WALKING THE TIGHTROPE: BALANCING ENERGY EFFICIENCY AND ACCURACY IN LLM-DRIVEN CODE GENERATION

by

Mats Buis

Supervisors Utrecht University: Prof. Dr. Yannis Velegrakis and Dr. Hakim Qahtan
Supervisors TNO: Aditya Ganesh and Ir. Jesse van Oort

A thesis submitted in conformity with the requirements
for the degree of Master of Science in Artificial Intelligence

Department of Information and Computing Sciences
Utrecht University

WALKING THE TIGHTROPE: BALANCING ENERGY EFFICIENCY AND ACCURACY IN LLM-DRIVEN CODE GENERATION

Mats Buis
Department of Information and Computing Sciences
Utrecht University
2024

Abstract

Large Language Models (LLMs) consume significant amounts of energy during inference, especially for computationally expensive tasks like code generation, which leads to environmental concerns. This work aims to reduce the energy consumption during inference without compromising model performance. The energy consumption of Qwen2.5-Coder-7B-Instruct, Meta-LLaMA 3.1-8B-Instruct, and DeepSeekCoder-V2-Instruct-16B was evaluated on BigCodeBench, a benchmark that consists of 1,140 diverse coding tasks, using a software-based energy measuring approach. The relations between task nature, batch size, model size, fine-tuning, Activation-Aware Weight Quantization (AWQ), and GPTQ with 8-bit and 4-bit precision were investigated for a variety of models including the Qwen2.5 models. Results indicate that task nature significantly affects energy consumption across all tested models, while batch size has a minor effect. Notably, the Meta-LLaMA model consumed 130.77% more energy than the DeepSeekCoder model while achieving lower accuracy. Fine-tuning, AWQ, GPTQ-INT8, and GPTQ-INT4 quantizations reducing energy consumption by up to 19%, 67%, 40% and 67%, respectively. GPTQ-INT8 models achieved these reductions without significantly reduced accuracy, whereas GPTQ-INT4 models showed slight decreases and AWQ showed substantially lower pass@1 scores. This work demonstrates that energy consumption of LLMs can effectively be reduced without significant performance loss, which demonstrates the importance and contributions of innovative research for sustainable AI practices.

Contents

1 Introduction **12**

1.1 Research Challenges . . . 14

1.2 Problem Statement . . . 15

1.3 Research Objectives . . . 16

2 Related Work **18**

2.1 Measuring the Energy Consumption of LLMs . . . 19

2.1.1 Hardware-Based Methods . . . 20

2.1.1.1 On-Chip Power Monitoring . . . 20

2.1.1.2 Hardware-Based Profiling . . . 20

2.1.2 Software-Based Methods . . . 20

2.1.2.1 Performance Counters and Software-Based Profiling Tools . . . 21

2.1.2.2 Energy Estimation Models . . . 21

2.1.2.3 Simulation and Emulation Tools . . . 21

2.1.2.4 Application Specific Energy Monitoring Tools . . . 21

2.1.2.5 Recent Energy Measurement Developments . . . 22

2.1.2.6 Insights on Energy Measurement . . . 22

2.2 Code Generation Efficiency . . . 23

2.2.1 Fine-tuning . . . 23

2.2.2 Model Compression Techniques . . . 23

2.2.2.1 Quantization . . . 23

2.2.2.2 Pruning . . . 24

2.2.3 Knowledge Distillation . . . 24

2.2.4 Model Architectures . . . 24

2.2.4.1 Transformer Variants . . . 24

2.2.4.2 Mixture of Experts (MoE) . . . 25

2.2.4.3 Sparse Attention Mechanisms . . . 25

2.2.5 Inference Optimisation . . . 25

2.2.5.1 Batching and Parallelization . . . 25

2.2.5.2 Hardware Acceleration . . . 26

2.2.6 Energy Measurement and Evaluation Metrics . . . 26
2.2.6.1 Energy Consumption per Token . . . 26
2.2.6.2 Carbon Footprint Estimation . . . 26
2.2.7 Insights on Efficient Code Generation . . . 26
2.3 The Efficiency-Accuracy Trade-Off . . . 26
2.3.1 Understanding the Trade-Off . . . 27
2.3.2 Factors Influencing the Trade-Off . . . 27
2.3.3 Optimal Configurations . . . 27
2.3.3.1 Hyper Parameter Optimisation . . . 27
2.3.3.2 Multi-Objective Optimisation . . . 27
2.3.3.3 Combining Optimisation Techniques . . . 28
2.3.4 Experimental Approaches . . . 28
2.3.4.1 Empirical Evaluation . . . 28
2.3.4.2 Theoretical Modeling . . . 28
2.3.5 Insights on the Energy-Accuracy Trade-Off . . . 28
2.4 Code Correctness and Functionality Preservation . . . 29
2.4.1 Code Correctness . . . 29
2.4.2 Evaluation Frameworks . . . 29
2.4.2.1 Unit Testing . . . 29
2.4.2.2 Regression Testing . . . 29
2.4.2.3 Static Analysis Tools . . . 29
2.4.2.4 Dynamic Analysis Tools . . . 30
2.4.2.5 Benchmarking . . . 30
2.4.3 Functionality Preservation . . . 30
2.4.4 Insights on Code Correctness and Functionality Preservation . . . 30
2.5 Gaps in the Literature . . . 30
2.6 Insights . . . 31

3 Methodology 33
3.1 Energy Measurement . . . 33
3.2 Data . . . 34
3.3 Materials . . . 34
3.4 Model Characteristics . . . 35
3.5 Model Architecture . . . 36
3.5.1 Decoder-Only Architectures . . . 37
3.5.1.1 Memory and Energy Consumption Estimation . . . 38
3.5.2 Mixture of Experts (MoE) Architectures . . . 38
3.6 Inference . . . 39
3.6.1 Parameter Calculation . . . 39
3.6.1.1 Word Embeddings . . . 40

3.6.1.2 Positional Encoding 40
3.6.1.3 Multi-Head Attention 40
3.6.1.4 Feedforward Networks 41
3.6.1.5 Layer Normalization 41
3.6.1.6 Output Layer 42
3.6.1.7 Mixture of Experts Layers 42
3.6.1.8 Experts (MoE) 42
3.6.1.9 Gating Network (MoE) 43
3.6.2 Total Number of Parameters 43
3.6.2.1 Decoder-Only 43
3.6.2.2 Mixture-of-Experts (MoE) 43
3.6.3 KV-cache 44
3.6.4 VRAM 46
3.6.5 Runtime 46
3.6.5.1 Prefill 46
3.6.5.2 Decode 47
3.6.5.3 Time-To-First-Token 47
3.6.5.4 Time-Per-Output Token 47
3.6.5.5 Total Generation Time (TGT) 47
3.6.6 Energy Consumption 48
3.7 Fine-Tuning 48
3.8 Quantization 48
3.8.1 Activation-aware Weight Quantization (AWQ) 48
3.8.2 GPTQ 48
3.9 Performance evaluation 49
3.10 Overview of Research Methods 49
3.11 Ethics and Privacy 49

4 Experimental Evaluation **50**
4.1 Experimental Setup 50
4.1.1 Evaluation Metrics 50
4.1.2 Scaling Laws 51
4.2 Experiments 54
4.2.1 High-Granularity Analysis 54
4.2.2 Batch Size and Energy Consumption With Coding LLMs 59
4.2.2.1 Statistical Analysis 59
4.2.3 Energy Impact of Fine-tuning, Quantization, and Model Size in Qwen2.5 Models 62
4.2.3.1 Impact of Increasing Model Size on Energy Consumption 62
4.2.3.2 Energy Impact of Fine-tuning 63

4.2.3.3 AWQ 64
4.2.3.4 GPTQ 65
4.2.4 The Energy-Accuracy Trade-Off 66
4.3 Discussion 67
4.4 Limitations 69
4.5 Future Work 70

5 Conclusion 71
5.1 Batch size 88

List of Figures

3.1 *Energy consumption of the GPU cluster during code generation* 34
3.2 *Visual representation of a 32-core CPU* 35
3.3 *Visual representation of a 1024-core GPU* 35
3.4 *Decoder-only architecture based on the Transformer architecture [70]* 38

4.1 *Total generation time build-up for* $output\_tokens = 1$ 51
4.2 *Total generation time build-up for* $output\_tokens = 10$ 51
4.3 *Scaling relationships for parameters in an LLM* 52
4.4 *Scaling relationships for parameters in the KV-cache* 53
4.5 *Scaling relationships for VRAM requirements* 54
4.6 *Energy consumption per Bigcodebench task* 55
4.7 *Number of tasks per BigCodeBench domain* 55
4.8 *Total energy consumption per task by domain* 55
4.9 *Total energy consumption per token by domain* 55
4.10 *Models used for granularity and batch size analysis* 59
4.11 *Qwen2.5 model family* 62
4.12 *Energy consumption for one Bigcodebench run vs Pass@1 rate for Qwen2.5 models* 67

List of Tables

3.1 *Explanation of Model Variables* . . . 36
3.2 *Explanation of KV-cache Calculation Variables (Aligned with Model Variables)* . . 45

4.1 *LLM characteristics* . . . 59

5.1 *Occurrence percentages of categories in BigCodeBench based on library presence in the solutions of coding tasks* . . . 77
5.2 *Durbin-Watson Test for Autocorrelation* . . . 78
5.3 *Shapiro-Wilk Normality Test Results* . . . 79
5.4 *Levene's Test for Homogeneity of Variance (Homoscedasticity)* . . . 80
5.5 *Qwen-Coder OLS regression results (batch size 1)* . . . 81
5.6 *Qwen-Coder OLS regression results (batch size 4)* . . . 82
5.7 *Qwen-Coder OLS regression results (batch size 8)* . . . 83
5.8 *Meta Llama3.1 OLS regression results (batch size 1)* . . . 84
5.9 *Meta Llama3.1 OLS regression results (batch size 4)* . . . 85
5.10 *Meta Llama3.1 OLS regression results (batch size 8)* . . . 86
5.11 *DeepSeekCoder-V2-Lite OLS regression results (batch size 1)* . . . 87
5.12 *DeepSeekCoder-V2-Lite OLS regression results (batch size 4)* . . . 87
5.13 *DeepSeekCoder-V2-Lite OLS regression results (batch size 8)* . . . 88
5.14 *LLM Computational Requirements* . . . 88
5.15 *Levene's Test for Homogeneity of Variance (Homoscedasticity)* . . . 89
5.16 *Kruskal-Wallis Test for Differences in Energy Consumption* . . . 89
5.17 *Dunn's Post-Hoc Test with Adjusted p-values* . . . 89
5.18 *Energy Consumption, Latency, Throughput, and Pass@1 for various batch sizes* . . 90
5.19 *Estimated Computational Requirements for Different Qwen2.5 Base Models* 90
5.20 *Shapiro-Wilk Test Statistics for Different Model Sizes* . . . 91
5.21 *Levene's Test for Differences in Energy Consumption by Model Size* . . . 91
5.22 *Kruskal-Wallis Test for Differences in Energy Consumption by Model Size* . . . 91
5.23 *Dunn's Test for Differences Among Model Sizes* . . . 91
5.24 *Energy Consumption, Latency, Throughput, Pass@1, and Pass@kWh for Five Qwen2.5 Base Models* . . . 92

