

Green Code or Costly Code? Evaluating the Energy Costs of Iterative LLM-Based Code Optimizations

1st Sadra Emami
University of Augsburg
Augsburg, Germany
sadra.emami@uni-a.de

2nd Benjamin Weigell
University of Augsburg
Augsburg, Germany
benjamin.weigell@uni-a.de

3rd Mohammad Reza Mousavi
King's College London
London, United Kingdom
mohammad.mousavi@kcl.ac.uk

4th Gunel Jahangirova
King's College London
London, United Kingdom
gunel.jahangirova@kcl.ac.uk

5th Hugo Araujo
King's College London
London, United Kingdom
hugo.araujo@kcl.ac.uk

6th Bernhard Bauer
University of Augsburg
Augsburg, Germany
bernhard.bauer@uni-a.de

Abstract— Large Language Models (LLMs) are increasingly used to optimize code for performance. This practice often assumes that improvements in execution speed naturally translate into overall lower energy consumption. We challenge this assumption empirically by measuring whether performance-optimized code produced by LLM self-refinement pipelines is also energy-efficient relative to canonical human reference solutions, and whether the energy cost of the optimization process can be offset by subsequent execution-time savings. We evaluate 11 open-source models across four decoding configurations on all 1,000 problems of EffiBench, a state-of-the-art performance benchmark, while collecting per-process, hardware-level energy measurements throughout. Inference energy spans 0.79–8.68 MJ across model-variant pairs, which is roughly equivalent to the energy of 250–2800 GPT-4.1 queries. Limiting the maximum number of output tokens has the largest impact, reducing energy use by up to 69% compared to the default configuration. On the execution side, 42 of 44 configurations produce code that consumes *more* energy per problem than the canonical human reference, with overheads ranging from +4% to +886%. The two exceptions achieve only marginal gains, with break-even points on the order of 10^6 – 10^7 executions. Mixture-of-Experts architectures consume substantially less inference energy than equivalently-sized dense models without sacrificing correctness, making architectural choice a more consequential sustainability lever than decoding parameters. Performance optimization and energy efficiency are not always aligned goals: LLM self-refinement consistently produces faster-seeking but energy-heavier code relative to what a human engineer would write. All artifacts are available at <https://doi.org/10.5281/zenodo.23598170>.

Index Terms—large language models, energy consumption, green software engineering, code efficiency, code optimization, break-even analysis

I. INTRODUCTION

Recent studies indicate that developers routinely integrate Large Language Models (LLMs) into their daily coding workflows [1], [2]. Beyond code generation, LLMs are used to iteratively refine code to reduce execution time and memory usage, tasks that previously required expert manual effort [3].

This growing practice implicitly assumes that performance improvements also translate into environmental benefits: that faster code is greener code. However, each optimization interaction carries two distinct energy costs: the *inference energy* consumed when the model produces code [4], [5], and the *execution energy* consumed when that code is later run [6]. Both costs are largely invisible to the developer, yet together determine whether an optimization yields a net energy benefit. However, it remains unclear whether the execution-time savings can offset the energy required to generate the optimization, or whether LLM-optimized code is more energy-efficient than code written by human developers [7], [8].

Coignon et al. [9] introduce the Break-Even Point (BEP), defined as the number of executions at which cumulative execution savings offset generation cost, and find that this point ranges from under 100 to over 200,000 executions depending on the large language model and the method that is being optimized. However, their analysis measures savings relative to each model's own unoptimized output rather than a canonical human-written solution. This distinction is important because a model can greatly improve upon its own baseline while still producing code that is less efficient than a competent human solution. As a result, it remains unclear whether LLM-optimized code is actually competitive with what a competent human engineer would produce.

We investigate this question by evaluating the energy consumption of LLM-based code optimization end-to-end. We use EffiLearner [3], a state-of-the-art iterative self-refinement pipeline for code performance optimization, to generate optimized solutions. We evaluate these solutions against the canonical human-written reference implementations in EffiBench [10], a benchmark of 1,000 LeetCode problems paired with top-rated human solutions. We evaluate only open-source models deployed on-premise. Such deployments are increasingly adopted in privacy-sensitive environments, while also allowing us to instrument the underlying hardware and obtain precise, hardware-level measurements of inference energy rather than relying on estimates or provider-reported statistics.

Addressing the practical value of LLM-based optimization requires examining three different aspects. First, the optimization process itself incurs an energy cost that must be quantified. Second, this cost may vary substantially across models and decoding strategies, potentially changing the trade-off between optimization quality and energy consumption. Finally, the generation investment is worthwhile only if the resulting code consumes less execution energy than a strong human-written baseline, allowing the initial energy cost to be recovered over repeated executions. These considerations motivate the following research questions:

- **RQ1.** What is the energy cost of generating performance-optimized code using LLMs?
- **RQ2.** How does the energy cost of generating optimized code vary across open-source LLMs and decoding configurations?
 - **RQ2a.** How do decoding parameters (temperature, context window, maximum output tokens) affect energy consumption across models?
 - **RQ2b.** Which model-configuration pairs offer the most favorable trade-off between inference energy and optimization quality?
- **RQ3.** Does performance-optimized code consume less execution energy than canonical human reference solutions, and if so, at what number of executions does the generation investment break even?

Contributions.

Our main contributions are as follows:

- 1) We present the first end-to-end energy evaluation of iterative LLM-based code optimization against canonical human-written reference solutions.
- 2) We conduct a large-scale empirical study of the energy cost of LLM-based code optimization across 11 open-source models and multiple decoding configurations, providing a systematic characterization of the factors that influence inference energy and optimization cost.
- 3) We release all experimental artifacts, including repaired EffiBench benchmark, to support reproducibility and future research on energy-efficient LLM-based code optimization.

Findings in brief. Our results show that inference energy ranges from 0.79 to 8.68 MJ across model-configuration pairs, equivalent to roughly 250–2800 GPT-4.1 queries [11]. Among the evaluated decoding parameters, limiting the maximum output length has by far the greatest impact, reducing inference energy by 13–69% relative to the default configuration.

On the execution side, the optimized code is almost never more energy-efficient than the canonical human reference: 42 of the 44 evaluated configurations consume between 4% and 886% more execution energy per problem. Only two configurations outperform the human baseline, requiring approximately 3.4×10^6 and 6.4×10^7 executions, respectively, to recover the energy invested in optimization.

The rest of this paper is organized as follows. **Section II** surveys related work. **Section III** describes the dataset, pipeline, models, hardware, and instrumentation. **Section IV** reports re-

sults for each RQ and the carbon-footprint analysis. **Section V** discusses the implications of our findings. **Section VI** outlines the limitations of our study and directions for future work. **Section VII** draws the main conclusions of the study.

II. RELATED WORK

The energy footprint of LLMs has been studied along two axes: the cost of *operating* the model, and the cost of *running the code* the model produces. Our study sits at the intersection of both.

A. Inference-Pipeline Energy

Luccioni et al. [12] reported that pre-training BLOOM consumed 433,196 kWh and 24.69 tonnes of CO₂eq. The systematic mapping by Verdecchia et al. [24] confirmed that the Green AI literature is heavily skewed toward training, with inconsistent measurement protocols hampering cross-study comparison. Budenny et al. [25] released `eco2AI`, exposing a 26-fold range in grid carbon intensity that makes *where* a model runs nearly as consequential as how efficiently it runs. These studies establish the measurement landscape our work builds on, but focus exclusively on training. The inference cost of iterative optimization pipelines remains unexamined.

