A Summary of EnergyBench: A Holistic and Systematic Benchmark for Measuring the Correctness and Energy-Efficiency of LLM-Generated Code

The growing adoption of Artificial Intelligence (AI) is a primary contributor to increased energy demands in data centers, raising environmental concerns for the Information, Communication, and Technology (ICT) sector as a whole. In the last five years, advances in areas like natural language processing and computer vision have significantly increased the global adoption of AI. An important topic of research lies at the intersection of Large Language Models (LLMs) with modern software development. The software industry is currently facing a steady transformation towards automated code generation, with LLM assistants and LLM-powered code editors at the forefront. The negative environmental impact of running these large AI models is certain, but questions still remain unanswered about the sustainability of LLM-generated code.

Striving to answer these questions, this paper introduces EnergyBench as a new correctness and energy-efficiency code generation LLM benchmark, which takes a holistic and systematic approach in the LLM evaluation process. This paper makes transparent what is the effect of combining different programming problems, programming languages and prompting strategies on the ability of LLMs to generate correct and energy-efficient code.

EnergyBench start with the process of taxonomizing three key areas: *1) Scenarios, 2) Adaptations* and *3) Metrics.* A Scenario contains detailed attributed related to a programming problem, how code must be executed, the programming language, and the available libraries and operating system dependencies that LLM-generated code must use to successfully run. Adaptation refers to the process of transforming a scenario into a prompt by converting its attributes into a set of directives, which can be systematically processed by an LLM. The taxonomy ends with the definition of the collected metrics which involve three complementary dimensions: correctness, energy and runtime. LLMs are evaluated by executing runs on different scenarios, using different adaptations and collecting different metrics. Using the metrics, the energy-efficiency of LLM-generated code can me measured, and LLMs can be compared against each other using a formula called the *Green Score*.

To demonstrate the importance of the holistic and systematic approach of EnergyBench, a set of experiments is defined consisting of 5 adaptations, 4 problems, 5 implementations, and 7 LLMs from different vendors. In addition, a set of tweaked scenarios for the worst performing problem is added to demonstrate the impact on Green Score when changes in scenario attributes are made. Altogether, the set of experiments in this paper amounts to 840, and offers a good start in exploring the effect that the interplay of different LLM prompt components has on the energy-efficiency of solutions.

From the experiments done using EnergyBench, only the closed source models are able to generate solutions that are both more accurate and energy-efficient when using optimization instructions, improving efficiency by up to 91.9%. The approach that EnergyBench takes identifies untested prompts that yield the most accurate and energy-efficient results. A tweaked experiment reveals that LLMs are highly sensitive to the content found in prompts: deleting half of a programming problem's description and removing the dependency version numbers led to GPT-4o achieving more than double the accuracy and over four times the energy efficiency. However, in all other cases, these values dropped significantly. Moreover, this paper compares LLM-generated solutions to human-optimized code, showing that LLMs lag considerably behind human solutions. The only notable exception comes from one of OpenAI's reasoning models, o3, which suggests that LLMs have the potential to surpass humans in the future for the task of correct and energy-efficient coding.

EnergyBench is implemented as a modular and extensible benchmark framework. Future work consists of reaching a broader evaluation of scenarios, adaptations and LLMs by adding new test cases. In addition, EnergyBench can be further developed to include the LLM hyperparameters as tunable components in the evaluation process, such as temperature, reasoning effort, and output budget tokens, which are elements that can greatly affect an LLM's performance.

EnergyBench: A Holistic and Systematic Benchmark for Measuring the Correctness and Energy-Efficiency of LLM-Generated Code

Dragoș Ionescu
Dept. of Computer Science
Aalborg University
Aalborg, Nordjylland, Denmark
diones23@student.aau.dk

Abstract*—The growing adoption of Artificial Intelligence (AI) is a primary contributor to increased energy demands in data centers, raising environmental concerns for the Information, Communication, and Technology (ICT) sector as a whole. In the last five years, advances in areas like natural language processing and computer vision have significantly increased the global adoption of AI. An important topic of research lies at the intersection of Large Language Models (LLMs) with modern software development. The software industry is currently facing a steady transformation towards automated code generation, with LLM assistants and LLM-powered code editors at the forefront. The negative environmental impact of running these large AI models is certain, but questions still remain unanswered about the sustainability of LLM-generated code. Current benchmarks show that LLMs have the ability to generate energy-efficient code when given optimization prompts. However, the results remain unclear when arguing for which problems, programming languages, and prompting strategies lead to the best trade-off between accuracy and energy efficiency. To address these gaps, EnergyBench is introduced—a benchmarking framework that takes a holistic and systematic approach to analyzing the elements that most impact the ability of LLMs to generate correct and energy-efficient code. Experiments show that prompts focused on reducing two key performance metrics make code more energy-efficient by as much as 91.9% in 5 out of the 7 tested LLMs, though this comes at the cost of lowered overall accuracy. Additional experiments show that the systematic approach EnergyBench takes reveals gaps unexplored by the current state-of-the-art. LLMs are highly sensitive to prompt contents: two tweaks made when defining a programming problem in an LLM prompt have drastic and opposite effects on both accuracy and energy efficiency. In one case, efficiency is increased by more than 4×, while in another, accuracy drops to zero. EnergyBench's implementation is released to the public, inviting the community to contribute to the existing set of tests in the hopes that a broader, more detailed view of the sustainability of LLM coding is reached.

1 Introduction

The Information, Communication, and Technology (ICT) sector is a fast-growing source of greenhouse gas (GHG) emissions. ICT is a broad sector encompassing personal computing devices like laptops and smartphones, or infrastructure like data centers and networks—all being key elements that enable digital communication. ICT supports businesses and the lives of people globally, and was responsible for 1.4–4% of total GHG emissions in 2020 [38]. A recent report made by Ericsson [18] estimates that the ICT sector released around 750 million metric tons of carbon dioxide (tCO_2e) into the atmosphere in 2023. In 2025, the European Union's (EU) Rolling Plan for ICT Standardization highlighted the sector's growing environmental footprint, noting that the energy demand for data centers alone is expected to reach 3.2% of the EU's total energy consumption in 2030, a concerning increase of 28% from 2018 [19]. In 2020, France reported a total of 17.2 million tCO_2e emissions from the ICT sector [7], with data centers and networks amounting to 3.6 million tCO_2e. These measurements are projected to increase to 25 million tons by 2030.

Global objectives to reduce these metrics have gained momentum, thanks to initiatives like EU's 2030 climate targets [20]. According to the EU's Rolling Plan for ICT Standardization, digital transformation has the chance to reduce Europe's total GHG emissions by 15–20% in the near future [19]. To reach these goals, there is a need for highly efficient data centers and an energy-aware digital infrastructure.

However, it remains uncertain whether these projections can be met in light of the recent emerging trends in Artificial Intelligence (AI) technologies, particularly in areas such as natural language processing and computer vision. These new AI models require significant computational resources for training and inference, which uses a substantial amount of energy. For example, USA's Data Center Energy Usage Report identifies that total data center energy demands have doubled between 2017 and 2023 due to demands in AI technology [39]. The negative environmental impact of these new technologies is clear and presents new challenges in reducing the carbon footprint of the ICT sector.

```python
# Power Hungry Average Calculator
import pandas as pd
import sys
df = pd.read_csv(sys.stdin)
means = df.mean(numeric_only=True)
print(means)
```

```python
# Optimized for Low Energy Consumption
import sys, csv
from collections import defaultdict
totals = defaultdict(float)
counts = defaultdict(int)
for row in csv.DictReader(sys.stdin):
    for col, val in row.items():
        try:
            totals[col] += (num := float(val))
            counts[col] += 1
        except (ValueError, TypeError):
            pass
for col in totals:
    print(f"{col:>10}
    ↪  {totals[col]/counts[col]:10.3f}")
```

Fig. 1: Two OpenAIGPT-4o solutions for the same problem. Left solution is the version where no optimization instructions were used to generate it. Right solution was generated when pairing the instruction *"Make your code energy-efficient"* with the problem description.

A rather unexplored topic is the environmental impact AI has in regards to its growing role in modern software development. With the advent of Large Language Models (LLMs), developers now have access to AI-powered assistants proficient at tackling a wide range of coding tasks. Code generation, debugging, and problem-solving are among the primary LLM use cases, with 76% of developers currently using or planning to use AI tools in their development process [49]. The reliance on tools that integrate these powerful features (e.g., editors like GitHub Copilot [23] or Cursor [37]) becomes more established in the industry as time passes. This drastic shift in how software is developed carries serious environmental concerns.

Consider a junior data scientist tasked with displaying the averages of every column in a table. If prompted with this problem, OpenAI's GPT-4o [44] will return a perfectly functional solution, as shown in Fig. 1 (Left). However, it uses up to 14.7× more energy than the modestly optimized version in Fig. 1 (Right), which is generated by the same model (the measurements are shown in App. A). If the same unoptimized solution is executed on multiple tables in the same day, the cumulative energy impact becomes significant. Assuming that the unoptimized code is executed 100 times on a similar-sized data input, it uses approximately 1.5 kJ more energy than the optimized version. The wasted energy is equivalent to powering a 60 W light bulb for 25 seconds.

The ability of LLMs to consistently generate energy-efficient code remains a difficult and uncertain question to answer [11, 50, 56], and providing evidence to support this ability is highly valuable. Currently, it is not well understood what are the effects of combining different programming problems, programming languages, and prompting strategies on the energy use of LLM-

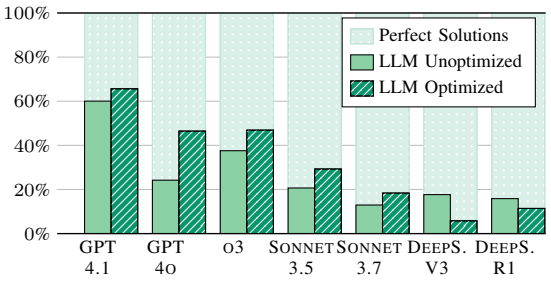

Fig. 2: Energy efficiency of 7 top-performing LLMs when prompted with optimization instructions versus standard, unoptimized instructions. Only closed source models improve in energy-efficiency when prompted for optimizations. Code produced by OpenAI's GPT-4o has the largest gain compared to unoptimized alternatives.

generated code. Recent studies show inconsistent results: LLMs can sometimes generate remarkably energy-efficient solutions compared to unoptimized code [15], while other experiments show that explicitly prompting for optimized code can increase energy use [11]. LLMs have shown impressive ability to tackle many aspects of code generation [12, 30], particularly in the realm of functional-level correctness [13]. However, it remains unclear whether adding energy-efficiency constraints on top of existing prompts creates a trade-off that compromises both correctness and energy consumption.

