

Smells Like Prompt Spirit: Evaluating Energy Efficiency and Performance of Local LLM Inference on CPU vs GPU

Simão Cunha

`simao.s.cunha@inesctec.pt`

HASLab/INESC TEC

Francisco Ribeiro

New York University Abu Dhabi

Luís Cruz

Delft University of Technology

João Saraiva

HASLab/INESC TEC

Research Article

Keywords: Large Language Models, Energy Consumption, GPU Inference, Few-Shot Prompting, Coding Tasks, Green AI

Posted Date: October 14th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10374549/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Smells Like Prompt Spirit: Evaluating Energy Efficiency and Performance of Local LLM Inference on CPU vs GPU

Simão Cunha^{1*}, Francisco Ribeiro^{2†}, Luís Cruz^{3†}, João Saraiva^{1†}

^{1*}INESC TEC & University of Minho, Braga, Portugal.

²New York University Abu Dhabi, Abu Dhabi, United Arab Emirates.

³Delft University of Technology, Delft, The Netherlands.

*Corresponding author(s). E-mail(s): simao.s.cunha@inesctec.pt;

Contributing authors: francisco.ribeiro@nyu.edu; L.Cruz@tudelft.nl; saraiva@di.uminho.pt;

[†]These authors contributed equally to this work.

Abstract

The integration of Large Language Models (LLMs) into daily software development workflows has fundamentally turned attention toward the sustainability of modern AI infrastructures. This paper presents an empirical study extending previous work by evaluating five local LLMs on a dedicated GPU cluster environment using the HumanEval-X and MBPP+ benchmarks. We analyze how changing the hardware environment interacts with prompting setups (0-shot vs 3-shot) to impact energy consumption, inference time, functional correctness and syntactic similarity. Although the GPU cluster cuts execution runtime by over 99.8% and reduces energy consumption by up to 89.4%, the results show that changing the hardware environment directly changes code generation, which can compromise output quality. Due to hardware-level nondeterminism, changing the computational backend causes model outputs to diverge, leading to unpredictable performance drops and structural code variations in 0-shot scenarios. In contrast, 3-shot configurations successfully suppress conversational noise, delivering greater operational stability and uniform energy-time savings across nearly all five local models. These findings show that infrastructure transitions merely boost execution efficiency, reinforcing that prompt structure and model selection must be designed together for reliable and green code generation.

Keywords: Large Language Models, Energy Consumption, GPU Inference, Few-Shot Prompting, Coding Tasks, Green AI

1 Introduction

Large Language Models, such as GPT-4 [1], are transforming software development. Initially created for natural language processing [2, 3], LLMs now operate as AI-driven coding assistants, supporting code generation [4], debugging [5] and optimization [6]. Tools like ChatGPT [7], GitHub Copilot [8], specialized open-weight architectures such as DeepSeek-Coder [9], and general-purpose model families like Llama 4 [10], illustrate their growing role in software engineering, assisting in many tasks such as code completion [11], test generation [12] and program repair [13].

The widespread use of AI coding assistants highlights their growing importance in software development. The 2025 Stack Overflow Developer Survey [14] indicates that over 84% of developers have actively integrated AI tools into their workflows, with the vast majority relying on them daily for both professional tasks and personal projects. This reflects a noticeable increase from the 76% adoption rate reported in the previous year. However, generating responses with these models requires substantial energy, and this continuous usage drives up both computational costs and carbon footprints [15]. It is therefore essential to shift focus towards *Green AI* paradigms [16], which advocate that energy consumption and processing time should be evaluated as primary metrics alongside traditional performance accuracy.

This expanding energy footprint is already visible in industry-wide environmental impact reports. For instance, Microsoft reported 20 million metric tons of carbon dioxide equivalent in 2025 (a 25% increase from the 16 million tons recorded in 2024) driven by the company’s rapid investment in AI infrastructure [17]. This trajectory aligns with broader global trends: the International Energy Agency projects that data center electricity emissions will nearly double by 2035, climbing from 180 million to 300 million metric tons [18].

Beyond sustainability, privacy and security remain primary drivers for local deployment [19, 20]. However, the landscape is also changing due to an unprecedented quantitative explosion in model availability.

By March 2026, an official report from Hugging Face showed that their hub had scaled to over 2 million public models, with specialized families like Qwen alone generating more than 113k derivative variants - such as fine-tuned models and specialized adapters [21]. This vast and highly fragmented diversity makes model selection increasingly difficult.

Prompting strategies such as *zero-shot* and *few-shot* prompting, where models receive no examples or only a few examples to guide their responses, directly influence output quality [22–27]. While the impact of n-shot prompting on hardware resource utilization and energy efficiency has begun to be characterized in high-performance or cloud-based setups, demonstrating that prompt design can influence inference energy consumption [28, 29], less focus has been placed on how these prompting configurations interact with CPU and GPU hardware environments to influence both the energy footprint and the functional reproducibility of local LLMs.

To address these combined challenges, this study presents a systematic empirical evaluation of five local LLMs across both CPU and GPU hardware environments. Evaluating their performance under different n-shot prompting setups allows us to assess how prompt configurations and hardware architectures interact to affect execution

efficiency, functional correctness and output consistency. Crucially, our cross-hardware analysis reveals that shifting execution from CPUs to GPUs goes beyond mere execution speedups. It introduces a behavioral variance where the exact same model, under identical prompting configurations and LLM parameters (such as *temperature*, *top_p* and *seed*), can diverge and produce structurally different code solutions due to hardware-level nondeterminism [30, 31].

This paper stands as an extended version of previous work [32]. In that initial study, we established a CPU baseline for local LLMs, yielding two main findings: (1) the significant variability in energy consumption and execution time across different local LLMs when run on CPU-only setups, highlighting the importance of model selection for energy-efficient deployment; and (2) the positive impact of few-shot prompting in improving both energy and time efficiency without compromising functional correctness (*i.e.*, test pass rates). Building directly upon those insights, this extended work evaluates the same five widely used local LLMs on the same two standard benchmarks for code generation and code completion, significantly broadening the scope to focus on the following key aspects: (1) energy efficiency and runtime performance, contrasting the CPU baseline against a dedicated GPU inference environment; (2) code generation effectiveness, evaluating how shifting the computing backend impacts syntactic alignment and functional correctness under different hardware architectures.

Thus, this work addresses the following research questions:

RQ1: How do local LLMs behave in terms of energy consumption and execution time when running under a GPU inference environment?

To answer this question, the empirical evaluation focuses on characterizing the energy consumption (measured in Joules) and execution time (in seconds) of five widely deployed quantized local LLMs: *Meta-Llama-3-8B-Instruct.Q6_K* [33], *deepseek-coder-6.7b-instruct.Q5_K_M* [34], *codellama-7b-instruct.Q5_K_M* [35], *codegeex4-all-9b.Q6_K_L* [36] and *starling-lm-7b-alpha.Q5_K_S* [37]. Energy consumption and runtime data are collected during inference on the HumanEval-X [38] and MBPP+ [39] benchmarks.

RQ2: How do local LLMs perform in terms of syntactic metrics (BLEU variants) and functional correctness (pass@k) under GPU inference?

To address this question, three syntactic metrics (CodeBLEU, SacreBLEU and GoogleBLEU) and two functional correctness metrics (*pass@1* and *pass@10*) are calculated for the code generated on the GPU. These metrics are analyzed under both 0-shot and 3-shot prompting setups to assess how n-shot prompting influence output quality.

RQ3: To what extent does transitioning from CPU to GPU inference impact the overall energy efficiency, inference time, output consistency, and functional reproducibility of local LLMs?

To address this final question, a cross-hardware comparative analysis directly contrasts the new GPU findings against the original CPU baseline from our original work [32]. This comparison examines whether changing the computational backend compromises code stability, tracking how output solutions diverge despite identical generation settings.

Our findings reveal a clear trade-off between hardware efficiency and code quality. While moving from the CPU to the GPU cluster cuts execution time by over 99.8% and reduces energy consumption by up to 89.4%, this hardware transition - as theoretically expected due to architectural differences - directly impacts the generation process, which can ultimately compromise output quality. In fact, the GPU environment causes unpredictable performance drops in 0-shot scenarios, whereas 3-shot prompting helps keep the code structure stable and ensures steady energy savings across nearly all models.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 describes our experimental methodology, including the selected models, benchmarks, energy measurement framework and cluster setup. Section 4 presents the empirical results and answers the proposed research questions. Section 5 addresses threats to validity. Finally, Section 6 summarizes our conclusions and outlines future work.

2 Related Work

Several approaches have targeted energy efficiency in software. These range from code-level techniques, such as removing obsolete code [40], to managing software lifecycles to avoid energy-intensive processes [41], optimizing hardware utilization [42], and, in distributed settings, improving data-transfer protocols [43]. All these efforts align with the Green AI agenda of minimizing computational carbon footprints [44]. Moreover, research shows that programming language choice significantly affects energy consumption [45]. Tools like Intel RAPL [46] and CodeCarbon [47] enable real-time energy analysis, allowing developers to make informed decisions during software optimization.

More recently, LLMs have been widely applied in software development, with specialized open-weight architectures such as DeepSeek-Coder [9] and the Llama 4 family [10] accelerating coding tasks. Other efforts, such as DeepDev-PERF [48] and Mentat [49], focus on performance enhancement and debugging. Additionally, multilingual approaches to program repair have been demonstrated by tools like AlphaRepair, which addresses software defects across various languages [50]. LLMs trained on large datasets support code generation, debugging, and optimization [51], with fine-tuning and prompt engineering enhancing performance [52, 53].

Structured benchmarks like HumanEval-X and MBPP+ provide a comprehensive framework to assess the quality and efficiency of LLM-generated code [4, 39, 54–56].

Several researchers have investigated key aspects of LLMs for Code (LLM4Code). A significant body of work has focused on the energy consumption of these models, particularly during the inference phase for code generation tasks [15, 57–59], as well as during the training phase [60–62].

While our previous work focused only on a local CPU baseline [32], running LLMs solely on CPUs is not the only local alternative. New benchmarks are evaluating how these models perform across different hardware setups and AI accelerators, measuring speed, latency and energy [63, 64]. However, these studies focus mostly on general text

tasks and system metrics. They do not look at how changing the hardware affects code-specific metrics or functional correctness under different prompting styles, nor do they account for the hardware-level nondeterminism that alters model outputs even when using the exact same hyperparameters, random seeds and sampling configurations [30, 31]. This paper bridges this gap by extending our previous baseline to a GPU setup, directly comparing how the new hardware environment impacts both the execution costs and the quality of the generated code.

3 Methodology

This section details the experimental setup, tools and evaluation infrastructure used in this study. The following subsections describe the selected LLMs (Section 3.1), the energy measurement pipeline (Section 3.2) and the evaluation benchmarks (Section 3.3). Next, the overall experimental setup (Section 3.4) is presented, detailing the system architecture (Section 3.4.1), the execution parameters (Section 3.4.2) and the statistical tests used to validate the results (Section 3.5). Finally, the hardware specifications of the cluster (Section 3.6) are described.

3.1 LLMs Under Analysis

This study explores running LLMs locally, inheriting the baseline constraints of a common CPU-only setup with 32 GB of RAM from the initial work [32]. To ensure a direct and correct cross-hardware comparison, this evaluation reuses exactly the same five models. Models that run efficiently under these hardware constraints were prioritized, focusing on quantized GGUF models, which reduce resource usage while maintaining reasonable performance [65]. To ensure diversity, models were selected from various families, avoiding bias toward any single architecture or training method.