On the inference side, Samsi et al. [5] provide the most detailed power profile of LLaMA inference across deployments, observing system draws between 300 W and 1 kW, while a study [26] estimates that at ChatGPT-scale traffic, inference matches the energy cost of training in just about 2 days. LASSI-EE [22] is the closest prior pipeline to ours in spirit: an iterative, self-correcting LLM loop with runtime power-profile feedback applied to HPC code, reporting 36% and 34% energy reductions on MI100 and A100 respectively. Like the training-focused studies above, however, it does not connect inference and execution energy through a break-even analysis, and targets bespoke HPC kernels rather than a standardised benchmark like EffiBench. Crucially, no prior work jointly measures inference and execution energy across a controlled decoding parameter sweep on a standardised benchmark, which is the gap this paper fills.

B. Effect of Model Configuration on Inference Energy

Several recent studies have identified output token length as the dominant inference energy driver. Poddar et al. [13] report a Pearson correlation of $r = 0.952$ between output tokens and energy, against $r = 0.697$ for input tokens. Adamska et al. [14] independently measure $r = 0.9$ on a separate corpus and find that prompt phrasing can influence response length. Both findings emerge as observational correlations across natural output variation rather than from controlled ablations of the output-token budget as an independent variable. To our knowledge, no prior study provides an isolated, controlled measurement of *sampling temperature*, *context window size*, or *maximum output tokens* as independent inference-energy levers while holding the remaining factors constant. Furthermore, both studies focus on general NLP tasks such as classification, question answering, and summarization; neither

TABLE I
COVERAGE OF THE DIMENSIONS RELEVANT TO OUR THREE RQS ACROSS PRIOR WORK, GROUPED BY CAPABILITY PROFILE. ○ = NOT ADDRESSED, ◐ = PARTIAL, ● = ADDRESSED.

Group	Task	Inference Energy	Runtime Energy	Standardized Benchmark	Parameter Sweep	Break-Even Analysis	Iterative Pipeline	Canonical Reference
Inference-energy studies [5], [12]–[14]	General LLM	●	○	○	◐	○	○	○
DL framework energy [15]	General DL	●	○	●	○	○	○	○
Inference energy for code gen. [16]–[19]	Code Generation	●	○	◐	○	○	○	○
Generated-code energy [7], [10], [20], [21]	Code Generation	○	●	●	○	○	○	●
Code-optimisation pipelines [3], [22], [23]	Code Optimization	○	●	◐	○	○	●	◐
Break-even analysis [9]	Code Optimization	●	●	●	○	●	◐	○
Our work	Code Optimization	●	●	●	●	●	●	●

examines code generation or optimization pipelines, where output structure and length distributions differ substantially from natural language responses. Controlled comparisons do exist for the software stack. Georgiou et al. [15] measure training and inference energy of six deep learning models under matched configurations in PyTorch and TensorFlow, and report statistically significant differences between the frameworks in 94% of the energy and runtime comparisons, with comparable accuracy and no single framework superior across all tasks. The factor they vary is the framework, however, and the models they study are not applied to code generation.

C. Runtime Energy of Generated Code and the Break-Even Point

Empirical consensus establishes that LLM-generated code is systematically less efficient than canonical human solutions. EffiBench [10], which evaluated 42 models, found that even the best open-source model (StarCoder2-15B) required $2.59\times$ the execution time and $1.71\times$ the peak memory of human solutions. Islam et al. [7] extend this picture on 878 EffiBench problems, finding that the best model (DeepSeek-v3) remains $1.17\times$ less energy-efficient than humans, with gaps reaching $449\times$ for dynamic programming and bit-manipulation tasks. Convergent findings come from Tuttle et al. [20], who report LLMs outperforming human solutions in only 5.7% of 192 LeetCode responses. Apsan et al. [21], who show the human–LLM gap persists across server, PC, and Raspberry Pi platforms. Solovyeva et al. [8], who find parity only in narrow regimes such as String and Tree tasks; and Rani et al. [23], who document that LLM-suggested optimizations on Matlab often unintentionally increase memory and energy usage. Whether iterative self-refinement can close this gap, and at what inference cost, motivates the methodological thread we build on.

The methodological precursor to our pipeline is EffiLearner [3], which iteratively refines LLM-generated code by feeding execution-overhead profiles back to the model as structured prompts, demonstrating consistent reductions

in Normalized Execution Time (NET) and Normalized Total Memory Usage (NTMU) across 22 models, but without measuring inference energy and therefore unable to answer whether the optimization investment is recovered at execution time. Coignon et al. [9] first formalised this question via the BEP ($BEP = E_{\text{optim}}/(e_{\text{orig}} - e_{\text{opt}})$), finding results spanning four orders of magnitude: DeepSeek-R1-14B with EoH yields 49% per-execution savings but requires 220,324 executions to break even, while pathologically inefficient code can drop from 318 J to 2.6 J and break even within 100 runs. Critically, their BEP measures savings relative to each model’s own unoptimized output rather than a canonical human reference, leaving open the question of whether LLM-optimized code is competitive with what a human engineer would write.

D. Gap Analysis

Most of the above studies address either (i) the inference energy of LLMs in general NLP settings, (ii) the runtime efficiency of LLM-generated code relative to human references, or (iii) the break-even economics of LLM-based optimization relative to each model’s own unoptimized output. No prior study jointly measures inference energy across a controlled decoding parameter sweep, evaluates the resulting code against canonical human solutions, and applies a break-even analysis within a multi-epoch self-refinement pipeline on a standardised benchmark. Our study is designed to fill this intersection.

Table I summarizes how our study differs from the closest prior work along the dimensions most relevant to our three RQs.

III. METHODS

In this section, we first describe the benchmark dataset and the self-refinement framework. We then present the evaluated models and decoding configurations, followed by the hardware platform and energy-measurement instrumentation. Finally, we explain how this experimental setup is used to answer each research question.

A. Dataset

We evaluate on EffiBench [10], a benchmark of 1,000 efficiency-critical LeetCode problems spanning 171 *Easy*, 589 *Medium*, and 240 *Hard* problems. Each entry pairs a problem description with an executable canonical solution drawn from top-starred LeetCode Discussion Forum submissions, a test-case generator, and 100 test cases. The canonical solutions serve as the human reference baseline for our RQ3 break-even analysis. Because they are top-starred submissions, they represent highly efficient human implementations rather than average developer code, and we discuss the implications in Section VI.

EffiBench is particularly well suited to our study because its algorithmically challenging problems exhibit substantial variation in execution time, making differences in execution energy measurable.

The EffiBench test cases were generated by GPT-3.5 Turbo, and during our validation run we found that 122 of the 1,000 problems contained defects preventing correct execution, all originating in the generated test cases rather than the solutions themselves. The most common defects were ordering-sensitive assertions on unordered outputs, missing helper utilities for tree and linked-list construction, object-address serialization embedding raw in-memory representations, wrong precomputed expected values, method-name mismatches, redundant class redefinitions shadowing the actual implementation, and empty test cases caused by generator logic that emitted no output. For each defect type we applied a targeted repair, recomputing expected values from the canonical solution, injecting missing helpers, rewriting object references as constructor calls, removing shadowing definitions, and converting ordering-sensitive assertions to order-independent equivalents. After repair, all 1,000 canonical solutions pass their respective test cases.

B. Self-Refinement Framework

We adopt EffiLearner [3] as our self-refinement pipeline because it directly targets execution-time and memory efficiency through line-level profiling feedback, and has been shown to produce significantly more efficient code across a wide range of model architectures, achieving state-of-the-art reductions in execution time and memory usage compared to alternative self-refinement approaches. EffiLearner structures code optimization as an iterative refinement loop comprising three stages: code generation, overhead profiling, and code refinement. Given a task description, the LLM first produces an initial code candidate.

