Energy Efficiency of AI Generated Code

by

DEVI VARAPRASAD REDDY JONNALA

May 12, 2025

A thesis submitted to the Faculty of the Graduate School of the University at Buffalo, The State University of New York in partial fulfillment of the requirements for the degree of

Master of Science (MS)

Department of Computer Science and Engineering

To my family

Acknowledgements

It has been a great privilege to work under the supervision of Dr. Tevfik Kosar. I am deeply grateful to Dr. Kosar for his invaluable guidance and support over the past year. I also thank Dr. Chen Wang for his support. My sincere thanks go to Arman Islam and Ritika Rekhi from the Data Intensive Distributed Computing Lab, as well as Pratik Pokharel and Siddharth Cilamkoti, for their thoughtful insights and contributions throughout the course of this project. I also thank the National Science Foundation (Award# 2343284), the State University of New York (SUNY), and IBM for partially sponsoring this work. Finally, I am profoundly grateful to my family for their unwavering support and confidence in me, which continues to empower and motivate me.

Abstract

ENERGY EFFICIENCY OF AI GENERATED CODE

Recent advances in AI have created new prospects for improving software development, particularly through the usage of Large Language Models (LLMs) for code generation. The increasing integration of code generation models into software development has heightened the need to evaluate not only the correctness but also the efficiency and sustainability of AI-generated code. More efficient code can significantly enhance the performance and execution efficiency of software developed using LLM-assisted programming.

In this work, we investigate the energy efficiency of code generated by several well-known LLMs. We use EFFIBENCH dataset containing 1000 LeetCode problems with the most efficient solutions known as canonical solutions. Our research aims to determine which models provide not only functionally correct but also energy-efficient code, thus contributing to more sustainable software development processes. We perform the analysis using *perf* to report the energy consumption in Joules. The results of this study can be used in the development of novel code generation and refactoring techniques to improve the overall energy efficiency of the code generated by LLMs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

RESEARCH CONCEPT

1.1 INTRODUCTION

In software development, code generators have become indispensable for boosting productivity, maintaining consistency, enforcing standards, and refining existing codebases. With the advent of LLM-based code generation tools, developers can swiftly create code structures tailored to their specific requirements, thereby simplifying the development process and minimizing errors. Traditionally, evaluations of code generators have centered around algorithmic efficiency and the quality of the produced code. However, a critical yet often neglected factor is the energy efficiency of the generated code.

Software applications are major consumers of computing resources, leading to substantial energy consumption and carbon emissions. Information Technology (IT) currently contributes about 10% of global energy usage [2] and is responsible for approximately 3% of global carbon emissions[3], exceeding the aviation industry's emissions of 2% [4]. The demand for computing power, especially due to trends like Generative AI, is expected to cause data centers to consume 20% of global electricity by

2030 [5]. The energy gains achieved in limiting the datacenter energy demand through the improvement of hardware systems and improved cooling systems can be outpaced by the increased demand for AI workloads and blockchain applications.

As computing systems grow more sensitive to power demands, evaluating and improving code for energy consumption is becoming increasingly essential. Beyond performance alone, considerations around power usage now encompass broader concerns such as environmental sustainability, cost efficiency, and hardware scalability challenges.

Our work aims to fill this gap by investigating the energy efficiency of code generated by LLMs. We use the EFFIBENCH [1] benchmark, consisting of 1000 Leetcode [6] problems with interview frequencies greater than 40% and with executable canonical solutions. We also select the latest versions of 20 widely used LLMs, and calculate the metrics like CPU-Energy (in Joules), RAM Energy (in Joules), execution time, and memory usage. By measuring and comparing the energy consumption of generated code snippets, we seek to provide insights into the environmental and economic implications of using code generated by LLMs. This thesis seeks to underscore the importance of incorporating energy efficiency considerations into code generation practices, paving the way for more sustainable and cost-effective software development processes.

1.2 CONTRIBUTIONS

This thesis makes the following contributions:

- We conduct an extensive evaluation of 20 LLMs, and compare the energy efficiency of the code generated by them.

- We ensure fair prompt inputs during code generation and fair comparison of LLMs against themselves and canonical solutions by only considering problems correctly addressed by all models.
- Our findings reveal that even the latest versions of state-of-the-art LLMs (like GPT-4o, Gemini-2.0) generate code that is inefficient when compared to human-written solutions (canonical solutions).

CHAPTER 2

RELATED WORK

2.1 GREEN COMPUTING

Green Computing is about the efficient use of computing resources to minimize environmental impact, primarily focusing on energy efficiency, reduced emissions, and sustainable product design. With rapid technological advancements especially in the fields of AI, Internet of Things, and Cloud Computing, there has been an exponential growth in the demand for compute[7] which leads to an exponential growth in energy consumption by data centers. If this issue is left unchecked, the emissions from the Information and Communication (ICT) industry will grow manyfold[5] leading to extreme weather conditions, habitat destruction, and global warming. Fortunately, to prevent this, many big corporations are targeting to become carbon neutral by 2050 or sooner. There has been extensive research in both academia and industry to design and promote green computing and sustainability practices.

2.1.1 INFRASTRUCTURE SUSTAINABILITY

Colleen Josephson et al. [8] outline a four-pillar framework to achieve zero-carbon ICT:

- Prioritizing Renewable Energy: transition from carbon-intensive energy sources to renewables, measuring carbon emissions accurately, and carbon-aware workload shifting (moving computing tasks geographically to where renewable energy is abundant).
- Efficient Use of Resources: improving server utilization, reducing zombie (inactive) virtual machines, and encouraging cloud-based on-demand resource usage.
- Addressing Embodied Carbon: designing reusable, modular, and recyclable hardware to minimize waste.
- Removing Institutional Barriers: cultural and structural changes to prioritize sustainability over traditional business practices.

Effective thermal management is critical for data centers, where cooling systems often consume up to 40% of total energy use. Research has increasingly focused on innovative strategies to improve energy efficiency and sustainability in data center cooling. Aayush Agrawal et al.[9] analyze different cooling strategies for various climatic conditions for reducing energy usage and the total cost of ownership in data centers. Liu et al.[10] propose a novel two-phase spray cooling system for data center racks, which eliminates the need for traditional chillers. Sarkar et al. (2023) [11] propose DC-CFR, a multi-agent reinforcement learning framework that optimizes data center operations by coordinating cooling, load shifting, and battery usage in real-time. Unlike prior static or forecast-based methods, DC-CFR reduces carbon emissions and energy use by over 14% using adaptive, cooperative control strategies.