Since the focus is on code generation, the selection targeted models that are either specialized or fine-tuned for programming tasks and have been widely evaluated on public code-related benchmarks [9, 38, 66–71]. These include specialized code-generation families like DeepSeek-Coder, CodeLlama and CodeGeeX. Additionally, general-purpose models outside the strict LLM4Code scope, such as Llama 3 and Starling-LM, were included to assess their performance in code generation tasks.

All selected models are open-weight or, at a minimum, publicly available and runnable locally, ensuring transparency, reproducibility, and practical feasibility for offline experimentation.

The specific models evaluated in this study, alongside their architectural parameters, precise quantized GGUF variants and short-form identifiers used throughout this paper, are summarized in Table 1.

3.2 Energy Measurements

Hardware energy consumption is tracked using CodeCarbon version 2.4.2, running continuously during the inference cycle. Reusing this specific version ensures absolute methodological consistency with the previous CPU baseline study [32]. While the original CPU baseline relied on Intel RAPL energy estimation [72–75], the GPU

Table 1: Evaluated models and GGUF variants

Model Full Name	Short Name	Params.	Quantized GGUF Variant
Meta-Llama-3-8B-Instruct	<i>Meta-Llama-3</i>	8B	Meta-Llama-3-8B-Instruct.Q6_K [33]
DeepSeek-Coder-6.7B-Instruct	<i>deepseek-coder</i>	6.7B	deepseek-coder-6.7b-instruct.Q5_K_M [34]
CodeLlama-7B-Instruct	<i>codellama</i>	7B	codellama-7b-instruct.Q5_K_M [35]
CodeGeeX4-All-9B	<i>codegeex4</i>	9B	codegeex4-all-9b.Q6_K_L [36]
Starling-LM-7B-Alpha	<i>starling-lm</i>	7B	starling-lm-7b-alpha.Q5_K_S [37]

evaluation environment configures the framework to query the NVIDIA Management Library (NVML) API via the `nvidia-ml-py` library.

3.3 LLM Benchmarks

To evaluate code generation and completion capabilities, this study reuses exactly the same two established benchmarks from the previous CPU baseline study [32]: HumanEval-X [76], which assesses code completion (completing a function body based on a given signature and docstring) in C++, Go, Java, JavaScript and Python [77], and MBPP+ [78], which evaluates Python code generation (writing a functional program from scratch based on a natural language prompt) [39]. Using the exact same datasets under identical testing conditions ensures a reliable cross-hardware comparison. These benchmarks remain widely established baselines in the literature for testing functional correctness through code execution [4, 79].

As in the previous work [32], functional performance is assessed via the unbiased *pass@k* estimator [4], computed for $k \in \{1, 10\}$ based on whether the outputs successfully pass the benchmarks’ predefined unit test suites. However, this study expands the analytical scope by also evaluating the syntactic and structural alignment of the generated code - a dimension left unexplored in the prior CPU analysis. To address this new layer of evaluation, three distinct syntactic metrics are applied. SacreBLEU is integrated using version 2.4.0 via the *sacrebleu* Python package, while CodeBLEU [80] and GoogleBLEU [81] are executed directly from the pre-configured Docker container environments provided in the HumanEval-X repository [76]. Although built for that repository, this container pipeline was also mapped to process the MBPP+ outputs without modification.

3.4 Experimental Setup

To preserve exact experimental alignment and guarantee a fair comparison with the previous CPU baseline study [32], all models were deployed locally using *LLaMa.cpp* [82] keeping identical execution arguments. Specifically, the environment used the *llama-cpp-python* bindings compiled with explicit CUDA support (CUDA version 12.4) to ensure full hardware acceleration and model offloading onto the GPU infrastructure. Inference parameters were kept constant across all runs: $max_tokens = 512$, $n_ctx = 4096$, $seed = 42$, $top_p = 0.95$ and $temperature = 0.6$ [66].

3.4.1 System Architecture

To ensure a modular and reproducible setup, the system architecture follows the exact design of the baseline study [32], as shown in Figure 1. Reusing the same pipeline ensures that any performance or energy differences found are strictly due to the hardware transition (CPU to GPU) rather than changes in the execution software.

The framework consists of a command-line interface (CLI) tool developed in Python to analyze the energy consumption and runtime of LLMs while generating code for the benchmarks described in Section 3.3. The tool also records the metrics obtained from each generation.

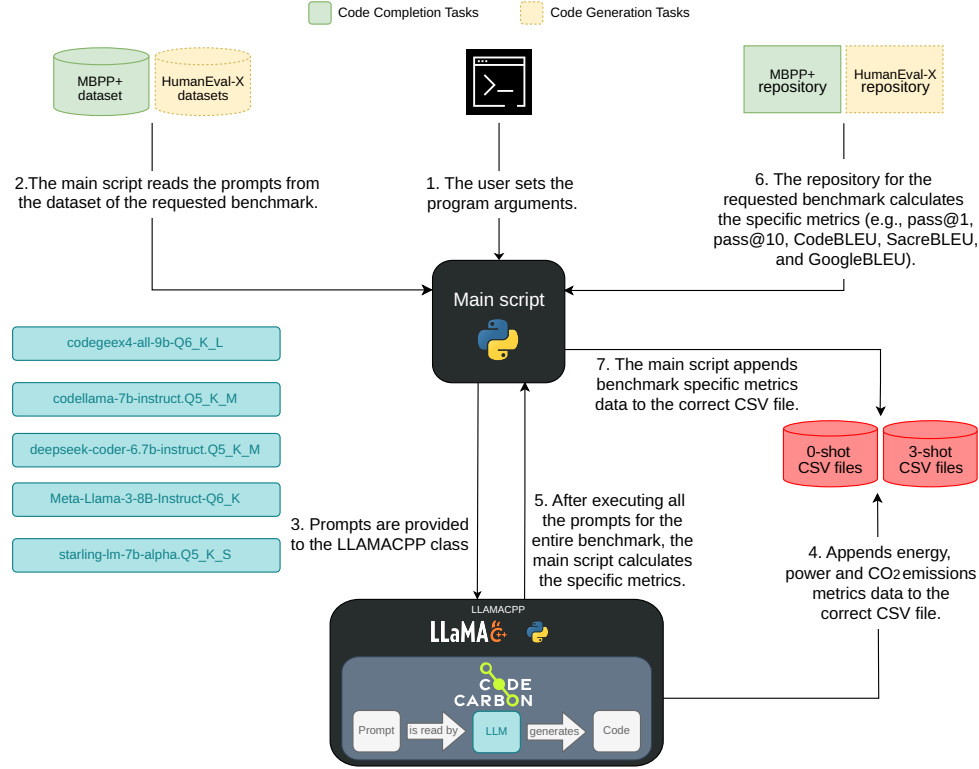


Fig. 1: Architecture for Execution and Measurement

3.4.2 Benchmark Execution

Following the setup from the original study [32], every prompt is executed 10 times consecutively per model to reduce hardware power fluctuations, adopting the empirical guidelines proposed in [83]. For energy and runtime profiling, each run acts as an individual data point, and anomalous data is filtered out using the Interquartile Range (IQR) method per prompt-language group. For the functional *pass@k* metrics, the 10

generations are grouped into a single pool to calculate the final success rate. Few-shot evaluation ($n = 3$) uses the first three tasks of each language’s dataset as fixed context examples to evaluate the remaining prompts. These three prompts are then excluded from the actual evaluation to avoid overlap. This ensures consistency in the examples used across all evaluations.

3.5 Statistical Analysis

To validate the observed differences in execution time and energy efficiency, the statistical analysis is structured around two main operational families, hereby defined as Dimension 1 (Dim. 1) and Dimension 2 (Dim. 2). First, the intra-hardware analysis (Dim. 1) evaluates the impact of few-shot prompting (0-shot vs 3-shot) strictly within the GPU environment. Second, the inter-hardware analysis (Dim. 2) measures the effect of the infrastructure transition by comparing the new GPU metrics against the baseline CPU distributions established in the prior study [32].

For each target comparison, the hypothesis framework is defined as follows:

- H_0 : There is no statistically significant difference between the two compared distributions.
- H_1 : There is a statistically significant difference between the two compared distributions.

The statistical pipeline follows a two-stage process. The Shapiro-Wilk test was first applied to assess the normality of the data samples. Since all distributions significantly departed from normality ($p < 0.05$), the non-parametric Mann-Whitney U test was used for all subsequent comparisons.

Finally, we address the risk of Type I errors (false positives). When performing a high volume of cross-comparisons, the probability of incorrectly rejecting a true null hypothesis purely by chance increases drastically. To prevent these false positives, the Holm-Bonferroni correction was applied independently for each metric within its respective analytical family. All matrices containing the raw test statistics and adjusted p -values generated by this pipeline are detailed in the online appendix (provided in the Data Availability Statement).

In total, this significance pipeline executed 180 independent hypothesis tests, fully stratified as follows:

- **HumanEval-X (150 tests total):**
 - *Dimension 1 (50 tests)*: 2 metrics (Energy and Time) evaluated across 25 combinations (5 LLMs $\times$ 5 programming languages) comparing 0-shot vs 3-shot setups.
 - *Dimension 2 (100 tests)*: 2 metrics evaluated across the 25 core combinations under both 0-shot and 3-shot prompting strategies, tracking the architectural transition from CPU to GPU hardware.
- **MBPP+ (30 tests total):**
 - *Dimension 1 (10 tests)*: 2 metrics evaluated across 5 configurations (5 LLMs $\times$ 1 programming language) comparing 0-shot vs 3-shot setups.

- *Dimension 2 (20 tests)*: 2 metrics evaluated across the 5 configurations under both 0-shot and 3-shot settings, measuring the infrastructure transition from CPU to GPU.

3.6 Experimental Equipment

The experiments were performed using a GPU-accelerated partition of the Deucalion supercomputer¹, running Rocky Linux 8 as the operating system. The evaluation was conducted on a single dedicated compute node featuring two 64-core AMD EPYC 7742 processors operating at 2.25 GHz, equipped with four NVIDIA Ampere A100 GPUs (40 GB VRAM each) and 512 GB of DDR4 RAM. Local storage is provided by a Micron 5300 480GB SATA SSD, with network connectivity handled by two NVIDIA ConnectX-6 200 Gb/s interfaces. To eliminate interference and ensure consistent performance measurements, all evaluations were executed on the compute node with no other active user workloads running on the same hardware during the experiments.

4 Results and Analysis

The entire GPU evaluation pipeline required a total execution time of 3 days and 11 hours, consuming an aggregate of 45 megajoules (MJ) of GPU energy. The experimental data reveals a strong, near-perfect positive Pearson correlation between execution time and GPU energy consumption, with correlation coefficients consistently falling between 0.98 and 0.99 across all tasks.

Before diving into specific research questions, all datasets - including the baseline metrics from the previous CPU study [32] - were subjected to the statistical testing pipeline outlined in Section 3.5. As detailed in the online appendix, the Shapiro-Wilk test confirmed that 100% of the collected runtime and energy distributions across both the HumanEval-X and MBPP+ benchmarks deviated significantly from normality ($p < 0.05$), fully justifying the use of non-parametric analysis.

Subsequently, the Mann-Whitney U tests, corrected via the Holm-Bonferroni method, led to a universal rejection of the null hypothesis (H_0) across all evaluated tasks, programming languages and models. This mathematical validation shows that both the intra-hardware prompting variations (0-shot vs 3-shot) and the inter-hardware architectural transition (CPU vs GPU) induce highly significant shifts in efficiency and performance.