5.25 *Expected Energy Impact of Fine-Tuning* 92
5.26 *Shapiro-Wilk Test for Normality of Fine-Tuned Models* 93
5.27 *Levene's Test for Differences in Energy Consumption for Fine-Tuned Models* 93
5.28 *Kruskal-Wallis Test for Differences in Energy Consumption for Fine-Tuned Models* 94
5.29 *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for Fine-Tuned Models* 94
5.30 *Estimated Computational Requirements for Activation-Aware Weight Quantization* 95
5.31 *Shapiro-Wilk Test for Normality* 96
5.32 *Levene's Test for Differences in Energy Consumption for Model Sizes* 96
5.33 *Kruskal-Wallis Test for Differences in Energy Consumption for Model Sizes* 97
5.34 *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for LLMs with and without AWQ* 97
5.35 *GPTQ* 98
5.36 *Shapiro-Wilk Test for Normality* 99
5.37 *Levene's Test for Differences in Energy Consumption for Model Sizes* 99
5.38 *Kruskal-Wallis Test for Differences in Energy Consumption for Model Sizes* 100
5.39 *Dunn's Post-Hoc Test with Adjusted p-values for GPTQ Models* 100
5.40 *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for GPTQ-Int8 and GPTQ-Int4* 101

Acknowledgements

I would like to express my gratitude to my supervisors Aditya Ganesh and Jesse van Oort for their guidance, feedback and support during my graduate internship at TNO. Their expertise and advice have been invaluable to the completion of this work.

Also, I would like to thank Prof. Dr. Yannis Velegrakis for the opportunity to conduct this work under his supervision at Utrecht University. His feedback, guidance and support have contributed significantly to this thesis.

Finally, I would like to thank my colleagues at TNO, whose collaboration and support are greatly appreciated.

List of Abbreviations

AI	Artificial Intelligence
AWQ	Activation-Aware Weight Quantization
CPU	Central Processing Unit
DW	Durbin-Watson
FFN	Feed-Forward Network
FLOPs	Floating Point Operations per Second
GDDR6	Graphics Double Data Rate 6
GPT	Generative Pre-trained Transformer
GPTQ	Generative Pre-trained Transformer Quantization
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
INT4	4-bit Integer Precision
INT8	8-bit Integer Precision
KV-cache	Key-Value Cache
LLM	Large Language Model
MHA	Multi-Head Attention
MoE	Mixture-of-Experts
NLP	Natural Language Processing
OLS	Ordinary Least Squares
POSIX	Portable Operating System Interface
Pthreads	POSIX Threads
PUE	Power Usage Effectiveness
SW	Shapiro-Wilk
TGT	Total Generation Time
TPOT	Time-Per-Output Token
TTFT	Time-To-First-Token
VRAM	Video Random Access Memory

Chapter 1

Introduction

Sustainability is increasingly becoming a focus in the ICT sector because of its rapidly growing demand for energy. It is recognized that hardware is not the only factor affecting the energy consumption of ICT systems. The software that controls the hardware also has a major influence on the total energy consumption of these systems [1, 2, 3]. Currently, the ICT sector faces a vast increase in energy demand due to the adoption of energy-demanding hardware and software alike [4]. With the rise of AI, this process is further reinforced [5, 6].

Large Language Models (LLMs) have become a significant factor in AI-related energy consumption. LLMs use deep learning architectures with billions of parameters, which are trained on large datasets to generate natural language [7]. These models are used in various applications such as machine translation, question answering and code generation. The widespread adoption of LLMs by the general public after the introduction of LLM-driven tools such as ChatGPT, DALL-E, and GitHub Copilot has led to an increased demand for data centers worldwide [8]. The training and inference processes of LLMs are computationally expensive and require substantial resources [9]. The size and complexity of these models have grown in recent years to improve model performance. However, this trend has also led to a significant increase in energy consumption during both training and inference [10].

In 2020, the ICT sector was responsible for 3.9% of worldwide energy consumption [11], with data centers accounting for 1% of worldwide energy consumption [6]. Furthermore, the ICT-related energy consumption is not expected to halt anytime soon [12]. The greenhouse gas (GHG) emissions related to data centers not only place a burden on the electricity grid but also pose a threat to the environment [13]. The increasing demand for energy in the ICT sector has been ongoing for a considerable period of time, and with the rapid adoption of AI and other energy-demanding software applications this trend has accelerated.

Despite recognition of their high energy consumption, the model size of LLMs has increased steadily, growing five-fold from 2018 to 2022 [14]. Considering that the worldwide electricity mix is predominantly dependent on fossil fuels such as coal, gas, and oil [15], this directly leads to a higher carbon footprint of LLMs because larger models generally require more energy. Currently, the ICT sector is estimated to be responsible for around 1.4% [11] to 1.8–2.8% [16] of global GHG emissions. Besides the direct consequence of higher energy consumption and a higher footprint, inefficient software may also impact the lifespan of hardware [17]. Given that 36% of the ICT sector's carbon footprint consists of embodied emissions [11], inefficient software can lead to significant indirect CO2 emissions. Moreover, hardware production often requires rare metals which lead to faster depletion of natural resources [18].

The economic implications of inefficient software are also impactful and have become a major topic for owners and users of computational facilities such as data centers [19], simply because increased energy consumption leads to higher energy-related costs. Furthermore, early breakdown of hardware not only leads to more e-waste [20] but also results in higher maintenance costs and capital expenditures for the organization that exploits the hardware. In turn, higher costs can lead to lower profitability, which can weaken a company's market position.

Writing code that runs fast is considered an energy-efficient coding practice because the relationship between speed and energy efficiency in code is well-known for particular use cases like sorting [21, 22, 23]. However, writing fast code is still a challenging task that requires skilled developers [24]. Optimizing the execution speed of code while considering software energy efficiency can be beneficial. However, currently available code generation tools lack the accuracy and efficiency to compete with skilled human developers [25]. Another reason for the lack of attention to energy efficiency in code is the absence of ready-to-use platforms that measure the energy consumption of code. If it is not possible to measure energy consumption and identify the causes of increased energy usage, then it is hard for developers or code generation tools to tweak software in such a way that it becomes energy efficient.

A promising set of models in the field of code production are LLMs. However, the recent introduction and widespread adoption of LLMs has further increased software-related energy consumption [26]. LLMs not only pose a threat, they also offer an opportunity to make software more sustainable by optimizing code for energy efficiency. Next Generation Software Development (NGSD) tools [27] such as Meta Llama, QwenCoder2.5, and DeepSeekCoder enable such automatic code generation. These tools have demonstrated fast code generation [24], raising questions about their energy efficiency and potential areas for optimization. In addition, it is shown that LLMs are able to solve simple coding tasks but might fail in solving more complex and challenging tasks [28]. The rapid development of LLMs toward larger and task-specific models has significantly increased their performance and applicability [29], but the energy efficiency of these models remains to be quantified.

The improved performance of coding LLMs has gained interest in both academia and the IT industry [30, 31]. In the IT industry, multiple applications could benefit from energy-efficient code generation, such as cloud computing, embedded devices and IoT devices. Higher energy efficiency of software leads to longer battery life in resource constrained environments such as in mobile and embedded technology [32, 33]. In energy-demanding environments such as cloud computing, not focusing on energy efficiency becomes a costly operation because a small percentage decrease in energy consumption can lead to large absolute energy savings.

From a societal and environmental perspective, energy-efficient code leads to lower CO2 emissions, resulting in a reduced carbon footprint. This work aims to contribute to the energy-efficient production of code in a fast, cost-effective and sustainable manner that is accessible to a wide public. Optimizing one line of code is not going to make much of a difference, but adopting energy efficient LLMs for code generation tasks on a large scale may help combat climate change. This aligns with important policies and regulations concerning the mitigation of climate change, such as the European Green Deal, which targets reducing GHG emissions by 55% by 2030 compared to 1990 levels [34]. On a global scale, reduced software-related energy consumption and GHG emissions are in line with the Paris Agreement, which aims to limit global warming to 1.5 degrees Celsius [35].

1.1 Research Challenges

Optimizing the energy efficiency of LLMs comes with various challenges with regard to the measurement of energy consumption, model selection, the energy-accuracy trade-off and ensuring the functional correctness of generated code. First, accurately measuring the energy consumption of LLMs is complex due to variability in hardware configurations and the lack of standardised measurement tools. Additionally, the feasibility of a measurement method is largely determined by the specific hardware and software environments. In cloud computing, users typically lack access to metadata about the cloud environment, making it extremely difficult to accurately collect and report individual energy consumption data. Also, making sure that the generated code is correct and usable in production is a challenge for LLM-generated code, often because of the lack of suitable testing frameworks and the artificial nature of LLM-based code generation that can suffer from hallucinations. Selecting the right LLM architecture for optimization is a crucial part of this process. Each LLM varies in accuracy and computational requirements for specific tasks. This trade-off has to be analysed depending on the specific context. Third, The size and complexity of the training dataset impacts the computational complexity of a task and the resources that it requires. This translates directly into energy consumption during training and inference. Carefully selecting and reducing the training dataset to a minimum without compromising the generalisability of the results is difficult. Finally, ensuring that optimised LLMs remain accurate and preserve the functionality of code is necessary for widespread adoption in automated code generation tools. The accuracy loss of a model should be acceptable for end-users of the generated code. Therefore, developing methods to optimise LLMs for energy efficiency while preserving their performance is still an open problem [36]. The overall complexity lies in integrating accurate energy measurement, defining and implementing LLM optimisation strategies, selecting suitable architectures, and ensuring model correctness into an approach that results in energy-efficient LLMs without degrading the coding capabilities of the model significantly.