At each iteration, the current candidate is executed locally against the set of open test cases provided by the benchmark, and two line-level overhead profiles are collected: an execution-time profile via `line_profiler` and a memory-usage profile via `memory_profiler`. Each profiling measurement is repeated 60 times per script, and the reported Execution Time (ET), Max Memory Usage (MU), and Total Memory Usage (TMU) values are the averages over those 60 runs. To minimize the influence of thermal throttling and

background load, a cooldown period of 15 seconds is enforced between consecutive runs. A run is only admitted once CPU load falls below 50% and the CPU package temperature is within the 50–60 °C range. Both profiles aggregate measurements across all open test cases and report, per line, the cumulative time and memory footprint. The current candidate, its overhead profiles, the task description, and the open test cases are then concatenated into a refinement prompt and passed back to the LLM, which emits a revised candidate. The revised candidate is validated for functional correctness against the open test cases only; if it passes, it replaces the current candidate as the input to the next iteration. The loop terminates after a fixed number of iterations. Final evaluation is conducted on a held-out set of private test cases disjoint from the open test cases, following the protocol of EffiBench [10].

C. Models and Parameter Configurations

Models were selected by querying the Arena Leaderboard on Hugging Face¹ and sorting by coding performance. We restricted our selection to open-source models deployed entirely on-premise, as local inference enables controlled energy measurement on a fixed, instrumented hardware environment and reflects the growing adoption of local deployment in privacy-sensitive settings. We selected representative size variants from the top-performing families, ensuring architectural diversity across both dense and Mixture-of-Experts (MoE) architectures to enable a controlled comparison of their energy and quality characteristics. The resulting set of 11 models spans four families [27]–[31], different size scales, and two architectural classes, as listed in Table II.

TABLE II
MODELS EVALUATED IN THIS WORK.

Family	Model	Parameters	Architecture
Qwen3.5	Qwen3.5-4B	4B	Dense
	Qwen3.5-9B	9B	Dense
	Qwen3.5-27B	27B	Dense
	Qwen3.5-35B-A3B	35B	MoE
GLM	GLM-4.7-Flash	31B	Dense
	GLM-4.6V-Flash	10B	Dense
Ministral	Ministral-3B	3B	Dense
	Ministral-8B	8B	Dense
	Ministral-14B	14B	Dense
Nemotron	Nemotron-Cascade	30B	MoE
	Nemotron-Nano	30B	MoE

For each model, we run four configurations that each change at most one parameter, so that energy differences can be attributed to a single cause. We vary only deployment levers that practitioners can adjust without retraining and that act on energy through the number of generated tokens or the memory provisioned for generation. The *default* configuration uses temperature 0 (greedy decoding) and a maximum output length of 2,048 tokens. The *sampling* variant raises the temperature to 0.5. In a pilot experiment, temperatures above 0.5 increased verbosity and inference energy without

¹<https://huggingface.co/spaces/Imarena-ai/arena-leaderboard>, accessed March 2025

improving code efficiency, so we did not sweep higher values. The *reduced-context* variant caps the context window at 64k tokens, since all evaluated models support windows three to four times larger by default. The *reduced-output* variant halves the maximum output length to 1,024 tokens to test whether a shorter generation budget lowers energy without hurting optimization quality. A full factorial sweep was timely infeasible.

D. Experimental Setup

All experiments were conducted on a server equipped with two Intel Xeon Gold 6438Y+ processors with 128 logical cores in total, 1 TB of system memory, and two NVIDIA L40 GPUs with 46 GB GDDR6 VRAM each. Simultaneous Multithreading (SMT) was enabled and the system exposed two Non-Uniform Memory Access (NUMA) nodes. The server ran Ubuntu 24.04 with Linux kernel 6.8.0-138, and models were served locally via vLLM [32]. Processes were pinned to physical cores, while the frequency governor, turbo boost, C-states, and NUMA policy remained at their operating system defaults to reflect standard deployments. The full experimental pipeline required approximately 30 days of continuous execution.

E. Energy Measurement

We instrument all experiments with Metrion [33], an energy measurement tool that attributes energy consumption to processes at the thread level. Most existing tools neglect the combined effects of frequency scaling, simultaneous multithreading, and non-uniform memory access in multi-socket architectures. Utilization-based tools like CodeCarbon [34] estimate process-level energy from coarse-grained CPU time and RAPL values without accounting for these hardware-level effects, while profiling utilities like `perf` expose RAPL metrics only at the package level. Metrion overcomes these limitations by combining Intel RAPL, `nvidia-smi`, and Hardware Performance Counters (HWPCs) to attribute CPU, DRAM, and GPU energy at the thread level, with HWPCs resolving per-thread footprints within each 1-second sampling window. Metrion does not require a process to span an entire sampling interval. It uses eBPF and HWPCs to trace when each process is scheduled on or off the CPU and to count its unhalted cycles, and it allocates the RAPL package energy, sampled at 1 Hz, to active processes in proportion to their unhalted cycles in each interval. This attributes energy correctly even to short-lived processes. Frequency scaling and hardware topology are handled at attribution time. Metrion weights active cycles by the APERF/MPERF ratio, which reflects the actual frequency under Dynamic Voltage and Frequency Scaling (DVFS) and Turbo Boost, scales overlapping SMT threads by a calibrated contention factor of 1.15, and attributes energy per socket. Energy is aggregated over all descendant threads of each solution process, yielding a per-solution breakdown into CPU, DRAM, and GPU, each decomposed into active and idle components.

1) *Measurement Protocol*: Inference energy and execution energy are measured under different repetition budgets, reflecting the difference in their computational cost.

a) *Inference energy*.: Each *(model, configuration)* pair is executed six times in full, covering all 1,000 problems per run. The reported $E_{\text{total}} (= E_{\text{gen}} + E_{\text{exec}})$ is the mean over these six runs. Running inference six times provides sufficient stability for the aggregate energy signal while keeping total experiment time feasible given the cost of a full five-epoch optimization pass. GPU energy applies only to this inference phase, whose runs last from hours to days, and is recorded continuously via NVML.

b) *Execution energy*.: Each solution script is executed 60 times sequentially within one persistent Python interpreter, and the reported E_{exec} is the mean over those 60 runs. Keeping the interpreter alive amortizes startup and import overhead, so the measurement reflects steady-state energy. Canonical and LLM-generated solutions share an identical harness, so any residual overhead affects both symmetrically. Individual executions average more than 0.1 seconds, so the 60 sequential executions of each solution give cumulative measurement windows well over one second. The higher repetition count relative to inference is feasible because individual script executions are cheaper. The same cooldown protocol applied to resource profiling is used here: a 15-second pause is enforced between consecutive executions, and a run is only admitted once CPU load is below 50% and the CPU package temperature is within the 50–60 °C range. These admission criteria prevent cross-experiment interference and thermal throttling.

F. Approaches to the Research Questions

The infrastructure described above produces two complementary energy measurements at different granularities. For each *(model, configuration)* pair, we record the aggregate *generation energy* E_{gen} expended by the LLM during the entire generation and optimization run across all 1,000 problems. For each *(model, configuration, problem)* triple, it additionally records the per-process *execution energy* E_{exec} consumed when running the resulting solution script. We use these measurements to answer each research question as follows.

1) *RQ1 – Energy Cost of Optimization*: To quantify the energy cost of generating performance-optimized code, we report E_{gen} at the granularity at which it is measured: one aggregate value per *(model, configuration)* pair, broken down by hardware component (CPU package, DRAM, GPU) and by dynamic versus idle contributions, where the dynamic component captures power draw above the static baseline and the idle component captures the baseline itself. E_{gen} includes the full generation cost across all five optimization epochs, including the initial code generation. We report an aggregate over all 1,000 problems because vLLM batches requests across problems and our research questions concern model-level and configuration-level energy budgets.

2) *RQ2 – Cross-Model and Parameter Variation*: RQ2 examines how generation energy varies with model choice and

decoding parameters. We evaluate this along two complementary axes.

a) *Parameter sensitivity (RQ2a).*: For every model in Table II, we compare the energy consumption of each model based on four parameter configurations defined in Section III-C. Holding the model fixed isolates the energy impact of temperature, context size, and maximum output length.

b) *Cross-model comparison (RQ2b).*: To identify configurations that simultaneously minimize generation cost and improve the performance of the optimized code, we plot correctness efficiency ($\text{pass}@1/E_{\text{total}}$, in % MJ^{-1}) against the NET of the last-epoch solutions for each of the four decoding configurations (default, t0.5, ctx65536, and mt1024), where $E_{\text{total}} = E_{\text{gen}} + E_{\text{exec}}$ is the sum of generation and execution energy.