2.1.2 CODE LEVEL SUSTAINABILITY

While the techniques discussed in section 2.1.1 deal only with optimizing the resource usage for computing infrastructure and data centers, it is also important to look at optimizing the underlying processes i.e., software and applications that cause the energy consumption. [12] propose the concept of "Energy-Aware Programming," emphasizing the importance of considering energy efficiency in software development. Their work underscores the need for developers to adopt practices that minimize energy consumption while maintaining code functionality and performance.

Addressing the carbon footprint of computation, the study on "Green Algorithms" [13] introduces a quantitative model for assessing the environmental impact of computational processes. The paper presents a methodological framework and an online tool called Green Algorithms for estimating and reporting the carbon footprint of computational tasks, addressing the environmental impact of high-performance computing. Through case studies on particle physics simulations, weather forecasts, and natural language processing, the study illustrates the applicability of the framework while advocating for greener computational practices.

In the domain of deep learning, the Zeus framework [14] focuses on an optimization framework designed to address the trade-off between DNN training performance and energy consumption by automatically adjusting job- and GPU-level configurations. Utilizing an online exploration-exploitation approach and just-in-time energy profiling, Zeus achieves significant improvements in energy efficiency, ranging from 15.3% to 75.8%, across various DNN training workloads without requiring expensive offline measurements.

2.2 CODE EVALUATION BENCHMARKS

Code generation [15] is one of the use cases of LLMs, where they generate code snippets based on the description given in the prompt. The inputs to the prompt can be a text explaining the expected behavior of code to be generated or a combination of text and code snippets to be fixed. To evaluate this capability of LLMs, several benchmarks have been proposed.

HumanEval [16] which is developed by researchers at OpenAI consists of 164 hand-written programming challenges and unit tests with Python method signatures. The challenges are hand-written to make sure they are not part of the training dataset. Subsequent works like HumanEval-X [17], and HumanEval-XL [18] extend the scope of the HumalEval to other programming languages like C++, Java, and Go. EvalPlus [19] improves the quality and quantity of test cases given in HumanEval. Another benchmark MBPP [15] consists of 974 basic Python problems with 3 test cases for each task, whereas MBXP [20] extends MBPP to include other programming languages. APPS [21] sources 10000 Python coding problems from open-access coding websites and provides 13.2 unit tests on average to measure coding ability and problem-solving.

As the scope of code generation widens, there are several benchmarks related to data science like DS-1000 [22] are proposed. For evaluating models on completing broader software engineering tasks, several benchmarks like APIBench [23], SWE-Bench [24], and RepoEval [25] are proposed. The above-proposed benchmarks focus only on the correctness of the generated code and do not consider the efficiency of the code.

EFFIBENCH [1] addresses this issue by constructing a benchmark with 1000 Leet-Code problems with each problem consisting of 100 test cases. They study 42 LLMs and evaluate the code generated for efficiency, using metrics like execution time and

memory usage.

2.3 ANALYSIS OF LLM-GENERATED CODE

Most of the benchmarks [15, 16] focus on the ability of LLMs to generate code correctly rather than how efficient is the generated code with respect to resource utilization.

The study titled ”Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation”, [26] investigates the sustainability of auto-generated code produced by LLMs like GitHub Copilot, OpenAI’s ChatGPT-3, and Amazon’s CodeWhisperer, compared to human-generated code. It proposes the metric ”Green Capacity” based on code correctness, runtime, memory usage, FLOPs, and energy consumption to assess the environmental impact of these AI models and their potential contribution to sustainable software development. This study considers only 3 problems from LeetCode without considering the baseline energy consumption of the testbed.

The study titled ”Can LLMs Generate Green Code - A Comprehensive Study Through LeetCode” [27] compares the energy efficiency of code generated by 8 LLMs using a python library called *PyJoules*. It only considers 8 problems from Leetcode and uses different prompt styles like *promptbase, promptCoT, prompteff* to see if they can generate efficient code. This study concludes that LLMs rarely generate efficient code because of already-seen data during training.

Coignion et al. (2024)[28] studies LLM-generated code on LeetCode, considering 18 LLMs specifically designed for code and 208 most recent problems sourced from LeetCode by Döderlein et al. [29]. They study the impact of temperature and success rate on code performance and find that there is little to no correlation between correctness, temperature, and performance.

Niu et al. (2024) [30] evaluates the efficiency of LLM-generated code based on runtime using two benchmarks, HumanEval [16] and MBPP [15], and extend the evaluation to more challenging problems from the online judge platform LeetCode. The findings reveal that the efficiency of code generated by LLMs is independent of both the correctness of the code and the size of the model. Additionally, Chain-of-Thought (CoT) prompting is found to improve the efficiency of generated code, particularly for complex problems.

Cursaru et al. (2024) [31] perform controlled experiments on Raspberry Pi with 3 problems from LeetCode using CodeLlama [32] for code generation. Erhabor et al. (2023) [33] evaluate Github Copilot on C++ problems with 32 human programmers working on 2 problems, one with Copilot's assistance and the other independently. Both these studies conclude that AI-generated code is worse in terms of energy efficiency and runtime performance.

Cappendijk et al. (2024) [34] study the energy consumption of the code generated by CodeLlama and DeepSeek-Code models using perf and GNU time commands. This study uses only 3 problems from LeetCode and does not consider a wide variety of problems with various algorithms and difficulties. They suggest prompt engineering could help improve the efficiency of auto-generated code, but not a single prompt can consistently result in improved efficiency with different models.

Solovyeva et al. (2025) [35] compare the energy efficiency of code generated by LLMs like GPT-4o, GPT-o1-mini, Github Copilot across three languages: Python, Java, and C++ on different hardware. This study takes 53 LeetCode "Hard" problems and finds out that C++ code is least accurate and Python code is mostly accurate and sometimes more efficient. For Models, the code generated by Copilot is less accurate but more energy-efficient, whereas OpenAI o1-mini is more accurate with less energy-

efficient code.

Overall, these studies underscore the importance of considering energy efficiency in code generation practices and highlight the potential for sustainable software development through optimized energy consumption.

CHAPTER 3

METHODOLOGY

3.1 BENCHMARK SELECTION

As discussed in section 2.2, benchmarks like MBPP, HumanEval, etc. are designed to evaluate the correctness of the code generated by LLMs. They are not best suitable to evaluate the efficiency of code generated due to one or many of the following reasons:

- **Choice of problems:** not including problems of all difficulties, and not including a variety of algorithms.
- **Number of test cases:** Most of the benchmarks have less number of test cases per problem.
- **Quality of test cases:** The test cases included are very basic with small inputs. The efficiency of an algorithm can only be known when the size of inputs is large.