1.2 Problem Statement

The rapid rise of AI has driven up energy consumption of data centers, networks and end-users because of increased hardware requirements, data transport and user activity [11]. This increase in energy use is accompanied by increased carbon dioxide emissions, contributing to climate change. The significance of AI in worldwide energy consumption has increased the importance of energy efficient computing [37].

One of the most direct ways to decrease the energy consumption of AI, is to ensure that popular AI tools like LLMs are energy efficient in tasks such as code generation. While methods exist for minimizing memory and computational requirements, their relationship to energy consumption during code generation tasks remains unclear, as they primarily focus on speed and memory optimizations. However, these metrics do not provide direct insight in the energy consumption of ICT-systems, which makes the carbon footprint of these systems opaque. To effectively tackle climate change, a reliable quantification of energy consumption and carbon emissions is necessary which is currently lacking [5]. Moreover, ensuring the correctness and functionality preservation of optimised code is challenging but necessary for saving resources [38]. The inadequacy of current methods requires new solutions that are able to optimise the energy efficiency of code.

LLMs are an emerging technology capable of optimizing code and scaling to large codebases. While early LLMs like GPT-2 have limited code optimization capabilities, newer LLMs like GPT-4 and Github Copilot reach accuracies of up to 85% on a variety of coding tasks [39]. Research has shown that the size of LLMs can be reduced without significantly impacting its performance with techniques like quantization [40, 41]. However, these insights have not yet been analysed in relation with energy consumption data, partly due to the difficulty in measuring the energy efficiency of code. Furthermore, selecting an LLM so that it is optimal with regard to energy consumption, is a challenging task. Additionally, ensuring code correctness and functionality preservation is non-trivial [38].

Solving these problems will provide insight in software-based energy consumption measurement methods and the ability of LLMs to efficiently produce code. If LLMs are able to efficiently generate code, this will lay the groundwork for a cost-efficient and scalable solution for automated code generation while minimising the environmental footprint of the software industry.

1.3 Research Objectives

The main goal of this work is to investigate to what extent various LLMs can be optimised to reduce their own energy consumption during code generation tasks, while preserving functionality and ensuring code correctness. The objective can be divided into the following sub-objectives:

- **Hypothesis 1:** Software-based energy monitoring tools can accurately measure and evaluate the energy consumption of LLM inference during code generation tasks. Lack of standardised measuring tools and variability in hardware make this a challenging task. To overcome this challenge, a software-based ensemble of tools is created that provide detailed GPU power consumption information during inference.

- **Hypothesis 2:** Techniques like fine-tuning and quantization can significantly reduce the energy consumption of LLMs without substantial accuracy loss. This is difficult because fine-tuning and quantizing models is a lengthy and energy intensive process. Also, it is unknown what the effect of these techniques will be on the accuracy and energy consumption. The energy consumption and model accuracy will be systematically evaluated to determine which configurations lead to minimal energy consumption but maintain accuracy.

- **Hypothesis 3:** There exists an optimal configuration of LLMs that minimises energy consumption and maintains model accuracy. Several variables have to be weighed and consensus has to be reached on the concept of optimality with regard to energy efficiency and accuracy. Exploring various LLM variants and configurations gives insight in the energy-accuracy trade-off and the concept of optimality.

- **Hypothesis 4:** Evaluation frameworks such as benchmarking and unit testing can ensure code correctness and functionality preservation for LLM-generated code. Ensuring code correctness and functionality is difficult due to the erroneous and hallucinative behaviour of LLMs. In addition, there is a lack of evaluation frameworks. The BigCodeBench benchmark will be used to employ benchmarking and unit testing for LLM-generated code and evaluate the correctness and functionality of the generated code.

The findings of this study have the potential to impact the field of artificial intelligence by laying the groundwork for incorporating sustainability in the performance analysis of LLMs. By focusing on sustainability as well as on traditional metrics like latency and throughput, this could ensure a minimal carbon footprint of AI while facilitating technological progress.

This work will use several open-source LLMs, which can be tweaked with the model optimization techniques to ensure minimal energy consumption. A software-based energy measuring approach is used to measure the energy consumption, using NVIDIA hardware data in combination with energy profiling tools to ensure accurate measurements. To evaluate the coding abilities of several LLMs, a widely accepted coding benchmark will be used. Code correctness and functionality preservation will be ensured through unit testing. Code correctness is defined as passing all of the unit tests of the BigCodeBench suite. This study focuses on both efficiency and correctness as an LLM's utility is limited if it produces incorrect code.

The structure of the thesis is as follows: chapter 2 provides an overview of related work and consists of four sections that each focus on one of the hypotheses. Chapter 3 delves deeper in the methodology which includes the selection of LLMs as well as optimization techniques, energy measurement methods and the evaluation framework. Chapter 4 presents the results of this work which includes quantitative analyses of energy consumption, model accuracy, code correctness and the efficiency-accuracy trade-off. This chapter also discusses the implications, limitations and suggestions for future work. Finally, in chapter 5 the work is summarised and addresses the significance of the study in a broader academic context.

Chapter 2

Related Work

The literature related to the energy efficiency of LLMs has multiple facets that provide insights for optimizing these models during code generation tasks. Optimizing LLMs is defined here as improving their performance while maintaining code correctness and functionality. Firstly, accurately measuring the energy consumption of LLMs during inference is necessary before reducing it. Furthermore, it is important to define the concept of energy efficiency and the available methods for improving it should be clarified. Also, the code correctness and functionality preservation must be ensured since this is not guaranteed with purely generating code. Finally, exploring the connections between energy measurement, LLM optimization techniques, code correctness and functionality, and the efficiency-accuracy trade-off will provide insight into the current state of research in this field.

The first aspect of improving the energy efficiency of LLMs is the ability to accurately measure the energy consumption of LLMs during inference [42]. Energy measurement methods can be broadly classified into hardware-based and software-based approaches [42, 43, 4]. Recent advancements have introduced software tools that can accurately measure the energy consumption of LLMs during inference [44, 45]. When hardware access is limited, such as in most cloud computing environments, software-based methods may be the only feasible option.

Various optimization techniques such as fine-tuning, pruning, model compression, distillation, and quantization are applied to improve the ability of a model on specific tasks or to reduce the memory requirements of LLMs. Recently, LLMs such as ChatGPT and Copilot have shown to rival human programmers on simple coding tasks [46]. While code generation has mainly focused on creating correct code that takes into account time and space complexity measures, it did not necessarily focus on creating energy-efficient code [47]. Optimizing LLMs for energy efficiency requires specialized techniques that balance reduced energy consumption with model accuracy.

The data used for LLMs is crucial because it largely determines how capable models will be in performing a task [48]. However, the training process of LLMs is typically also energy-intensive, making it crucial to focus on energy efficiency both in the training and inference phases. Various methods have been proposed to reduce the energy consumption of LLMs that target both the training and the inference phase.

Additionally, the functionality and the correctness of the optimised code also have to be guaranteed. However, the correctness is not always ensured, as even the state-of-the-art LLMs achieve accuracies of up to 56% on a challenging coding benchmark such as BigCodeBench. This indicates that LLM-generated code does not always produce correct results and still requires a human in the loop to ensure correctness [49].

This chapter situates this work within the broader field of LLM energy efficiency by examining the available methods and approaches in the literature. Finally, the chapter highlights how this work contributes to bridging the gap between LLM code generation accuracy and energy efficiency. This is achieved by focusing on four key aspects: measuring the energy consumption of LLMs, exploring optimization techniques for energy efficiency, analysing the energy efficiency-accuracy trade-off, and assessing the correctness of LLM-generated code.

2.1 Measuring the Energy Consumption of LLMs

Measuring the energy consumption of LLMs is necessary to quantify how much energy is required for a task such as code generation. If it is unknown, it is challenging to identify factors that contribute to higher energy use in LLMS and consequently apply strategies to optimise the energy efficiency of LLMs. Therefore, energy measurement is a crucial aspect in improving the energy efficiency of LLMs during both training and inference.

Energy measurement methods can be divided into hardware-based and software-based approaches [42, 43, 50]. Measuring the energy consumption of LLMs might not always be feasible although they are renowned for their accuracy for low-granular measurements [43]. Continuous advancements in software-based methods have made them reliable enough to measure the energy consumption of LLMs during training and inference [51].

Measuring energy consumption of LLMs is a complex task because of the many interactions between software and hardware components. LLMs often use both CPUs and GPUs and the energy consumption differs significantly between idle and active states [52]. Additionally, attributing energy consumption to specific tasks, algorithms or workloads is challenging because modern techniques like Dynamic Voltage and Frequency Scaling (DVFS) adjust hardware parameters based on workload [53]. Fine-grained methods are required to address these challenges. Understanding the characteristics of both LLMs and the underlying hardware is essential in the energy measurement and the optimization process. The following sections explore the most common and reliable methods for measuring the energy consumption of LLMs which forms the basis for strategies aiming to enhance the energy efficiency of LLMs during code generation tasks.

The techniques for measuring the energy consumption of LLMs can roughly be divided into two categories: hardware-based and software-based methods. Hardware-based methods use physical power meters or sensors to measure the energy consumption of software. This typically involves placing a physical device between the hardware running the LLMs and the power source. Consequently, the device measures the energy that flows from the power source to the computer. Software-based methods use models and tools to estimate the energy consumption based on log-files or information that is available via the performance monitoring tools of hardware [51].

2.1.1 Hardware-Based Methods

Hardware-based methods use specific physical power meters (e.g. Kill-a-Watt meters) to measure the energy consumption of the hardware running the LLMs. These device provide real-time data but usually offer low-granularity information which makes it difficult to attribute energy consumption to specific processes or tasks within an LLM inference task. Moreover, physical access to the hardware is needed which is often impractical in cloud computing environments which is nowadays a fairly common practice.

2.1.1.1 On-Chip Power Monitoring

Hardware like CPUs and GPUs sometimes have on-chip power monitoring which uses built-in sensors to measure the energy consumption of workloads. Tools like Intel's Running Average Power Limit (RAPL) [54] and NVIDIA's NVML library [55] provide real-time power draw data with higher granularity. These methods are useful for determining the energy consumption of LLMs during both training and inference because they offer detailed insights on the process level. It must be noted that the accuracy of these methods depends on the precision of the built-in sensors which are often platform specific. This limits the applicability across different hardware setups.