3) *RQ3 – Break-Even of Generation vs. Execution Savings:* RQ3 asks whether the energy invested in generating optimized code is recovered through reduced energy consumption at execution time. We treat the canonical solution provided by EffiBench as the reference baseline against which optimized solutions are compared. Each canonical solution is a top-rated, human-written solution selected from competitive programming submissions, representing a strong but attainable efficiency ceiling. We aggregate at the (model, configuration) level rather than per task because our goal is to characterize the overall energy trade-off of deploying a given model under a given decoding configuration, not to identify which individual problems benefit from refinement.

For each (model, configuration) pair, we compute the mean execution energy of the optimized solutions $\bar{E}_{\text{exec}}^*$ and the mean execution energy of the canonical solutions $\bar{E}_{\text{exec}}^{\text{canon}}$ over the benchmark, and define the per-run saving as

$$\Delta E = \bar{E}_{\text{exec}}^{\text{canon}} - \bar{E}_{\text{exec}}^*.$$

The break-even point is the number of executions at which the cumulative execution-time saving offsets the up-front generation cost,

$$N^* = \left\lceil \frac{E_{\text{total}}}{\Delta E} \right\rceil \quad \text{when } \Delta E > 0,$$

where E_{total} is the aggregate generation energy of the optimization run. N^* is thus a single scalar per (model, configuration) pair, expressing how many executions of a representative benchmark problem are required before the energy invested in optimization is recouped relative to the canonical baseline. When the mean optimized solution is no more energy-efficient than the canonical baseline ($\Delta E \leq 0$), no break-even point exists and the optimization constitutes a net energy loss in aggregate. We report this outcome explicitly where it occurs.

IV. RESULTS

This section presents results for three research questions: the energy cost of LLM-based code optimization (Section IV-A), how that cost varies across models and decoding configurations (Section IV-B), and whether the generation investment

breaks even against execution-time savings relative to canonical solutions (Section IV-C).

A. *RQ1: What is the energy cost of generating performance-optimized code using LLMs?*

All 11 models first produced an initial solution through base generation, which was then subjected to five epochs of iterative self-optimization. We evaluate functional correctness via the pass rate on the open test cases, and resource efficiency via ET, MU, and TMU, each reported in absolute terms and normalized against the profiling results of the canonical solutions, resulting in NET, Normalized Max Memory Usage (NMU) and NTMU. Table III reports base generation values (top row) and post-optimization values (bottom row) for all metrics, with the relative change in parentheses.

a) *Execution Time.*: NET at base generation ranges from 0.97 (Minstral-14B) to 1.43 (Nemotron-Nano), with the GLM and Nemotron families consistently above 1.0. After self-optimization, ET reductions of up to 29.0% are achieved, with Nemotron-Nano showing the largest relative improvement. The Qwen family already starts below or near 1.0 at base generation and shows modest further reductions of 1.5–1.7%.

b) *Memory Usage.*: Normalized Max Memory Usage (NMU) at base generation ranges from 1.0029 (GLM-4.6V-Flash) to 1.0411 (Nemotron-Nano). After self-optimization, some models reduce NMU by up to 2.1%, converging toward the canonical footprint. NTMU reductions are more pronounced, reaching 36.6% for Nemotron-Nano, and several models, including Minstral-3B, Minstral-8B and GLM-4.7-Flash achieve NTMU values below 1.0 after optimization.

c) *Functional Correctness.*: The highest pass@1 is achieved by Qwen3.5-27B (81.0%), followed by Qwen3.5-35B-A3B (78.55%) and Qwen3.5-9B (66.80%). GLM models achieve the lowest correctness, with GLM-4.7-Flash at 25.02% and GLM-4.6V-Flash at 25.42%.

d) *Inference Energy.*: Table IV reports the inference energy consumed during the five-epoch optimization loop and shows that total inference energy varies considerably across models. Minstral-3B is the most energy-efficient at 1.27 MJ, followed by Qwen3.5-4B at 1.38 MJ. Qwen3.5-27B incurs the highest cost at 8.58 MJ, reflecting its larger parameter count. GLM-4.7-Flash is the second most expensive at 6.08 MJ despite achieving only 25.02% pass@1, making it the least cost-effective configuration overall. Notably, Qwen3.5-35B-A3B consumes only 3.12 MJ despite having more total parameters than Qwen3.5-27B. This is due to its MoE architecture, which activates only a subset of parameters per token and therefore reduces the effective compute per inference step. CPU energy dominates across all models, accounting for 59–65% of E_{total} , with GPU contributing the remainder and DRAM a negligible fraction (<0.05%).

TABLE III

RESOURCE EFFICIENCY BEFORE (TOP ROW) AND AFTER (BOTTOM ROW) SELF-OPTIMIZATION AND PASS@1, WITH RELATIVE CHANGE IN PARENTHESES. ET: EXECUTION TIME (S); MU: MAX MEMORY USAGE (MB); TMU: TOTAL MEMORY USAGE (MB-S). NET, NMU AND NTMU ARE ET, MU AND TMU NORMALIZED BY THE CANONICAL SOLUTION; VALUES BELOW 1.0 BEAT THE CANONICAL REFERENCE.

Model	ET (s)	NET	MU (MB)	NMU	TMU	NTMU	pass@1 (%)
GLM-4.6V-Flash	0.133 0.131 (1.4%)	1.052 1.038 (1.4%)	23.098 23.091 (0.0%)	1.003 1.003 (0.0%)	2.215 2.189 (1.2%)	1.053 1.040 (1.3%)	25.4
GLM-4.7-Flash	0.129 0.123 (4.7%)	1.034 0.984 (4.8%)	23.150 23.133 (0.1%)	1.005 1.005 (0.1%)	2.168 2.062 (4.9%)	1.039 0.989 (4.8%)	25.0
Ministral-3B	0.126 0.120 (4.8%)	0.994 0.946 (4.9%)	23.439 23.248 (0.8%)	1.018 1.009 (0.8%)	2.170 2.026 (6.7%)	1.024 0.956 (6.7%)	32.7
Ministral-8B	0.135 0.118 (12.6%)	1.089 0.949 (12.9%)	23.635 23.385 (1.1%)	1.026 1.015 (1.1%)	2.425 2.005 (17.3%)	1.167 0.966 (17.2%)	52.6
Ministral-14B	0.121 0.119 (1.7%)	0.971 0.954 (1.8%)	23.303 23.231 (0.3%)	1.011 1.008 (0.3%)	2.031 1.996 (1.7%)	0.980 0.963 (1.8%)	55.3
Nemotron-Cascade	0.142 0.130 (8.4%)	1.143 1.050 (8.2%)	23.743 23.480 (1.1%)	1.030 1.019 (1.1%)	2.529 2.234 (11.7%)	1.213 1.079 (11.1%)	43.2
Nemotron-Nano	0.180 0.128 (29.0%)	1.431 1.020 (28.7%)	23.985 23.476 (2.1%)	1.041 1.019 (2.1%)	3.453 2.190 (36.6%)	1.641 1.045 (36.4%)	41.5
Qwen3.5-4B	0.124 0.122 (2.1%)	0.988 0.972 (1.6%)	23.307 23.283 (0.1%)	1.011 1.010 (0.1%)	2.069 2.024 (2.1%)	0.995 0.979 (1.6%)	51.3
Qwen3.5-9B	0.127 0.126 (1.1%)	1.020 1.003 (1.7%)	23.303 23.298 (0.0%)	1.011 1.011 (0.0%)	2.126 2.082 (2.0%)	1.029 1.012 (1.7%)	66.8
Qwen3.5-27B	0.124 0.122 (1.5%)	0.989 0.974 (1.6%)	23.184 23.173 (0.0%)	1.006 1.005 (0.0%)	2.035 2.003 (1.6%)	0.993 0.977 (1.6%)	81.0
Qwen3.5-35B-A3B	0.123 0.121 (1.8%)	0.986 0.971 (1.5%)	23.207 23.197 (0.0%)	1.007 1.006 (0.0%)	2.030 1.998 (1.6%)	0.989 0.974 (1.5%)	78.6

TABLE IV

INFERENCE ENERGY CONSUMPTION PER MODEL DURING SELF-OPTIMIZATION, IN MEGAJOULES (MJ). CPU % AND GPU % ARE EACH COMPONENT'S SHARE OF E_{total} .