To address the aforementioned gaps in current research, EnergyBench is introduced as a new correctness and energy-efficiency code generation LLM benchmark, which takes a holistic and systematic approach in the LLM evaluation process. This paper makes transparent what is the effect of combining different programming problems, programming languages and prompting strate-

gies on the ability of LLMs to generate correct and energy-efficient code. From the experiments done using EnergyBench, only the closed source models are able to generate solutions that are both more accurate and energy-efficient when using optimization instructions, improving efficiency by up to 91.9%, as shown in Fig. 2. The approach that EnergyBench takes identifies untested prompts that yield the most accurate and energy-efficient results. A tweaked experiment reveals that LLMs are highly sensitive to the content found in prompts: deleting half of a programming problem's description and removing the dependency version numbers led to GPT-4o achieving more than 2× the accuracy and over 4× the energy efficiency. However, in all other cases, these values dropped significantly. Moreover, this paper compares LLM-generated solutions to human-optimized code, showing that LLMs lag considerably behind human solutions. The only notable exception comes from one of OpenAI's reasoning models, o3, which suggests that LLMs have the potential to surpass humans in the future for the task of correct and energy-efficient coding.

Key Contributions: EnergyBench is introduced in Sec. 4, a framework made to benchmark LLMs abilities at generating correct and energy-efficient source code in any programming language. The framework takes a holistic approach to evaluate this ability, by systematically analyzing the components found in code generation prompts that affect energy efficiency the most.

An implementation of EnergyBench is introduced in Sec. 4.6: a command-line utility written in Python, designed to automate the evaluation process used in the framework. In addition, it is used to compare how well LLMs fare against each other and against human-optimized solutions. The implementation is released to the community in an open source repository [27]. It features a modular and extensible framework which enables the continuous evaluation of future LLMs.

Evaluation of 7 top-performing LLMs from three different vendors (OpenAI, Anthropic and DeepSeek) in Sec. 5. The experiments include 5 closed source and 2 open source models, and their evaluation is facilitated by EnergyBench and its implementation. The set of experiments in this paper is comprised of 840 individual runs, which use 4 challenging problems across 5 programming language implementations. In addition, several LLM instructions and prompting strategies are tested in Sec. 5.1, including optimization instructions and one-shot examples respectively. The results in Sec. 5.2 demonstrate that all evaluated closed source LLMs have the ability to improve the energy efficiency of generated code by using one of these setups.

2 Background

Measuring Energy: In the literature, two main approaches for measuring the overall energy consumption of a computer are commonly cited [21, 28]: the use of external power meters to measure the current draw of the entire computer or individual components [8], and the use of built-in sensors that don't require any additional hardware.

For measuring the energy consumption of software, *Running Average Power Limit (RAPL)* [26] is a built-in energy measuring technology that has gained widespread adoption [31, 51, 52] thanks to its convenience and integration into the CPU. RAPL is a technology available on Intel and most AMD CPUs, and exposes energy consumption metrics for several parts of the computer [54]. The CPU package is the only metric universally available on all RAPL-supported platforms. Depending on the CPU vendor and model, additional metrics are exposed. These include directly attached main memory, CPU cores, integrated GPU, and an overall platform consumption [26]. *Perf* [53] is a command-line tool available for Linux which allows users to collect the aforementioned RAPL energy metrics, as well as other metrics related to the runtime performance of a process, via *Perf Events*. Fig. 3 shows the energy consumption of three energy metrics for the unoptimized version of the power hungry calculator in Fig. 1 (Left).

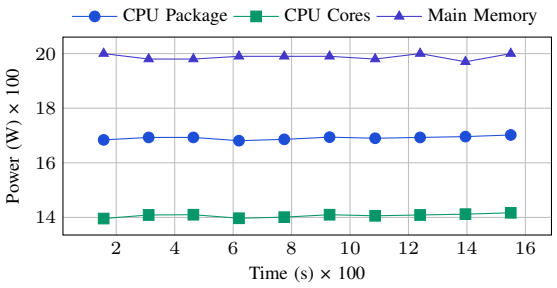

Fig. 3: Line chart showing the power consumption of the unoptimized average calculator in Fig. 1 (Left). Chart shows three energy metrics captured by Perf. CPU Package includes CPU Cores.

The accuracy of modern RAPL is generally considered high, and correlated with external meters, with studies reporting a strong Pearson correlation coefficient of around 0.95 when implemented in software measuring tools [21, 28]. RAPL stores its energy metrics in unsigned 32-bit CPU registers. These registers are updated approximately every millisecond and wrap around every minute on a heavy workload [26], which leads to a loss of accuracy if not accounted for. In this work, Perf is used to collect RAPL metrics using a high

sampling rate to minimize issues related to accuracy and robustness commonly encountered when using RAPL [54]. However, no specific requirements about how to collect energy metrics is enforced throughout this paper, recognizing that RAPL is less accurate than external meters, particularly when measuring the energy use of memory hardware [2]. A disadvantage of using RAPL, and by extension the tools that use it internally, is that energy measurements cannot be performed on a per-process basis, only on the whole computer. This makes interference from other background processes very likely while measuring.

LLM Benchmarks for Coding: The task of *Code Generation* is the ability of an LLM to output source code given a user description in natural language [25]. The quality of LLM-generated code has been extensively studied in recent years, with research addressing aspects such as number of introduced security exploits [63], code correctness and mathematical reasoning [43], time and space complexity [12], and the ability to solve issues commonly found in code bases hosted on GitHub [30].

A recurring challenge for these benchmarks is *saturation*, which is a phenomenon encountered when LLMs achieve scores close to 100% on benchmark test cases, making it impossible to gauge further improvements beyond the benchmark's scope. For example, both OPENAI and ANTHROPIC models achieve performances above 90% [48] on HUMANEVAL [13], which is a benchmark that tests functional correctness of code. When newer, more capable models surface, they will no longer be challenged by benchmarks like HUMANEVAL, thus requiring extended and more difficult test cases. Currently, many dimensions of evaluating LLM generated code remain overlooked [29], including the environmental impact and sustainability of the generated code.

Terminology: Several terms are used to explain related concepts throughout this paper. The *problem* refers to a programming problem, which can be solved with source code. A *solution* is the code attempting to solve a problem, either made by a human, or generated by an LLM. The *host system* refers to the computer where the solution will be executed on, for the purpose of acquiring measurements. A *scenario* stores the necessary information to describe the problem, what are its required host system dependencies, and how its solution must be executed.

3 Related Work

Related Energy Benchmarks: A number of recent papers have investigated the energy costs of executing LLM solutions. Vartziotis et al. [62] takes a comprehensive approach by collecting several *sustainability metrics* using Perf, that go beyond raw energy consumption. These include the number of Floating-Point Operations (FLOPs) and peak memory use. Solely focusing on Python, they found that prompting three commercial LLMs for energy optimization can reduce energy use by as much as 61%. However, the results are inconsistent because in many cases, these prompts have little effect or can even increase consumption.

Cursaru et al. [15] extend the evaluation scope with two programming languages (C++, JavaScript), and measure consumption using an external hardware meter. The evaluation using Code Llama shows that human written solutions generally use substantially less energy. Adding the prompt *"Make the code as energy-efficient as possible"* can reduce energy use, but these results are inconsistent and highly dependent on the choice of programming language and scenario. For example, Code Llama generates a solution for the Two Sum problem in C++ that has the single largest drop of about 250× less energy compared to when the optimization prompt is not used, while gains in other experiments are negligible.

Perhaps the broadest analysis of an emerging phenomenon among the surveyed papers is found in Solovyeva, Weidmann, and Castor [56], which evaluates 53 LeetCode problems across multiple programming languages and LLMs. The study shows that, while LLMs can generate code that uses less energy than top human solutions, their performance is generally unreliable. Notably, the largest energy savings are confined to a small subset of scenario categories, while solutions for C++ are almost always less efficient than those written by humans.

Finally, Cappendijk, Reus, and Oprescu [11] test three prompt instructions: 1) *"Give me an energy-optimized solution..."*, 2) *"Use library functions..."*, and 3) *"Use a for loop instead of a while loop..."*, all intended at reducing the computational costs of executing solutions. Both *for loop* and *library* prompts have large single energy drops of about 59%, yet they can also lead to increases by more than four times, demonstrating that no prompt is universally effective at saving energy. The study concludes with a note on the importance of covering a larger area of experimentation, considering multiple prompts and hyperparameter tuning.

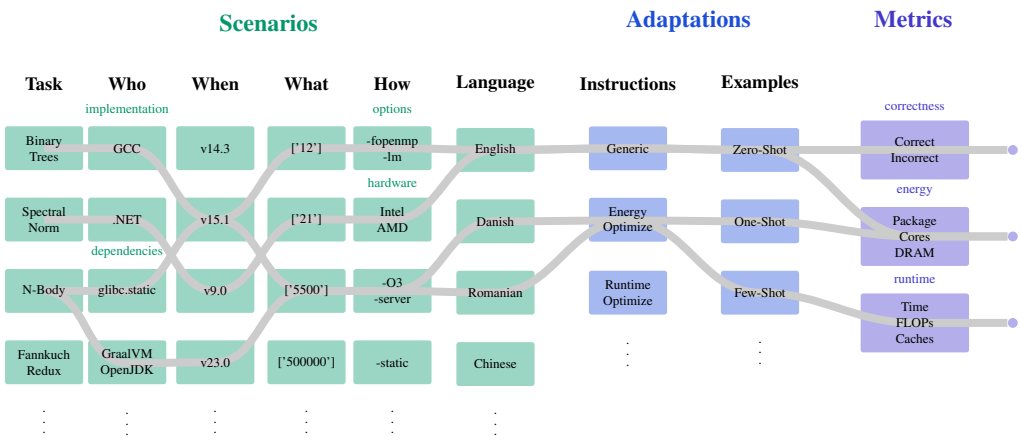

Fig. 4: Taxonomy process of ENERGYBENCH visualized, inspired by HELM [35]. Includes the three key areas and highlights the importance of analyzing gaps in the evaluation process.