All the above-mentioned issues were addressed by the dataset – EFFIBENCH [1]. It is inspired by the common practice of evaluating developers' coding ability using problems from the competitive coding platform – LeetCode. Problems that are asked in

interviews frequently (>40%) are considered and paired with the most efficient solutions from the LeetCode discussion forum. Test cases for each problem are generated using a test case generator created using GPT-3.5-turbo. The comparison of Effibench with other datasets is shown in Table 3.1

Dataset	Number of Problems	Evaluation Support	Avg. Test Cases	Avg. Lines of Canonical Solution	Data Source	Assessment
HumanEval	164	Test Cases	7.7	6.3	Hand-Written	Correctness
MBPP	974	Test Cases	3.0	6.7	Crowd-sourced	Correctness
APPS	10000	Test Cases	13.2	18.0	Competitions	Correctness
DSP	1119	Test Cases	2.1	4.5	Notebooks	Correctness
DS-1000	1000	Test Cases	1.6	3.6	StackOverflow	Correctness
EFFIBENCH	1000	Test Cases	100	14.6	LeetCode	Efficiency & Correctness

Table 3.1: Comparison of benchmark datasets [1].

3.1.1 BENCHMARK STATISTICS

The coding problems in Effibench can be categorized based on difficulty and algorithms. As per LeetCode, there are three difficulty types - Easy, Medium, and Hard. Based on algorithms, the problems can be divided into 11 categories. As shown in Table 3.2, there are 171 easy, 589 medium, and 240 hard problems in the dataset. One problem can be tagged to more than one algorithm and hence the sum of the number of problems across different algorithms for a given difficulty will be greater than the reported total.

Algorithm	Greedy	DP	Backtracking	Divide & Conquer	DFS	BFS	Binary Search	Two Pointers	Sliding Window	Bit Manipulation	Sorting	Total
Number of problems	243	277	48	21	108	86	148	105	70	102	238	1000
Number of Easy problems	32	8	1	4	18	8	23	39	9	26	63	171
Number of Medium problems	170	151	37	8	72	52	75	59	47	58	133	589
Number of Hard problems	41	118	10	9	18	26	50	7	14	18	42	240

Table 3.2: Statistics of EFFIBENCH [1].

3.1.2 BENCHMARK EVALUATION

Before using the dataset for our study, we thoroughly analyzed and tested the dataset. The following are the findings from our analysis:

- 12 problems do not have comprehensive test cases.
- 110 problems throw errors when we run canonical solutions against comprehensive test cases. This is due to one or all of the following reasons: syntax errors in canonical solutions, syntax errors in comprehensive test cases, and improper definition of test cases for TreeNode, GraphNode, and LinkedList problems.

We excluded the above-mentioned 122 problems and considered 878 problems for our study. The updated statistics for this filtered dataset can be seen in Table 3.3.

Algorithm	Sliding Window	Divide & Conquer	Binary Search	DP	Greedy	Two Pointers	Backtracking	Bit Manipulation	Sorting	DFS	BFS	**Total**
Total problems	62	19	139	258	231	88	40	97	230	53	58	878
Easy problems	8	4	23	7	32	31	1	25	62	3	0	145
Medium problems	40	6	68	140	161	52	31	54	128	37	36	510
Hard problems	14	9	48	111	38	5	8	18	40	13	22	223

Table 3.3: Statistics of filtered EFFIBENCH

3.2 LLMS UNDER STUDY

We chose the latest versions of the most popular LLMs which are widely used by developers for code generation tasks. A Total of 21 LLMs are studied of which 7 models from Meta (Llama-3.1-8b, Llama-3.1-70b, Llama-3.3-70b), Mistral (Mistral-large-2407, Codestral-mamba-2407, Pixtral-large-2411) and DeepSeek-v3 are open-souced. The models and their parameters are shown in Table 3.4

Vendor	LLM	# params
OpenAI	GPT 3.5-turbo-0125	175B
OpenAI	GPT 4-turbo (2024-04-09)	Not Public
OpenAI	GPT 4o (2024-08-06)	Not Public
Meta	Llama 3.1 (8B)	8B
Meta	Llama 3.1 (70B)	70B
Meta	Llama 3.3	70B
Anthropic	Claude 3.5 Sonnet (2024-10-22)	Not Public
Anthropic	Claude 3.5 Haiku (2024-10-22)	Not Public
Google	Gemini 1.5 Flash-002	Not Public
Google	Gemini 1.5 Pro-002	Not Public
Google	Gemini 2.0 Flash-001	Not Public
Google	Gemini 2.0 Flash-Lite-001	Not Public
Mistral AI	Mistral-large-2407	123B
Mistral AI	Pixtral-large-2411	124B
Mistral AI	Codestral-mamba-2407	7B
Amazon	Nova - Micro	Not Public
Amazon	Nova - Lite	Not Public
Amazon	Nova - Pro	Not Public
xAI	Grok	Not Public
DeepSeek	DeepSeek v3	37B
EFFIBENCH	Canonical	Not Applicable

Table 3.4: LLMs under study.

3.3 CODE GENERATION

The overall workflow of this study is shown in Figure. 3.1. We considered 878 problems from EFFIBENCH and generated solutions using the LLMs discussed in section 3.2.

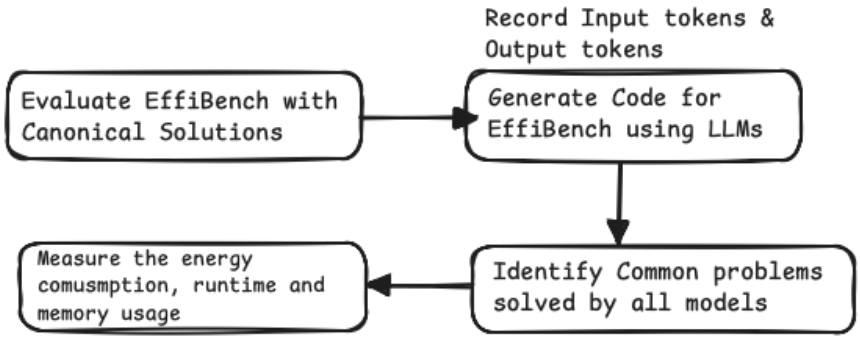

Figure 3.1: Overall worflow

LLMs are accessed using APIs as shown in Table **??**. We utilized batch inference wherever possible to save on time and costs. Prompts in batch inference are executed independently and the results are made available in a single file.

Vendor/Model	Access Type
OpenAI	Batch API
Amazon	AWS Bedrock Batch Inference
LLaMA	Groq Cloud
LLaMA 3.1-70B	AWS Bedrock Batch Inference
Google	Vertex AI Batch Prediction
xAI	API
Deepseek	Fireworks.ai API
Mistral	API
Anthropic	API

Table 3.5: Model Access Types by Vendor

If the LLMs are not able to generate the correct code in the first go, retries are allowed till it generates a correct solution or up to 24 times. After each iteration, python files for each problem are generated, they contain the solution generated by LLMs followed by test cases in the dataset. All the files are then executed and errors are logged, in the next regeneration, the error message is included in the prompt to correct its mistake. To ensure fairness, all models follow the same prompt format. Once a

correct solution is generated, that problem will not be considered for regeneration in the next iteration. During each iteration, the number of input tokens and the number of output tokens to LLMs are noted along with the number of problems passed till that iteration.

