires reliable instrumentation. Teams must measure both the energy consumed by the optimizer (*e.g.*, via GPU power meters or software hooks) and the downstream savings achieved

on representative workloads. Integrating energy regression detection directly into the CI/CD pipeline can help developers focus their use of LLM-based optimizers [45]. Only with this telemetry can teams enforce energy budgets, detect regressions, and adapt optimization policies according to changing usage patterns.

B. Decoupling performance and energy

A core insight from our study is that improvements in performance metrics do not automatically translate into energy savings. We observe a weak negative Pearson correlation (-0.29) between normalized performance gains ($\mathsf{DPS}_{\mathsf{norm}}$) and measured energy reduction at the configuration level. Consequently, benchmarks that report only runtime metrics have the risk of overstating environmental benefits. Our results underscore the need for benchmarks that report execution time, energy consumption, and optimization overhead. Such multi-metric evaluations will help practitioners and researchers avoid "greenwashing" speedups and ensure that reported performance gains correspond to genuine reductions in energy footprint. Further research is needed to evaluate the correlation between energy reduction and other performance metrics, such as Beyond [26], eff@k [25] or even CPU instructions-based speedups [23].

C. The pitfalls of chasing efficiency alone

While our study demonstrates that LLM-based optimization methods can yield significant runtime and energy improvements, pursuing efficiency (runtime or energy) in isolation risks unintended environmental and societal consequences. First, efficiency gains often trigger rebound effects, whereby lower unit costs or faster runtimes induce a greater overall usage. For example, a development team that reduces optimization latency by 50% may be tempted to apply the optimizer not only in production pipelines but also in exploratory and CI workloads, effectively increasing total energy consumption rather than reducing it [46]. In extreme cases, rapid, near-free optimizations might even encourage "optimizing for optimization's sake," multiplying inference calls and model training with negligible per-run cost but huge aggregate footprint.

Second, energy consumption is only one dimension of AI's environmental impact. LLM inference depends on extensive GPU clusters, whose manufacture and operation deplete abiotic resources, consume vast quantities of water for cooling, and generate e-waste when hardware is replaced [15], [47], [48]. These embodied and indirect impacts are not captured by our energy-only metrics but can outweigh operational savings over the hardware life-cycle. For instance, a single optimizer rollout on a state-of-the-art model might save 100 J of energy per invocation, but the GPU fabrication and data-center cooling required to support that model can entail thousands of liters of water and tens of kilograms of rare-earth mining per server.

Finally, by treating efficiency as entirely beneficial, we risk normalizing ever-greater dependence on opaque, proprietary LLM services. Organizations pursuing "green code" may inadvertently reinforce centralized cloud infrastructures, further entrenching water-intensive data-center construction and prolonging hardware refresh cycles. A more holistic approach, one that couples efficiency targets with usage caps, transparency in resource accounting, and incentives for lightweight or on-device solutions, is therefore essential to truly reducing environmental impact rather than simply shifting the burden [46].

VI. Limitations & Threats to Validity

While our study provides novel insights into the energy and performance trade-offs of LLM-based code optimization, several limitations constrain the generalizability of our findings. First, we rely exclusively on the EvalPerf benchmark, which focuses on CPU-bound algorithmic tasks. This dataset consists of small programs, often containing one or two functions that solve an algorithmic problem. As such, it does not capture I/O-bound programs, large data-processing pipelines, or real-world microservices, all of which exhibit distinct performance and energy profiles. As a result, our break-even estimates and energy savings percentages may not generalize to domains where memory bandwidth, network latency, or storage I/O dominate. Consequently, there is an urgent need for new LLM-evaluation datasets that encompasses broader software engineering tasks. However there is a lower representativeness for large-scale industrial code, the benchmark includes diverse algorithmic and application-level challenges derived from MBPP and HumanEval, both widely accepted in LLM literature. These tasks are aligned with early-phase development, algorithm design, and performance-critical routines. Hence, we believe the findings are still meaningful and generalizable to an important subset of real-world coding scenarios.