The remainder of this section is organized as follows: Section 4.1 evaluates the intra-hardware resource efficiency metrics under few-shot prompting (RQ1); Section 4.2 tracks the resulting functional and syntactic quality shifts (RQ2); Section 4.3 details the cross-hardware infrastructure trade-offs by comparing CPU and GPU implementations (RQ3); and Section 4.4 synthesizes these dimensions into a consolidated energy-based ranking of the evaluated LLMs.

¹<https://docs.deucalion.macc.fcn.pt/deucalion/>

4.1 Energy Consumption and Execution Time in the GPU Environment

To answer **RQ1: How do local LLMs behave in terms of energy consumption and execution time when running under a GPU inference environment?**, this subsection evaluates the intra-hardware resource efficiency of the selected LLMs strictly within the accelerated GPU computing environment. The operational impact of the prompting strategy (0-shot vs 3-shot) on energy consumption and execution time is analyzed across the multi-lingual tasks of HumanEval-X (Section 4.1.1) and the Python-centric generation parameters of MBPP+ (Section 4.1.2).

4.1.1 Code Completion: HumanEval-X

The experimental data reveals a universal trend regarding the prompting strategy: switching from 0-shot to 3-shot systematically cuts down both execution time and energy consumption across every single evaluated model and programming language (as shown in Figures 2 and 3). This efficiency gain happens because providing in-context examples guides the LLM to generate code directly, avoiding the verbose text, markdown wrappers or filler explanations that often occur in a 0-shot setup. As empirically demonstrated in Figure 4, transitioning to a 3-shot prompting forces a significant drop in the volume of generated output tokens across all evaluated models and languages. For instance, *deepseek-coder* reduces its mean generated output token count from 315 to 86 in C++ and from 356 to 95 in Java, directly aligning with the overall reduction in energy consumption and execution time. By stripping away this verbose conversational output, the model requires less computational effort from the GPU, thereby minimizing resource expenditure. A similar behavioral impact on reducing computational waste was also observed in our previous study on CPU-only environments [32].

When comparing overall performance under the 0-shot prompting, *codellama* is the most efficient and fastest baseline across all benchmarks, with energy consumption remaining below 500 J (ranging from 161 J in Python to 461 J in Go) and execution times between 0.91 s and 2.57 s. Conversely, *starling-lm* and *Meta-Llama-3* consistently demand the most resources in this baseline setup, with energy costs frequently scaling between 700 J and 850 J, and execution times exceeding 4 s across nearly all languages. Looking at individual programming languages, *codellama* establishes absolute dominance, achieving the lowest energy consumption and shortest execution times for C++, Go, Java, JavaScript and Python in 0-shot settings. Under 3-shot prompting, *codellama* loses its monopoly on efficiency. While it remains the most efficient model for C++ and Go, it is surpassed by *deepseek-coder* in Java (172 J vs *codellama*’s 175 J) and JavaScript (138 J vs *codellama*’s 145 J), while both models achieve a virtual tie in Python at 102 J.

4.1.2 Code Generation: MBPP+

The experimental results for the MBPP+ benchmark, presented in Figures 5 and 6, confirm that the positive impact of 3-shot prompting remains highly consistent during code generation tasks. Across all evaluated model families, providing in-context

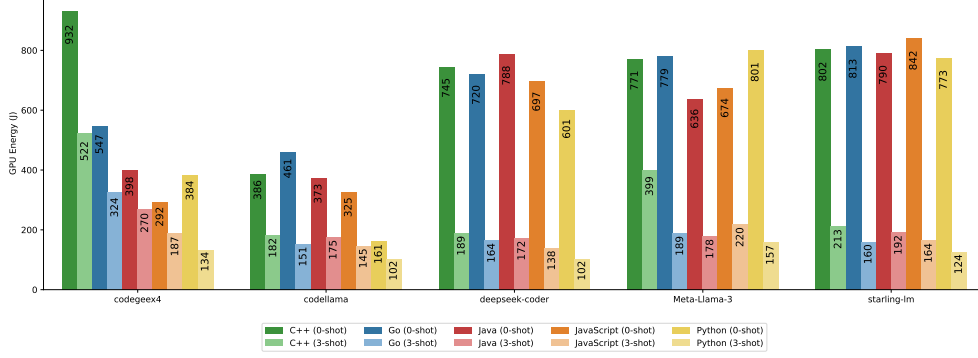


Fig. 2: GPU energy consumption (J) across LLMs and programming languages for the HumanEval-X benchmark under 0-shot and 3-shot prompting

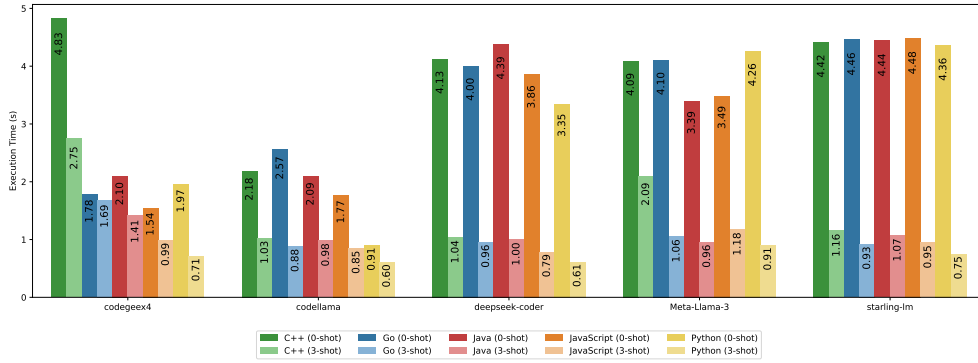


Fig. 3: Execution time (s) across evaluated LLMs and programming languages for the HumanEval-X benchmark under 0-shot and 3-shot prompting

examples leads to a decrease in both energy consumption and runtime, with the single exception of the *codellama* family. As mapped out in Figure 7, this optimization is driven by a reduction in token volume for the worst-performing models in 0-shot: *starling-lm* reduces its mean generated output volume from 328 down to just 39 tokens, triggering an energy drop from a peak of 729.23 J (4.08 s) in 0-shot down to 85.04 J (0.50 s) in 3-shot.

When evaluating baseline efficiency, *codellama* again establishes itself as the most optimal model under 0-shot conditions, requiring only 75.27 J and 0.47 s to complete the benchmark. However, the MBPP+ dataset highlights a unique behavioral inversion for this specific model: *codellama* is the only LLM that does not benefit from few-shot prompting in this setup, showing a slight performance degradation with its energy rising to 83.09 J and execution time increasing to 0.49 s under 3-shot prompting. Figure 7 exposes the root cause of this anomaly: under 0-shot, *codellama* already features an extremely lean generation profile of only 43 tokens, leaving no room for

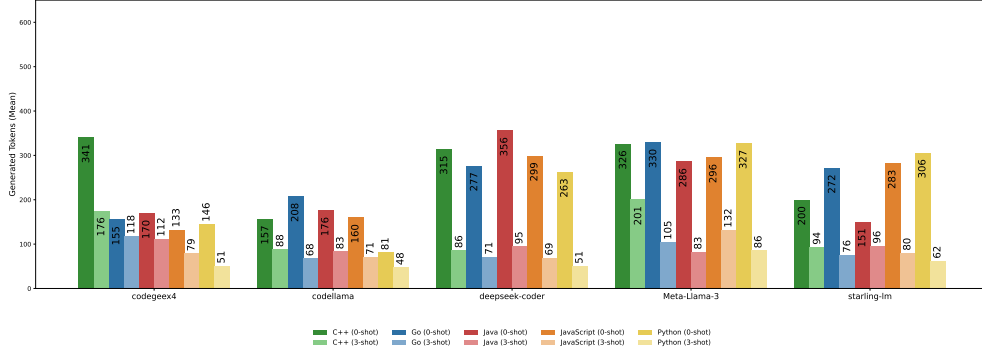


Fig. 4: Mean number of generated tokens across LLMs and programming languages for the HumanEval-X benchmark under 0-shot and 3-shot prompting

optimization via few-shot filters, shifting down marginally to 36 tokens. Because this performance penalty mirrors the exact behavioral anomaly observed in the previous CPU study [32], it highlights a potential architectural or fine-tuning particularity unique to the *codellama* family, although the exact computational root cause requires further isolated investigation. Across all other models, *deepseek-coder* emerges as the most responsive to the inclusion of in-context examples, achieving the absolute best overall efficiency in the 3-shot prompting by dropping to a minimal consumption of 73.58 J and a runtime of 0.46 s.

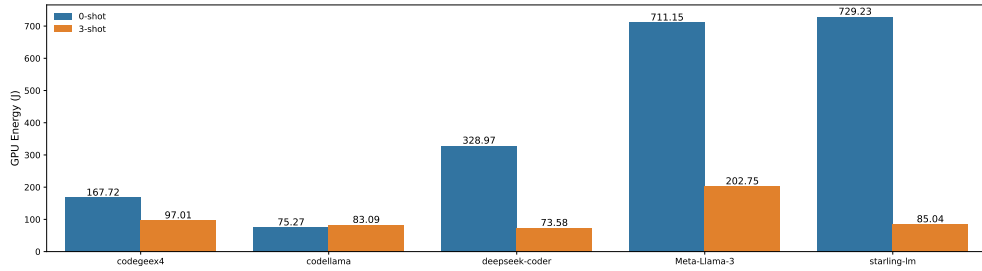


Fig. 5: GPU energy consumption (J) across evaluated LLMs for the MBPP+ benchmark under 0-shot and 3-shot prompting

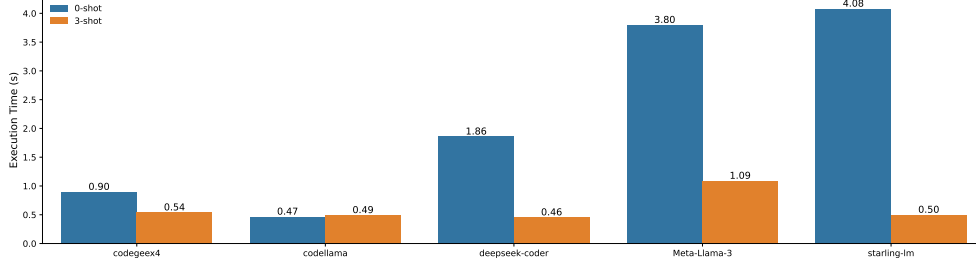


Fig. 6: Execution time (s) across evaluated LLMs for the MBPP+ benchmark under 0-shot and 3-shot prompting

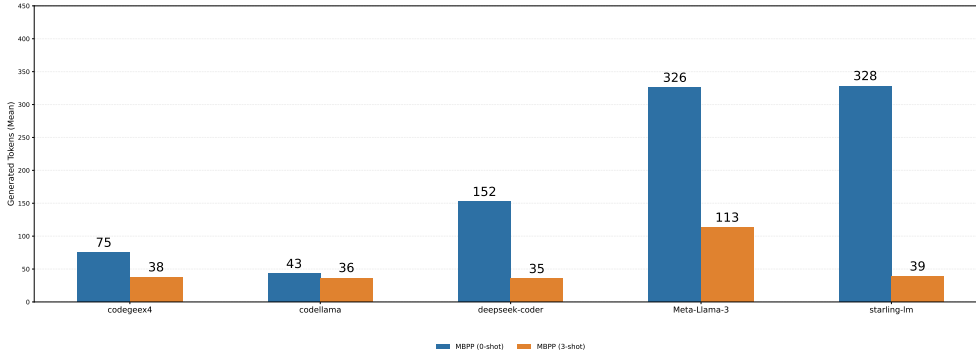


Fig. 7: Mean number of generated tokens across evaluated LLMs for the MBPP+ benchmark under 0-shot and 3-shot prompting

RQ1: How do local LLMs behave in terms of energy consumption and execution time when running under a GPU inference environment?