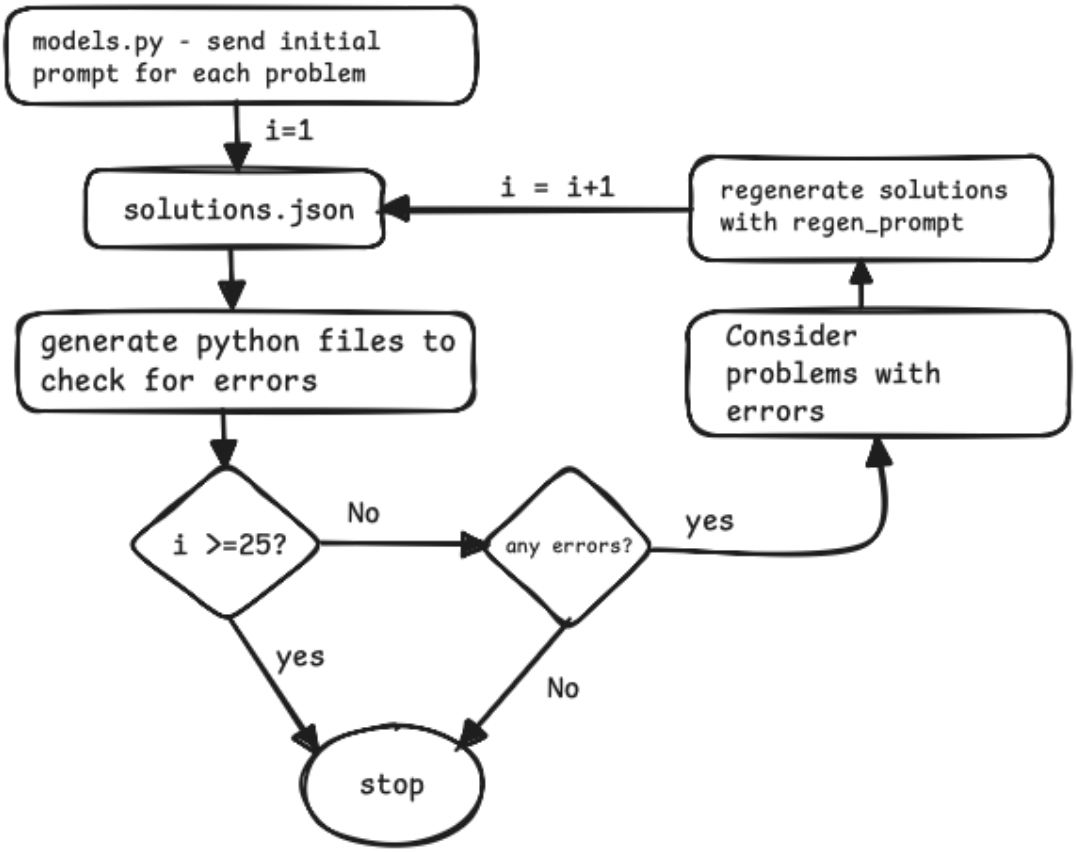

Figure 3.2: Code Generation Workflow

We defined ***'pass@k'*** as the number of problems passing the test cases after k iterations of generating code with LLMs. The results for various LLMs are shown in Table 3.6

We used a prompt structure similar to the one used in EFFIBENCH, which follows the MBPP code generation prompt. The prompt input format is shown below:

Initial Prompt Format

Please write a solution based on the task description that passes the provided test cases. You must follow these rules:

- First, the code should be in ```` ```python\n[Code]\n``` ```` block. There should be only one such block in your response.
- Second, you should not add the provided test cases into your ```` ```\npython[Code]\n``` ```` block.
- Third, you do not need to write the test cases, we will provide the test cases for you.
- Finally, you should make sure that the provided test cases can pass your solution.

Sample Problem Statement, Code and Solution

Now the actual task for you:

Problem Statement

Small Test Cases

Constraints

If the code has to be regenerated, the below prompt format is used:

Regen Prompt Format

Please write a solution based on the task description that passes the provided test cases.

You must follow these rules:

- First, the code should be in ```` ```python\n[Code]\n``` ```` block. There should be only one such block in your response.
- Second, you should not add the provided test cases into your ```` ```\npython[Code]\n``` ```` block.
- Third, you do not need to write the test cases, we will provide the test cases for you.
- Finally, you should make sure that the provided test cases can pass your solution.
- your last generated code is not passing all test cases, please correct your logic and regenerate.

Problem Statement

Small Test Cases

Constraints

for which your last generated code was this: **Previous Solution**

All the given test cases are correct so you have to correct the logic as it fails the below test case: **Error Message**

Problem-3 Description

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a
    substring.
```

Constraints:

- $0 \leq$ `s.length` $\leq 5 \times 10^4$
- `s` consists of English letters, digits, symbols, and spaces.

Test case (Python):

```
solution = Solution()
assert solution.lengthOfLongestSubstring("abcabcbb") == 3
assert solution.lengthOfLongestSubstring("bbbbb") == 1
assert solution.lengthOfLongestSubstring("pwwkew") == 3
```

Generated Code - Amazon Nova