Second, our measurements were conducted on NVIDIA A100 GPUs in a controlled environment. Consumer-grade GPUs, CPU inference, and diverse datacenter cooling and power infrastructures can exhibit substantially different power draw curves and thermal behaviors. Therefore, the absolute energy figures we report (*e.g.*, joules per optimization) should be interpreted as indicative rather than definitive, and practitioners should recalibrate our findings to their target hardware.

Third, to increase the precision of the results, we evaluate the code samples with a minimum measurement duration of 1 second. While we noticed a strong correlation between the energy usage and the number of CPU instructions of the samples–indicating a stable enough measuring setup, this measurement duration might still be too short for an accurate comparison of the programs' energy measurements, which, in turn, might incur biases and errors in our results.

Finally, we evaluated only five LLMs that represent current state-of-the-art code-specialized and general-purpose architectures. Given the rapid pace of model development, quantized, distilled, or next-generation transformers may offer different energy–performance trade-offs than those we observed. Thus, our conclusions about model ranking and break-even behavior may shift as new architectures emerge.

While the exact results may shift with newer models, we believe that our core findings remain valid: (1) optimization effectiveness varies greatly with the choice of LLM and method, and (2) the energy cost of optimization is non-negligible and must be weighed against its potential energy gains. Extending our framework to include a broader array of models will be necessary to confirm the robustness of these trends.

Concerning the BEP. The *Break-Even Point* (BEP) is designed to assess the cost-effectiveness of LLM-based code optimization. While it reflects energy efficiency in a practical sense, the BEP is influenced not only by the model's energy use, but also by the quality of the optimization, and the relative efficiency gains achieved.

For capturing the inherent efficiency of an LLM (at inference), complementary metrics such as energy per token or energy per response may offer a better view of the model's energy characteristics.

The training costs of the LLM itself are intentionally not included in our paper, as pretraining is a shared cost across potentially millions of users and use cases, and it is difficult, if not impossible, to reliably estimate how many times a given model will be used over its lifetime.

VII. Conclusion

In this work, we systematically evaluated a broad spectrum of LLM-based optimization methods and models, quantifying their performance improvements, correctness preservation, and energy implications. Our results demonstrate that batching strategies, such as EoH and LLM4EFFI, deliver the strongest performance improvement while maintaining correctness over multiple iterations at the cost of a higher energy consumption. Energy profiling revealed that generating the optimization dominates the optimization pipeline, and that larger models incur disproportionately high energy costs. Most importantly, we showed that the energy "break-even" point for these methods can range from under 100 to over 10^5 executions, underscoring the necessity of usage-aware, energy-centric decision making when adopting LLM-driven optimizers.

Looking forward, our study highlights both the potential and the pitfalls of integrating LLMs into energy-sensitive software workflows. While "heavyweight" configurations can yield dramatic energy reductions in hot-path code, "lightweight" setups or even selective forgoing of optimization may be more prudent for less frequently run tasks. Moreover, the broader environmental footprint of LLM inference, encompassing resource extraction, water use, and data-center operations, demands a nuanced perspective beyond operational energy metrics alone. We hope this work serves as a foundation for future research on adaptive, energy-aware optimization frameworks and green AI methods that jointly balance performance and sustainability.

Acknowledgment

This work received support from the French government through the *Agence Nationale de la Recherche* (ANR) under the France 2030 program, including partial funding from the CARECloud (ANR-23-PECL-0003), DISTILLER (ANR-21-CE25-0022), and KOALA (ANR-19-CE25-0003-01) projects. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.[1]

References

[1] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu *et al.*, "Qwen2.5-Coder Technical Report," arXiv, Nov. 2024.

[2] DeepSeek-AI *et al.*, "DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence," arXiv, Jun. 2024.