2.1.1.2 Hardware-Based Profiling

Hardware manufacturers also have provided energy profiling tools that can be used to determine the energy consumption of LLMs. For instance, NVIDIA's nvidia-smi tool allows users to monitor GPU power usage during LLM inference [56]. Similarly, AMD and other manufacturers provide tools that allow for energy profiling of their hardware platforms. However, they are usually platform specific so changes in hardware may require different tools for fine-grained energy measurements, which complicates cross-platform analysis.

2.1.2 Software-Based Methods

When hardware access is limited or impractical, software-based methods are crucial for measuring the energy consumption of LLM workloads. These methods provide valuable insights without the need for additional hardware while they can be faster and more cost-effective [54]. Software-based methods include performance counters and profiling tools, energy estimation models, power profiling libraries and APIs, simulation and emulation tools, and application-specific energy monitoring tools.

2.1.2.1 Performance Counters and Software-Based Profiling Tools

During LLM workloads, performance counters and profiling tools can collect utilization data from CPUs and GPUs. Tools like Linux's perf, Intel VTune, and AMD uProf can be used to profile LLM workloads [42]. These tools help to identify energy spikes in code and understand the hardware utilization patterns of LLMs. However, these tools are often hardware-specific and may not provide direct energy measurements but require additional modeling to accurately estimate the energy consumption of workloads.

2.1.2.2 Energy Estimation Models

Energy estimation models use statistical methods to estimate the energy consumption of LLMs based on several software performance metrics. Applications like Microsoft's Joulemeter [57] and the PowerAPI toolkit [58] allow estimation without directly accessing the underlying hardware. The advantage of these methods is that they are platform independent. However, their accuracy heavily depends on the quality of the underlying model and the assumptions made about the hardware, which can be challenging given the complexity of LLM workloads.

2.1.2.3 Simulation and Emulation Tools

Simulation and emulation tools like gem5 [59] and QEMU [60] can model energy consumption of hardware that run LLMs under different conditions. In addition to the energy estimation models these simulation and emulation tools can test these models under various conditions to observe how changes in the environment affect energy consumption. However, these simulations can be computationally expensive and the models may not fully capture the real-world dynamics of running LLMs, which may limit their practical applicability.

2.1.2.4 Application Specific Energy Monitoring Tools

Certain applications sometimes have their own energy monitoring tools to measure their energy consumption. Tools like MySQL Power Consumption Monitoring [61] and the energy Profiler in Android Apps [62] are not directly applicable to LLMs but they do illustrate have specialized tools can be developed for LLM workloads. Developing tools for LLMs could provide detailed insights on the interaction between various LLM components and their effect on energy consumption. However, this remains an area for future research.

2.1.2.5 Recent Energy Measurement Developments

With increasing interest in the energy efficiency of LLMs, new tools and frameworks have been presented to accurately measure the energy consumption of software. For instance, CodeCarbon [63] is an open-source software package that estimates the energy consumption and carbon footprint of code, including LLMs. Additionally, the EnergyVis framework provides [64] visualization tools to improve the accuracy of measurements.

2.1.2.6 Insights on Energy Measurement

Hardware-based methods offer accurate measurements but their applicability is limited with regard to LLM energy measurement because of their low-granularity. Besides, the need for physical access to the hardware makes these methods infeasible to use for cloud-based deployments which is fairly common for LLM training and inference. Tools like NVIDIA's nvidia-smi and Intel's RAPL provide valuable energy consumption data but are platform-specific hence lacking cross-platform compatibility. The scale on which LLMs are used nowadays often requires energy consumption measurement across multiple nodes or GPUs, which further limits the use of hardware-based measurements. Therefore, while hardware-based methods are valuable, they may not always be feasible to use for the LLM energy consumption measurement in real-world scenarios.

Software-based methods offer flexibility and are often more practical for energy measurement of LLMs, especially in environments where access is restricted. However, many tools are platform-specific which limits their generalizability. In addition, accurately modeling the energy consumption of LLMs can be challenging due to their complexity and the scale of resources used. Despite these challenges, software-based methods offer valuable insights to researchers and developers that aim to minimise the energy consumption of LLMs.

In summary, measuring the energy consumption of LLMs requires careful assessment of the available methods and their applicability to a specific use case. The choice between either hardware-based and software-based methods depends on a number of factors such as hardware accessibility, required measurement granularity, and the deployment environment of the LLMs. Given the popularity of cloud-based LLM deployments, software-based methods seem to be the most feasible option. However, a combination of hardware-based and software-based methods may be necessary to achieve high-granularity insight in the energy consumption of LLMs and arrive at accurate and reliable measurements. Understanding the available methods is crucial to develop effective measurement strategies to improve the energy efficiency of LLMs, which is a key focus of this work.

2.2 Code Generation Efficiency

To optimise the energy efficiency of LLMs during code generation tasks, it is essential to understand how the inference processes of these models work. Energy efficiency in this context refers to the minimal use of energy for LLMs during inference, and specifically during code generation tasks. Several factors influence the energy consumption of LLMs such as model architecture, computational complexity, and inference optimisations. Metrics used to determine the energy efficiency of LLMs can be energy consumption per generated line of code, latency, memory utilization and throughput.

Optimising the energy efficiency of LLMs during code generation can be approached at multiple levels and includes fine-tuning, model compression techniques, efficient architectural design and inference optimisations [65]. Techniques such as pruning, quantization, and knowledge distillation have been explored to minimise the size of LLMs without affecting their performance significantly [41, 40].

2.2.1 Fine-tuning

Fine-tuning methods show promising results for improving the coding performance of LLMs while reducing energy consumption [66]. The fine-tuning process focuses on adjusting certain parameters in the LLM that are associated with a specific tasks such as coding. The advantage of such an approach is that only a limited amount of parameters need to be adjusted which saves time and energy while the performance of the model improves on that task.

2.2.2 Model Compression Techniques

Compression techniques are essential for minimising the size and computational requirements of LLMs, which increases the energy efficiency during inference [67]. Key methods include fine-tuning, quantization, pruning and knowledge distillation.

2.2.2.1 Quantization

Quantization is the process of reducing the precision of numerical representations of a model's parameters or aspects of parameters. Precisions typically range from 32-bit floating-point to lower-bit representations such as 8-bit, 4-bit integers or even lower precisions [40]. Reducing the precision decreases the computational load and memory usage during LLM inference. Recent works like GPTQ [40] and AWQ [41] have demonstrated methods to shrink model size by adjusting the precision of parameters with minimal effect on model accuracy.

Activation-aware Weight Quantization is a quantization method that focuses on the optimisation of deep neural networks by reducing the precision of the weights while taking into account the behaviour of the weight activations within the networks. The precision of parameter weights ranges from FP32 where a 32-bit floating point number is used to store the weights, to INT4 where a 4-bit integer is used to store the weights and sometimes goes even lower to 1-bit numbers. This approach is used to minimise the performance loss of the model by adjusting weight precision based on the specific distribution of activations in a layer. The precision of the weights is not changed uniformly but instead based on activation ranges and patterns. By adjusting the quantization levels dynamically, AWQ reduces computational requirements while maintaining accuracy. Research demonstrates AWQ significantly reduces model size with minimal performance [41].

A more rigorous approach to compress LLMs in size is GPTQ which applies quantization to all tensor types, weights, activations, gradients, and feature maps within a model. This approach aims to balance between computational efficiency and model accuracy by applying quantization across multiple facets of a tensor. Often applied precisions are 8-bit and 4-bit, which can reduce the memory usage and the corresponding computational requirements without substantial model degradation [40].

2.2.2.2 Pruning

Pruning techniques aim to identify less important weights or connections in a model and consequently removing them to minimise the size of the model [68]. By making models sparse, pruning reduces the number of computations during inference which in turn can reduce the energy consumption. However, there is a balance between pruning and model performance since it could decrease model performance.

2.2.3 Knowledge Distillation

The process of a larger model ”teaching” a smaller model by transferring information, is known as knowledge distillation. The smaller ”student” model is trained to replicate the outputs of the larger model to achieve similar performance with significantly less parameters and computational requirements. This in turn can lead to reduced energy consumption. This techniques is mainly used to ensure that they are deployable on devices with limited resources [69].

2.2.4 Model Architectures

Designing efficient model architectures is another strategy that can be taken to improve the energy efficiency of LLMs. Innovations in architecture can reduce the computational complexity and thereby reduce the energy requirements of models.

2.2.4.1 Transformer Variants

The original Transformer architecture [70] has been adapted multiple times to create more computationally efficient variants. Models like Albert [71] and Reformer [72] introduce architectural adjustments that reduce memory consumption and computational requirements. Albert uses parameter sharing and factorizes embeddings while Reformer replaces full attention with locality-sensitive hashing, that both decrease computational requirements. Also, Decoder-only models such as GPT-3[7], have been used to optimise the inference process in text generation tasks and do not use the encoder part of the Transformer architecture.

2.2.4.2 Mixture of Experts (MoE)

Mixture of Experts models bundle multiple neural network architectures connected by multiple experts. Together with a gating mechanism these models determine to which submodel input is directed [73]. The advantage of these models is that the computational requirements do not increase linearly increase with the number of parameters because only a subset of parameters is active during inference [74].

In code generation tasks these models offer additional advantages since multiple experts can specialise in different languages, techniques or domains which can lead to higher performance [75]. The sparse activation mechanism present in MoE models ensures that computational resources are focused where they are most needed which minimises computational resources.

2.2.4.3 Sparse Attention Mechanisms

Reducing the quadratic complexity of standard attention mechanism to computationally less demanding variants that limit the attention to a subset of tokens, is what sparse attention mechanisms do in practice [76]. Models like Sparse Transformer and Longformer [77] use these sparse attention mechanisms to handle longer sequences more efficiently.

2.2.5 Inference Optimisation

Optimising the inference process of LLMs is crucial to decrease the overall energy consumption of LLMs, especially for LLMs that are being frequently used. Strategies to improve this process involve efficient batching, parallelization, and leveraging hardware accelerators.

2.2.5.1 Batching and Parallelization

Batching input sequences together allows for parallel processing, which increases hardware utilization and reduces the per-sample energy consumption [78]. Efficiency can be further improved by parallelize across multiple GPUs or using multi-threading. It must be noted that these techniques should take into account throughput and latency constraints to ensure optimal performance.