In GPU environments, there are significant variations in energy consumption and runtime across different models. The results highlight that resource usage is heavily influenced by the specific family of each LLM, the programming language under evaluation, and the chosen prompting configuration (0-shot vs 3-shot). Crucially, the data shows that the number of generated tokens directly determines execution time and energy costs, since 3-shot prompting saves resources by preventing models from generating noisy text (*e.g.* conversational style) alongside the code. Specifically, when transitioning from 0-shot to 3-shot prompting, the average energy consumption for the HumanEval-X benchmark was reduced by 64.4% (dropping from 644.0 J to 229.0 J), while runtime decreased by 66.1% (from 3.33 s to 1.13 s). A similar optimization trend was observed for the MBPP+ benchmark, which achieved a reduction of 47.7% in energy consumption (from 402.4 J to 210.3 J) and 59.3% in execution time (from 2.22 s to 0.90 s).

4.2 Syntactic Quality and Functional Correctness under GPU Inference

To answer **RQ2: How do local LLMs perform in terms of syntactic metrics (BLEU variants) and functional correctness (pass@k) under GPU inference?**, this subsection analyzes the generative quality and correctness of the code produced under GPU inference. The evaluation measures how the transition from a 0-shot baseline to a 3-shot prompting impacts functional correctness through pass@ k distributions, alongside structural and lexical syntactic alignment via the CodeBLEU, SacreBLEU and GoogleBLEU metrics. These dimensions are detailed sequentially for code completion in HumanEval-X (Section 4.2.1) and code generation in MBPP+ (Section 4.2.2).

4.2.1 Code Completion: HumanEval-X

The functional correctness and syntactic similarity metrics for the HumanEval-X benchmark are detailed in Figures 8, 9 and 10. Across almost all evaluated models and programming languages, the transition from 0-shot to 3-shot prompting produces a significant improvement in both $pass@1$ and $pass@10$ scores. Under 0-shot, functional correctness is exceptionally low for multi-lingual tasks (C++, Go, Java and JavaScript), with scores frequently dropping below 0.10 (representing a low 10% success rate). In contrast, 3-shot prompting guides the models to comply with the expected execution format, allowing performance to surge. For instance, *deepseek-coder* increases its $pass@1$ score from 0.010 to 0.537 in C++ and from 0.030 to 0.612 in Java. Python stands out as the only language where 0-shot baselines are already high, with *codegeex4* achieving the highest baseline score of 0.698, although its performance slightly decreases to 0.629 under 3-shot prompting. Across all other languages and shot setups, *deepseek-coder* consistently delivers the highest functional correctness values.

The $pass@10$ distributions follow an identical pattern. While 0-shot results remain severely constrained for non-Python languages, 3-shot prompting unlocks higher functional correctness. Under 3-shot, *deepseek-coder* achieves the highest functional correctness overall, reaching a $pass@10$ of 0.758 in C++, 0.832 in Go, and 0.857 in Java. Interestingly, while *codellama* suffered an energy and runtime degradation under 3-shot prompting (as seen in RQ1), its functional correctness still increases compared to its 0-shot baseline, moving from a $pass@1$ of 0.327 to 0.383 in Python, and from a $pass@10$ of 0.232 to 0.553 in C++.

Syntactic and structural similarity metrics, assessed through CodeBLEU, GoogleBLEU and SacreBLEU (Figure 10), closely mirror the functional correctness trends while highlighting distinct aspects of code generation quality. For the majority of models, the 3-shot prompting (represented by dashed lines) consistently outperform the 0-shot baselines (solid lines). This gap is particularly wide for surface-level text metrics like SacreBLEU and GoogleBLEU in languages such as C++, Java and JavaScript, where 0-shot prompting suffer heavy penalties due to conversational verbosity, with scores frequently dropping into the 0.05 to 0.12 range. Conversely, CodeBLEU, distinguished as the only metric in our evaluation that incorporates abstract syntax tree (AST) and data flow tracking alongside standard n-grams, exhibits higher resilience under 0-shot conditions, maintaining significantly higher baseline scores between 0.18

and 0.34, as it isolates and evaluates the structural components of the generated code rather than relying solely on lexical overlap.

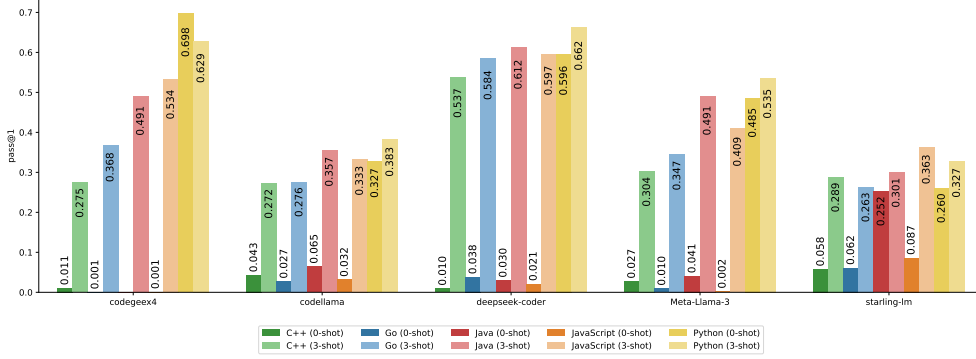


Fig. 8: Functional correctness ($pass@1$) across LLMs and programming languages for the HumanEval-X benchmark under 0-shot and 3-shot prompting

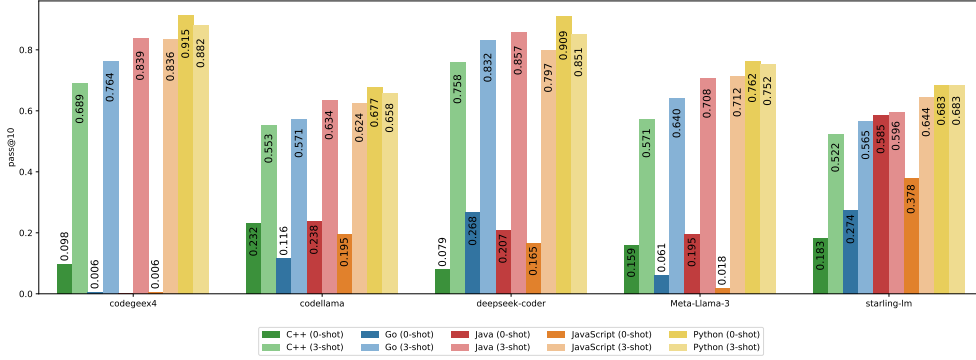


Fig. 9: Functional correctness ($pass@10$) across LLMs and programming languages for the HumanEval-X benchmark under 0-shot and 3-shot prompting

4.2.2 Code Generation: MBPP+

Figures 11 and 12 show the $pass@1$ and $pass@10$ results for both the original MBPP and the enhanced MBPP+ test suites across unsanitized and sanitized splits. The MBPP results are included for completeness, as the MBPP+ evaluation automatically reports both. Since the two benchmarks use the same tasks but MBPP+ adds more test cases per task, we compare them briefly to show the impact of stricter validation.

Among the models under 0-shot conditions, *codegeex4* and *Meta-Llama-3* show the lowest overall performance, with $pass@1$ scores dropping to 0.00 and 0.01 on

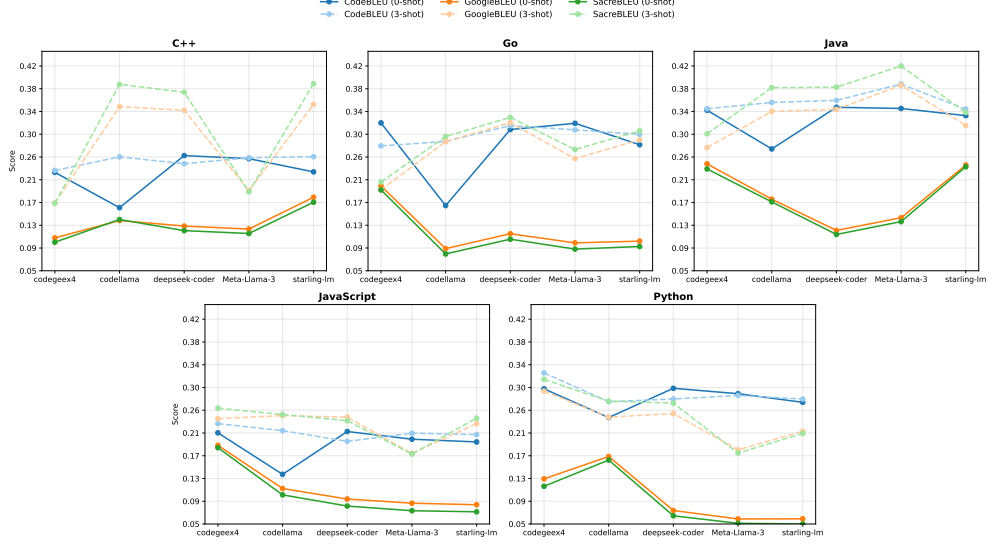


Fig. 10: Syntactic metrics (CodeBLEU, GoogleBLEU and SacreBLEU) comparison across LLMs for the HumanEval-X benchmark under 0-shot and 3-shot prompting

MBPP+ (unsanitized). In contrast, *codellama* achieves the highest baseline scores in this setting, with a $pass@1$ of 0.25 and a $pass@10$ of 0.58 on the same split. However, when transitioning to 3-shot prompting, *deepseek-coder* emerges as the top-performing model, achieving a peak $pass@1$ of 0.65 and a $pass@10$ of 0.78 on the MBPP+ sanitized split. As expected, scores are generally lower on MBPP+ due to the larger number of test cases. Also, $pass@10$ consistently outperforms $pass@1$, since generating multiple outputs increases the likelihood of a correct solution. All LLMs show improved $pass@1$ and $pass@10$ scores on both MBPP and MBPP+ after code sanitization, which removes extraneous code and isolates the function of interest. The drop in performance from MBPP to MBPP+ remains consistent post-sanitization.

The syntactic similarity evaluation (Figure 13) confirms this behavior across both subsets. The 3-shot implementations (dashed lines) maintain a steady and higher structural baseline across all variants of BLEU compared to 0-shot executions (solid lines). For *Meta-Llama-3* and *starling-lm* in the unsanitized split, surface-level lexical metrics like SacreBLEU and GoogleBLEU drop close to 0.05 under 0-shot conditions due to conversational verbosity, whereas CodeBLEU exhibits higher resilience by maintaining scores above 0.30, as it isolates and evaluates the structural syntax tree components rather than relying solely on lexical overlap.