[3] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828

[4] C. Reux, M. Acher, D. E. Khelladi, O. Barais, and C. Quinton, "LLM Code Customization with Visual Results: A Benchmark on TikZ," in *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).*, Istanbul, Turkey, Jun. 2025. [Online]. Available: https://hal.science/hal-05049250

[5] S. S. Dvivedi, V. Vijay, S. L. R. Pujari, S. Lodh, and D. Kumar, "A Comparative Analysis of Large Language Models for Code Documentation Generation," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware 2024, Jul. 2024, pp. 65–73.

[6] T. Ye, W. Huang, X. Zhang, T. Ma, P. Liu, J. Yin, and W. Wang, "LLM4EFFI: Leveraging Large Language Models to Enhance Code Efficiency and Correctness," arXiv, Feb. 2025.

[7] D. Huang, J. Dai, H. Weng, P. Wu, Y. Qing, H. Cui, Z. Guo, and J. M. Zhang, "EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization," arXiv, May 2025.

[8] Y. Peng, A. D. Gotmare, M. Lyu, C. Xiong, S. Savarese, and D. Sahoo, "PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback," arXiv, Nov. 2024.

[9] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Welleck, B. P. Majumder, S. Gupta, A. Yazdanbakhsh, and P. Clark, "Self-Refine: Iterative Refinement with Self-Feedback," arXiv, Mar. 2023.

[10] World Bank Group, *Measuring the Emissions and Energy Footprint of the ICT Sector: Implications for Climate Action*, ser. Other Environmental Study. Washington, D.C: The World Bank, 2024.

[11] ADEME, "Numérique & environnement : Entre opportunités et nécessaire sobriété," Jan. 2025.

[12] C. Morand, A.-L. Ligozat, and A. Névéol, "How Green Can AI Be? A Study of Trends in Machine Learning Environmental Impacts," arXiv, Dec. 2024.

[13] S. Samsi, D. Zhao, J. McDonald, B. Li, A. Michaleas, M. Jones, W. Bergeron, J. Kepner, D. Tiwari, and V. Gadepally, "From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2023, pp. 1–9.

[14] S. Luccioni, Y. Jernite, and E. Strubell, "Power Hungry Processing: Watts Driving the Cost of AI Deployment?" in *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 2024, pp. 85–99.

[15] A. Berthelot, E. Caron, M. Jay, and L. Lefèvre, "Estimating the environmental impact of Generative-AI services using an LCA-based methodology," *Procedia CIRP*, vol. 122, pp. 707–712, 2024.

[16] T. Coignion, C. Quinton, and R. Rouvoy, "Green My LLM: Studying the key factors affecting the energy consumption of code assistants," arXiv, Nov. 2024.

[17] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating Language Models for Efficient Code Generation," arXiv, Aug. 2024.

[18] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "RAPGen An Approach for Fixing Code Inefficiencies in Zero-Shot," arXiv, Jul. 2024.

[19] S. Gao, C. Gao, W. Gu, and M. Lyu, "Search-Based LLMs for Code Optimization," arXiv, Aug. 2024.

[1] See https://www.grid5000.fr

[20] A. G. Shypula, A. Madaan, Y. Zeng, U. Alon, J. R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning Performance-Improving Code Edits," in *The Twelfth International Conference on Learning Representations*, Oct. 2023.

[21] D. Huang, G. Zeng, J. Dai, M. Luo, H. Weng, Y. Qing, H. Cui, Z. Guo, and J. M. Zhang, "SwiftCoder: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning," arXiv, Mar. 2025.

[22] H. Lin, M. Maas, M. Roquemore *et al.*, "ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers," arXiv, Mar. 2025.

[23] Y. Peng, J. Wan, Y. Li, and X. Ren, "COFFE: A Code Efficiency Benchmark for Code Generation," arXiv, Feb. 2025.

[24] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "EffiBench: Benchmarking the Efficiency of Automatically Generated Code," arXiv, Jul. 2024.

[25] R. Qiu, W. W. Zeng, H. Tong, J. Ezick, and C. Lott, "How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark," arXiv, Jun. 2024.

[26] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A Code Efficiency Benchmark for Code Large Language Models," arXiv, Jun. 2024.