Model	E_{total}	E_{CPU}	E_{GPU}	CPU %	GPU %
GLM-4.6V-Flash	1.926	1.231	0.695	63.9	36.1
GLM-4.7-Flash	6.077	3.636	2.440	59.8	40.2
Ministral-3B	1.273	0.829	0.444	65.1	34.8
Ministral-8B	2.625	1.693	0.932	64.5	35.5
Ministral-14B	3.512	2.220	1.291	63.2	36.8
Nemotron-Cascade	3.085	1.913	1.171	62.0	38.0
Nemotron-Nano	3.262	2.009	1.252	61.6	38.4
Qwen3.5-4B	1.382	0.903	0.478	65.4	34.6
Qwen3.5-9B	2.259	1.447	0.812	64.0	35.9
Qwen3.5-27B	8.577	5.125	3.450	59.8	40.2
Qwen3.5-35B-A3B	3.125	1.990	1.134	63.7	36.3

RQ1: Self-optimization consistently reduces ET, MU, and TMU across all models, with the Nemotron family showing the largest relative improvements. Pass@1 varies substantially across families, with Qwen models achieving the highest correctness. Inference energy is strongly driven by model size, though MoE models consume considerably less energy than equivalently-sized dense models. The GLM family combines high inference cost with the lowest correctness, making it the least cost-effective choice overall.

B. RQ2: How does the energy cost of generating optimized code vary across open-source LLMs and decoding configurations?

Table V presents the resource-efficiency metrics, inference energy, and pass@1 for all 11 models across the four evaluated variants after five epochs of self-optimization. Figure 1 summarizes total inference energy per model and variant. Figure 2

plots correctness efficiency against normalized execution time to support the cross-model comparison in RQ2b.

1) *RQ2a: How do decoding parameters affect energy consumption across models?:*

a) *Effect on inference energy:* As shown in Figure 1, *mt1024* produces the lowest total inference energy across all models, with reductions of up to 69% relative to *default* (mean 2.26 MJ vs. 3.37 MJ). The savings are largest for GLM-4.7-Flash (6.08 MJ $\rightarrow$ 1.87 MJ, -69.3%) and smallest for Qwen3.5-4B (1.38 MJ $\rightarrow$ 1.33 MJ, -3.8%), reflecting how strongly maximum output tokens constrains generation cost. The *ctx65536* and *t0.5* variants are broadly comparable to *default* in energy, with *t0.5* reaching the highest absolute consumption for Qwen3.5-27B (8.68 MJ), likely due to more diverse and verbose outputs under sampling.

b) *Effect on execution quality:* The *default* variant achieves NET values of 0.95–1.05 across all models, indicating near-canonical execution time. Non-default variants span a wider range of 0.93–2.51, where the upper extreme is driven by the Nemotron-Nano *mt1024*. NMU is largely insensitive to variant choice (range 1.002–1.088), while NTMU varies more substantially, reaching 3.19 for Nemotron-Nano *mt1024* and 1.52 for Ministral-8B *ctx65536*.

2) *RQ2b: Which model-configuration pairs offer the most favorable trade-off between inference energy and optimization quality?:* Figure 2 plots correctness efficiency (pass@1/ E_{total} , in $\% \text{MJ}^{-1}$) against NET for all model-variant pairs. The optimal region—low NET and high correctness per invested joule—is shaded in the upper-left quadrant.

Qwen3.5-4B dominates across all four variants, achieving the highest efficiency of 37–40 $\% \text{MJ}^{-1}$ while maintaining NET below 1.0. Its *ctx65536* variant reaches 40.15 $\% \text{MJ}^{-1}$, the best single configuration in the study. Ministral-3B *mt1024* is the best non-Qwen entry in the optimal region at

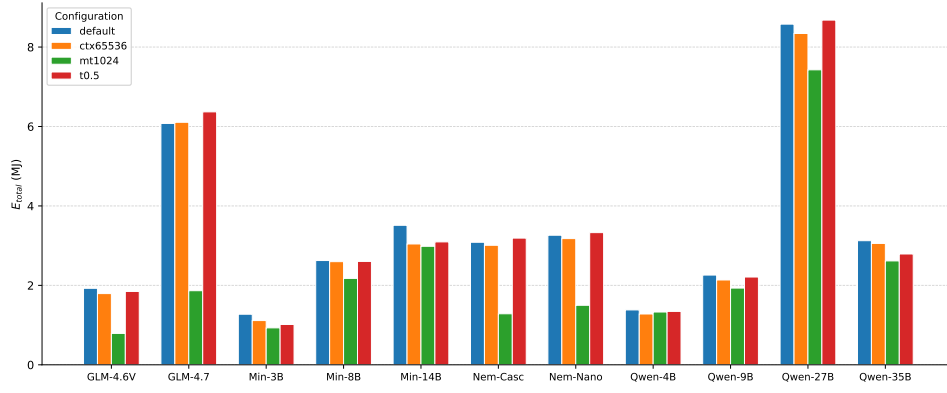


Fig. 1. Total inference energy E_{total} (MJ) per model and decoding variant after five epochs of self-optimization. *mt1024* is almost always the lowest-energy variant and *t0.5* is highest in most cases.

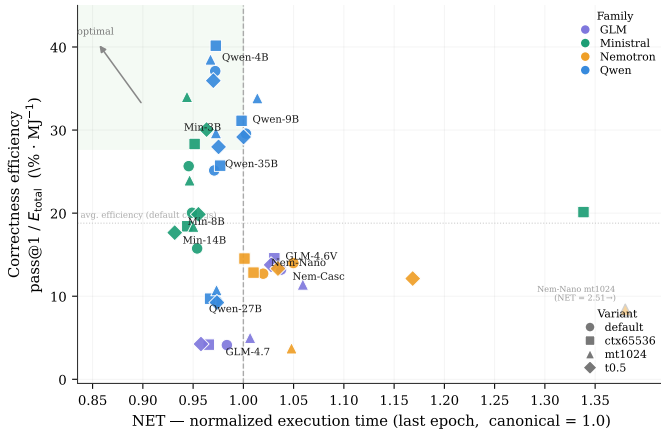


Fig. 2. Correctness efficiency ($\text{pass@1} / E_{\text{total}}$) vs. NET for all model-variant pairs. Shaded region: optimal zone. Dashed line: canonical baseline (NET = 1.0). Nemotron-Nano *mt1024* is clipped due to pass@1 collapse.

34.0 % MJ⁻¹. At the opposite extreme, GLM-4.7-Flash and Nemotron-Cascade *mt1024* are the least efficient configurations (4.1 and 3.7 % MJ⁻¹ respectively), combining high inference cost or correctness collapse with no compensating gain in execution quality. Qwen3.5-27B illustrates the cost of scaling: despite the highest absolute pass@1 (80.98%), its 8.58 MJ generation cost depresses its efficiency to ~ 9.4 % MJ⁻¹, well below Qwen3.5-4B.