Summary of Related Energy Benchmarks: Studying the related work revealed several common themes:

- **Optimized vs. Unoptimized:** Using instructions designed to produce energy-efficient or runtime-optimized solutions [11, 15, 62].
- **LLM vs. Human Optimized:** All discussed papers compare LLM solutions against human-made, optimized code to evaluate relative energy consumption.
- **Grouping Scenarios:** Some form of grouping such as computational bottlenecks [15], algorithm topic [56] or implementation difficulty [11, 62]. These groups are used to analyze the energy performance trends of scenarios that relate to each other in certain aspects.
- **Reporting More Than Just Energy:** Notably, FLOPs, peak memory use, and runtime, either as part of the sustainability metrics [62] or for direct comparisons [11].

In addition, two of the four papers explore individual unique themes, which are not considered by the others.

- **Using Different Optimization Strategies:** Various optimization prompts demonstrate to have different effects over the performance of the solution [11].
- **Host System Transparency:** An overlooked aspect identified by Solovyeva, Weidmann, and Castor [56] is the potential for LLM solutions to perform differently depending on the host system.

ENERGYBENCH builds on all the concepts explored in the related energy benchmarks, involving both the common and unique themes of each paper to reach a broader perspective than the current state-of-the-art. An evaluation method similar to Vartziotis et al. [62] is adopted, where multiple metrics beyond energy consumption are considered. In addition, model accuracy is evaluated as the ability of an LLM to produce a correct solution on the first attempt, akin to the $pass@1$ metric [13]. Comparisons against human-optimized solutions are also included. Scenarios with related attributes are grouped to achieve a better understanding of why their performance behaves in similar ways.

Holistic Benchmarks: The space of scenarios and prompting strategies involved in evaluating LLMs is vast and complex, and certain factors are often overlooked or underrepresented. For instance, the *contextual time and place* and choice of *natural language* within the prompt can significantly impact the performance of a model, yet such variables are rarely captured in standard LLM benchmarks. Liang et al. [35] address this by taking a systematic approach at describing the full range of possible scenarios, prompt strategies and metrics. They introduce HELM, a generalized framework for creating LLM benchmarks through an in-depth *taxonomy* of the components that make up an accurate LLM evaluation.

ENERGYBENCH draws inspiration from HELM and its many adopters [32, 33, 34] to analyze, with a high degree of detail, which aspects of the code generation prompt affect energy use the most. This work differentiates itself from existing energy benchmarks [15, 56, 62] by taking a holistic approach, as defined in HELM: *"The taxonomy not only facilitates the systematic selection of scenarios and metrics, but it also makes explicit what is missing."* [35].

This methodology allows gaps to be easily identified. For instance, prior work makes no use of advanced prompting strategies like few-shot or chain-of-thought (these strategies are explained in Sec. 4.4), and disregards the use of main memory from their sensor-based measure-

ments [11, 56, 62]. These shortcomings are addressed in ENERGYBENCH by a systematic review of the problems, scenarios, prompt strategies, captured metrics and host system dependencies, all being components that play a significant role in the evaluation process of an LLM for the task of energy-efficient code generation. The holistic approach in this work allows for greater transparency in the evaluation process which reduces gaps related to the interplay of different components.

4 The ENERGYBENCH Framework

ENERGYBENCH starts with the process of taxonomizing key areas. Fig. 4 offers an overview of this process.

4.1 Scenario

A scenario contains detailed *attributes* related to a problem. Besides a description in natural language, it also specifies how the code must be executed, the programming language implementation to be used, and the available libraries and operating system dependencies. All attributes defined in this section are part of the prompt given to an LLM. A scenario is comprised of three high-level components:

Task: Contains `name` and `description`. They are the name and the description in natural language of the problem. Simply using the task, without additional scenario attributes, provides enough information such that an LLM can produce accurate solutions. An example of an LLM-generated solution for a scenario containing only a name and a description is shown in Fig. 13. However, many aspects of the task would be left out to interpretation.

Domain: To align the scenario to a particular programming setup, to minimize the chance of hallucinations [36, 57], and to enforce the holistic view, all scenarios must be grounded using a set of domain attributes. These attributes relate to the contextual time and place of the scenario. Moreover, they are grouped in four subcategories, each corresponding to a guiding question, three from HELM (Who, When, What) and the last being an extension introduced by ENERGYBENCH (How).

- **Who** Includes the `implementation` (e.g. C, Python, Java) and `dependencies` as the set of technologies, compilers, interpreters and runtime environments (e.g. GCC, .NET, NumPy) needed for the solution to execute.
- **When** Dependencies are frozen in time by defining their `version`. Often, dependencies have version numbers suffixed to their names (e.g. GCC 15.1, .NET 10.0, NumPy 2.2), which makes having to explicitly state the versions redundant.
- **What** Defines a set of test cases for the scenario, `tests`. Each test is a triple which includes `args`, `stdin`, and the `expected_stdout`. All attributes in a test are optional, as some scenarios may not take any arguments or input, nor output anything. When an expected output is defined in a test case, it will be used to compare against the actual output of the solution given the arguments and input of that case.
- **How** Includes `hardware` and `options`. The first attribute refers to the host system specifications which are involved in the solution execution, while the second refers to build flags used to create the executable, or the options passed to the runtime environment of the implementation. The hardware attribute is included in the hopes that LLMs will leverage the specifications to reach more efficient, host system-tailored solutions.

```yaml
name: Spectral Norm
implementation: C # Compiled implementation.
description: | # CPU-bound and technical task description.
    Write a program that computes the spectral norm of an infinite matrix A,
    where the entries follow A[i][j] = 1 / ( (i + j) * (i + j + 1) / 2 + i + 1 )
    (This defines a symmetric infinite matrix where indices start from 0.)
    Your program must:
    - Print the result with correct formatting for N = 100 to verify correctness
    - Support a larger input value like N = 5500 to evaluate performance
dependencies: [gcc14] # Must use GCC version 14.2.0.
options: [-O3, -fopenmp, -lm] # Has access to 'libm' and 'OpenMP'.
hardware: Ubuntu 22.04.5 x86_64, Intel i7-8700, 16GB RAM
--- # tests
args: [5500] # A single command-line argument.
expected_stdout: 1.274224153 # Output should be a single float.
```

Fig. 5: Example of a CPU-bound, technical scenario involving a compiled language implementation and explicit hints about the host system, in YAML format. It shows the Spectral Norm problem, which has its task description copied from *The Computer Language Benchmarks Game (CLBG)* [61].

Language: Refers to the natural language used in the task. HELM identifies that there is a disproportionate quantity of training data for LLMs in a select few languages, such as English and Chinese [35]. This component is used to reveal gaps in the ability of an LLM to generate correct and energy-efficient code in underrepresented languages.

The formal definition of the scenario can be established using the aforementioned concepts. It is a triple which makes up a central part of the LLM prompt.

$$scenario = (task, domain, language) \quad (1)$$

Fig. 5 shows an example scenario of the Spectral Norm problem [61], which is computationally intensive and challenging to solve. The scenario is grouped as CPU-bound due to its arithmetic-heavy nature, and as technical because of the specific terminology used in the task description. The next section discusses how these groups are created and argues their importance.

4.2 Scenario Groups:

Scenarios can be grouped based on attributes that share similar traits. This process provides insight into why certain attributes yield similar results across multiple scenarios. It is the responsibility of the user to discern which scenarios are related, and the groups are not limited to a fixed set. Three group definitions that users can adopt are considered.

Technicality: Is the binary classification given to a scenario based on the level of programming specificity found in the task. Non-technical users typically emphasize desired output in a goal-oriented fashion. This happens without placing particular effort into describing program flow, time complexity or data structures [41]. The difference in task specificity is illustrated in Fig. 6, for a simple text reversal program.

It can also be argued that different levels or technicality exist and have the potential to fundamentally change the solution [12, 42]. To simplify this group, *Non-Technical* tasks are those that do not explicitly state any code concepts like function definitions, control flow structures, data structures, or primitives like threads and mutexes. Descriptions that include any of these are grouped as *Technical*.

Performance Bounds: Runtime of all software is computationally bound in some regard [1, 22, 55]. Similarly, ENERGYBENCH assumes that every problem will fall under one of these bounds once a solution is implemented. Users can place scenarios into one of the following groups, solely by visually inspecting the task:

```
--- # Non-Technical Task
name: Text Flipper
description: Write a program that takes some text
↪  and flips it backwards. For example, if given
↪  'energy', it should output 'ygrene'. The output
↪  text should be colored green.
--- # Technical Task
name: String Reversal Function
description: Implement a string-reversal function
↪  with the signature 'reverse(text: str) -> str'.
↪  It must handle empty strings and display output
↪  using the color code \#00ff00.
```

Fig. 6: Non-Technical and Technical tasks describing the same string reversal problem.

CPU-Bound for programs that are limited by the CPU's processing speed, *Memory-Bound* for programs making frequent memory accesses in locations with large chunks of data which do not fit in the CPU cache effectively, or *I/O-Bound* for programs needing to spend more time reading and writing data compared to performing CPU operations.

Energy consumption and runtime efficiency is shaped by the kinds of operations a program is doing. Memory-bound tasks exhibit a large number of main memory accesses and high rates of cache misses, due to large data segments, poor spatial locality, or irregular access patterns. Similarly, I/O-bound tasks are characterized by a high number of read and write system calls, often resulting in longer lasting CPU residency in both low-power (C-states) and high-performance (P-states) due to I/O idle waiting periods and computation bursts respectively.