[27] T. Coignion, C. Quinton, and R. Rouvoy, "A Performance Study of LLM-Generated Code on Leetcode," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno Italy: ACM, Jun. 2024, pp. 79–89.

[28] S. Li, Y. Cheng, J. Chen, J. Xuan, S. He, and W. Shang, "Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. Tsukuba, Japan: IEEE, Oct. 2024, pp. 216–227.

[29] T. Vartziotis, I. Dellatolas, G. Dasoulas, M. Schmidt, F. Schneider, T. Hoffmann, S. Kotsopoulos, and M. Keckeisen, "Learn to Code Sustainably: An Empirical Study on Green Code Generation," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, ser. LLM4Code '24, Sep. 2024, pp. 30–37.

[30] J. F. Tuttle, D. Chen, A. Nasrin, N. Soto, and Z. Zong, "Can LLMs Generate Green Code - A Comprehensive Study Through LeetCode," in *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*, Nov. 2024, pp. 39–44.

[31] V.-A. Cursaru, L. Duits, J. Milligan, D. Ural, B. R. Sanchez, V. Stoico, and I. Malavolta, "A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama," arXiv, May 2024.

[32] P. Rani, J.-A. Bard, J. Sallou, A. Boll, T. Kehrer, and A. Bacchelli, "Can We Make Code Green? Understanding Trade-Offs in LLMs vs. Human Code Optimizations," arXiv, Mar. 2025.

[33] H. Peng, A. Gupte, N. J. Eliopoulos, C. C. Ho, R. Mantri, L. Deng, W. Jiang, Y.-H. Lu, K. Läufer, G. K. Thiruvathukal, and J. C. Davis, "Large Language Models for Energy-Efficient Code: Emerging Results and Future Directions," arXiv, Oct. 2024.

[34] N. Jegham, M. Abdelatti, L. Elmoubarki, and A. Hendawi, "How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference," arXiv, May 2025.

[35] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv, Jan. 2023.

[36] S. Xu, W. Xie, L. Zhao, and P. He, "Chain of Draft: Thinking Faster by Writing Less," arXiv, Mar. 2025.

[37] S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried, "ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness?" arXiv, Jul. 2024.

[38] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, "Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model," arXiv, Jun. 2024.

[39] A. Jagannadharao, N. Beckage, S. Biswas, H. Egan, J. Gafur, T. Metsch, D. Nafus, G. Raffa, and C. Tripp, "A Beginner's Guide to Power and Energy Measurement and Estimation for Computing and Machine Learning," arXiv, Dec. 2024.

[40] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," arXiv, Sep. 2023.

[41] Z. Yang, K. Adamek, and W. Armour, "Part-time Power Measurements: Nvidia-smi's Lack of Attention," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2024, pp. 1–17.

[42] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, "Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model," *Journal of Machine Learning Research*, vol. 24, no. 253, pp. 1–15, 2023.

[43] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto *et al.*, "Evaluating Large Language Models Trained on Code," arXiv, Jul. 2021.

[44] R. R. Wilcox, *Introduction to robust estimation and hypothesis testing*. Academic press, 2011.

[45] B. Danglot, J.-R. Falleri, and R. Rouvoy, "Can we spot energy regressions using developers tests?" *Empirical Software Engineering*, vol. 29, no. 5, p. 121, Jul. 2024.

[46] A. S. Luccioni, E. Strubell, and K. Crawford, "From Efficiency Gains to Rebound Effects: The Problem of Jevons' Paradox in AI's Polarized Environmental Debate," arXiv, Jan. 2025.

[47] D. Wright, C. Igel, G. Samuel, and R. Selvan, "Efficiency is Not Enough: A Critical Perspective of Environmentally Sustainable AI," Mar. 2025.

[48] S. Cerf, A. Luxey-Bitri, C. Quinton, R. Rouvoy, T. Simon, and C. Truffert, "Untangling the Critical Minerals Knot: when ICT hits the Energy Transitions," Dec. 2023.