RQ2a: Constraining output tokens (*mt1024*) is the most effective knob for reducing inference energy across all models, though it can severely collapse pass@1 for some configurations. **RQ2b:** Qwen3.5-4B dominates the correctness-efficiency frontier across all variants. Ministral-3B is the strongest non-Qwen alternative. The GLM family is strictly dominated on both axes, and high-parameter models such as Qwen3.5-27B are penalized by disproportionate inference cost despite leading on absolute pass@1.

C. RQ3: Does the energy invested in generation translate into reduced execution energy when the optimized code is run relative to canonical human reference solutions, and if so, at what number of executions does the investment break even?

For each model and decoding variant we measure the per-execution CPU energy of the last-epoch solution and compare it against the canonical solution. Figure 3 (left) reports the per-problem execution energy delta $(E_{\text{canon}} - E_{\text{opt}})/n$ for all model-variant pairs, and Figure 3 (right) shows the per-problem spread between the base and optimized epoch, $(E_{\text{base}} - E_{\text{opt}})/n$.

a) Canonical solutions dominate at execution time:

Figure 3 (left) shows that 42 of the 44 model-variant configurations produce optimized code that consumes *more* execution energy per problem than the canonical reference ($\Delta < 0$). The two exceptions are Qwen3.5-9B *t0.5* (0.65 J per problem) and Nemotron-Cascade *mt1024* (0.02 J per problem), the only configurations whose last-epoch solutions undercut the canonical baseline. The largest overheads are concentrated in the GLM family: GLM-4.6V-Flash across all variants exceeds 1.5 J per problem, peaking at 1.94 J (*default*). Qwen variants cluster at the lower end of the distribution (mean 0.26 J), while Ministral sits in the middle (mean 0.66 J).

Across decoding configurations, *mt1024* and *t0.5* produce the lowest mean deltas (0.43 J and 0.44 J respectively), compared to *default* and *ctx65536* (both ≈ 0.65 J). However, even the most favourable configuration (Qwen3.5-9B *t0.5*) achieves only a marginal gain over the canonical baseline, and the result is driven by a single model-variant pair rather than a systematic trend.

b) Optimization does not reliably improve upon its own base generation: Figure 3 (right) reveals that self-optimization fails to consistently reduce execution energy even relative to its own base generation: only 20 of the 44 configurations (45.5%) show improvement ($\bar{E}_{\text{base}} > \bar{E}_{\text{opt}}$), while 24 (54.5%) regress. The Ministral family fares best at 58.3% improved, while Qwen and Nemotron both sit at 37.5%. The largest regressions are observed for Nemotron-Nano *t0.5* (-0.54 J per problem) and Nemotron-Nano *default* (-0.44 J), where the best epoch

TABLE V

RESOURCE EFFICIENCY, INFERENCE ENERGY, AND PASS@1 AFTER SELF-OPTIMIZATION. N-PREFIXED METRICS ARE NORMALIZED AGAINST THE CANONICAL SOLUTION; VALUES BELOW 1.0 INDICATE BETTER PERFORMANCE THAN THE CANONICAL REFERENCE. E_{total} IN MJ.

Model	Variant	NET	NMU	NTMU	pass@1 (%)	E_{total}	CPU %	GPU %
GLM-4.6V-Flash	default	1.038	1.003	1.040	25.4	1.930	63.9	36.1
	ctx65536	1.031	1.002	1.033	26.1	1.790	61.5	38.5
	mt1024	1.059	1.002	1.058	9.0	0.790	61.2	38.8
	t0.5	1.028	1.003	1.028	25.4	1.850	61.7	38.3
GLM-4.7-Flash	default	0.984	1.005	0.989	25.0	6.080	59.8	40.2
	ctx65536	0.966	1.004	0.971	25.5	6.110	60.2	39.8
	mt1024	1.007	1.004	1.009	9.3	1.870	60.3	39.7
	t0.5	0.958	1.005	0.965	27.0	6.370	60.1	39.9
Ministral-3B	default	0.946	1.009	0.956	32.7	1.270	65.1	34.8
	ctx65536	0.952	1.009	0.961	31.6	1.110	61.9	38.1
	mt1024	0.944	1.007	0.951	31.6	0.930	62.7	37.3
	t0.5	0.963	1.010	0.974	30.5	1.020	58.1	41.9
Ministral-8B	default	0.949	1.015	0.966	52.6	2.630	64.5	35.5
	ctx65536	1.338	1.033	1.524	52.3	2.600	64.4	35.6
	mt1024	0.947	1.011	0.965	52.1	2.170	63.8	36.1
	t0.5	0.955	1.015	0.974	51.7	2.600	64.5	35.4
Ministral-14B	default	0.954	1.008	0.963	55.3	3.510	63.2	36.8
	ctx65536	0.944	1.008	0.952	56.0	3.040	57.7	42.3
	mt1024	0.950	1.009	0.959	54.8	2.980	62.7	37.3
	t0.5	0.932	1.009	0.940	54.6	3.090	58.7	41.3
Nemotron-Cascade	default	1.050	1.019	1.079	43.2	3.080	62.0	38.0
	ctx65536	1.001	1.014	1.017	43.7	3.010	61.1	38.9
	mt1024	1.048	1.009	1.059	4.8	1.280	61.9	38.1
	t0.5	1.034	1.018	1.056	42.5	3.190	62.3	37.7
Nemotron-Nano	default	1.020	1.019	1.045	41.5	3.260	61.6	38.4
	ctx65536	1.010	1.018	1.037	40.8	3.180	60.5	39.5
	mt1024	2.507	1.088	3.189	12.5	1.500	61.7	38.3
	t0.5	1.169	1.039	1.266	40.4	3.330	62.4	37.6
Qwen3.5-4B	default	0.972	1.010	0.979	51.3	1.380	65.4	34.6
	ctx65536	0.973	1.011	0.979	51.4	1.280	64.0	36.0
	mt1024	0.967	1.011	0.974	51.2	1.330	67.7	32.3
	t0.5	0.970	1.011	0.977	48.3	1.340	65.2	34.8
Qwen3.5-9B	default	1.003	1.011	1.012	66.8	2.260	64.0	35.9
	ctx65536	0.998	1.011	1.008	66.5	2.140	63.8	36.2
	mt1024	1.014	1.011	1.023	65.4	1.930	63.3	36.7
	t0.5	1.000	1.011	1.008	64.5	2.210	62.9	37.1
Qwen3.5-27B	default	0.974	1.005	0.977	81.0	8.580	59.8	40.2
	ctx65536	0.967	1.005	0.970	80.9	8.340	58.8	41.2
	mt1024	0.973	1.005	0.976	79.8	7.430	59.9	40.0
	t0.5	0.974	1.006	0.977	80.2	8.680	60.0	40.0
Qwen3.5-35B-A3B	default	0.971	1.006	0.974	78.6	3.120	63.7	36.3
	ctx65536	0.977	1.006	0.980	78.6	3.060	63.8	36.2
	mt1024	0.973	1.006	0.977	77.5	2.620	62.6	37.4
	t0.5	0.975	1.008	0.981	78.1	2.790	59.9	40.1

produced significantly more energy-expensive code than the worst.

c) *Break-even analysis*: A break-even point N^* exists only when the optimized code consumes less execution energy than the canonical solution on average. Under this criterion, 42 of the 44 configurations admit *no break-even point*: the energy invested in generation is never recovered regardless of how often the resulting code is executed. For the two configurations where a saving exists, the thresholds are prohibitively large. Qwen3.5-9B *t0.5* requires approximately 3.4×10^6 executions per problem to break even, and Nemotron-Cascade *mt1024* requires approximately 6.4×10^7 , thresholds reachable only in exceptional high-volume deployment scenarios. We discuss the broader implications for the energy economics of LLM-based code optimization in Section V.