```
from typing import *


class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        max_length = 0
        start = 0
        char_index_map = {}
        for end in range(len(s)):
            if s[end] in char_index_map and char_index_map[s[end]] >= start:
                start = char_index_map[s[end]] + 1
            char_index_map[s[end]] = end
            max_length = max(max_length, end - start + 1)
        return max_length
```

LLM	Pass@1 (%)	Pass@10 (%)	Pass@25 (%)
GPT 3.5 (gpt-3.5-turbo-0125)	56.9%	67.0%	71.0%
GPT 4 Turbo (gpt-4-turbo-2024-04-09)	56.3%	87.0%	89.4%
GPT 4o (gpt-4o-2024-08-06)	78.6%	90.0%	92.4%
Llama 3.1 (8B)	20.6%	55.2%	62.2%
Llama 3.1 (70B)	54.7%	75.5%	81.2%
Llama 3.3 (70B)	66.1%	79.3%	82.9%
Claude 3.5 Sonnet (2024-10-22)	78.6%	89.7%	89.5%
Claude 3.5 Haiku (2024-10-22)	71.4%	83.8%	85.3%
Gemini 1.5 Flash-002	66.1%	75.2%	77.4%
Gemini 1.5 Pro-002	79.7%	87.1%	87.7%
Gemini 2.0 Flash-001	80.9%	88.5%	90.1%
Gemini 2.0 Flash-Lite-001	71.3%	82.0%	84.4%
Mistral-large-2407	64.8%	72.8%	76.2%
Pixtral-large-2411	56.2%	73.5%	78.2%
Codestral-mamba-2407	43.8%	55.5%	63.0%
Nova - Micro	33.7%	49.1%	51.5%
Nova - Lite	43.1%	56.7%	58.9%
Nova - Pro	60.6%	73.2%	75.3%
Grok	75.3%	84.1%	85.0%
DeepSeek v3	83.7%	90.0%	91.3%
Canonical	100.0%	100.0%	100.0%

Table 3.6: LLMs under study.

DeepSeek-v3 has the highest pass@1 score of 83.7%, whereas both DeepSeek-v3 and GPT-4o have pass@10 score of 90% and, finally GPT-4o has the highest pass@25 score of 92.4%.

CHAPTER 4

EVALUATION

4.1 EXPERIMENT DESIGN

Once the code generation with LLMs is done, we proceed to measure the energy consumed by the generated code. As a first step, based on ***pass@k*** metric, we select the LLMs and find the problems that are correctly solved by all the LLMs, hereafter referred to as ***common problems***. These common problems are then executed again to check for errors and to record minimum runtime. The correctness of the code is tested by the test cases in the dataset, and the code passing all test cases can be considered correct. Based on the runtime, we assign a problem a bucket category (as shown in Table 4.1), so that running the problems in each bucket multiple times makes sure the runtime crosses 1 second, which ensures accurate readings by the profiling tool *perf* [36]. *perf* is used to compute the runtime, CPU-Energy (`power/energy-pkg`), RAM- Energy (`power/energy-ram`). This approach gives the energy consumption of the entire system in *Joules*. To get the energy consumption only during the code execution, we consider the ***baseline energy consumption*** and subtract it from the actual

energy reported.

Bucket Category	Execution Time
40 runs	< 50 ms
25 runs	50 ms – 100 ms
10 runs	100 ms – 500 ms
2 runs	> 500 ms

Table 4.1: Execution Time by Bucket Category

Baseline power consumption: We measure the energy consumed by the system while no process is running on it for 30 seconds.

$$P_{\text{baseline}} = \frac{E_{\text{idle, 30s}}}{30\ \text{s}} \tag{4.1}$$

Now, we can calculate the Energy consumed by code E_{adjusted} as,

$$E_{\text{adjusted}} = E_{\text{actual}} - P_{\text{baseline}} \times t_{\text{code}} \tag{4.2}$$

where E_{actual} is the energy reported by Perf while the code runs for t_{code} seconds.

The solutions generated by different LLMs along with canonical solutions are organized in folders named after problem_index. During the experiment, the problems are processes based on folders, with each folder containing the solutions generated by different LLMs for that particular problem along with canonical solutions. All the sets of solutions in a folder are run 5 times in a random order each time, with each solution being run multiple times according to its bucket category. Baseline Power consumption is measured before processing each folder and after running each solution a cool-down period of 10 seconds is maintained to reduce the effect of previous executions. To avoid the usage of compute for tasks other than the execution of solutions, the measurements

are simply logged and computations of averages are only performed after the execution of all problems. The adjusted energy consumption for each solution is then computed and the results are analyzed using the metrics discussed in section 4.2.

In addition to the above experiment, we calculated the memory used by each solution using the python library `Memory_Profiler`. This library helps us sample the memory used by the process at the given intervals (in this case 0.001 seconds) and store the readings in a .dat file for each solution. This experiment is done 3 times independently of the previous one with perf and the results are averaged to reduce noise.

4.1.1 EXPERIMENT ENVIRONMENT

All the experiments are conducted on a Ubuntu 24.04 LTS (Kernel 6.8.0-51-generic) machine equipped with an Intel(R) Xeon(R) Gold 6126 (24 cores) processor with 192 GB of main memory, hosted on Chameleon cloud [37] and Python 3.12.3. While the experiments are running, we make sure that the machine is not being used for any other tasks and that no updates are being installed on it.

4.2 METRICS

We use the following metrics to evaluate the results:

Execution Time: represents the time taken by the problem to complete execution, represented by T_{code}. It is measured using *perf*.

Average Execution Time: is the sum of the execution times of problems considered

for the experiment divided by the number of problems considered (N).

$$\text{Avg. ET} = \frac{1}{N}\sum_{i=1}^{N} T_{\text{code}_i} \tag{4.3}$$

Memory Consumption over Time: is a metric that measures memory consumption taking into account the time for which it is being consumed. It is measured using the python library `memory_profiler`.

$$\text{mem-sec} = \sum_{i=1}^{n-1}\left(\frac{M_i + M_{i+1}}{2} \times (T_{i+1} - T_i)\right) \tag{4.4}$$

where,

- M_i is the memory consumption at time T_i
- n is the number of sampling points
- The unit is memory-seconds (MB-sec)

Average Memory Consumption over Time: is the sum of the mem-sec of the problems considered for the experiment divided by the number of problems considered (N).

$$\text{Avg. mem-sec} = \frac{1}{N}\sum_{i=1}^{N} memsec_{\text{code}_i} \tag{4.5}$$

Package Energy: is the energy consumed by the CPU during the execution of code as measured by perf. It is the adjusted package energy calculated using equation 4.2.

RAM Energy: is the energy consumed by memory during the execution of the code, measured using perf. It is then adjusted considering the baseline power consumption using the equation 4.2.

Total Energy: is the sum of CPU Energy and RAM Energy for a particular problem.

Average Total Energy: is the sum of the total energy consumed by problems considered for the experiment divided by the number of problems considered.

$$\text{Avg. TE} = \frac{1}{N}\sum_{i=1}^{N} TE_{\text{code}_i} \tag{4.6}$$

4.3 RESULTS FOR COMMON PROBLEMS

After generating solutions to the problems, we select the batches of problems and LLMs based on pass@k to compute the metrics discussed in section 4.2. This is to ensure that all LLMs are considered in at least one analysis. The analysis of results is discussed in the following subsections. Overall, DeepSeek-v3 outperforms all other models in generating energy-efficient code.

4.3.1 BATCH-1: LLMS WITH PASS@25 $>$ 65%