2.2.5.2 Hardware Acceleration

Specialised hardware accelerators like Tensor Processing Units (TPUs) [79] or newer GPUs optimised for AI workloads can significantly improve the energy efficiency of LLMs. These accelerators are designed to efficiently handle matrix operations compared to general-purpose CPUs.

2.2.6 Energy Measurement and Evaluation Metrics

As elaborately discussed in 2.1, accurate energy measurement and appropriate evaluation metrics are essential in determining the energy efficiency of LLMs during code generation.

2.2.6.1 Energy Consumption per Token

The energy consumption per token is a metric that measures the average energy required to generate each token during inference [80]. This metric provides a standardised way to compare the energy efficiency of different LLMs. With estimation model the total code generation time can be calculated per token which can then be translated into energy consumption per token based on the energy consumption of the underlying hardware.

2.2.6.2 Carbon Footprint Estimation

Based on metrics like the energy consumption per token, the carbon footprint associated with LLMs can provide insight in the environmental impact of deploying these models [81]. Tools like CodeCarbon [63] allow researchers to estimate and track the carbon emissions of code which can also be applied to the LLMs deployed during code generation tasks.

2.2.7 Insights on Efficient Code Generation

Integrating and using these techniques and strategies before running LLM workloads can significantly decrease the computational requirements of models during code generation tasks. By focusing on techniques like model compressions, efficient architectures and inference optimizations, it is possible to reduce the computational load without substantially compromising the performance of LLMs. This integrated approach of combining multiple methods is essential for sustainable AI in which the environmental impact of LLMs is minimised.

2.3 The Efficiency-Accuracy Trade-Off

The introduction of LLMs for code generation tasks has already disrupted software development since they are widely used by developers. However, these models are computationally demanding which leads to significant energy consumption during training and inference [39]. From a sustainability perspective, finding efficient or even optimal configurations for LLMs with regard to energy efficiency and accuracy can greatly reduce the energy consumption of these models. This section further explores the opportunities to find the optimal configuration for LLMs, focusing on techniques and strategies that balance energy efficiency and performance.

2.3.1 Understanding the Trade-Off

Optimising LLMs involves navigating the space of energy efficiency and model accuracy. Reducing energy consumption often means simplifying the model to ensure it uses minimal computational resources which can result in performance loss. On the other hand, enhancing accuracy may increase necessary computational resources and corresponding energy consumption. Therefore, finding an optimal configuration is necessary to balance these sometimes conflicting goals. [9].

2.3.2 Factors Influencing the Trade-Off

Several factors influence the energy consumption and accuracy of LLMs such as model size and architecture. Larger models with more parameters typically achieve higher accuracy but require more computational resources [7]. Furthermore, the settings of a model influence the trade-off such as learning rate, batch size and the number of layers [82]. Methods like fine-tuning, quantization, and pruning can alter the computational requirements of the model [65].

2.3.3 Optimal Configurations

2.3.3.1 Hyper Parameter Optimisation

Hyper parameters involve the settings that are provided to a model instead of being learned by the model, which is also part of the optimal configuration of LLMs. Techniques such as grid search, random search, and Bayesian optimisation can be applied to explore the hyper parameter space efficiently [83].

Incorporating energy consumption as an objective in hyper parameter tuning allows for the identification of the balance between energy consumption and accuracy. Multi-objective frameworks can be used to to optimise for both objectives simultaneously [84].

2.3.3.2 Multi-Objective Optimisation

In multi-objective optimisation, multiple objectives are being optimised at the same time such as minimising energy consumption and maximising accuracy. In multi-objective optimisation, Pareto optimality is a key concept, where a solution is considered optimal if no other solution can optimise one of the objective without worsening another.

By applying multi-objective optimisation to LLM configurations, it is possible to generate a Pareto front of potential optimal solutions, offering a range of trade-offs between energy efficiency and accuracy. The place on the Pareto optimal front can be determined based on the requirements of the user. Leveraging AutoML tools can automatically search for optimal configurations [85].

2.3.3.3 Combining Optimisation Techniques

Integrating multiple optimisation techniques such as fine-tuning, quantization, pruning and knowledge distillation can lead to new Pareto optimal configurations. For instance, fine-tuning can be combined with quantization. Applying techniques like AWQ and GPTQ adjusts the model to lower precisions without significant accuracy loss [86]. Furthermore pruning and knowledge distillation can be combined. Pruning reduces model size while knowledge distillation transfers knowledge to a smaller model which could lead to improved performance [87]. Careful calibration and analysis is needed to determine which optimisations could be suitable for optimising LLMs.

2.3.4 Experimental Approaches

2.3.4.1 Empirical Evaluation

Besides a theoretical basis for the experiments conducted for optimal configurations, empirical data will provide the eventual evidence for applicability of these methods. Measuring metrics such as Wh per generated token and pass@k rate help in comparing different configurations. A study that might help finding optimal configurations can train different models with various levels of quantization. Consequently, the energy consumption of the models can be measured by using tools like CodeCarbon [63] during code generation tasks. Afterwards, the generated code can be evaluated on coding benchmarks such as HumanEval [88] and BigCodeBench [89].

2.3.4.2 Theoretical Modeling

The empirical results can in turn steer new research into the right direction and based on theoretical frameworks the search space of optimal configurations can be narrowed down.

2.3.5 Insights on the Energy-Accuracy Trade-Off

Identifying optimal configurations of LLMs that minimise energy consumption while maintaining model accuracy is a challenging due to the multidimensional nature of the problem. Due to the complexity of the size of the search space it is infeasible to do an exhaustive search in the (hyper) parameter space of LLMs. Besides, optimal configurations can also vary depending on the nature of the workload. Hardware constraints can also limit the options and put constraints on the compatibility of certain optimisation techniques. By understanding the components of the problem it is possible to gain insight in the factors affecting energy consumption and accuracy in LLMs. Besides, techniques like hyper parameter optimisation, multi-objective optimisation, and combining several model compression methods, it is possible to find configurations that satisfy these criteria. This does not only enhance the sustainability of LLM inference but can be applied in general in the broader context of sustainable AI practices.

2.4 Code Correctness and Functionality Preservation

LLMs are powerful tools for developers in programming tasks. However, LLM-generated code also raises concerns about correctness and functionality due to potential errors or unintended behaviours present in the code [88]. Code correctness and functionality preservation are essential for the reliability and energy efficiency of code. Incorrect code can lead to additional computing cycles and resources spent on debugging and re-running programs which increases energy consumption [90]. This section further explores how evaluation frameworks such as benchmarking and unit testing can ensure code correctness and functionality preservation for LLM-generated code.

2.4.1 Code Correctness

LLMs can generate faulty code that is syntactically correct but semantically flawed which can lead to bugs or security vulnerabilities [88]. These issues arise when LLMs hallucinate, which is fairly common in LLMs [91]. Therefore, robust frameworks are required to ensure that generate code is correct and preserves functionality.

2.4.2 Evaluation Frameworks

Evaluation frameworks are important in verifying the code correctness and functionality of LLM-generated code. Techniques such as unit testing, regression testing [92], static and dynamic analysis tools [93] and benchmarking [88, 89] can identify issues and provide insight in the quality of LLM-generated code.

2.4.2.1 Unit Testing

Unit testing involves the isolation and testing of individual code components to verify if each part operates as intended in isolation [94]. Unit tests can validate whether each module or function behaves as expected and catch errors that are not directly apparent in most use cases. There exist automated unit testing tools that can be utilized to create test case for generated code [95].

2.4.2.2 Regression Testing

Regression testing ensures that new code does not distort the functionality of the code. In the context of LLM-generated code this method can test whether integrating LLM-generated code does not introduce bugs or break existing features [96].

2.4.2.3 Static Analysis Tools

Executing code to test for errors and energy efficiency is a computationally demanding task. Therefore, static analysis tools exist to examine code without executing it. These tools can identify errors, code smells and security vulnerabilities [97]. Tools like SonarQube and Pylint can analyse LLM-generated code to test it for coding standards and detect issues in an early stage [98]. These tools are essential for code quality and minimising the number of errors.

2.4.2.4 Dynamic Analysis Tools

Dynamic analysis tools involves executing code and consequently monitoring its behaviour to detect issues such as memory leaks, performance bottlenecks and concurrency issues [99]. Dynamic analysis can uncover runtime errors that static analysis might miss which complements static analysis tools [100].

2.4.2.5 Benchmarking

A standardized way to evaluate the performance and correctness of code is provided by benchmarking tools [101]. Frameworks like HumanEval [88], MBPP (MultiPL-E Benchmark for Programming Problems) [102] and BigCodeBench [89] were designed to assess and compare the capacities of LLMs on programming tasks. Models can be evaluated on these benchmarks which often also have built-in testing suits to assess the correctness and functionality of the generated code.

2.4.3 Functionality Preservation

Functionality preservation ensures that the generated code is not only syntactically correct and compilable but also maintains its functionality as stated in the prompt or code that needs to be optimised. This component is crucial for LLM-generated code as it can be difficult to translate a prompt, with potential ambiguous or unclear instructions, to correct and functional code. Techniques like unit testing testing and formal verification can be used to ensure that the correctness and functionality remain intact after integration or optimisation [103].

2.4.4 Insights on Code Correctness and Functionality Preservation

Ensuring code correctness and functionality preservation are necessary components to enable automatic code generation and incorporate machine generate code in software development processes. Frameworks and techniques like unit testing, regression testing, static and dynamic analysis tools and benchmarking play an important role in verifying the quality of code. By applying these techniques developers can reduce the errors in LLM-generated code and improve sustainability by minimising the computational resources required to take correct and functional code in production.

2.5 Gaps in the Literature

The literature review has exposed several gaps in the literature in understanding and improving the energy efficiency of LLMs. These gaps align with the hypotheses formulated in the introduction of this study. First of all, the accuracy and effectiveness of software-based energy measurement methods are not yet fully established for LLM inference [42, 51]. Most existing tools either lack the ability for high-granularity measurement or are platform- and application specific and might not be suitable for the energy measurement during LLM inference tasks. Second, methods like fine-tuning, quantization, pruning and model distillation have shown to decrease computational requirements significantly without substantial accuracy loss [40, 41]. Nonetheless, there is a lack of studies quantifying the energy savings as a result of these methods, even in the field of LLMs. In addition, the trade-offs between energy and accuracy are not yet well understood, which stresses the importance of further research to quantify the effects of these methods on the energy consumption of LLMs. Moreover, the concept of configuring LLMs to minimise energy consumption while maintaining model accuracy remains under explored [104]. Existing studies often focus on maximising the accuracy of a model or on one optimisation method instead of focusing on energy consumption. There is a need for research that incorporates energy efficiency as part of the multi-objective optimisation problem. Finally, evaluation frameworks such as benchmarking and unit testing are commonly used in software development. Although research on code correctness and functionality preservation is gaining traction, in-depth research for LLM-generated code is still scarce and not thoroughly investigated yet [88, 93]. Taking into account that LLMs can produce syntactically and compilable code but are semantically flawed shows the need for these methods for LLM-generated code. To conclude, literature shows increased interest in improving the energy efficiency of LLMs but significant gaps exists for energy measurement, understanding the energy-accuracy trade-off, finding optimal trade-offs and ensuring the code correctness and preserving the functionality of LLM-generated code.