Execution Model: Similar to how tasks can be used to group scenarios that have the same technicality or performance bound, the choice of domain attributes can determine a separate group related to an execution model. Particularly, the pair $(implementation, options)$ derives one of the following: *Compiled* if the pair produces a native executable, *Managed* if the produced binary runs on a virtual machine or *Interpreted* if the execution process involves some version of code interpretation. This distinction is important because runtime performance and energy consumption patterns are closely linked to the chosen execution model [51, 52].

4.3 Metrics

Next, a multi-metric strategy is adopted. This section discusses three complementary dimensions: *correctness*, *energy*, and *runtime*, with a particular focus on the energy metrics. A list of proposed energy and runtime metrics, along with their mapped Perf Events, is available in Tab. 1.

Energy Metrics		
Metric	**Perf Events**	**Description**
Core (PP0)	`power/energy-cores/`	Total energy consumption of all CPU cores.
Uncore (PP1)	`power/energy-gpu/`	Energy consumption of integrated GPU.
DRAM	`power/energy-ram/`	Energy consumption of main memory directly attached to the CPU.
Package (Pkg)	`power/energy-pkg/`	Combined energy of the entire CPU package, including Core (PP0), Uncore (PP1), last-level cache (LLC), and internal I/O controllers.
Platform (Psys)	`power/energy-psys/`	Overall system energy consumption, which may include Package (Pkg), DRAM, dedicated GPUs, cooling fans, displays and disks depending on system configuration.
Runtime Metrics		
Elapsed Time	`---`	Wall-clock runtime in milliseconds (ms).
FLOPs	`fp_arith_inst_retired.scalar`	Number of floating-point operations.
Peak Memory	`---`	Maximum resident set size in megabytes (MB).
Branch/Cache Misses	`branch-misses,cache-misses`	Number of branch-prediction/cache misses.
C-State Residency	`cstate_core/c*-residency/`	Time spent in low-power CPU states (ms or %).
P-State Residency	`---`	Time spent in high-performance CPU states (ms or %).

Tab. 1: Examples of energy and runtime metrics used in LLM evaluation. Most metrics are mapped to a Perf Event. *Platform (Psys)* is the most comprehensive energy metric exposed by RAPL; it makes measuring all three computational-bound groups possible. When not available on the host system, the next closest approximation is *Package (Pkg) + DRAM*, which only makes CPU-bound and memory-bound scenarios accurate.

Correctness: Is a binary classification indicating whether all tests defined in the scenario produce the expected output. To determine correctness, each test will have its expected output compared byte by byte with the actual output of the solution. Correctness carries the most weight in the evaluation of an LLM, since scoring a highly efficient solution is irrelevant if the output is incorrect.

Energy Metrics: Are key metrics used to evaluate the energy consumption of an LLM solution. To avoid depending on external hardware meters, ENERGYBENCH only imposes the energy consumed by the entire CPU package and main memory directly attached to the CPU of the host system.

Unfortunately, CPU and RAM metrics alone will underestimate the energy used by I/O-bound scenarios, since other components are active and draw power during execution: storage devices, network interfaces, and other peripherals. Ideally, comprehensive energy measurements for I/O-bound scenarios would capture every joule. In practice, obtaining such metrics is challenging because 1) orchestrating an energy measuring setup using external hardware sensors requires extra effort [15] and is error-prone if not synchronized with the running software, and 2) built-in sensors typically do not provide these metrics. Therefore, the framework makes accurate energy measurements for I/O-bound scenarios entirely optional. If precise evaluation of I/O-bound tasks is desired, the recommended minimal set of components includes *internal CPU I/O controllers*, *network controllers*, and *disks.*

Runtime Metrics: Several additional metrics related to runtime performance are captured to provide a more comprehensive view of the solution's efficiency. These metrics, separate from raw energy consumption, are also used in the final LLM evaluation, similar to the approach used by Vartziotis et al. [62] and Cappendijk, Reus, and Oprescu [11]. A list of runtime metrics is shown in Tab. 1.

4.4 Adaptation

Adaptation is the process of transforming a scenario into a prompt by converting its attributes into a set of *directives*, which can be systematically processed by an LLM. To achieve this, a structured prompting technique is used, leveraging XML-style formatting [5, 10, 14]. Each attribute is converted into a directive by encapsulating its data in opening and closing XML tags. In addition, attributes containing sequences of items (i.e. `dependencies`, `tests` and `args`) are formatted as JSON arrays. Fig. 7 shows the Spectral Norm scenario from Fig. 5, adapted into directives using the structured format.

Structured prompting is a widely adopted prompt engineering technique [5, 10, 14] because it is effective at helping LLMs parse and organize complex prompts that involve multiple components. This approach is suitable for ENERGYBENCH because a scenario can potentially contain a lot of data, especially if multiple test cases are defined.

Instructions: The adaptation makes use of a sequence of instructions to help guide the solution generation towards meeting specific objectives, given the scenario

```
<!-- Scenario -->
<name>Spectral Norm</name>
<description>Write a program that computes the spectral norm...</description>
<implementation>C</implementation>
<dependencies>['gcc14']</dependencies>
<options>['-O3', '-fopenmp', '-lm']</options>
<hardware>Ubuntu 22.04.5 x86_64, Intel i7-8700, 16GB RAM</hardware>
<args>['5500']</args>
<expected_stdout>1.274224153</expected_stdout>
<!-- Instructions -->
- Solve the programming problem above.
- Solution must be production-ready.
- Solution must prioritize energy efficiency above all else for the provided hardware specs.
<!-- One-Shot Example -->
<name>Matrix Multiplication</name>
<description>Multiply two square matrices for size N×N; print the last cell</description>
<implementation>C</implementation>
<code>double *A = aligned_alloc(64, n*n*sizeof(double));...</code>
<!-- LLM-Generated Solution -->
<code>printf("%.9f\n", spectral_game(N));...</code>
```

Fig. 7: Scenario adapted into a set of encapsulated directives, instructions containing an energy optimization, a one-shot example scenario and an LLM-generated solution at the bottom. The Spectral Norm scenario from Fig. 5 and a one-shot example of the Matrix Multiplication scenario are referenced here.

context. Initially, ENERGYBENCH users can experiment with generic instructions such as *"solve the coding problem..."* or *"solution must be production-ready..."*. Instructions provide good flexibility. For example, they can specify that the solution must be able to run multiple times in a loop, making it possible to observe changes in energy consumption over each iteration. Alternatively, a more sophisticated instruction, called chain-of-thought, can be used [64], which guides the LLM to generate a sequence of reasoning steps leading to a final solution, akin to the thought patterns of a human. Most notable is the use of the *optimization instruction*, which is an instruction aimed at reducing some metrics, either runtime performance or energy consumption. In Fig. 7, there are two generic code-generation instructions and an energy optimization instruction targeted at the host system specifications.

Examples: Refers to zero-shot, one-shot, and few-shot prompting strategies. Optionally, including an example in the adaptation increases the likelihood that the LLM aligns with the given instructions [9]. For this approach to be effective, the examples must reference relevant aspects mentioned in the instructions. For instance, when using chain-of-thought, examples suggesting a reasoning pattern must be provided: *"step 1: analyze the problem... step 2: consider a naive solution... step 3: evaluate energy efficiency and optimize code..."*. Similarly, when instructing for runtime or energy optimizations, a separate scenario including a human-optimized solution written in the same implementation as the problem scenario must be included as reference. Fig. 7 illustrates this with the Matrix Multiplication scenario provided as a one-shot example.

4.5 Evaluating Accuracy and Efficiency

Run: Is a function that takes a scenario, an adaptation and a solution produced by either an LLM or a human, which is then executed on the host system to return the tuple (C, M). $C \in \{0, 1\}$ is the correctness metric and M is either a runtime or energy metric. For human solutions, the adaptation value is null.

$$run(scenario, adaptation, solution) = (C, M) \quad (2)$$

Green Score: Is a real number that shows how well an LLM or human can produce correct solutions while also minimizing a chosen runtime or energy metric. This score is inspired by the *Green Capacity* formula found in Vartziotis et al. [62]. Given a set of runs containing solutions from either an LLM or a human, and a chosen metric, the Green Score (GS) is calculated as follows:

$$GS_M = \frac{1}{n} \sum_{i=1}^{n} \frac{C_i}{M_i + 1} \quad (3)$$

- C_i: Indicates if the solution in run i is correct (1 if correct, 0 if incorrect).
- M_i: Runtime or energy metric for run i once the solution is executed and measured.
- n: Is the total number of runs.

This function returns real numbers between $[0, 1]$ where a value of 0 means that no correct solutions are generated, and a value of 1 is the theoretical *perfect score* where all solutions are correct and the chosen metric measures 0. A higher GS is better. To facilitate the

direct comparison of multiple GS's, returned values are normalized to a common scale (e.g. by multiplying with 100).

Human Delta Percentage: Is the GS percentage difference between LLM-generated and human-made solutions, given the same metric. A positive delta percentage means that the LLM outperforms the human, while a negative percentage means that the LLM performed worse. This evaluation quantifies the performance difference by returning a concrete real number.

$$\Delta = \frac{GS^{LLM} - GS^{human}}{GS^{human}}\% \tag{4}$$

As mentioned in Sec. 4.4, LLMs can use adaptations to influence the generated solution in some regard. The most meaningful way to use the human delta percentage involves comparing three categories of solutions: 1) LLM solutions that have not used any optimization instructions, 2) LLM solutions instructed to perform a runtime or energy optimization, and 3) human-optimized solutions serving as baselines. By comparing the performance gap between each LLM approach and the human baseline, the percentage improvement provided by the adaptation in closing the human delta gap can be quantified.

4.6 Implementation

To automate the process of generating solutions, metric collection, and LLM evaluation, a Python command-line tool implementing the features of EnergyBench is developed and is made available for Linux in a public repository [27] under the MIT license. This tool effectively mitigates benchmarking saturation concerns by using a flexible implementation of scenarios, which makes it possible to change or add new problems by creating scenario files in YAML format. In addition, the collected metrics are defined in an environment file, and adaptations are modular because they can be extended by creating new Python classes that define which instructions are used in the prompt.