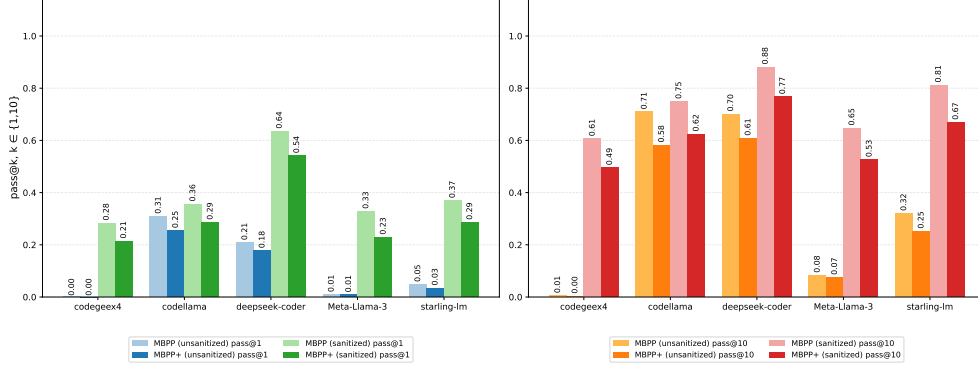


Fig. 11: Functional correctness ($pass@1$ and $pass@10$) across evaluated LLMs for the MBPP+ benchmark splits under a 0-shot prompting

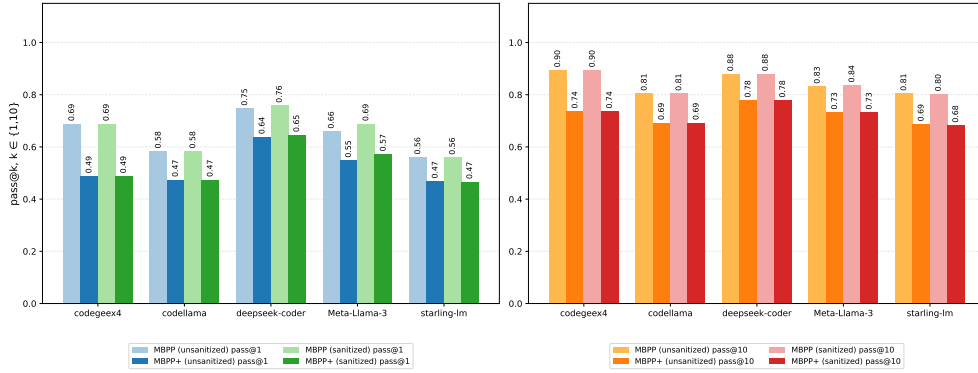


Fig. 12: Functional correctness ($pass@1$ and $pass@10$) across evaluated LLMs for the MBPP+ benchmark splits under a 3-shot prompting

RQ2: How do local LLMs perform in terms of syntactic metrics (BLEU variants) and functional correctness ($pass@k$) under GPU inference?

The results under a GPU inference environment show that transitioning from 0-shot to 3-shot prompting improves both functional correctness and syntactic metrics. For functional correctness, 3-shot learning increases $pass@1$ and $pass@10$ scores across all architectures. For syntactic similarity, the inclusion of in-context examples reduces conversational noise, leading to higher SacreBLEU and GoogleBLEU scores, while CodeBLEU exhibits a more moderate increase due to its structural evaluation of syntax trees and data flow.

4.3 Impact of the CPU-to-GPU Inference Transition

To answer **RQ3: To what extent does transitioning from CPU to GPU inference impact the overall energy efficiency, inference time, output consistency,**

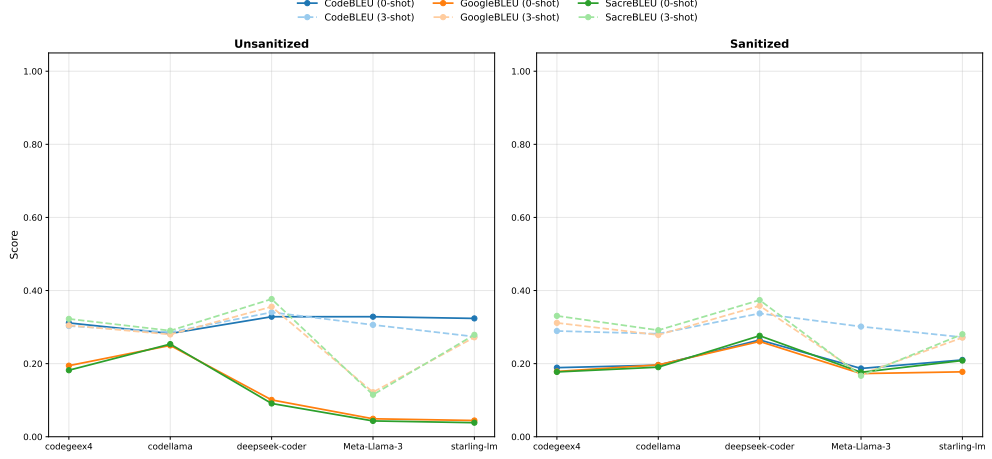


Fig. 13: Syntactic metric comparison (CodeBLEU, GoogleBLEU, and SacreBLEU) across unsanitized and sanitized splits for the MBPP+ benchmark

and functional reproducibility of local LLMs?, this section compares execution efficiency (energy and execution time) alongside generative quality ($pass@k$ and BLEU variants) to evaluate the cross-architecture trade-offs between CPU and GPU inference. For this macro-analysis, the standard MBPP dataset is integrated alongside the enhanced MBPP+ suite. This combined evaluation serves as a cross-benchmark validation mechanism to determine whether the architectural shift induces uniform impacts when moving from standard evaluation suites (represented by MBPP) to highly strict testing environments with expanded test cases (represented by MBPP+).

Figures 14 and 15 illustrate the absolute savings achieved in energy (Joules) and execution time (Seconds) when transitioning from CPU to GPU hardware, alongside their relative percentage reductions. The data reveals an absolute efficiency dominance for the GPU across all evaluated cases. Energy consumption exhibits a consistent downward trend, with the majority of these model-language pairs achieving relative savings between 82.5% and 89.4%. The only notable exceptions occur in specific few-shot setups: *codegeex4* in Go (0-shot) shows a slightly lower relative energy reduction of 78.5% (saving 2002 J), while *codellama* under the grey-labeled MBPP+ (3-shot) baseline achieves a more constrained reduction of 66.1% (saving 395 J), echoing the prompting degradation behavior identified in RQ1. Following an even more pronounced and uniform optimization pattern, execution time reductions remain highly stable across all models, languages and prompting strategies, consistently yielding relative savings of either 99.8% or 99.9%.

The impact of hardware translation on generation quality is detailed in Figures 16 and 17, which plot the average absolute gain or loss across metrics (measured on a 0.0 to 1.0 scale) between GPU and CPU execution. Under 3-shot prompting, the GPU environment induces a substantial and consistent absolute quality improvement for the Python subsets under unsanitized conditions (MBPP Unsanitized and MBPP+ Unsanitized). For instance, in $pass@1$, *deepseek-coder* achieves an absolute metric

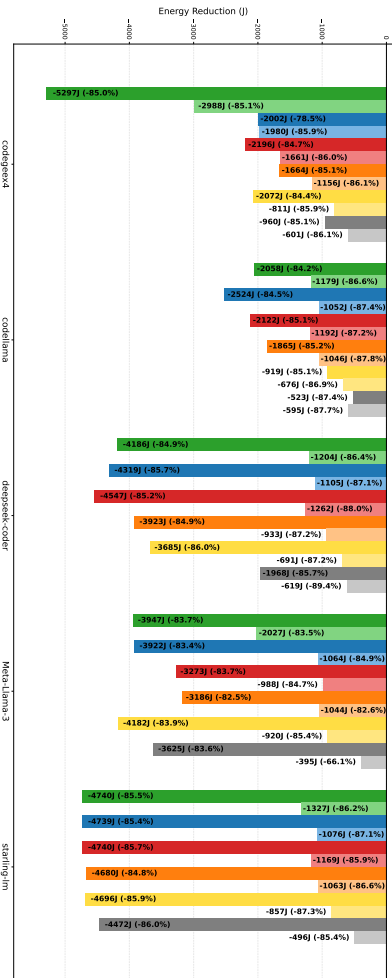


Fig. 14: Absolute and relative GPU energy reduction (J) across LLMs, languages and benchmarks compared to CPU execution

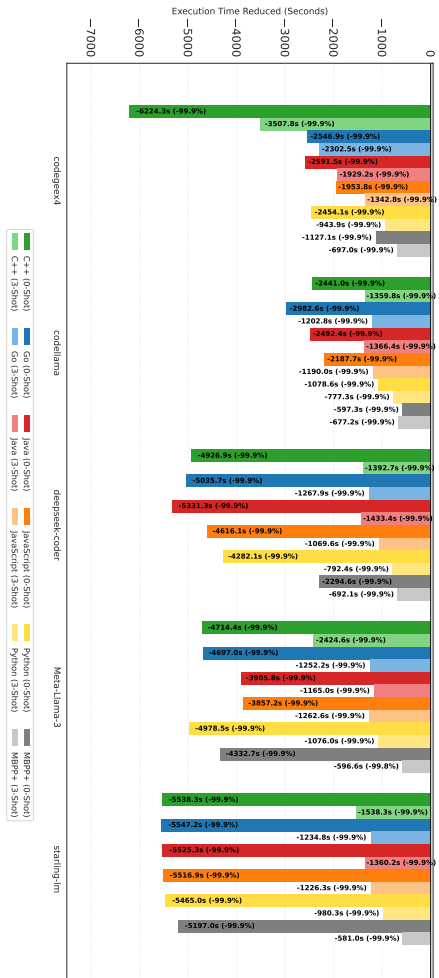


Fig. 15: Absolute and relative execution time reduction (s) across LLMs, languages and benchmarks compared to CPU execution

gain of +0.748 (MBPP) and +0.639 (MBPP+), while *codegeex4* exhibits absolute gains of +0.686 and +0.486, respectively. This improvement indicates that the GPU environment successfully handles long-context dependencies introduced by few-shot examples.

In contrast, 0-shot prompting exposes severe performance inconsistencies during hardware transition. While some models like *codellama* show minor functional correctness gains under unsanitized settings, others suffer catastrophic degradation; most notably, *Meta-Llama-3* experiences an absolute performance drop of -0.645 in *pass@1* and -0.752 in *pass@10* on the MBPP Unsanzitized set. This cross-hardware divergence occurs because keeping identical execution parameters and hyperparameter

configurations across different studies cannot guarantee determinism when changing computational backends [31]. Due to the non-associative nature of floating-point arithmetic, the highly parallel reduction paths and optimization kernels within CUDA introduce micro-numerical deviations in the outputs compared to CPU execution [30]. Even though sampling parameters like *temperature* = 0.6, *top_p* = 0.95 and *seed* = 42 are kept perfectly constant, the sampling layer processes these slightly altered logit distributions. This leads the pseudo-random token selection to draw different paths, inherently causing the final code solutions to diverge between the two hardware platforms. This pattern is closely mirrored in the syntactic matrices (Figure 17), where 3-shot lexical metrics (SacreBLEU and GoogleBLEU) experience prominent absolute gains on unsanitized blocks (e.g., up to +0.225 and +0.207 for *codellama*) due to the suppression of conversational text under accelerated inference, whereas CodeBLEU metrics exhibit negligible absolute variations between hardware executions.