There are 16 LLMs with pass@25 metric greater than 65%. There are 426 problems solved correctly, by all of these LLMs. As shown in table 4.2, there are 102 easy, 265 medium, and 59 hard problems. Table 4.3 summarizes the average energy and memory consumption of these problems.

Algorithm	Sliding Window	Divide & Conquer	Binary Search	DP	Greedy	Two Pointers	Backtracking	Bit Manipulation	Sorting	DFS	BFS	**Total**
Total	32	6	63	109	106	56	14	49	120	23	22	426
Easy	4	1	15	7	20	24	0	18	41	3	0	102
Medium	24	3	37	74	73	32	12	26	72	18	17	265
Hard	4	2	11	28	13	0	2	5	7	2	5	59

Table 4.2: Batch-1: Problem difficulty and algorithm distribution

Model	Avg Total Energy	Avg Runtime	Avg mem-sec	Avg Input Tokens	Avg Output Tokens	Avg Pass_at
canonical	5.52	0.07	7.63			
claude-haiku	7.24	0.09	8.60	1103	610	1.2089
claude-sonnet	7.12	0.09	8.64	912	514	1.0493
Deepseek-v3	6.30	0.08	8.04	811	155	1.0213
Gemini-1.5-flash	9.02	0.11	9.25	1158	195	1.2324
Gemini-1.5-pro	8.34	0.10	8.84	873	159	1.0213
Gemini-2.0-flash	6.57	0.08	8.36	927	180	1.0425
Gemini-2.0-flash-lite	7.36	0.09	8.85	1043	206	1.1643
GPT-3.5-turbo	7.34	0.09	8.80	1448	245	1.6197
GPT-4-turbo	10.66	0.13	10.15	974	223	1.1573
GPT-4o	6.98	0.09	8.54	817	175	1.0213
grok-2-1212	9.32	0.12	9.09	877	137	1.0610
llama-3.1-70b	9.37	0.12	9.31	1516	350	1.5962
llama-3.3-70b	9.21	0.11	9.71	1021	218	1.7512
mistral-large-2407	9.21	0.11	9.46	1476	208	1.3474
Nova-Pro	7.24	0.09	8.74	1263	239	1.3568
pixtral-large-2411	7.54	0.09	8.48	1596	222	1.3709

Table 4.3: Batch-1: Model performance and efficiency metrics

It can be seen that canonical solutions have the least energy consumption of 5.52 Joules, whereas, among LLMs, DeepSeek-v3 has the least energy consumption of 6.3 Joules. GPT-4-Turbo has the overall worse energy efficiency when we consider this set of problems. While analyzing the energy consumption based on difficulty, all LLMs are equally energy efficient on Easy problems, the most significant difference is observed with Medium problems, where DeepSeek-v3 performs the best and GPT-4-turbo performs the worst consuming 1.2 times and 2.16 times the energy consumed by canonical solutions. Overall among all the algorithms, DeepSeek-v3 consistently performed better than all LLMs, whereas Llama 3.3-70b, Grok-2-1212, Llama-3.1-70b, and GPT-4-turbo performed the worst. We also observed that problems involving dynamic programming and sorting consume more energy than the other ones.

4.3.2 BATCH-2: LLMS WITH PASS@25 $>$ 80%

There are 11 LLMs with pass@25 metric greater than 80%. There are 576 problems solved correctly, by all of these LLMs. As shown in table 4.4, there are 119 easy, 347 medium, and 110 hard problems. Table 4.5 summarizes the average energy and memory consumption of these problems.

Algorithm	Sliding Window	Divide & Conquer	Binary Search	DP	Greedy	Two Pointers	Backtracking	Bit Manipulation	Sorting	DFS	BFS	**Total**
Total	48	12	86	158	149	62	25	66	155	33	30	576
Easy	7	3	16	7	28	24	1	22	48	3	0	119
Medium	31	4	46	98	101	37	19	34	89	27	24	347
Hard	10	5	24	53	20	1	5	10	18	3	6	110

Table 4.4: Batch-2 Problem difficulty and algorithm distribution

As compared to batch-1 (see table 4.2), this batch has double the number of hard problems and more number of problems across various algorithms. It can be considered that this batch of problems includes complex problems that could not be solved by most of the LLMs.

Model	**Avg Total Energy**	**Avg Runtime**	**Avg mem-sec**	**Avg Input Tokens**	**Avg Output Tokens**	**Avg Pass_at**
canonical	5.46	0.07	6.62			
claude-haiku	7.05	0.09	7.65	1421	719	1.4149
claude-sonnet	6.74	0.08	7.33	964	532	1.0764
Deepseek-v3	6.37	0.08	6.90	890	191	1.0747
Gemini-1.5-pro	11.15	0.14	8.43	927	174	1.0556
Gemini-2.0-flash (1)	9.81	0.12	7.77	1084	231	1.1510
Gemini-2.0-flash (2)	7.31	0.09	8.27	1230	241	1.3073
GPT-4-turbo	9.40	0.12	9.08	1378	343	1.4479
GPT-4o	6.63	0.08	7.44	935	218	1.0920
grok-2-1212	10.92	0.14	9.34	982	156	1.1458
llama-3.1-70b	8.99	0.11	8.22	2562	645	2.3090
llama-3.3-70b	10.52	0.13	10.73	1249	291	2.4392

Table 4.5: Batch-2 model performance and resource usage comparison

No LLMs can outperform the canonical solutions in terms of energy consumption. Among LLMs, DeepSeek-v3 consistently performs the best, closely followed by GPT-

4o and Claude-Sonnet-3.5 across all algorithms and problem difficulties. Whereas Llama, Grok-2-1212, and GPT-4-turbo have significantly higher energy consumption.

4.3.3 BATCH-3: ALL LLMS

In this Batch, we consider all the 21 LLMs shown in table 3.4. There are 298 common problems solved correctly by all of these LLMs. As shown in table 4.6, there are 89 easy, 179 medium, and 30 hard problems. Table 4.7 summarizes the average energy and memory consumption of these problems.

Algorithm	Sliding Window	Divide & Conquer	Binary Search	DP	Greedy	Two Pointers	Backtracking	Bit Manipulation	Sorting	DFS	BFS	**Total**
Total	23	4	38	74	66	50	11	35	85	20	16	298
Easy	4	1	13	7	14	22	0	18	33	3	0	89
Medium	17	3	21	49	46	28	10	14	51	15	13	179
Hard	2	0	4	18	6	0	1	3	1	2	3	30

Table 4.6: Batch-3: Problem difficulty and algorithm distribution

Even in this batch of experiments, no LLM can beat canonical solutions in overall energy consumption. DeepSeek-v3 performs the best among all LLMs, whereas GPT-4-turbo performs the worst. It can be observed that among the worst performing LLMs we see a slight change in order, models like Llama-3.3-70b, and Llama-3.1-70b perform better than GPT-4-turbo in Batch-3 but not in Batch-2. This could be because of relatively simple problems in Batch 3 and when they are tested against more complex problems, their performance is less than that of all other models.

Model	Avg Total Energy	Avg Runtime	Avg mem-sec	Avg Input Tokens	Avg Output Tokens	Avg Pass_at
canonical	5.78	0.07	8.33			
claude-haiku	7.09	0.09	9.12	1049	598	1.1913
claude-sonnet	6.41	0.08	8.71	886	508	1.0470
codestral-mamba-2407	7.86	0.10	7.00	6973	773	4.3792
Deepseek-v3	5.91	0.08	8.30	790	146	1.0168
Gemini-1.5-flash	9.54	0.12	9.92	1007	163	1.0738
Gemini-1.5-pro	8.66	0.11	9.32	835	147	1.0067
Gemini-2.0-flash	6.12	0.08	8.66	872	167	1.0201
Gemini-2.0-flash-lite	7.15	0.09	9.21	935	181	1.0906
GPT-3.5-turbo	7.19	0.09	9.28	1130	182	1.3389
GPT-4-turbo	12.01	0.15	11.26	882	192	1.0906
GPT-4o	6.77	0.09	8.99	784	161	1.0101
grok-2-1212	9.98	0.12	9.77	800	121	1.0134
llama-3.1-70b	10.09	0.13	10.10	977	160	1.1846
llama-3.1-8b	8.11	0.10	8.46	944	179	1.5805
llama-3.3-70b	9.79	0.12	10.51	1257	250	1.3423
mistral-large-2407	9.82	0.12	10.17	1275	178	1.2282
Nova-lite	6.92	0.09	8.90	1303	268	1.4094
Nova-micro	8.12	0.10	9.80	2133	439	2.0671
Nova-Pro	7.12	0.09	9.21	1147	215	1.2712
pixtral-large-2411	7.47	0.09	8.81	1288	173	1.1879

Table 4.7: Batch-3: performance and efficiency metrics for 20 LLMs

4.4 WORST CASE ANALYSIS

In this section, we analyze the inefficient code generated by the best-performing LLM, DeepSeek-v3, and worst performers GTP-4-turbo and Grok-2-1212. We filter out the 10 most inefficient pieces of code generated by these LLMs and manually analyze the implementation for inefficiency. In the case of DeepSeek-v3, all the top 10 inefficient problems are associated with dynamic programming and backtracking. The comparison of one such problem is shown in figures 4.2, 4.1 for the problem with ID: 1638. The task is to find the count of sub-strings that differ by one character. The solution generated by DeepSeek-v3 is shown in figure 4.1, where it uses a brute-force approach of checking all sub-strings with a time complexity of $O(m^3n)$, making it inefficient for larger inputs.