Technical Considerations: The tool uses the same notion of scenarios as in Sec. 4.1, implemented as YAML files. A scenario file consists of all the attributes necessary to generate, build, and execute solutions, as well as separate YAML documents used to test functional correctness. Next, the scenarios are adapted as discussed in Sec. 4.4, and fed into an LLM sequentially, or by leveraging batch processing for platforms that support it, like OpenAI and Anthropic. Solutions that yield valid source code are copied back into the scenario YAML, and the files are stored on the system. Finally, a scenario file that stores a solution can be executed, leveraging the Who and How domain attributes (i.e., `options`, `implementation`, `dependencies`).

To collect runtime and energy metrics, internally the EnergyBench tool uses Perf. Several Perf Events directly correspond to runtime and energy metrics as shown in Tab. 1, which are used to calculate the GS. The accuracy and robustness of the measurements is ensured by setting a high sampling frequency of 100Hz. The total value for any captured event using Perf is computed by summing all of its samples during one measurement.

5 Experiments

To demonstrate the importance of the holistic and systematic approach of EnergyBench, a set of experiments is defined consisting of 5 adaptations, 4 problems, 5 implementations, and 7 LLMs from different vendors. In addition, a set of tweaked scenarios for the worst performing problem is added to demonstrate the impact on Green Score when changes in scenario attributes are made. These tweaked scenarios apply to 4 out of 5 implementations. Altogether, the set of experiments amounts to *700 base runs* (5 adaptations × 4 problems × 5 implementations × 7 LLMs) and *140 runs involving tweaked scenarios* for a single problem (5 adaptations × 1 problem × 4 implementations × 7 LLMs), for a total of *840 individual experiment runs.* All chosen problems are selected from *The Computer Language Benchmarks Game (CLBG)* software project [24], which hosts a collection of challenging micro benchmark descriptions. These have been selected because they are accompanied by community-made solutions that are optimized for runtime performance, which will be used as human baselines to compare against LLM solutions.

5.1 Setup

Evaluated LLMs: The chosen LLMs are released by three vendors: OpenAI, Anthropic and DeepSeek. An overview of all evaluated models is shown in Tab. 3. Each set of LLMs from a vendor includes a single reasoning model, which can enter an extended thinking mode before generating a solution. The thinking output of these types of models is separate from the actual solution, and is not taken into account when evaluating the correctness or efficiency. Reasoning models are specially trained LLMs that use techniques like supervised learning or reinforcement learning to incentivize the chain-of-thought process described in Sec. 4.4, without the need for special prompt strategies [16]. The choice of these models is attractive because they display higher performance in mathematical and code generation tasks compared to standard LLMs [48].

Implementations			Problems		
Implementation	**Options**	**Group**	**Task**	**Description**	**Group**
C GCC 14.2.0	-O3	Compiled	Binary Trees [58]	Allocate, traverse and deallocate many trees	Mem-Bound
C++ GCC 14.2.0	-O3	Compiled	Fannkuch Redux [59]	Indexed access to tiny integer sequence	CPU-Bound
C# .NET 9.0	-c Release	Managed	N-Body [60]	Double precision N-body simulation	CPU-Bound
Java OpenJDK 23	—	Managed	Spectral Norm [61]	Eigenvalue using the power method	CPU-Bound
Java GraalVM 23	—	Managed			

Adaptations			Optimization Instructions	
Optimization	**Prompting**	**Example**	**Optimization**	**Description**
Unoptimized	Zero-Shot	—	Energy Optimized	*"CRITICAL! Your solution MUST prioritize energy-efficiency..."*
Unoptimized	One-Shot	MatMul	Runtime Optimized	*"CRITICAL! Your solution MUST be as efficient as possible..."*
Energy Optimized	Zero-Shot	—		
Energy Optimized	One-Shot	MatMul		
Runtime Optimized	Zero-Shot	—		

Tab. 2: Overview of the experimental setup. Implementations are defined along with their version numbers. Scenarios are grouped based on the choice of `(implementation, options)` pair, and task. Adaptations use an optimization instruction and a prompting strategy. The bottom-right table shows the contents of the two evaluated optimization instructions. Matrix Multiplication is abbreviated as MatMul.

All models are configured with temperature 0 to eliminate randomness in output [6]. LLMs are probabilistic models, which means that the probability of generating the next token (text word) in an output sequence is determined by the previously seen context (e.g., a human prompt or previously generated output). Configuring all LLMs to the lowest possible temperature ensures that they stick to the probabilities of their pretrained weights, making the output more deterministic and better suited for code generation. Following OPENAI'S recommendations [47], all closed source reasoning models are limited to 25,000 output tokens. For consistency, all closed source non-reasoning models are limited to 8,192 output tokens, as this represents the maximum limit supported by CLAUDE 3.5 SONNET. Moreover, O3 is the only reasoning model that accepts a reasoning effort parameter, which will be set to the default value of `medium`. DEEPSEEK'S models only support the temperature parameter.

Evaluated Problems: Each CLBG problem has its description manually cleaned to remove content unrelated to the actual task, like author notes and comments. Using the cleaned problem and an implementation, a unique scenario is created. The remaining attributes reflect up-to-date dependencies and production-level options. Finally, the scenarios are grouped as described in Sec. 4.2, based on the execution model of the chosen $(implementation, options)$ pair, and the derived performance bounds of the task description, as shown in Tab. 2.

Vendor	Model	Reasoning	Open
OPENAI	GPT-4.1 [45]	No	No
OPENAI	GPT-4O [44]	No	No
OPENAI	O3 [46]	Yes	No
ANTHROPIC	CLAUDE 3.5 SONNET [4]	No	No
ANTHROPIC	CLAUDE 3.7 SONNET [3]	Hybrid	No
DEEPSEEK	R1 [16]	Yes	Yes
DEEPSEEK	V3 [17]	No	Yes

Tab. 3: Evaluated LLMs, reasoning capabilities and open source status. For brevity in other figures, referring to the models is kept concise: GPT 4.1, GPT 4O, O3, SONNET 3.5, SONNET 3.7, DEEPS. R1, DEEPS. V3.

Evaluated Host System and Metrics: All experiments are executed on a headless Linux host system with a minimal set of background processes. The host system runs on a 64-bit architecture and an Intel i7-8700 CPU with 16 GB of RAM. See Tab. 7 for a lengthier list of specifications. To provide LLMs with context about the host system where their generated solutions will execute, scenarios are assigned the same specifications for the `hardware` attribute. Experiment runs are executed 10 times, and the results are averaged. This is done to reduce the effect that outlier data has on the Green Score, as energy measurements using Perf are unpredictable and are influenced by the activity of background processes. The captured energy metrics are Pkg and DRAM, because they accurately reflect the consumption of the chosen problems. The captured runtime metrics are Elapsed Time and FLOPs. Energy consumption is generally correlated with the time spent running a program [51]. Measuring the Elapsed Time metric offers a straightforward way to interpret the energy efficiency of different solutions. In addition, FLOPs is a widely accepted metric

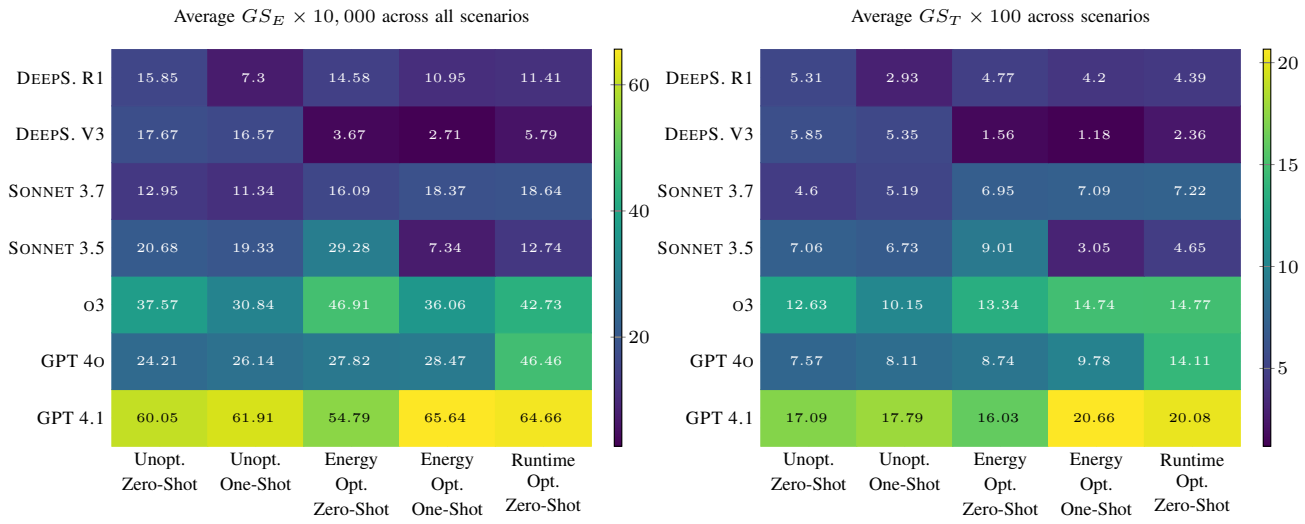

Fig. 8: Heatmaps showing the GS_E (left) and GS_T (right) of all evaluated LLMs. The results show a strong correlation between the energy consumption and the time required to execute correct solutions.

for measuring the computational demand of software [40], and energy use generally scales with this metric too. Floating-point operations are computationally expensive, and reducing their number is an effective way to lower energy use. Throughout the experiments, GS_E is the combined Green Score of the captured energy metrics Pkg + DRAM, while GS_T and GS_F are the Green Scores for the runtime metrics Elapsed Time and FLOPs, respectively.

Evaluated Adaptations: The chosen adaptations involve optimization instructions along with zero- or one-shot prompting strategies. When one-shot is used, the *Matrix Multiplication (MatMul)* scenario serves as the reference example due to its simplicity and high optimization potential. The MatMul solution is modified to reflect the specified optimization instruction. If the Energy Optimized instruction is used, the solution will be an energy-efficient version. See Tab. 2 for an overview and Fig. 15 for the source code.

Selecting Human-Optimized Solutions: A human-optimized solution is selected alongside the problem based on two criteria:

- Must be the most efficient among all candidates for the given problem.
- Must use the same hardware, dependencies and implementation defined in the scenario. By extension, it must be able to execute on the host system.