RQ3: Self-optimized code almost universally consumes more execution energy than the human canonical reference, with only two of 44 configurations achieving a marginal gain. Most configurations also regress relative to their own base epoch. Where a break-even point exists, it requires millions of executions, making LLM-based self-

optimization a net energy loss in all but extreme high-volume deployment scenarios.

D. Carbon Footprint of the Optimization Loop

To translate energy measurements into sustainability-relevant units, we convert inference energy into operational carbon emissions using the global average grid intensity of $\gamma = 217 \text{ gCO}_2\text{eq/kWh}$ from Cheung et al. [35] ($\gamma = 60.3 \mu\text{gCO}_2\text{eq/J}$). This value lies within the 23.92–625.57 kg/MWh range across 365 regional grids tracked by eco2AI [25]. Region-specific estimates can be obtained by scaling our figures by $\gamma_{\text{local}}/217$.

a) *Generation-side footprint*: Table VI reports emissions for the default configuration. Total footprint ranges from 76.8 gCO₂eq (Ministral-3B) to 517.0 gCO₂eq (Qwen3.5-27B) for optimizing 1,000 problems.

b) *Execution-side footprint and break-even in carbon units*: Execution-side emissions are three orders of magnitude smaller than generation. For the two configurations where the optimized solution undercuts the canonical baseline (Section IV-C), the per-execution savings in carbon terms are negligible: Qwen3.5-9B *t0.5* saves $\sim 39 \mu\text{gCO}_2\text{eq}$ per problem

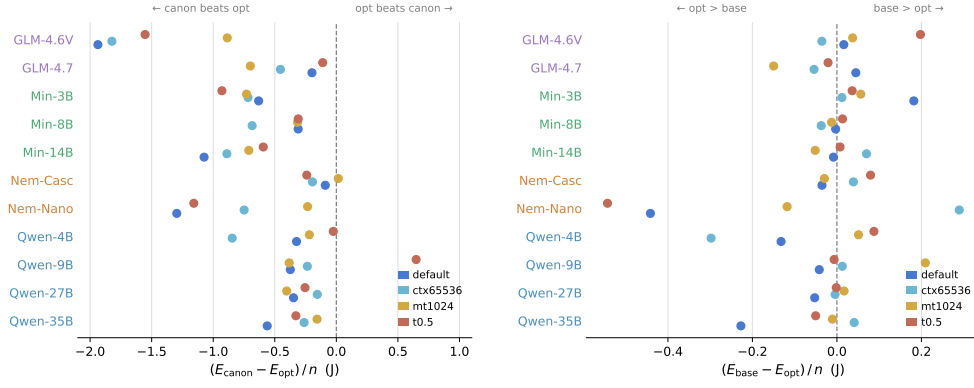


Fig. 3. *Left:* $(E_{\text{canon}} - E_{\text{opt}})/n$; points left of zero beat the canonical baseline. *Right:* $(E_{\text{base}} - E_{\text{opt}})/n$; positive values indicate improvement over the base epoch.

TABLE VI
CARBON FOOTPRINT OF THE FIVE-EPOCH OPTIMIZATION LOOP (DEFAULT VARIANT), COMPUTED USING $\gamma = 217 \text{ gCO}_2\text{EQ/kWh}$ [35].

Model	E_{total} (MJ)	gCO_2eq
GLM-4.6V-Flash	1.926	116.1
GLM-4.7-Flash	6.077	366.3
Ministral-3B	1.273	76.8
Ministral-8B	2.625	158.2
Ministral-14B	3.512	211.7
Nemotron-Cascade	3.085	186.0
Nemotron-Nano	3.262	196.6
Qwen3.5-4B	1.382	83.3
Qwen3.5-9B	2.259	136.2
Qwen3.5-27B	8.577	517.0
Qwen3.5-35B-A3B	3.125	188.3

per execution, against a generation investment of $133 \text{ gCO}_2\text{eq}$, yielding $N^* \approx 3.4 \times 10^6$ executions, which is identical to the energy-denominated result since the conversion is linear in γ .

V. DISCUSSION

We now interpret the results across the three research questions and their broader implications for practitioners considering LLM-based code optimization.

A. Generation energy as a hidden cost of optimization

A central finding of this study is that the energy consumed during LLM-based self-optimization is substantial and largely unrecoverable. Across 42 of the 44 evaluated model-variant configurations, the optimized code consumes more execution energy per problem than the canonical human-written reference, and the generation investment requires millions of executions to break even in the two cases where a saving exists. Our results suggest that performance optimization and energy efficiency are not aligned goals: faster code does not automatically mean greener code when the cost of generation is accounted for. Coignon et al. [9] reached a similar conclusion on a different benchmark and pipeline, reporting weak correlation between performance gains and energy reduction.

B. Model and architecture choice matter more than decoding parameters

Our RQ2 analysis shows that the dominant factor in both inference energy and correctness-efficiency is the choice of model and its architecture, not decoding configuration. Constraining output length (*mt1024*) consistently reduces inference energy across all models, but at the cost of correctness collapse in several configurations. The MoE architecture of Qwen3.5-35B-A3B achieves near-identical correctness to the larger Qwen3.5-27B dense model at less than half the inference energy, owing to sparse parameter activation, suggesting that architectural choices made upstream of deployment have a larger sustainability impact than parameter tuning.

C. Self-optimization does not reliably reduce execution energy across epochs

A subtler finding from RQ3 is that the EffiLearner pipeline does not drive consistent execution-energy reductions even relative to its own base epoch, with 54.5% of configurations regressing across the optimization trajectory. Future pipelines should incorporate execution energy as an explicit selection criterion alongside correctness.

D. Practical implications

LLM-based self-optimization is only energetically justified for code that will be executed at very high volume. For one-off scripts or lightly used utilities, the generation cost will never be recouped. Practitioners aiming to reduce overall energy consumption should restrict optimization to hot-path code with known high execution frequency and prefer smaller models such as Ministral-3B or Qwen3.5-4B when correctness requirements allow. The carbon footprint analysis (Section IV-D) confirms that these differences are non-trivial at scale: the generation footprint of Qwen3.5-27B is approximately seven times that of Ministral-3B for the same optimization task.

VI. LIMITATIONS AND FUTURE WORK

The findings reported in this study are bounded by deliberate experimental choices, each pointing to a concrete direction for future work.

a) *Benchmark scope*: EffiBench problems are short, algorithmically dense, and complete in under a second, which is ideal for energy measurement at scale but unrepresentative of long-running workloads where break-even would most plausibly favor optimization. Future work should investigate whether the pass@1 collapse observed under aggressive decoding constraints persists on harder, more diverse tasks such as I/O-bound programs or data-processing pipelines.

b) *Human baseline*: The canonical EffiBench solutions are top-rated submissions from a competitive programming forum, so they represent highly efficient human implementations rather than the code an average practitioner would write. Our RQ3 comparison therefore measures LLM-optimized code against a strong reference. We also assign no authoring cost to the canonical solutions, whereas the generation cost of the LLM is fully counted. This omission does not affect our main finding. In 42 of 44 configurations the optimized code consumes more execution energy per run than the canonical code, so the recurring difference is negative and no authoring cost for the baseline can turn the generation energy into a net saving through repeated execution. It could only favor LLM-based optimization when a solution is executed few times. Future work should compare against solutions from a broader pool of practitioners and measure human authoring effort explicitly.

c) *Pipeline design*: The EffiLearner pipeline replaces solutions with no gate on execution energy, which explains a significant portion of the regressions we observed and makes it difficult to attribute results solely to model capability. Future pipelines should incorporate execution energy as an explicit selection criterion to isolate its effect.

d) *Language and platform coverage*: All experiments use Python, whose interpreted execution model shapes the energy profiles we report. The execution-energy gap between LLM-generated and canonical solutions may narrow in compiled languages such as C++ or Rust [6], and a multi-language replication would establish whether our patterns generalize.

e) *Closed-source models*: Our open-source-only model selection is required by the need for hardware-level energy attribution, so our findings do not extend to frontier models served through commercial APIs. Closing this gap will require provider-published per-request energy disclosures or instrumented inference stacks.

f) *Broader environmental impact*: Our measurements cover operational energy only. Embodied costs of GPU manufacturing, data-center cooling, model training, and hardware refresh cycles are not captured yet can outweigh operational savings over the hardware lifecycle. Future work should adopt a lifecycle perspective accounting for these indirect impacts.