2.6 Insights

The literature provides valuable insights in the challenges associated with energy efficient use of LLMs in code generation tasks. They have the potential to disrupt the software development industry further if the accuracy in code generation tasks increases, but their energy consumption still raises sustainability concerns.

Accurate energy measurement of LLM inference is essential to optimise these models with regard to energy efficiency. However, existing tools may not yet be able to accurately measure the energy consumption of LLMs on a high-granular level during code generation tasks which stresses the need for more fine-grained measurement methods. Optmisation methods such as fine-tuning, quantization offer promising results for the energy reduction in LLMs without significantly impacting their accuracy [40, 41]. Nonetheless, these methods also consume energy and require evaluation to determine their effectiveness. Striking the right balance between energy efficiency and accuracy is here of importance. This involves a detailed analysis of model architecture, hyper parameters, and necessary computational resources. However, available methods in the field of multi-objective optimisation can play a role in finding this balance. The accuracy of LLMs to produce correct and functional code is here an essential aspect to prevent additional energy consumption of LLMs. Evaluation frameworks such as benchmarks, and unit testing, are suitable tools to assess the reliability of LLM-generated code.

These insights show the importance of a comprehensive frameworks that integrates these four aspects to ensure that code can be produced with minimal energy by LLMs. This aligns with the broader vision of Sustainable AI which seeks to strike a balance between technological progress and environmental responsibility [105].

Chapter 3

Framework and Methodology

While speed and space complexity are well-known metrics for evaluating the efficiency of LLMs, the energy efficiency of code generation is rarely incorporated into such evaluations [47]. This work aims to address the gaps in the literature presented in the previous chapter and focuses on accurate energy measurement and optimisation of energy consumption in LLM code generation tasks while maintaining code correctness and preserving functionality. Open-source LLMs are used to enable modifications to parameters and be able to run these models in isolation which is not typically possible with commercial models that often restrict access to the underlying structure of an LLM [106]. By leveraging open-source models, techniques like fine-tuning and quantization can be applied to optimise for energy efficiency while aiming for minimal accuracy loss.

3.1 Energy Measurement

A software-based approach is taken to measure the energy consumption of the LLMs, consistent with findings in the literature that such methods are practical and effective for energy measurement in LLMs. The energy consumption is measured in isolation by ensuring that the LLMs are fully run on a single GPU. For the NVIDIA L40S GPUs, the energy consumption of a process running on a GPU can be monitored via nvtop, which provides access to queries and kernel execution details [55]. This software-based approach enables users to read hardware information in near real-time, which makes it possible to identify how running an LLM affects the energy consumption of a GPU. Additionally, the Multi-Threaded Synchronized Monitoring approach is used to verify the initial measurement. This method identifies the window of kernel execution. Consequently, POSIX threads (Pthreads) are used for synchronization, dedicating one thread to recording the power consumption. The power consumption during model inference minus the idle state power consumption represents the total energy consumption, as illustrated in Figure 3.1.

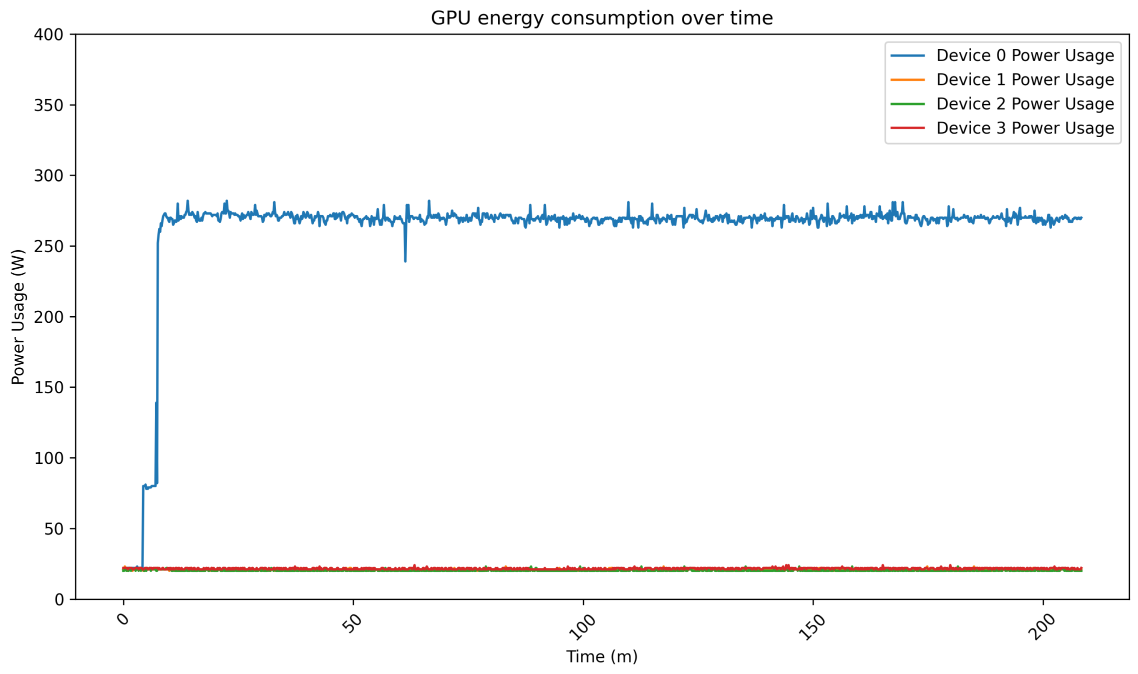


Figure 3.1: *Energy consumption of the GPU cluster during code generation*

3.2 Data

The data used in this work comes from the BigCodeBench benchmark, which is a diverse set of 1,140 programming tasks designed to test the real-world coding capacities of LLMs in as diverse ways as possible [89]. BigCodeBench builds on other benchmarks such as HumanEval, MBPP, and DS-1000 which each focus on specific qualities of coding expertise. BigCodeBench aims to bundle these quality measures into one benchmark. The LLMs are prompted to generate code solutions for particular programming tasks. Subsequently, the correctness of the generated code is assessed by calculating the pass@k metric, which provides the chance that at least one of the top k generated samples is correct [88]. The power consumption of a programming task is calculated by adding the energy consumed during code generation and code evaluation. The LLMs are compared based on power consumption and model performance for code generation tasks.

3.3 Materials

The experiments are run on LLMs from the Qwen2.5 model family [107] and range in size from 0.5B to 14B. All models are run on a single Nvidia L40S GPU within a four Nvidia L40S GPU cluster. Each Nvidia L40S GPU has 48GB of Graphics Double Data Rate 6 (GDDR6) memory with Error-Correcting Code (ECC). This means that the GPU offers faster data transfer rates and higher reliability. The energy consumption of an Nvidia L40S GPU is 350 watt at 100% utilization. An AMD EPYC 9354 32-Core Processor is used to handle essential functions like task management and data transfer between the CPU and GPU memory. The CPU has a managing function in running a workload and therefore also consumes some energy but this is minimal compared to the energy consumption of the GPU and the CPU energy consumption is more complex, hence this aspect is excluded from this work.

Figure 3.2: *Visual representation of a 32-core CPU*

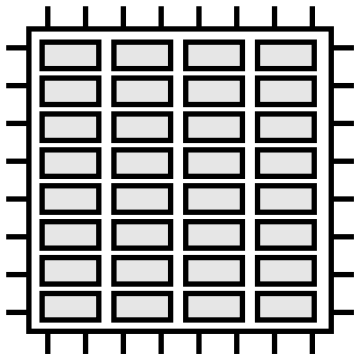
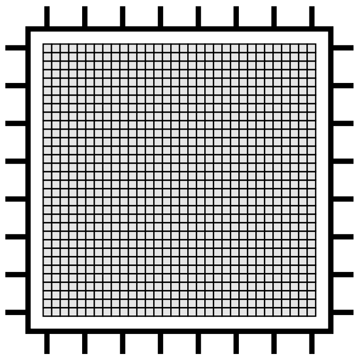


Figure 3.3: *Visual representation of a 1024-core GPU*



3.4 Model Characteristics

To accurately estimate the power consumption of an LLM it is necessary to understand the interactions and relationships between variables of an LLM. Nowadays, LLMs often use a transformer architecture that consists of an encoder and a decoder part. Decoder-only models are mostly used for text generation or code completion. The architecture, model size and computational complexity directly influence the computational resources needed during inference [104]. Key variables that impact the computational requirements of LLMs are the number of layers (I), model dimensionality (m), number of attention heads (h) and the vocabulary size (v). Table 3.1 summarises the most important variables of LLMs. This work focuses on several decoder-only models with varying model sizes and a Mixture-of-Experts (MoE) model to see how model characteristics relate to energy consumption during code generation tasks

Table 3.1: *Explanation of Model Variables*

Variable	Description
Model Name	The name of the model (e.g., Qwen2.5-3B-Instruct, Meta-LLaMA 3.1-8B-Instruct)
$n_{params}(n)$	The number of parameters in the model which refers to the total learnable weights in the model.
$l_{layers}(l)$	The number of layers in the model. This is the total number of transformer blocks stacked in the model.
$m_{model}(m)$	The dimensionality of the model, representing the size of each input and output vector in the transformer layers.
$h_{heads}(h)$	The number of attention heads in each transformer block. Each head is responsible for capturing different aspects of attention.
$d_{head}(d)$	The dimensionality of each attention head. This is usually equal to m_{model}/n_{heads}.
$n_{vocab}(v)$	The size of the model's vocabulary, i.e., the total number of unique tokens the model can recognize.
$n_{ctx}(c)$	The maximum context length or the maximum number of tokens the model can process for a single input.