5.2 Evaluation Results

Fig. 8 compares the GS_E and GS_T of all models across each adaptation. Inspecting the left heatmap reveals that all closed source models improve their GS_E by using one of the three optimized adaptations. The largest score increases using any optimized zero-shot adaptation are seen in 3.7 Sonnet (64.37%), 3.5 Sonnet (41.59%), o3 (24.86%), and GPT-4o (91.9%), when comparing to their unoptimized alternatives. On the other hand, GPT-4.1 achieves the highest score increase with the energy optimized one-shot adaptation (6.02%), compared to the unoptimized alternative. The only exceptions are the open source DeepSeek V3 and R1 models, which universally gained lower scores using optimized adaptations compared to the best unoptimized alternatives.

A notable aspect in this part of the evaluation is the choice of optimization instruction. Prompting for runtime efficiency instead of energy efficiency increases the GS_E of GPT-4.1 by 18.01%, and of GPT-4o by 67%, showing that energy efficiency instructions are not always better at generating energy-efficient solutions. In contrast, 3.5 Sonnet achieves its largest score with an energy optimized zero-shot adaptation, which is more than 2× better than the runtime alternative. As previously shown, Fig. 2 highlights how much each model's GS_E improves when the top performing optimized and unoptimized adaptations are compared.

Surprisingly, all closed source reasoning models performed worse on average compared to a non-reasoning model from the same vendor. R1 is the only reasoning model fairing better, with a 29.48% increase in average

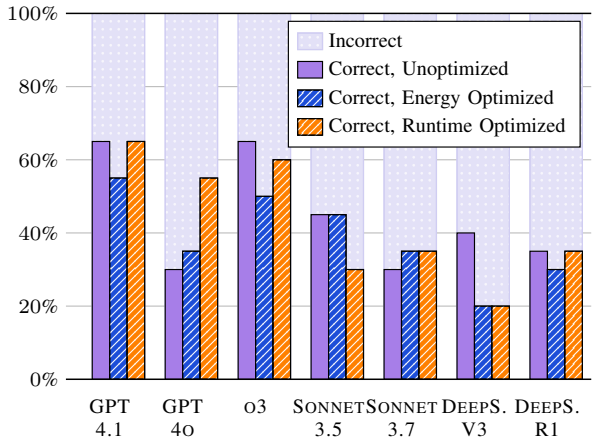

Fig. 9: Accuracy of evaluated LLMs with respect to the ratio of correct solutions over attempted runs. Striped bars indicate the use of an optimization instruction.

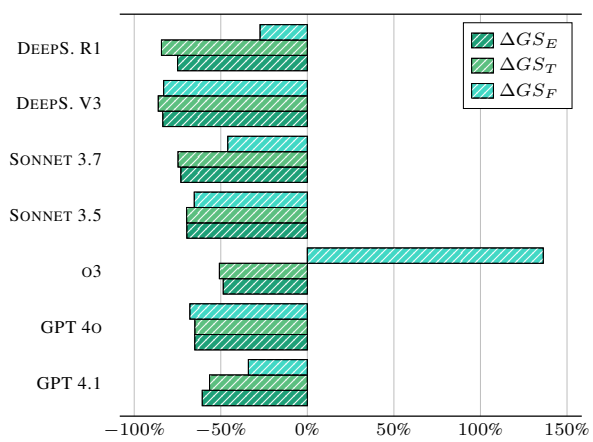

Fig. 10: Green Score Delta Percentage of evaluated LLMs compared to human-optimized solutions. Only the best-rated optimized adaptations are chosen.

GS_E compared to V3. These results are counterintuitive because reasoning models are known to perform better at tasks that require problem-solving, which is a highly relevant trait for correct and energy-efficient code generation. It seems that the added reasoning capabilities of o3, 3.7 Sonnet and R1 do little to improve the GS_E.

Model Accuracy: Fig. 9 shows the accuracy of each model with respect to the number of correct solutions it is able to generate using the zero-shot prompting strategy. The figure compares results that use unoptimized instructions against energy- and runtime-optimized instructions. GPT-4.1 and o3 are the only models to achieve at least 50% accuracy across these adaptations, while the others generally remain below 40%. Surprisingly, GPT-4o surpasses its unoptimized accuracy when using the runtime optimization instruction. This shows that the model produces more correct solutions when it is also optimizing for runtime efficiency.

Accuracy directly affects the individual Green Scores in Fig. 8, as models that generate more correct solutions achieve higher scores. This relationship is evident when comparing Fig. 8 and Fig. 9: GPT-4o achieves the highest GS_E, GS_T and accuracy when using the runtime optimization instruction. Similarly, GPT-4.1 has a strong overall performance, as illustrated by the high accuracy and consistently high values on both heatmaps.

LLM vs. Human Optimized: LLM-generated solutions remain behind human-optimized solutions, performing between 48.57% and 86.11% worse on average for ΔGS_E and ΔGS_T as shown in Fig. 10. The GS_F score is the only evaluated metric with the largest number of positive delta instances (14) found across all adaptations, problems, and LLMs, though they are rare and far between. In Tab. 4, three models are highlighted showing positive GS_F deltas for the Binary Trees and N-Body problems.

Model	Problem	Adaptation	ΔGS_F
GPT 4.1	Binary Trees	Energy Opt. One-Shot	+23.73%
o3	N-Body	Runtime Opt. Zero-Shot	+3.61%
Sonnet 3.5	N-Body	Energy Opt. One-Shot	+2.82%

Tab. 4: Three instances where the GS_F has positive delta values across models, problems and adaptations. All scenarios of a problem are averaged in this table.

There are only two notable exceptions in all experiments. In Tab. 5, o3 generates solutions for the Binary Trees problem that outperforms the human-optimized baseline across two different adaptations.

Adaptation	ΔGS_E	ΔGS_T	ΔGS_F
Energy Opt. Zero-Shot	+4.73%	-40.79%	+216.79%
Energy Opt.One-Shot	+18.68%	+6.53%	+743.47%

Tab. 5: o3 scoring better energy efficiency and lower FLOPs count than the human-optimized baseline for the Binary Trees problem.

Although these single successes are overshadowed by the lower performance across all other scenarios, o3 achieves the highest average delta against human solutions. This demonstrates the model's ability to produce optimized solutions that come closest to the human-optimized baseline.

Tweaked Experiment: Out of all evaluated problems, Fannkuch Redux performs the worst, with the lowest average GS_E and the second-lowest average accuracy across all runs, as shown in Tab. 6. This last experiment explores the effects on the performance of Fannkuch Redux when applying *two tweaks* to the scenario attributes. First, the dependency version numbers are removed, using the intuition that LLMs are typically trained on source code that does not leverage the most up-to-date features of the specified dependencies. If this hypothesis is correct, solutions will be at least more accurate because LLMs will not generate code using the dependencies they have not been trained on. Second, half of the task description is omitted because it consists of a lengthy list of implementation guidelines, which are not essential to the task description itself. A new set of scenarios is created using these tweaks for 4 out of 5 implementations (C, C++, C#, and Java with OpenJDK).

Problem	Avg. Correct	Avg. GS_E
Binary Trees	81.71%	34.09%
Fannkuch Redux	19.42%	2.11%
N-Body	41.71%	32.2%
Spectral Norm	18.85%	37.59%

Tab. 6: Average accuracy and GS_E of evaluated problems across all LLM runs, normalized to a scale of 100 and 10,000, respectively. All scenarios of a problem are averaged in this table.

Fig. 11 shows that the tweaked version of Fannkuch Redux only increases the accuracy and GS_E for GPT-4O by more 2× and 4× respectively, and V3 which only increases the GS_E by 14.99%. All other models show that the tweaks do not improve accuracy or the Green Score, and in many cases it decreases them. A concerning result appears for 3.5 SONNET, as its accuracy drops to zero after the tweaks, indicating that at least a full task description and dependency versions are needed to generate some correct solutions. This experiment further demonstrates the importance of analyzing each scenario attribute carefully, as even minor changes can have a great impact on the perceived performance of LLMs for correct and energy-efficient code generation.

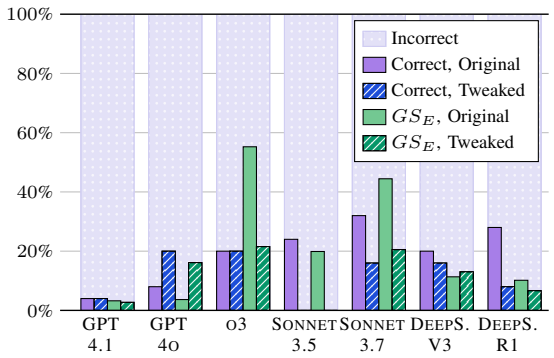

Fig. 11: LLM accuracy and GS_E for the tweaked version of Fannkuch Redux, which has half of its description and all dependency version numbers removed. GPT-4O and V3 are the only models benefiting from this change.

6 Conclusion

The environmental impact of the ICT sector within the next five years are expected to rise, with the energy demand of data centers being a primary driver. The energy cost of operating new AI models and the widespread adoption of LLMs in modern software development has the potential to slow down global climate change initiatives considerably.

Motivated by these concerns, this paper introduces ENERGYBENCH, a benchmark framework capable of evaluating the correctness and energy-efficiency of LLM-generated code. To expand the current state-of-the-art in energy-efficiency benchmarks for code generation, ENERGYBENCH adopts a holistic and systematic approach in the evaluation process. This approach identifies with a high degree of granularity, what are the elements of an LLM prompt that have the biggest impact on the generation of correct and energy-efficient solutions. Experiments reveal large shifts when two tweaks are made to the definition of a programming problem. OPENAI'S GPT-4O achieves 4× better energy-efficiency when half of the information found in the task description is removed, while ANTHROPIC'S CLAUDE 3.5 SONNET accuracy drops to zero.