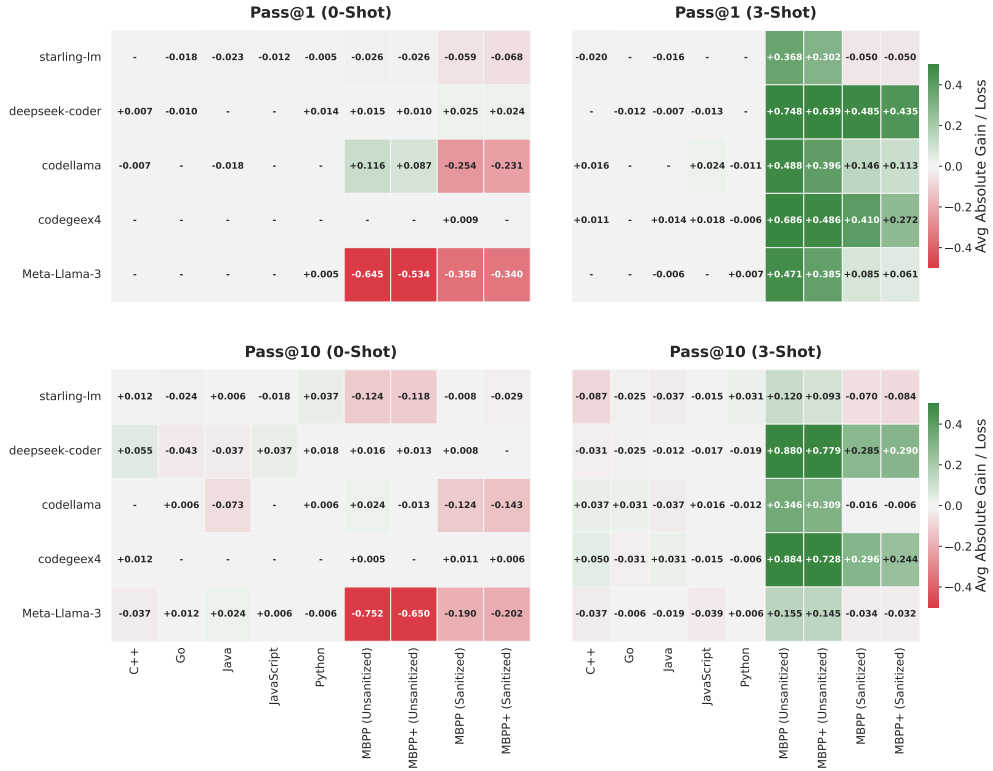


Fig. 16: Heatmaps of the average absolute gain or loss in functional correctness (*pass@1* and *pass@10*) between GPU and CPU hardware execution under 0-shot and 3-shot prompting

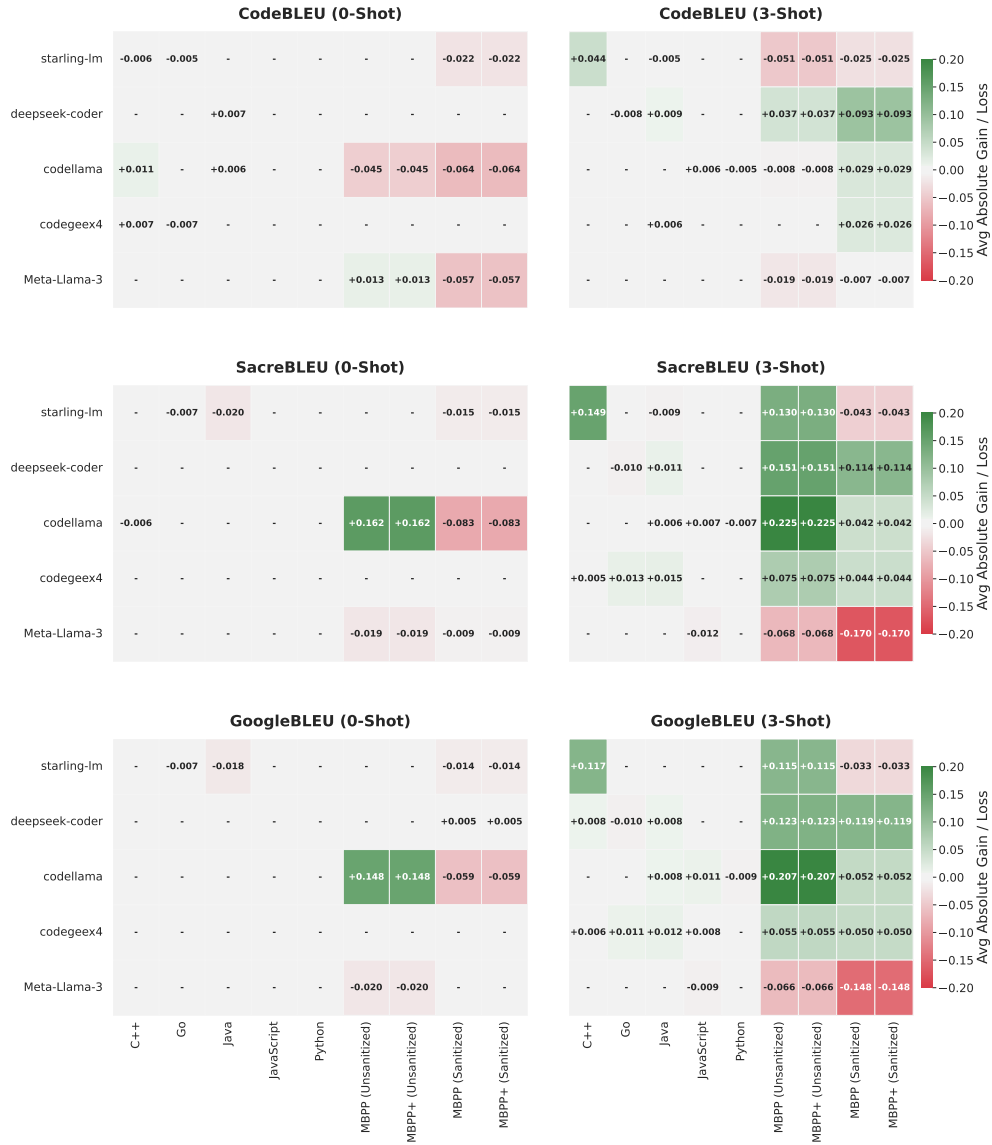


Fig. 17: Heatmaps of the average absolute gain or loss across CodeBLEU, SacreBLEU and GoogleBLEU metrics between GPU and CPU hardware execution under 0-shot and 3-shot prompting

RQ3: To what extent does transitioning from CPU to GPU inference impact the overall energy efficiency, inference time, output consistency, and functional reproducibility of local LLMs?

Transitioning from CPU to GPU hardware significantly reduces both energy consumption and execution time across all evaluated models and programming languages, with energy savings scaling up to 89.4% and runtime consistently dropping by 99.8% to 99.9%. Regarding functional correctness ($pass@k$), the hardware transition improves scores under 3-shot prompting - particularly for the MBPP+ benchmark - but introduces volatility under 0-shot, where performance changes vary depending on the model. For syntactic metrics, the hardware transition increases lexical similarity (SacreBLEU and GoogleBLEU) under 3-shot prompting due to reduced conversational noise, whereas structural similarity (CodeBLEU) exhibits no significant variations between the two hardware platforms.

4.4 LLMs’ Energy-Based Ranking

To synthesize the experimental results, Table 2 presents a multi-dimensional ranking of the LLMs prioritized by their mean GPU energy consumption across all tasks, mapping efficiency alongside execution time, functional correctness ($pass@1$, $pass@10$) and structural similarity (CodeBLEU). We report both $pass@1$ and $pass@10$ to capture functional correctness under different generation constraints. For the syntactic metrics, we choose to report only CodeBLEU for simplification, as it focuses specifically on code structure - evaluating AST and data flow - rather than basic text overlap. Within this ranking, the model in Rank 1 represents the optimal efficiency reference for that specific n-shot. For the remaining rows, the energy and time sub-scripts indicate the performance penalty multiplier relative to this leader (e.g., in 0-shot, *codegeex4* requires $1.53\times$ more energy and $1.31\times$ more time than *codellama*). Furthermore, the colored arrows and adjacent sub-scripts denote the absolute metric shifts and direction when transitioning from the baseline CPU environment [32] to the accelerated GPU setup, where upward green arrows ($\uparrow$) represent beneficial quality improvements, downward red arrows ($\downarrow$) indicate performance degradation and rightward grey arrows ($\rightarrow$) signal stable, unchanged metrics. For example, moving to a GPU environment under a 3-shot setup allows the leading *deepseek-coder* to achieve an absolute functional boost of $+0.07$ in $pass@1$ and $+0.03$ in $pass@10$ compared to its CPU execution, while also slightly increasing its structural similarity score ($+0.02$).

Table 2: Ranking of LLMs with Energy and Time, as well as Pass@1, Pass@10 and CodeBLEU in Each Shot Setting (GPU environment)

	Rank	LLM	Energy (J)	Time (s)	pass@1	pass@10	CodeBLEU
0-Shot	1	codellama	296	1.7	0.13 $\downarrow_{-0.04}$	0.35 $\downarrow_{-0.03}$	0.20 $\rightarrow_{+0}$
	2	codegeex4	453 $_{1.53\times}$	2.2 $_{1.31\times}$	0.15 $\rightarrow_{+0}$	0.25 $\rightarrow_{+0}$	0.26 $\downarrow_{-0.01}$
	3	deepseek-coder	646 $_{2.18\times}$	3.6 $_{2.16\times}$	0.21 $\uparrow_{+0.01}$	0.40 $\uparrow_{+0.01}$	0.28 $\rightarrow_{+0}$
	4	Meta-Llama-3	728 $_{2.46\times}$	3.9 $_{2.32\times}$	0.13 $\downarrow_{-0.06}$	0.29 $\downarrow_{-0.03}$	0.27 $\downarrow_{-0.01}$
	5	starling-lm	791 $_{2.67\times}$	4.4 $_{2.63\times}$	0.17 $\downarrow_{-0.02}$	0.46 $\rightarrow_{+0}$	0.25 $\downarrow_{-0.01}$
3-Shot	1	deepseek-coder	139	0.8	0.61 $\uparrow_{+0.07}$	0.81 $\uparrow_{+0.03}$	0.29 $\uparrow_{+0.02}$
	2	codellama	139 $_{1.00\times}$	0.8 $_{1.00\times}$	0.35 $\uparrow_{+0.02}$	0.62 $\rightarrow_{+0}$	0.28 $\uparrow_{+0.01}$
	3	starling-lm	156 $_{1.12\times}$	0.9 $_{1.10\times}$	0.33 $\downarrow_{-0.02}$	0.62 $\downarrow_{-0.03}$	0.28 $\uparrow_{+0.01}$
	4	Meta-Llama-3	224 $_{1.60\times}$	1.2 $_{1.50\times}$	0.44 $\uparrow_{+0.01}$	0.69 $\downarrow_{-0.02}$	0.29 $\rightarrow_{+0}$
	5	codegeex4	255 $_{1.83\times}$	1.3 $_{1.67\times}$	0.46 $\uparrow_{+0.05}$	0.79 $\uparrow_{+0.04}$	0.28 $\rightarrow_{+0}$

The consolidated data highlights a critical trade-off between resource efficiency and generative quality. Under 0-shot conditions, *codellama* secures the top rank, demanding a minimal mean energy consumption of 296 J and an execution time of 1.7 s. However, this efficiency comes at a high quality cost, as it exhibits low functional correctness scores ($pass@1 = 0.13$, $pass@10 = 0.35$). When transitioning to a 3-shot prompting, the entire hierarchy changes. While *deepseek-coder* and *codellama* share the peak efficiency tier (139 J, 0.8 s), *deepseek-coder* achieves absolute functional superiority,

delivering a *pass@1* of 0.61 and a *pass@10* of 0.81, establishing itself as the most well-balanced model in GPU environments.

5 Threats to Validity

This section addresses the construct, internal and external threats to validity identified in our experimental design, along with the mitigation strategies applied.