```python
class Solution:
    def countSubstrings(self, s: str, t: str) -> int:
        count = 0
        for i in range(len(s)):
            for j in range(i + 1, len(s) + 1):
                s_sub = s[i:j]
                for k in range(len(t) - len(s_sub) + 1):
                    t_sub = t[k:k + len(s_sub)]
                    diff = sum(1 for a, b in zip(s_sub, t_sub) if a != b)
                    if diff == 1:
                        count += 1
        return count
```

Figure 4.1: problem 1638 - DeepSeek-v3

The canonical solution shown in figure 4.2, uses a dynamic programming approach to solve it in $O(mn)$ time. Though this solution has some space complexity due to the usage of dp matrices, this solution is approximately 50 times more energy efficient than the one in figure 4.1.

```python
class Solution:
    def countSubstrings(self, s: str, t: str) -> int:
        ans = 0
        m, n = len(s), len(t)
        f = [[0] * (n + 1) for _ in range(m + 1)]
        g = [[0] * (n + 1) for _ in range(m + 1)]
        for i, a in enumerate(s, 1):
            for j, b in enumerate(t, 1):
                if a == b:
                    f[i][j] = f[i - 1][j - 1] + 1
        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                if s[i] == t[j]:
                    g[i][j] = g[i + 1][j + 1] + 1
                else:
                    ans += (f[i][j] + 1) * (g[i + 1][j + 1] + 1)
        return ans
```

Figure 4.2: problem 1638 - Canonical

A similar pattern is seen in another inefficient solution generated by DeepSeek-v3, where it fails to use memoization/caching of already computed results, to avoid computing them again and again.

Grok-2-1212 also lags in generating efficient solutions for problems associated with DP and backtracking. GPT-4-turbo fails to perform well in problems associated with dynamic programming and relatively simple technique sorting. If we consider problem: 945, the minimum number of increments to make an array unique. The time complexity of the solution generated by GPT-4-turbo (see figure 4.3) can be considered $O(n)$ as it has only one 'for loop' with a large iteration. Even for small inputs the loop runs a lot of times, consuming more resources. The canonical solution uses a sorting-based greedy approach as shown in figure 4.4 which consumes less energy with the same time complexity.

```python
class Solution:
    def minIncrementForUnique(self, nums: List[int]) -> int:
        count = Counter(nums)
        taken = []

        ans = 0
        for x in range(10 ** 5 * 2):
            if count[x] > 1:
                taken.extend([x] * (count[x] - 1))
            elif taken and x > taken[0]:
                ans += x - taken.pop(0)
        return ans
```

Figure 4.3: problem 945 - GPT-4-turbo