VII. CONCLUSION

This paper presented a systematic measurement study of the energy economics of LLM-based code self-optimization, covering inference and execution cost across 11 open-source models, four decoding configurations, and 1,000 EffiBench

problems. The central question was whether performance-optimized code produced by LLM self-refinement is also energy-efficient relative to canonical solutions.

Optimized code consumes more execution energy than the canonical reference in 42 of 44 model-variant pairs; the two exceptions achieve only marginal gains, with break-even points requiring millions of executions. Performance optimization and energy efficiency are not aligned even within the refinement trajectory: more than half of all configurations regress in execution energy relative to their base epoch, reflecting the absence of any energy-aware selection criterion in the pipeline.

On the generation side, inference energy varies by nearly an order of magnitude across models, driven primarily by model size and architecture. MoE models such as Qwen3.5-35B-A3B offer substantially lower inference cost than equivalently-sized dense counterparts without sacrificing correctness, making architectural choice a more consequential sustainability lever than decoding parameters. Constraining output tokens (*mt1024*) is the most effective parameter adjustment for reducing inference energy, but risks correctness collapse that must be weighed per model.

Practitioners should restrict LLM-based self-optimization to hot-path code with reliably high execution frequency, prefer small and MoE-based models when correctness allows, and treat generation energy as a first-class cost. Future work should extend this framework to compiled languages, long-running workloads, and energy-aware refinement pipelines.

ACKNOWLEDGMENT

Funding. This research was supported by ITEA GreenCode, ITEA Genius and the Federal Ministry of Research, Technology and Space of Germany under grant number 16IS24070G. **Transparency note.** The readability of this text has been improved with the support of AI technologies, which were also used in the code generation process. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

REFERENCES

- [1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.
- [2] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.
- [3] 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," *Advances in Neural Information Processing Systems*, vol. 37, pp. 84 482–84 522, 2024.
- [4] S. Luccioni, Y. Jernite, and E. Strubell, "Power hungry processing: Watts driving the cost of AI deployment?" in *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, 2024, pp. 85–99.
- [5] 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)*. IEEE, 2023, pp. 1–9.

- [6] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "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*, 2017, pp. 256–267.
- [7] M. A. Islam, D. V. Jonnala, R. Rekhi, P. Pokharel, S. Cilamkoti, A. Imran, T. Kosar, and B. Turkkan, "Evaluating the energy-efficiency of the code generated by LLMs," *arXiv preprint arXiv:2505.20324*, 2025.
- [8] L. Solovyeva, S. Weidmann, and F. Castor, "AI-powered, but power-hungry? energy efficiency of LLM-generated code," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 2025, pp. 49–60.
- [9] T. Coignon, C. Quinton, and R. Rouvoy, "When faster isn't greener: The hidden costs of LLM-based code optimization," in *2025 International Conference on Automated Software Engineering*, 2025.
- [10] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "Effibench: Benchmarking the efficiency of automatically generated code," *Advances in Neural Information Processing Systems*, vol. 37, pp. 11 506–11 544, 2024.
- [11] N. Jegham, M. Abdelatti, C. Y. Koh, L. Elmoubarki, and A. Hendawi, "How hungry is ai? benchmarking energy, water, and carbon footprint of LLM inference," *arXiv preprint arXiv:2505.09598*, 2025.
- [12] 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.
- [13] S. Poddar, P. Koley, J. Misra, N. Ganguly, and S. Ghosh, "Towards sustainable NLP: Insights from benchmarking inference energy in large language models," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 12 688–12 704.
- [14] M. Adamska, D. Smirnova, H. Nasiri, Z. Yu, and P. Garraghan, "Green prompting: Characterizing prompt-driven energy costs of LLM inference," 2026. [Online]. Available: <https://arxiv.org/abs/2503.10666>
- [15] S. Georgiou, M. Kechagia, T. Sharma, F. Sarro, and Y. Zou, "Green AI: Do deep learning frameworks have different costs?" in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1082–1094.
- [16] R. Rubei, A. Moussaid, C. Di Sipio, and D. Di Ruscio, "Prompt engineering and its implications on the energy consumption of large language models," *arXiv preprint arXiv:2501.05899*, 2025.
- [17] H. Ashraf, S. M. Danish, S. Rahman, and Z. Sattar, "Toward green code: Prompting small language models for energy-efficient code generation," *arXiv preprint arXiv:2509.09947*, 2025.
- [18] S. Ilager, L. F. Briem, and I. Brandic, "Green-code: Learning to optimize energy efficiency in LLM-based code generation," in *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2025, pp. 559–569.
- [19] J. Shi, Z. Yang, H. J. Kang, B. Xu, J. He, and D. Lo, "Greening large language models of code," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 2024, pp. 142–153.
- [20] 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)*. IEEE, 2024, pp. 39–44.
- [21] R. Apsan, V. Stoico, M. Albonico, R. Dhar, K. Vaidhyanathan, and I. Malavolta, "Generating energy-efficient code via large-language models—where are we now?" *arXiv preprint arXiv:2509.10099*, 2025.
- [22] M. T. Dearing, Y. Tao, X. Wu, Z. Lan, and V. Taylor, "Leveraging LLMs to automate energy-aware refactoring of parallel scientific codes," *arXiv preprint arXiv:2505.02184*, 2025.
- [23] P. Rani, J.-A. Bard, J. Sallou, A. Boll, T. Kehrler, and A. Bacchelli, "Can we make code green? understanding trade-offs in LLMs vs. human code optimizations," *arXiv preprint arXiv:2503.20126*, 2025.
- [24] R. Verdecchia, J. Sallou, and L. Cruz, "A systematic review of green AI," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 13, no. 4, p. e1507, 2023.
- [25] S. A. Budennyy *et al.*, "Eco2AI: carbon emissions tracking of machine learning models as the first step towards sustainable ai," in *Doklady mathematics*, vol. 106, no. Suppl 1. Springer, 2022, pp. S118–S128.
- [26] A. De Vries, "The growing energy footprint of artificial intelligence," *Joule*, vol. 7, no. 10, pp. 2191–2194, 2023.
- [27] A. Yang *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.
- [28] Team GLM *et al.*, "ChatGLM: A family of large language models from GLM-130b to GLM-4 all tools," *arXiv preprint arXiv:2406.12793*, 2024.
- [29] A. H. Liu *et al.*, "Minstral 3," *arXiv preprint arXiv:2601.08584*, 2026.
- [30] B. Wang *et al.*, "Nemotron-cascade: Scaling cascaded reinforcement learning for general-purpose reasoning models," *arXiv preprint arXiv:2512.13607*, 2025.
- [31] A. Blakeman *et al.*, "Nemotron 3 nano: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning," *arXiv preprint arXiv:2512.20848*, 2025.
- [32] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th symposium on operating systems principles*, 2023, pp. 611–626.
- [33] B. Weigell, S. Hornung, and B. Bauer, "Mettrion: A framework for accurate software energy measurement," *arXiv preprint arXiv:2512.06806*, 2025.
- [34] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.
- [35] K. S. Cheung, M. Kaul, G. Jahangirova, M. R. Mousavi, and E. Zie, "Comparative analysis of carbon footprint in manual vs. LLM-assisted code development," in *Proceedings of the 1st International Workshop on Responsible Software Engineering*, 2025, pp. 13–20.