3.5 Model Architecture

The architecture is a determining factor for the eventual computational complexity of the model, which is closely related to the power consumption of a model [10]. Except for the DeepSeekCoder-V2-Lite-16B model, ll of the models used in this work are Decoder-only models. This means that they do not make use of the entire encoder part of the original encoder-decoder architecture [70]. It must be noted that models that use a decoder-only architecture also require some steps that are normally part of the encoder. Figure 3.4 depicts a visual representation of a decoder-only model and the steps necessary for code generation.

Encoder

1. Input embedding
2. Positional encoding
3. Multi-head attention (self-attention)
4. Feed-forward networks
5. Layer normalization

Decoder

1. Masked multi-head attention (self-attention)
2. Cross attention (optional)
3. Feed-forward networks
4. Layer normalization
5. Output generation

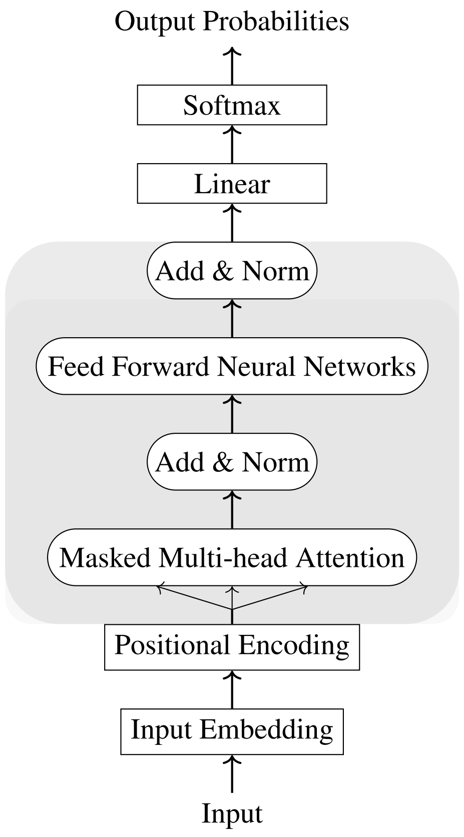
3.5.1 Decoder-Only Architectures

Decoder-only models such as Qwen2.5-Coder-Instruct and Meta-LLaMA 3.1-8B-Instruct only use the decoder component of the Transformer architecture for tasks like code generation [107, 108].

As shown in Figure 3.4, the model processes input sequences through an embedding layer and adds positional encodings to capture the order of the input tokens. The core component of the decoder model is the multi-head self-attention mechanism, which allows the model to attend to previous tokens in a sequence while preventing access to future tokens to ensure causality [70]. After the multi-head attention mechanism, the feed-forward networks and layer-normalization layers are placed. Directly after this phase, the tokens are generated one-by-one whereby each tokens depends on previously generated tokens.

Decoder-only models handle resources more efficiently compared to encoder-decoder models. This makes it a suitable option in environments where resources are limited. Understanding the interactions in an LLM architecture is essential to be able to accurately estimate the computational requirements of running a model. Model architecture directly relates to memory usage and energy consumption during inference. By gaining insight in the number of layers, attention heads, and other model parameters, it is possible to predict the resource demands and optimise the energy efficiency of models without degrading the performance of the model.

Figure 3.4: *Decoder-only architecture based on the Transformer architecture [70]*



3.5.1.1 Memory and Energy Consumption Estimation

As concluded from the decoder-only architecture analysis, memory requirements are influenced by the size of the model parameters and the key-value (KV) cache used during inference. The keys and values of the previously generated tokens are stored in the KV-cache. This techniques allows for more efficient computations. The size of the KV-cache grows linearly with the sequence length and the number of attention heads on which will be elaborate further in the scaling laws section.

The energy consumption during inference can be estimated by considering both the computational operations (e.g. FLOPs) and memory accesses. The number of FLOPs are in turn determined by the model size and the sequence length. Memory access patterns on the other hand also contribute significantly to energy usage.

By integrating these factors, it is possible to establish a cost model to estimate the energy consumption of the model during inference, which crucial in effectively planning LLM workloads.

3.5.2 Mixture of Experts (MoE) Architectures

The DeepSeekCoder-V2-Lite-16B model is a so-called Mixture of Experts model which trains multiple experts to which an input can be assigned. Based on the ”specialisation” of the expert an input is assigned to it and each expert has its own specialisation [109]. This generally results in cheaper inference since an input does not interact with all the parameters in a model. A MoE model also has a gating layer that directs the input to an expert. To obtain the probabilities of directing it to an expert a softmax function is used:

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_j^N e^{h(x)_j}} \tag{3.1}$$

where $h(x)_i$ is the gating function's output for expert i, and N is the total number of experts.

The output of the MoE layer is the convex combination of the weighted sum of the expert outputs:

$$y = \sum_{i \in \mathbb{T}} p_i(x) E_i(x) \tag{3.2}$$

Typically, only the top-k experts with the highest gating probabilities are considered, where $k < N$.

Shaping a formula that estimates the number of parameters for every MoE model is challenging due to variations in their architectures. In this case our approximation model will take into account the architecture of the DeepSeekCoder-V2 model.

3.6 Inference

Inference of an LLM can roughly be divided into the prefill and the decode parts. The prefill part is defined as the part in which the context used by the LLM is processed. The output tokens are only generated after the prefill stage is completed. The decode part determines the speed of code generation. Import metrics during the inference part of running an LLM are time-to-first-token (TTFT), time-per-output-token (TPOT) and total-generation-time (TGT) [110].

Roughly speaking, two aspects influence the memory requirements of an LLM: the parameter weights and KV-cache. The KV-cache holds the key-value embeddings of the input context and earlier generated tokens. Consequently, the VRAM usage is influenced by the batch size and the sequence length. A thorough understanding of the model architecture and characteristics is important to effectively estimate the power consumption of an LLM.

3.6.1 Parameter Calculation

The number of parameters is usually stated on the model card for an LLM, but it can also be calculated based on the architecture of an LLM. This is not only useful for situations in which only the characteristics of a model are known but it can also be used to make accurate estimations of memory requirements and desired resource capacities before developing or running models. The process of estimating the number of parameters is based on the steps taken in Table 3.4. To arrive at an accurate estimate of the number of parameters, memory requirements and the power consumption of an LLM, the number of parameters are counted for each part of the decoder transformer.

3.6.1.1 Word Embeddings

The first layer that requires parameters to operate, is the word embeddings layer, where the word embedding matrix (W_e) is computed. The embedding layer embeds the input tokens whereby the number of tokens is represented by the vocabulary size v.

$$h_0 = UW_e + W_p \tag{3.3}$$

In Equation 3.3 [111], it can be seen that during the computation of the sum of token embeddings and their positional encodings (h_0), the word embedding matrix is computed and multiplied with matrix U. $U = (u_{-k}, ..., u_{-1})$ is the context of vector tokens before the position embedding matrix is added to it. Matrix U, which is of shape (c, v), is here transformed into h_0 which is of size (c, m). Therefore, we know that matrix W_e must be of size (v, m) according to the matrix multiplication rule. Consequently, the total number of embedding parameters can be calculated by the following formula:

$$n_{embedding\,params} = v \cdot m \tag{3.4}$$

3.6.1.2 Positional Encoding

The next part of the LLM that requires parameters is the position embedding matrix (W_p) [70]. According to matrix addition rules matrices can only be added when the number of rows and columns is equal. Hence, the position embedding matrix (W_p) is of size (v, m). This ensures the embedding of the positions of the input tokens. The positional encoding aim to capture the order in which words occur in a text. The number of parameters required to store the positional encodings can be calculated by:

$$n_{positional\,params} = c \cdot m \tag{3.5}$$

3.6.1.3 Multi-Head Attention

Multi-head attention is used within the initial transformer architecture [70] and allows a model to focus on multiple positions of the input sequence:

The total number of parameters for the multi-head attention (MHA) component is calculated as:

$$\begin{aligned} MultiHead(Q,K,V) &= Concat(head_1,\ldots,head_h)W^O \\ \text{where} \quad head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.6}$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ [70].

Each attention head has three parameter matrices: one for Q, K and V that are all of size (m, d). Each attention head has thus size $3 \cdot (m, d)$. Additionally, there is the output projection matrix (W^O) that combines the attention head matrices in each layer. The number of parameters for (W^O) can be calculated by $h \cdot d \cdot m$. Now that the number of parameters for each individual attention head and for the concatenated heads can be calculated separately, the total number of parameters for the Multi-head attention part is:

$$n_{MHA\,params} = (3 \cdot mdh + mdh) \cdot l = 4 \cdot m \cdot d \cdot h \cdot l \tag{3.7}$$

3.6.1.4 Feedforward Networks

In the feedforward networks (FFN) of an LLM, parameters are needed to transform the inputs of the model. From [70] it can be learned that:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{3.8}$$

Here W_1 and W_2 are weight matrices and b_1 and b_2 are the bias terms. The parameters in each FFN layer are composed of W_1, W_2, b_1, b_2. Here the concept of intermediate size is introduced which is denoted by d_{ff}. Oftentimes $d_{ff} = 4m$ but this is not always the case. Therefore the term d_{ff} is used instead of 4m. The weight matrices W_1 and W_2 are of size (m, d_{ff}) and (d_{ff}, m) respectively. The bias terms are and the bias terms are of size d_{ff} and m so the total number of parameters per weight matrix is $d_{ff} + m$. The total number of parameters per FFN layer is equal to:

$$n_{FFN\,params\,per\,layer} = (d_{ff} \cdot m)^2 + d_{ff} + m \tag{3.9}$$

The total number of parameters in the FFN layers is:

$$n_{FFN\,params} = ((d_{ff} \cdot m)^2 + d_{ff} + m) \cdot l \tag{3.10}$$

3.6.1.5 Layer Normalization

The layer normalization step normalizes the input before passing it on to the attention layer. The layer normalization layer formula is [112]:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{Var[x] + \epsilon}} \cdot \gamma + \beta \tag{3.11}$$

The learnable parameters in this formula are λ and β which are both of size $m.$ This holds for every normalization layer (k) that needs to be normalized hence the formula for the total number of parameters in the layer normalization step is equal to:

$$n_{LN\,params} = 2 \cdot m \cdot k \tag{3.12}$$

3.6.1.6 Output Layer

The output layer projects the hidden states back to the vocabulary size. Hence the number of output layer parameters can be calculated by:

$$n_{output\,params} = m \cdot v + v \tag{3.13}$$

3.6.1.7 Mixture of Experts Layers

In a mixture of experts model other parameters come into play, such as the intermediate size of the MoE model which differs from the intermediate size of the larger decoder model. For simplicity the intermediate size of a model is set to be four times the hidden size of the model. However in recent times, the complexity of the models has increased and the architectures have become more diverse. Therefore it does not hold to set the intermediate size of a model to be four times the hidden size. For the DeepSeekCoder model the intermediate size is approximately 1.5 times the hidden size.