```python
class Solution:
    def minIncrementForUnique(self, nums: List[int]) -> int:
        nums.sort()
        ans = 0
        for i in range(1, len(nums)):
            if nums[i] <= nums[i - 1]:
                d = nums[i - 1] - nums[i] + 1
                nums[i] += d
                ans += d
        return ans
```

Figure 4.4: problem 945 - Canonical

CHAPTER 5

CONCLUSION

In this thesis, we study the energy efficiency of code generated by various popular state-of-the-art Large Language Models. We compare the efficiency of code generated against the most optimized human-written canonical solutions and against every other LLM. It is observed that LLMs can not generate energy-efficient code in most of the cases, and can only generate code as efficiently as humans in some of the cases. This implies that sustainability goals are not embedded into these models during the training phase.

Among the studied LLMs, DeepSeek-v3 and GPT-4o generate the most efficient code, with higher accuracy. It is important to consider using models, that generate energy-efficient code in less number of retries, to reduce the impact of energy consumed in regenerating the code. We can also employ prompt engineering techniques to make the LLMs generate more energy efficient. Overall, this thesis addresses the most overlooked part i.e., the capability of generating energy-efficient code, in evaluating the performance of LLMs.

5.1 LIMITATIONS AND FUTURE WORK

Though this thesis performs a comprehensive study in evaluating LLMs' ability to generate efficient code, it has some limitations:

- LLMs are not deterministic. The same LLM might generate a different code, the next time we ask it.
- Datasets used could have been seen by LLMs during the training phase, resulting in it memorizing and generating the most efficient solution.
- This thesis is currently limited to evaluating codes generated in Python only and does not consider other programming languages.

To address the above limitations, future studies can focus on including a diverse range of programming languages and problems that were created recently but not seen by LLMs during their training phase. The impact of various prompt engineering techniques on the code generation aspect of LLMs can also be thoroughly investigated.

BIBLIOGRAPHY

[1] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: Benchmarking the efficiency of automatically generated code. In *Advances in Neural Information Processing Systems*, volume 37, pages 11506–11544. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/15807b6e09d691fe5e96cdecde6d7b80-Paper-Datasets_and_Benchmarks_Track.pdf. ix, 2, 7, 11, 12

[2] Erol Gelenbe. Electricity consumption by ict: Facts, trends, and measurements. *Ubiquity*, 2023(August), August 2023. doi: 10.1145/3613207. URL https://doi.org/10.1145/3613207. 1

[3] State of the Planet. Ai's growing carbon footprint, June 2023. URL https://news.climate.columbia.edu/2023/06/09/ais-growing-carbon-footprint/. Accessed: 2025-04-21. 1

[4] Air Transport Action Group. Facts & figures, 2023. URL https://www.atag.org/facts-figures/. Accessed: 2025-04-21. 1

[5] Lotfi Belkhir and Ahmed Elmeligi. Assessing ict global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production*, 177:448–463, 2018. 2, 4

[6] LeetCode. Problems, 2025. URL https://www.leetcode.com/problems. Accessed: 2025-04-21. 2

[7] Peter J. Denning and Ted G. Lewis. Exponential laws of computing growth. *Commun. ACM*, 60(1):54–65, December 2016. ISSN 0001-0782. doi: 10.1145/2976758. URL https://doi.org/10.1145/2976758. 4

[8] Peter J. Denning and Ted G. Lewis. Exponential laws of computing growth. *Commun. ACM*, 60(1):54–65, December 2016. ISSN 0001-0782. doi: 10.1145/2976758. URL https://doi.org/10.1145/2976758. 5

[9] Aayush Agrawal, Mayank Khichar, and Sanjeev Jain. Transient simulation of wet cooling strategies for a data center in worldwide climate zones. *Energy and Buildings*, 127:352–359, 2016. ISSN 0378-7788. doi: https://doi.org/10.1016/j.enbuild.2016.06.011. URL https://www.sciencedirect.com/science/article/pii/S037877881630500X. 5

[10] Pengfei Liu, Ranjith Kandasamy, Jin Yao Ho, Teck Neng Wong, and Kok Chuan Toh. Dynamic performance analysis and thermal modelling of a novel two-phase spray cooled rack system for data center cooling. *Energy*, 269:126835, 2023. ISSN 0360-5442. doi: https://doi.org/10.1016/j.energy.2023.126835. URL https://www.sciencedirect.com/science/article/pii/S0360544223002293. 5

[11] Soumyendu Sarkar, Avisek Naug, Ricardo Luna Gutierrez, Antonio Guillen, Vineet Gundecha, Ashwin Ramesh Babu, and Cullen Bash. Real-time carbon footprint minimization in sustainable data centers wth reinforcement learning. In *NeurIPS 2023 Workshop on Tackling Climate Change with Machine Learning*, 2023. URL https://www.climatechange.ai/papers/neurips2023/28. 5

[12] Xueliang Li and John P. Gallagher. An energy-aware programming approach for mobile application development guided by a fine-grained energy model. *CoRR*, abs/1605.05234, 2016. URL http://arxiv.org/abs/1605.05234. 6

[13] Loïc Lannelongue, Jason Grealey, and Michael Inouye. Green algorithms: Quantifying the carbon footprint of computation. 2020. 6

[14] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 119–139, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/you. 6

[15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732. 7, 8, 9

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder,

Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374. 7, 8, 9

[17] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x, 2024. URL https://arxiv.org/abs/2303.17568. 7

[18] Qiwei Peng, Yekun Chai, and Xuhong Li. HumanEval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8383–8394, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.735/. 7

[19] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7. 7

[20] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023. URL https://arxiv.org/abs/2210.14868. 7

[21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. URL https://arxiv.org/abs/2105.09938. 7

[22] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022. URL https://arxiv.org/abs/2211.11501. 7

[23] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael Lyu. Revisiting, benchmarking and exploring api recommendation: How far are we?, 2021. 7

[24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770. 7

[25] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023. URL https://arxiv.org/abs/2303.12570. 7

[26] Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. Learn to code sustainably: An empirical study on green code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, page 30–37, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705793. doi: 10.1145/3643795.3648394. URL https://doi.org/10.1145/3643795.3648394. 8

[27] Jonas F. Tuttle, Dayuan Chen, Amina Nasrin, Noe Soto, and Ziliang Zong. Can llms generate green code - a comprehensive study through leetcode. In *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*, pages 39–44, 2024. doi: 10.1109/IGSC64514.2024.00017. 8

[28] Tristan Coignion, Clément Quinton, and Romain Rouvoy. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 79–89, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717017. doi: 10.1145/3661167.3661221. URL https://doi.org/10.1145/3661167.3661221. 8

[29] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. Piloting copilot and codex: Hot temperature, cold prompts, or black magic? https://ssrn.com/abstract=4496380, 2023. Available at SSRN: https://ssrn.com/abstract=4496380 or http://dx.doi.org/10.2139/ssrn.4496380. 8

[30] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. On evaluating the efficiency of source code generated by llms. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, page 103–107, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706097. doi: 10.1145/3650105.3652295. URL https://doi.org/10.1145/3650105.3652295. 9

[31] Vlad-Andrei Cursaru, Laura Duits, Joel Milligan, Damla Ural, Berta Rodriguez Sanchez, Vincenzo Stoico, and Ivano Malavolta. A controlled experiment on the energy efficiency of the source code generated by code llama. In Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, editors, *Quality of Information and Communications Technology*, pages 161–176, Cham, 2024. Springer Nature Switzerland. 9

[32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950. 9

[33] Daniel Erhabor, Sreeharsha Udayashankar, Meiyappan Nagappan, and Samer Al-Kiswany. Measuring the runtime performance of c++ code written by humans using github copilot, 2024. URL https://arxiv.org/abs/2305.06439. 9

[34] Tom Cappendijk, Pepijn de Reus, and Ana Oprescu. Generating energy-efficient code with llms, 2024. URL https://arxiv.org/abs/2411.10599. 9

[35] Lola Solovyeva, Sophie Weidmann, and Fernando Castor. Ai-powered, but power-hungry? energy efficiency of llm-generated code, 2025. URL https://arxiv.org/abs/2502.02412. 9

[36] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010. 22

[37] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020. 24