5.1 Construct Validity

Standardized benchmarks like HumanEval-X and MBPP+ focus on isolated algorithmic tasks, which do not fully replicate real-world programming where software development is interactive and context-heavy [84]. However, as noted in Section 3.3, we selected these specific datasets because they are widely adopted across the literature for evaluating LLM-generated code, ensuring our cross-model comparisons remain consistent with existing academic papers. Regarding the energy measurement process, tracking software-based consumption can capture background system activities. We mitigated this by running all experiments in isolated node environments on the Deucalion cluster, allowing CodeCarbon to automatically interface with the NVIDIA Management Library (NVML) via the `nvidia-ml-py` dependency to collect the required metrics.

5.2 Internal Validity

Internal threats involve typical hardware power variations and system-level processes that can alter energy and runtime measurements. To minimize this, we ran all models using the same *LLaMa.cpp* software stack with identical parameters (seed, temperature, context window and max tokens). We also executed every prompt 10 times consecutively to account for any performance fluctuations. Outliers were filtered out using the Interquartile Range (IQR) method, and we validated the final significance trends using the 180 independent hypothesis tests described in Section 3.5. Furthermore, for the cross-hardware analysis, the baseline CPU metrics were collected on a physically different machine. Since the two environments do not share the same motherboard or RAM architecture, these infrastructure differences inevitably influence the energy baselines independently of the transition from CPU to GPU execution. This architectural change also introduces hardware-level nondeterminism that causes minor fluctuations in the raw logit outputs. Although this variance can lead the final code solutions to diverge even under identical configuration settings, we isolated its operational impact on our energy and runtime comparisons by maintaining strict parameter symmetry across all runs. Finally, we applied a uniform prompt structure across all evaluated LLMs, omitting the model-specific special tokens required by certain architectures. While this design choice was intentional to maintain strict parity with the original CPU baseline study, ignoring these specialized formats may introduce operational bias and alter the generation behavior of individual models.

5.3 External Validity

The generalizability of our findings is constrained by our specific choice of models and configurations. All LLMs were evaluated using quantized GGUF variants at Q5 and Q6 precision. Testing other quantization levels (such as Q4 or Q8) or choosing different model families could yield different energy and performance rankings. Additionally, because new LLMs are released continuously, newer versions might alter the specific positions in our ranking. However, our selection includes models from different training backgrounds - covering both highly specialized models for code and general-purpose systems - and the evaluation framework is designed to be fully repeatable for benchmarking future models under the same criteria.

6 Conclusions

This paper presented an extension of our previous study [32]. While the original work benchmarked five local LLMs on a CPU-only setup, we replicated the exact setup on a GPU cluster to evaluate how the hardware platform used and the number of shots in the prompt influence energy consumption, inference time, functional correctness ($\text{pass}@1$, $\text{pass}@10$) and code similarity metrics (CodeBLEU, SacreBLEU and GoogleBLEU).

Our main findings show that energy and runtime are heavily tied to the volume of generated tokens. Moving from 0-shot to 3-shot prompting acts as a filter that cuts out conversational noise, explanations and markdown wrappers, lowering both energy and execution times across nearly all models by guiding them toward concise responses. Specifically, 3-shot prompting reduces average energy consumption and runtime by 64.4% and 66.1% for HumanEval-X, and by 47.7% and 59.3% for MBPP+. Regarding the transition from CPU to GPU, moving to the GPU cluster cuts energy between 78.5% and 89.4% for HumanEval-X (and 66.1% to 89.4% for MBPP+), while reducing runtime by 99.8% to 99.9% across all tasks. However, this hardware transition does not lead to a clear trend in code quality; while we see small variations in the metrics, the overall impact is inconsistent. Due to hardware-level nondeterminism from parallel floating-point operations (inherent to GPUs), changing the execution platform causes the raw model outputs to diverge. In 0-shot scenarios, this divergence introduces noticeable performance fluctuations and structural code variations where functional correctness and similarity metrics vary unpredictably or even degrade. In contrast, the 3-shot configuration provides greater operational stability, maintaining structural CodeBLEU scores while preventing text noise and hardware-induced variance from harming the output quality.

Based on our consolidated energy ranking, the choice of the optimal model depends directly on the prompting strategy, with clear shifts when compared to CPU execution. Under 0-shot conditions, *codellama* remains the most energy-efficient architecture across both hardware platforms, though it delivers poor functional correctness. However, under 3-shot prompting, *deepseek-coder* achieves the best overall balance, matching *codellama* as the peak efficiency leader on the GPU while delivering the highest functional accuracy ($\text{pass}@1 = 0.61$, $\text{pass}@10 = 0.81$) and the largest quality improvements over its CPU baseline. These findings reinforce that hardware acceleration alone is purely an efficiency optimization and not a quality booster. To deploy local LLMs effectively,

model selection and prompt structure must be considered together, with few-shot configurations on models like *deepseek-coder* offering the most balanced trade-off by sustaining high functional correctness at the lowest hardware energy tier.

Future Work:

Our study suggests several directions for future research. Evaluating these local LLMs on broader benchmarks like RepoCod [85], LiveCodeBench [86] and SWE-Bench [84] could better capture repository-level dependencies and real-world coding complexity. Developing precise methodology to isolate LLM-specific energy consumption from background system overhead would also improve the granularity of sustainability assessments. Furthermore, testing newer model families across a wider range of hardware architectures and alternative quantization levels could generalize our findings and track industry progress in energy-efficient code generation. Finally, future work should explore how advanced prompting techniques (such as Chain-of-Thought [87]) and internal model parameters like *temperature* and *top-p* influence the energy-efficiency and correctness trade-offs during GPU inference.

Data Availability Statement

The datasets, source code and experimental artifacts supporting the findings of this study are publicly available in the Figshare repository at <https://doi.org/10.6084/m9.figshare.32984774>. Additional experimental results and supplementary materials are available in the online appendix at <https://sites.google.com/view/energy-local-llms-for-code/new-results>.

Acknowledgements. This work is funded by national funds through FCT - Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (<https://doi.org/10.54499/UID/50014/2025>). Simão Cunha is also funded by FCT grant 2025.04596.BDANA (<https://doi.org/10.54499/2025.04596.BDANA>).

References

- [1] OpenAI, et al.: GPT-4 Technical Report (2024). <https://arxiv.org/abs/2303.08774>
- [2] Devlin, J., Chang, M.-W., Lee, K., et al.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019). <https://arxiv.org/abs/1810.04805>
- [3] Li, L., Wang, P., Ren, K., et al.: Origin Tracing and Detecting of LLMs (2023). <https://arxiv.org/abs/2304.14072>
- [4] Chen, M., et al.: Evaluating Large Language Models Trained on Code (2021). <https://arxiv.org/abs/2107.03374>
- [5] Sobania, D., Briesch, M., Hanna, C., et al.: An Analysis of the Automatic Bug Fixing Performance of ChatGPT (2023). <https://arxiv.org/abs/2301.08653>

- [6] Madaan, A., Tandon, N., Gupta, P., et al.: Self-Refine: Iterative Refinement with Self-Feedback (2023). <https://arxiv.org/abs/2303.17651>
- [7] OpenAI: ChatGPT. <https://chat.openai.com>. Accessed: July 13, 2026
- [8] GitHub: GitHub Copilot. <https://github.com/features/copilot>. Accessed: July 13, 2026
- [9] DeepSeek-AI, Zhu, Q., Guo, D., et al.: DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence (2024). <https://arxiv.org/abs/2406.11931>
- [10] Meta AI: The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. Meta AI Blog. Accessed: July 17, 2026 (2025). <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>
- [11] Izadi, M., Katzy, J., Dam, T., et al.: Language Models for Code Completion: A Practical Evaluation (2024). <https://arxiv.org/abs/2402.16197>
- [12] Li, K., Yuan, Y.: Large Language Models as Test Case Generators: Performance Evaluation and Enhancement (2024). <https://arxiv.org/abs/2404.13340>
- [13] Santos, S., Saraiva, J., Ribeiro, F.: Large language models in automated repair of haskell type errors. In: Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair. APR '24, pp. 42–45. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3643788.3648012>
- [14] Stack Overflow: 2025 Developer Survey. Accessed: July 13, 2026 (2025). <https://survey.stackoverflow.co/2025/>
- [15] Luccioni, S., Jernite, Y., Strubell, E.: Power hungry processing: Watts driving the cost of ai deployment? In: Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency. FAccT '24, pp. 85–99. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3630106.3658542>
- [16] Schwartz, R., Dodge, J., Smith, N.A., Etzioni, O.: Green ai. *Commun. ACM* **63**(12), 54–63 (2020) <https://doi.org/10.1145/3381831>
- [17] Roush, T.: Microsoft’s AI Buildout Sent Carbon Emissions Up 25% Last Year. *Forbes*. Accessed: July 17, 2026 (2026). <https://www.forbes.com/sites/tylerroush/2026/07/09/microsofts-ai-buildout-sent-carbon-emissions-up-25-last-year/>
- [18] International Energy Agency: Energy and ai: Executive summary. Technical report, IEA (2025). Accessed: July 17, 2026. <https://www.iea.org/reports/energy-and-ai>
- [19] Scholz, S., Lawall, A., Schaaff, K.: The impact of large language models on it security in the corporate environment. In: 2024 2nd International Conference

- on Foundation and Large Language Models (FLLM), pp. 109–115 (2024). <https://doi.org/10.1109/FLLM63129.2024.10852476>
- [20] Kumar, B., Shaheer Ahmed, M.: Beyond clouds: Locally runnable llms as a secure solution for ai applications. *Digital Society* **3** (2024) <https://doi.org/10.1007/s44206-024-00141-y>
 - [21] Ghosh, A., Kaffee, L.-A., Jernite, Y., Solaiman, I.: State of Open Source on Hugging Face: Spring 2026. Accessed: July 17, 2026 (2026). <https://huggingface.co/blog/huggingface/state-of-os-hf-spring-2026>
 - [22] Yoshida, L.: The Impact of Example Selection in Few-Shot Prompting on Automated Essay Scoring Using GPT Models, pp. 61–73 (2024). https://doi.org/10.1007/978-3-031-64315-6_5
 - [23] Brown, T.B., Mann, B., Ryder, N., et al.: Language Models are Few-Shot Learners (2020). <https://arxiv.org/abs/2005.14165>
 - [24] Zhao, W.X., Zhou, K., Li, J., et al.: A Survey of Large Language Models (2025). <https://arxiv.org/abs/2303.18223>
 - [25] Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., Steinhardt, J.: Measuring Massive Multitask Language Understanding (2021). <https://arxiv.org/abs/2009.03300>
 - [26] Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., Steinhardt, J.: Measuring Mathematical Problem Solving With the MATH Dataset (2021). <https://arxiv.org/abs/2103.03874>
 - [27] Singhal, K., Azizi, S., Tu, T., et al.: Large Language Models Encode Clinical Knowledge (2022). <https://arxiv.org/abs/2212.13138>
 - [28] Adamska, M., Smirnova, D., Nasiri, H., Yu, Z., Garraghan, P.: Green Prompting: Characterizing Prompt-driven Energy Costs of LLM Inference (2026). <https://arxiv.org/abs/2503.10666>
 - [29] Podder, S., Date, H., Murthy, S.: Green prompt engineering for sustainable generative ai. *Environmental Science and Ecotechnology* **30**, 100684 (2026) <https://doi.org/10.1016/j.esec.2026.100684>
 - [30] Yuan, J., Li, H., Ding, X., Xie, W., Li, Y.-J., Zhao, W., Wan, K., Shi, J., Hu, X., Liu, Z.: Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference (2025). <https://arxiv.org/abs/2506.09501>
 - [31] Gond, R., Kamath, A.K., Ramjee, R., Panwar, A.: LLM-42: Enabling Determinism in LLM Inference with Verified Speculation (2026). <https://arxiv.org/abs/2601.17768>