Additional experiments show that, out of 7 evaluated LLMs, 5 have the ability to generate correct and energy-efficient code when being prompted with an optimization instruction. The improvements measure up to 91.9% compared to LLM-generated solutions that are not optimized. However, LLMs tend to decrease in overall accuracy when these optimizations are used, indicating that the added complexity of energy efficiency prompts makes it harder for LLMs to correctly solve problems. Finally, LLMs are almost universally worse at generating energy-efficient solutions compared to humans, with

only one notable exception. OpenAI's o3 reasoning model was capable of beating human-optimized solutions in the Binary Trees problem by as much as 18.68%. This result would suggest that ongoing LLM advances can close this gap.

EnergyBench is implemented as a modular and extensible benchmark framework. Future work consists of reaching a broader evaluation of scenarios, adaptations and LLMs by adding new test cases. In addition, EnergyBench can be further developed to include the LLM hyperparameters as tunable components in the evaluation process, such as temperature, reasoning effort, and output budget tokens, which are elements that can greatly affect an LLM's performance.

References

[1] Martin Abadi et al. "Moderately hard, memory-bound functions". In: *ACM Transactions on Internet Technology (TOIT)* 5.2 (2005), pp. 299–327.

[2] Lukas Alt et al. "An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. London, United Kingdom: Association for Computing Machinery, 2024, pp. 71–82. ISBN: 9798400704444. DOI: 10.1145/3629526.3645052. URL: https://doi.org/10.1145/3629526.3645052.

[3] Anthropic. *Claude 3.7 Sonnet System Card*. https://docs.anthropic.com/en/resources/claude-3-7-system-card. Accessed: 2025-05-28. 2025.

[4] Anthropic. *Introducing Claude 3.5 Sonnet*. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2025-05-28. 2024.

[5] Anthropic. *Use XML tags to structure prompts*. https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/use-xml-tags. Accessed: 2025-06-02. 2024.

[6] Chetan Arora et al. *Optimizing Large Language Model Hyperparameters for Code Generation*. 2024. arXiv: 2408.10577 [cs.SE]. URL: https://arxiv.org/abs/2408.10577.

[7] World Bank and International Telecommunication Union. *Measuring the Emissions & Energy Footprint of the ICT Sector: Implications for Climate Action*. Tech. rep. World Bank and ITU, 2024. URL: https://documents1.worldbank.org/curated/en/099121223165540890/pdf/P17859712a98880541a4b71d57876048abb.pdf.

[8] Daniel Bedard et al. "PowerMon: Fine-grained and integrated power monitoring for commodity computer systems". In: *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*. 2010, pp. 479–484. DOI: 10.1109/SECON.2010.5453824.

[9] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[10] Steve Campbell. *Better LLM Prompts Using XML*. https://www.aecyberpro.com/blog/general/2024-10-20-Better-LLM-Prompts-Using-XML/. Accessed: 2025-06-02. 2024.

[11] Tom Cappendijk, Pepijn de Reus, and Ana Oprescu. *Generating Energy-efficient code with LLMs*. 2024. arXiv: 2411.10599 [cs.SE]. URL: https://arxiv.org/abs/2411.10599.

[12] Pierre Chambon et al. *BigO(Bench) – Can LLMs Generate Code with Controlled Time and Space Complexity?* 2025. arXiv: 2503.15242 [cs.CL]. URL: https://arxiv.org/abs/2503.15242.

[13] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[14] Google Cloud. *Structure prompts for generative AI models*. https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/structure-prompts. Accessed: 2025-06-02. 2024.

[15] Vlad-Andrei Cursaru et al. *A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama*. 2024. arXiv: 2405.03616 [cs.SE]. URL: https://arxiv.org/abs/2405.03616.

[16] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[17] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2024. arXiv: 2412.19437 [cs.CL].

[18] Ericsson. *ICT Sustainability: Shrinking the sector's carbon footprint*. Ericsson Mobility Report, November 2024. 2024. URL: https://www.ericsson.com/4acd55/assets/local/reports-papers/mobility-report/documents/2024/emr-november-2024-ict-sustainability-article.pdf.

[19] European Commission. *ICT Environmental Impact (RP2025)*. https://interoperable-europe.ec.europa.eu/collection/rolling-plan-ict-standardisation/ict-environmental-impact-rp2025. Accessed: 2025-06-03. 2025.

[20] European Commission. *Progress on climate action*. Accessed: 2025-06-03. European Commission. 2025. URL: https://climate.ec.europa.eu/eu-action/climate-strategies-targets/progress-climate-action_en.

[21] David Freina, Matthijs Jansen, and Animesh Trivedi. *A Survey of Energy Measurement Methodologies for Computer Systems*. https://

atlarge-research.com/pdfs/2024-dfreina-litsurvey.pdf. 2024.

[22] PS Gill. *Operating systems concepts*. Firewall Media, 2006.

[23] GitHub. *The State of Generative AI in 2024*. Accessed: 2025-05-29. 2024. URL: https://github.blog/news-insights/octoverse/octoverse-2024/#the-state-of-generative-ai-in-2024.

[24] Isaac Gouy and Brent Fulgham. *The Computer Language Benchmarks Game*. Version 25.03, accessed 29 May 2025. 2025. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html (visited on 05/29/2025).

[25] Nam Huynh and Beiyu Lin. *Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications*. 2025. arXiv: 2503.01245 [cs.SE]. URL: https://arxiv.org/abs/2503.01245.

[26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel. Dec. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[27] Dragoș Ionescu. *energy-bench: A novel LLM evaluation benchmark focusing on energy efficient and correct code generation*. Accessed: 2025-05-25. 2025. URL: https://github.com/diones23/energy-bench.

[28] Mathilde Jay et al. "An experimental comparison of software-based power meters: focus on CPU and GPU". In: *CCGrid 2023 - 23rd IEEE/ACM international symposium on cluster, cloud and internet computing*. Bangalore, India: IEEE, 2023, pp. 1–13. DOI: 10.1109/CCGrid57682.2023.00020. URL: https://inria.hal.science/hal-04030223.

[29] Juyong Jiang et al. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL]. URL: https://arxiv.org/abs/2406.00515.

[30] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770.

[31] Nicolas van Kempen et al. *It's Not Easy Being Green: On the Energy Efficiency of Programming Languages*. 2025. arXiv: 2410.05460 [cs.PL]. URL: https://arxiv.org/abs/2410.05460.

[32] Tony Lee et al. *Holistic Evaluation of Text-To-Image Models*. 2023. arXiv: 2311.04287 [cs.CV]. URL: https://arxiv.org/abs/2311.04287.

[33] Tony Lee et al. *VHELM: A Holistic Evaluation of Vision Language Models*. 2024. arXiv: 2410.07112 [cs.CV]. URL: https://arxiv.org/abs/2410.07112.

[34] Yanyang Li et al. *CLEVA: Chinese Language Models EVAluation Platform*. 2023. arXiv: 2308.04813 [cs.CL]. URL: https://arxiv.org/abs/2308.04813.

[35] Percy Liang et al. "Holistic Evaluation of Language Models". In: *Transactions on Machine Learning Research* (Oct. 2023). arXiv: 2211.09110 [cs.CL]. URL: https://arxiv.org/abs/2211.09110.

[36] Fang Liu et al. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. 2024. arXiv: 2404.00971 [cs.SE]. URL: https://arxiv.org/abs/2404.00971.

[37] Analytics India Magazine. *Cursor is the Fastest Growing SaaS in History*. Accessed: 2025-06-04. 2024. URL: https://analyticsindiamag.com/ai-features/cursor-is-the-fastest-growing-saas-in-history/.

[38] Jens Malmodin et al. "ICT sector electricity consumption and greenhouse gas emissions – 2020 outcome". In: *Telecommunications Policy* 48.3 (2024), p. 102701. ISSN: 0308-5961. DOI: https://doi.org/10.1016/j.telpol.2023.102701. URL: https://www.sciencedirect.com/science/article/pii/S0308596123002124.

[39] Eric Masanet et al. *United States Data Center Energy Usage Report*. Tech. rep. LBNL-2001836. Lawrence Berkeley National Laboratory, 2024. URL: https://eta-publications.lbl.gov/sites/default/files/2024-12/lbnl-2024-united-states-data-center-energy-usage-report.pdf.

[40] Hans Meuer et al. *Green500 List - November 2024*. https://www.top500.org/lists/green500/list/2024/11/. Accessed: 2025-06-05.

[41] Asaf Achi Mordechai, Yoav Goldberg, and Reut Tsarfaty. *NoviCode: Generating Programs from Natural Language Utterances by Novices*. 2024. arXiv: 2407.10626 [cs.CL]. URL: https://arxiv.org/abs/2407.10626.

[42] Lincoln Murr, Morgan Grainger, and David Gao. *Testing LLMs on Code Generation with Varying Levels of Prompt Specificity*. 2023. arXiv: 2311.07599 [cs.SE]. URL: https://arxiv.org/abs/2311.07599.

[43] Ansong Ni et al. *L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models*. 2023. arXiv: 2309.17446 [cs.CL]. URL: https://arxiv.org/abs/2309.17446.

[44] OpenAI. "GPT-4o System Card". In: *arXiv preprint arXiv:2410.21276* (2024). URL: https://arxiv.org/abs/2410.21276.

[45] OpenAI. *Introducing GPT-4.1 in the API*. https://openai.com/index/gpt-4-1/. Accessed: 2025-05-28. Apr. 2025.

[46] OpenAI. *Introducing OpenAI o3 and o4-mini*. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-05-28. Apr. 2025.

[47] OpenAI. *Reasoning models*. https://platform.openai.com/docs/guides/reasoning?api-mode=chat#allocating-space-for-reasoning. Accessed: 2025-05-29. 2025.

[48] OpenAI. *simple-evals: OpenAI Evaluation Framework and Benchmarks*. https://github.com/openai/simple-evals. Accessed: 2025-05-22.

[49] Stack Overflow. *2024 Developer Survey: AI Sentiment and Usage*. Accessed: 2025-05-29. 2024. URL: https://survey.stackoverflow.co/2024/ai#sentiment-and-usage-ai-select.

[50] Huiyun Peng et al. *Large Language Models for Energy-Efficient Code: Emerging Results and Future Directions*. 2024. arXiv: 2410.09241 [cs.SE]. URL: https://arxiv.org/abs/2410.09241.

[51] Rui Pereira et al. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 9781450355254. DOI: 10.1145/3136014.3136031. URL: https://doi-org.zorac.aub.aau.dk/10.1145/3136014.3136031.