- [32] Cunha, S., Ribeiro, F., Cruz, L., Saraiva, J.: Not all local llms are equal: A benchmark of energy and performance. In: Proceedings of the IEEE/ACM 10th International Workshop on Green and Sustainable Software. GREENS '26, pp. 99–106. Association for Computing Machinery, New York, NY, USA (2026). <https://doi.org/10.1145/3786148.3788630>
- [33] Bartowski: Meta-Llama-3-8B-Instruct-Q6_K. https://huggingface.co/bartowski/Meta-Llama-3-8B-Instruct-GGUF/blob/main/Meta-Llama-3-8B-Instruct-Q6_K.gguf (2024)
- [34] Tom Jobbins: deepseek-coder-6.7b-instruct.Q5_K_M. https://huggingface.co/TheBloke/deepseek-coder-6.7B-instruct-GGUF/blob/main/deepseek-coder-6.7b-instruct.Q5_K_M.gguf (2023)
- [35] Tom Jobbins: codellama-7b-instruct.Q5_K_M. https://huggingface.co/TheBloke/CodeLlama-7B-Instruct-GGUF/blob/main/codellama-7b-instruct.Q5_K_M.gguf (2023)
- [36] Bartowski: codegeex4-all-9b-Q6_K_L. https://huggingface.co/bartowski/codegeex4-all-9b-GGUF/blob/main/codegeex4-all-9b-Q6_K_L.gguf (2024)
- [37] Tom Jobbins: starling-lm-7b-alpha.Q5_K_S. https://huggingface.co/TheBloke/Starling-LM-7B-alpha-GGUF/blob/main/starling-lm-7b-alpha.Q5_K_S.gguf (2023)
- [38] Zheng, Q., Xia, X., Zou, X., *et al.*: Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. KDD '23, pp. 5673–5684. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3580305.3599790>
- [39] Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation (2023). <https://arxiv.org/abs/2305.01210>
- [40] Chen, M., Li, G., Wu, L.-I., *et al.*: DCE-LLM: Dead Code Elimination with Large Language Models (2025). <https://arxiv.org/abs/2506.11076>
- [41] Peslalz, T.L.J., Katz, B.: Educated energy efficiency optimization of distributed software: Measure, monitor, mitigate. In: 2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS), pp. 20–27 (2025). <https://doi.org/10.1109/GREENS66463.2025.00008>
- [42] Kaffes, K., Sbirlea, D., Lin, Y., *et al.*: Leveraging application classes to save power in highly-utilized data centers. In: Proceedings of the 11th ACM Symposium on Cloud Computing. SoCC '20, pp. 134–149. Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3419111.3421274>

- [43] Costa, G.D., Pierson, J.-M.: Dvfs governor for hpc: Higher, faster, greener. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 533–540 (2015). <https://doi.org/10.1109/PDP.2015.73>
- [44] Verdecchia, R., Sallou, J., Cruz, L.: A Systematic Review of Green AI (2023). <https://arxiv.org/abs/2301.11047>
- [45] Pereira, R., Couto, M., Ribeiro, F., *et al.*: Ranking programming languages by energy efficiency. *Science of Computer Programming* **205**, 102609 (2021) <https://doi.org/10.1016/j.scico.2021.102609>
- [46] Intel Corporation: Running Average Power Limit (RAPL) Energy Reporting. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html> (2022)
- [47] Courty, B., Schmidt, V., Luccioni, S., Goyal-Kamal, MarionCoutarel, Feld, B., Lecourt, J., LiamConnell, Saboni, A., Inimaz, supatomic, Léval, M., Blanche, L., Cruveiller, A., ouminasara, Zhao, F., Joshi, A., Bogroff, A., Lavoreille, H., Laskaris, N., Abati, E., Blank, D., Wang, Z., Catovic, A., Alencon, M., Stęchły, M., Bauer, C., Araújo, L.O.N., JPW, MinervaBooks: mlco2/codecarbon: v2.4.1. Zenodo (2024). <https://doi.org/10.5281/zenodo.11171501> . <https://doi.org/10.5281/zenodo.11171501>
- [48] Garg, S., Moghaddam, R.Z., Clement, C.B., *et al.*: Deepdev-perf: a deep learning-based approach for improving software performance. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022, pp. 948–958. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3540250.3549096>
- [49] Ribeiro, F., Macedo, J.N.C., Tsushima, K., *et al.*: Gpt-3-powered type error debugging: Investigating the use of large language models for code repair. In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2023, pp. 111–124. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3623476.3623522>
- [50] Xia, C.S., Zhang, L.: Less training, more repairing please: revisiting automated program repair via zero-shot learning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022, pp. 959–971. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3540250.3549101>
- [51] Hadi, M.U., Tashi, Q.A., Qureshi, R., *et al.*: A survey on large language models: Applications, challenges, limitations, and practical usage. *TechRxiv* (2023) <https://doi.org/10.36227/techrxiv.23589741.v1>

- [52] Kaddour, J., Harris, J., Mozes, M., Bradley, H., Raileanu, R., McHardy, R.: Challenges and Applications of Large Language Models (2023). <https://arxiv.org/abs/2307.10169>
- [53] Chen, B., Zhang, Z., Langrené, N., Zhu, S.: Unleashing the potential of prompt engineering for large language models. *Patterns* **6**(6), 101260 (2025) <https://doi.org/10.1016/j.patter.2025.101260>
- [54] Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T.Y., Singh, S., Tang, X., Werra, L., Longpre, S.: OctoPack: Instruction Tuning Code Large Language Models (2024). <https://arxiv.org/abs/2308.07124>
- [55] CodeParrot: InstructHumanEval. <https://huggingface.co/datasets/codeparrot/instructhumaneval>. Accessed: January 26, 2025 (2024)
- [56] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program Synthesis with Large Language Models (2021). <https://arxiv.org/abs/2108.07732>
- [57] Alizadeh, N., Belchev, B., Saurabh, N., et al.: Analyzing the Energy and Accuracy of LLMs in Software Development (2024). <https://arxiv.org/abs/2412.00329>
- [58] Durán, F., Martínez, M., Lago, P., et al.: Energy consumption of code small language models serving with runtime engines and execution providers (2024). <https://arxiv.org/abs/2412.15441>
- [59] Shi, J., Yang, Z., Kang, H.J., *et al.*: Greening large language models of code. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society. ICSE-SEIS'24*, pp. 142–153. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3639475.3640097>
- [60] Vartziotis, T., Dellatolas, I., Dasoulas, G., *et al.*: Learn to code sustainably: An empirical study on green code generation. In: *Proceedings of the 1st International Workshop on Large Language Models for Code. LLM4Code '24*, pp. 30–37. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3643795.3648394> . <https://doi.org/10.1145/3643795.3648394>
- [61] Wan, Z., Wang, X., Liu, C., et al.: Efficient Large Language Models: A Survey (2024). <https://arxiv.org/abs/2312.03863>
- [62] Shi, J., Yang, Z., Lo, D.: Efficient and green large language models for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.* **34**(5) (2025) <https://doi.org/10.1145/3708525>
- [63] Stuhlmann, L., Argerich, M.F., Fürst, J.: Bench360: Benchmarking Local LLM Inference from 360 Degrees (2026). <https://arxiv.org/abs/2511.16682>

- [64] Chitty-Venkata, K.T., Raskar, S., Kale, B., Ferdous, F., Tanikanti, A., Raffanetti, K., Taylor, V., Emani, M., Vishwanath, V.: Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1362–1379 (2024). <https://doi.org/10.1109/SCW63240.2024.00178>
- [65] Mucci, T.: GGUF versus GGML. <https://www.ibm.com/think/topics/gguf-versus-ggml> (2024)
- [66] Rozière, B., Gehring, J., Gloeckle, F., et al.: Code Llama: Open Foundation Models for Code (2024). <https://arxiv.org/abs/2308.12950>
- [67] Tong, W., Zhang, T.: CodeJudge: Evaluating Code Generation with Large Language Models (2024). <https://arxiv.org/abs/2410.02184>
- [68] Grattafiori, A., Dubey, A., Jauhri, A., et al.: The Llama 3 Herd of Models (2024). <https://arxiv.org/abs/2407.21783>
- [69] Szalontai, B., Szalay, G., Márton, T., et al.: Large Language Models for Code Summarization (2024). <https://arxiv.org/abs/2405.19032>
- [70] Li, B., Wu, W., Tang, Z., et al.: Prompting Large Language Models to Tackle the Full Software Development Lifecycle: A Case Study (2024). <https://arxiv.org/abs/2403.08604>
- [71] Miller, S., Rai, D., Yao, Z.: Mechanistic Understanding of Language Models in Syntactic Code Completion (2025). <https://arxiv.org/abs/2502.18499>
- [72] Raffin, G., Trystram, D.: Dissecting the software-based measurement of CPU energy consumption: a comparative analysis (2024). <https://arxiv.org/abs/2401.15985>
- [73] Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., Rajwan, D.: Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro* **32**(2), 20–27 (2012) <https://doi.org/10.1109/MM.2012.12>
- [74] Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **3**(2) (2018) <https://doi.org/10.1145/3177754>
- [75] Hähnel, M., Döbel, B., Völpl, M., Härtig, H.: Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.* **40**(3), 13–17 (2012) <https://doi.org/10.1145/2425248.2425252>
- [76] Cunha, S.: Adapted HumanEval-X Benchmark. <https://github.com/simaocunha71/codefuse-evaluation>. Forked from original HumanEval-X repository.

Last commit SHA: 90b0148256c254caff6abc61dcada5d0c7555400 (2024)

- [77] CodeFuse: HumanEval-X. <https://github.com/codefuse-ai/codefuse-evaluation>. Accessed: January 10, 2025 (2023)
- [78] Cunha, S.: Adapted MBPP+ Benchmark. <https://github.com/simaocunha71/evalplus>. Forked from original MBPP+ repository. Last commit SHA: f86db47fd3150b480376fe149c3ad047b7fff0b0 (2024)
- [79] Bui, T.-D., Vu, T.T., Nguyen, T.-T., et al.: Correctness Assessment of Code Generated by Large Language Models Using Internal Representations (2025). <https://arxiv.org/abs/2501.12934>
- [80] Ren, S., Guo, D., Lu, S., et al.: CodeBLEU: a Method for Automatic Evaluation of Code Synthesis (2020). <https://arxiv.org/abs/2009.10297>
- [81] Wu, Y., Schuster, M., Chen, Z., et al.: Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation (2016). <https://arxiv.org/abs/1609.08144>
- [82] Gerganov, G.: LLaMa.cpp: LLM Inference in C/C++. <https://github.com/ggerganov/llama.cpp>. Accessed: January 16, 2024 (2023)
- [83] Castor, F.: Estimating the Energy Footprint of Software Systems: a Primer (2024). <https://arxiv.org/abs/2407.11611>
- [84] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (2024). <https://arxiv.org/abs/2310.06770>
- [85] Liang, S., Hu, Y., Jiang, N., et al.: Can Language Models Replace Programmers for Coding? REPOCOD Says ‘Not Yet’ (2025). <https://arxiv.org/abs/2410.21647>
- [86] Jain, N., Han, K., Gu, A., et al.: LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code (2024). <https://arxiv.org/abs/2403.07974>
- [87] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models (2023). <https://arxiv.org/abs/2201.11903>