[52] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[53] PerfWiki Contributors. *PerfWiki*. Accessed: 2025-05-26. 2025. URL: https://perfwiki.github.io/main/.

[54] Guillaume Raffin and Denis Trystram. *Dissecting the software-based measurement of CPU energy consumption: a comparative analysis*. 2024. arXiv: 2401.15985 [cs.DC]. URL: https://arxiv.org/abs/2401.15985.

[55] K. Salah et al. "Mitigating starvation of Linux CPU-bound processes in the presence of network I/O". In: *Journal of Systems and Software* 85.8 (2012), pp. 1899–1914. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.02.042. URL: https://www.sciencedirect.com/science/article/pii/S0164121212000660.

[56] Lola Solovyeva, Sophie Weidmann, and Fernando Castor. *AI-Powered, But Power-Hungry? Energy Efficiency of LLM-Generated Code*. 2025. arXiv: 2502.02412 [cs.SE]. URL: https://arxiv.org/abs/2502.02412.

[57] Joseph Spracklen et al. *We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs*. 2025. arXiv: 2406.10279 [cs.SE]. URL: https://arxiv.org/abs/2406.10279.

[58] The Computer Language Benchmarks Game. *Binary Trees*. Accessed: 2025-05-04. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees.

[59] The Computer Language Benchmarks Game. *Fannkuch Redux*. Accessed: 2025-05-04. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux.

[60] The Computer Language Benchmarks Game. *N-Body*. Accessed: 2025-05-04. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html#nbody.

[61] The Computer Language Benchmarks Game. *Spectral Norm*. Accessed: 2025-05-04. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/spectralnorm.html#spectralnorm.

[62] Tina Vartziotis et al. *Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation*. 2024. arXiv: 2403.03344 [cs.SE]. URL: https://arxiv.org/abs/2403.03344.

[63] Mark Vero et al. *BaxBench: Can LLMs Generate Correct and Secure Backends?* 2025. arXiv: 2502.11844 [cs.CR]. URL: https://arxiv.org/abs/2502.11844.

[64] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

Appendix

A Optimizing the Power Hungry Average Calculator

The simple average calculator in Fig. 1 is power hungry because it depends on Pandas and invokes `pandas.read_csv`, which loads the entire dataset into memory and calculates the average all at once using `pandas.DataFrame.mean`. The optimized version reads and processes one row at a time to calculate the average.

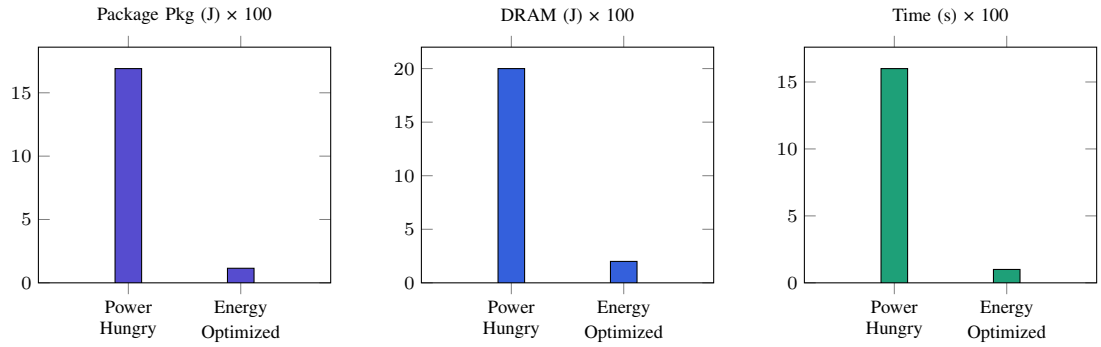

Fig. 12: Direct metric comparison between the power hungry calculator and the optimized version. To conduct this experiment, 100 iterations were executed and averaged for both versions using the same setup in Sec. 5.1.

B Generating Solutions Without Grounding an LLM with Domain Attributes

In isolation, a task contains sufficient data such that, by passing it to an LLM without specifying any other attributes, it could generate a correct solution. See Fig. 13. However, two main problems arise from this lack of specificity. First, unpredictable solutions are to be expected from vague descriptions [25, 42], especially when key aspects such as desired programming language or required libraries are missing. Unpredictable solutions are those where it can't be determined with a high degree of certainty, how to execute them before prompting the LLM. The main problem arises from the fact that plain descriptions offer no clues about the technologies required to execute the code. Second, it is well documented that current models are prone to hallucinations [36, 57]—the phenomenon where models reference non-existent functions or libraries.

```
name: Fibonacci Finder
descrtiption: |
    Given a number, the program should determine whether it appears in the
    Fibonacci sequence. If it does, the program should return its position in the sequence;
    otherwise, it should indicate that the number is not part of the sequence.
code: |
    def find_fibonacci_position(n):
        if n < 0:
            return -1
        a, b = 0, 1
        position = 1
        while a < n:
            a, b = b, a + b
            position += 1
        return position if a == n else -1
    number = int(input("Enter a number: "))
    position = find_fibonacci_position(number)
    if position != -1:
        print(f"{number} is at position {position} in the Fibonacci sequence.")
    else:
        print(f"{number} is not in the Fibonacci sequence.")
```

Fig. 13: Example of a scenario which had its solution generated by OpenAI's GPT-4o, only using the `name` and `description` attributes. The solution is written in Python even though a programming language is never explicitly mentioned.

C Host System Specifications

Tab. 7 shows a partial list of the most important components involved in executing experiments.

Component	Specification
OS	Ubuntu 22.04.5 LTS x86_64
Host	Komplett PC
Kernel	6.8.0-52-generic
Shell	bash 5.3.0(1)-beta
CPU	Intel i7-8700 (12 cores) @ 800 MHz–4.6 GHz
CPU Governor	performance
CPU Min Frequency	1 GHz
CPU Max Frequency	4.6 GHz
GPU	NVIDIA GeForce GTX 1060 3GB
Memory	15,911 MiB
L1d Cache	192 KiB (6 instances)
L1i Cache	192 KiB (6 instances)
L2 Cache	1.5 MiB (6 instances)
L3 Cache	12 MiB (1 instance)
NUMA Nodes	1
NUMA Node0 CPUs	0–11

Tab. 7: Host system specifications used to execute all experiments, including the operating system energy consumption profile.

D Matrix Multiplication Solutions Given as Example to One-Shot Adaptations

The solution in Fig. 14 is given to a run that uses a one-shot unoptimized adaptation, while the solution in Fig. 15 is for one-shot energy optimized adaptation. Adaptations that use a zero-shot strategy do not receive any examples. The specific Matrix Multiplication example solution is tied to the chosen implementation, which is Java GraalVM in the referenced figures. Moreover, the optimized example solution is created for the same host system specifications where the experiments are executed.

```yaml
implementation: graalvm
name: Matrix Multiplication
description: |
    Create two square matrices of size N×N.
    Each cell within these two matrices will be initialized with the sum of their row and column indices.
    Multiply the two matrices and output the contents of the last cell.
code: |
    public class Program {
        public static void main(String[] args) {
            int n = Integer.parseInt(args[0]);
            double[] A = new double[n * n];
            double[] B = new double[n * n];
            double[] C = new double[n * n];
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    A[i*n + j] = B[i*n + j] = i + j;
                }
            }
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    double sum = 0.0;
                    for (int k = 0; k < n; k++) {
                        sum += A[i*n + k] * B[k*n + j];
                    }
                    C[i*n + j] = sum;
                }
            }
            System.out.printf("%.0f\n", C[n*n - 1]);
        }
    }
dependencies: [graalvm-ce] # 23.0.0
---
args: [200]
expected_stdout: 18487100
```

Fig. 14: Example solution for Matrix Multiplication given as reference to runs that use a one-shot unoptimized adaptation.

```
implementation: graalvm
name: Matrix Multiplication Optimized
description: |
    Create two square matrices of size N×N.
    Each cell within these two matrices will be initialized with the sum of their row and column indices.
    Multiply the two matrices and output the contents of the last cell.
code: |
    import java.util.concurrent.*;
    import java.util.stream.IntStream;
    public class Program {
        private static final int L1 = 64;
        private static final int L2 = 256;
        public static void main(String[] args) {
            int n = Integer.parseInt(args[0]);
            double[] A = new double[n * n];
            double[] B = new double[n * n];
            double[] C = new double[n * n];
            IntStream.range(0, n).parallel().forEach(i -> {
                for (int j = 0; j < n; j++) A[i*n + j] = B[i*n + j] = i + j;
            });
            int threads = Math.min(Runtime.getRuntime().availableProcessors(), 6);
            ForkJoinPool pool = new ForkJoinPool(threads);
            pool.submit(() -> IntStream.range(0, (n + L2 - 1) / L2).parallel().forEach(i2Block -> {
                int i2 = i2Block * L2;
                for (int j2 = 0; j2 < n; j2 += L2)
                    for (int k2 = 0; k2 < n; k2 += L2)
                        for (int i1 = i2; i1 < i2 + L2 && i1 < n; i1 += L1)
                            for (int k1 = k2; k1 < k2 + L2 && k1 < n; k1 += L1)
                                for (int j1 = j2; j1 < j2 + L2 && j1 < n; j1 += L1)
                                    mulBlock(A, B, C, n, i1, j1, k1);
            })).join();
            pool.shutdown();
            System.out.printf("%.0f%n", C[n*n - 1]);
        }
        private static void mulBlock(double[] A, double[] B, double[] C, int n, int i0, int j0, int k0) {
            for (int i = i0; i < i0 + L1 && i < n; i++) {
                for (int k = k0; k < k0 + L1 && k < n; k++) {
                    double aik = A[i*n + k];
                    int jEnd = Math.min(j0 + L1, n);
                    for (int j = j0; j < jEnd; j++)
                        C[i*n + j] += aik * B[k*n + j];
                }
            }
        }
    }
dependencies: [graalvm-ce] # 23.0.0
---
args: [200]
expected_stdout: 18487100
```

Fig. 15: Example solution for Matrix Multiplication given as reference to runs that use a one-shot energy optimized adaptation.