3.6.1.8 Experts (MoE)

Within the MoE model there are stil FFN layers which are structured in the following way there are also two weight matrices W_1 and W_2 and two bias terms b_1 and b_2 [113]. When the moe intermediate size is given we scale this with a scaling factor of 1.5 to account for overhead within the MoE layers and denote the result as j.

$$j = 1.5 \cdot moe\,intermediate\,size \tag{3.14}$$

$$\begin{aligned} n_{params\,per\,expert} &= W_1 + W_2 + b_1 + b_2 \\ &= (m \cdot j) + (j \cdot m) + i + m \end{aligned} \tag{3.15}$$

To obtain the total number of experts we add the number of routed experts to the number of shared experts in the model.

$$total\_experts = n_{routed\_experts} + n_{shared\_experts} \tag{3.16}$$

Consequently the number of experts is multiplied by the number of number of parameters per expert.

$$n_{expert\,params} = n_{params\,per\,expert} \cdot total\_experts \tag{3.17}$$

3.6.1.9 Gating Network (MoE)

The gating network that ensures that each input is connected to the right experts also requires parameters. This is calculated by multiplying the hidden size of the model times the number of routes experts.

$$n_{gating\,params} = m \cdot n_{routed\,experts} \tag{3.18}$$

3.6.2 Total Number of Parameters

To estimate the total number of parameters in an LLM the following formulas can be used. The first formula is for a dense transformer model and the second formula is for a MoE model.

3.6.2.1 Decoder-Only

The total number of parameters can be calculated by adding all the parameters from previous steps to each other. This boils down to the embedding layer, positional encoding, attention, feedforward network, layer normalization and output parameters.

$$\begin{aligned} n_{params} &= n_{emb\,params} + n_{enc\,params} + n_{MHA\,params} + n_{FFN\,params} + n_{LN\,params} + n_{output\,params} \\ &= vm + cm + 4mdhl + (d_{ff} \cdot m)^2 l + 5ml + 2mk + mv + v \end{aligned} \tag{3.19}$$

3.6.2.2 Mixture-of-Experts (MoE)

The total number of parameters for a MoE model is slightly different since this involves the calculation of the Mixture of Experts model parameters based on the number of experts in a layer, the number of shared feed-forward network parameters and the gating network parameters:

$$n_{MoE\,params\,per\,layer} = n_{expert\,params} + n_{shared\,FFN\,params} + n_{gating\,params} \tag{3.20}$$

The number of MoE parameters is then multiplied with the number of layers in the model:

$$n_{MoE\,params} = n_{MoE\,params\,per\,layer} \cdot MoE\,layers \tag{3.21}$$

Finally, the calculation is more or less similar to the number of parameters in the decoder-only model:

$$n_{MoE\,params} = n_{emb\,params} + n_{enc\,params} + n_{MHA\,params} + n_{MoE\,params} + n_{LN\,params} + n_{output\,params} \tag{3.22}$$

3.6.3 KV-cache

The KV-cache requires memory for the storage of the key and value for each of the attention heads of a tensor. The KV-cache enables more efficient computation of the attention scores in an LLM [7]. How much memory every tensor parameter requires, depends on the precision of the parameter. Roughly speaking, there are four commonly used precision levels: full-precision (FP32) 4 bytes/parameter, half-precision (BF16, FP16) 2 bytes/parameter, quantized data (INT8, FP8) 1 byte/parameter and quantized data (INT4) 0.5 byte/parameter.

Table 3.2: *Explanation of KV-cache Calculation Variables (Aligned with Model Variables)*

Variable	**Description**
$n_{params}(n)$	The number of parameters in the model which refers to the total learnable weights in the model.
$l_{layers}(l)$	The number of attention layers in the model, corresponding to the number of transformer blocks.
$h_{heads}(h)$	The number of attention heads per attention layer. Each head performs its own self-attention calculation.
$d_{model}(m)$	The dimensionality of each attention layer, aligning with the model's dimensionality.
p	The number of bytes required for storing a parameter. This is typically 4 bytes (32-bit floating point), 2 bytes (16-bit), or 1 byte (8-bit).
b	The batch size, i.e., the number of input sequences processed together in parallel.
s	The total sequence length, representing the number of tokens per sequence input to the model.
2	A constant, representing each layer's key and value cache.

The size of the KV-cache per token in bytes can be calculated with the following formula:

$$KV_{token} = 2 \cdot n_{params} \cdot (h_{heads} \cdot d_{model}) \cdot p \tag{3.23}$$

As a general rule of thumb we can calculate the required memory space for the parameters of an LLM in the following way: every 32-bit parameter requires 32 bits which is equal to 4 bytes. When we multiply this with the number of parameters in the model, we arrive at a memory requirement of 4*1,000,000,000 = 4 GB. Also, the size of the KV-cache grows linearly with batch size and sequence length. It is thus perfectly possible that the required memory for the KV-cache exceeds the required memory of the model weights. The size of the total KV-cache can be calculated with the following formula [114]:

$$KV_{total} = 2 \cdot b \cdot s \cdot n_{params} \cdot h_{heads} \cdot d_{model} \cdot p \tag{3.24}$$

The size of the KV-cache can be reduced by reducing the number of attention heads. This can be done by either applying multi-head attention (MHA) or multi-query attention (MQA). When these methods are applied, the number of heads can be reduced depending on the method applied. The parameter that determines how much groups of heads there are is given by (g). The new formula then becomes:

$$KV_{total} = 2 \cdot b \cdot s \cdot n_{params} \cdot \frac{h_{heads}}{g} \cdot d_{model} \cdot p \tag{3.25}$$

where b is the batch size and s is the sequence length.

Reducing the number of attention heads or using techniques like multi-query attention (MQA) can help reduce the size of the KV-cache [115].

3.6.4 VRAM

The total VRAM usage in bytes of a GPU can be calculated by multiplying the number of parameters by p, usually $p = 2$ given that there are 16-bits or 2 bytes necessary for each parameter. The second part of the equation calculates the VRAM requirement for the KV-cache, which is multiplied by the precision as well. The same applies here, p is 2 for half-precision. It must be noted that p for the parameters and the KV-cache can differ, hence there are two p denotations incorporated in the equation.

$$VRAM = (n_{parameters}) \cdot p_{parameters} + (KV - cache \cdot p_{KV-cache}) \tag{3.26}$$

3.6.5 Runtime

The runtime of an LLM eventually determines the energy consumption of a programming task. The longer a machine is dedicated to running that specific task, the more energy has to be assigned to that task to complete it. The task can roughly be divided into to parts: prefill and decode that are both related to the FLOPs and HBM rate of a GPU. The relationships are shown in this section.

3.6.5.1 Prefill

The formula for the prefill is the number of multiplications and summations for each of the parameters in the model. This where $2 \cdot N$ comes from. In the prefill stage the number of tokens loaded is equal to s hence, it is multiplied by s. The batch size determines how often this process is repeated hence the compute is multiplied by s.

$$Prefill\, compute = 2 \cdot n_{params} \cdot b \cdot s \tag{3.27}$$

$$Prefill\, memory = 2 \cdot n_{params} \tag{3.28}$$

3.6.5.2 Decode

In the decode phase, the number of tokens processed in the decode phase is equal to 1. Again the number of parameters is multiplied by 2 since there is a multiplication and summation for each parameter. Finally it is multiplied by the batch size since this determines how often this process is repeated. For the decode stage:

$$Decode\,compute = 2 \cdot n_{params} \cdot b \cdot 1 \tag{3.29}$$

$$Decode\,memory = 2 \cdot n_{params} \tag{3.30}$$

3.6.5.3 Time-To-First-Token

The time-to-first-token meausres the latency from the moment a request is made to when the first output token is received. In NLP applications the waiting time is important for user experience and they aim to minimise the TTFT. The TTFT is obtained by dividing the prefill compute by the FLOPs rate. The FLOPs rate is the total number of floating point operations per second. In addition, this number is added to the the prefill memory divided by the high bandwidth memory (HBM) times to each other [110]. The HBM rate gives here the data transfer speed in high-performance computing systems such as GPUs.

$$TTFT = \frac{Prefill\,Compute}{FLOPs\,rate} + \frac{Prefill\,Memory}{HBM\,rate} \tag{3.31}$$

3.6.5.4 Time-Per-Output Token

The TTFT is not the only aspect that determines the time it takes to generate an output. The time-per-output token is the time necessary to generate one subsequent token. This metric is obtained by dividing the decode compute or divided by the FLOPs rate and the decode memory divided by the HBM rate [110].

$$TPOT = \frac{Decode\,Compute}{FLOPs\,rate} + \frac{Decode\,Memory}{HBM\,rate} \tag{3.32}$$

3.6.5.5 Total Generation Time (TGT)

The total generation time represents the time to complete a code generation task. The TGT can be obtained by taking the time to first token and multiplying the time per output token times the sequence length as given in Equation 3.33.

$$TGT = TTFT + TPOT \cdot n_{tokens} \tag{3.33}$$

3.6.6 Energy Consumption

The energy consumption of the model is related to both computational operations and memory accesses. Using the power draw in Watt of a machine P and the time the machine is ran t, the energy efficiency can be expressed as:

$$Energy\,consumption = P \cdot t \tag{3.34}$$

3.7 Fine-Tuning

Fine-tuning is commonly used to increase the performance of LLMs in code generation tasks by pre-training an LLM on a task or domain specific dataset. As such, the accuracy of a model can be improved during inference tasks [116].This not only improves the performance of the models on coding benchmarks, it also increases the practical use of the model since it is increasingly able to handle user instructions. The models that will be used in the first part of the experiments are all fine-tuned models and the DeepSeekCoder and the CodeQwen models are also fine-tuned specifically on coding tasks.

3.8 Quantization

Quantization methods reduce the precison of a model parameters so that a lower number of bits is required for each parameter. The effect is that the memory requirements and thus the space that a model requires on hardware diminishes [86]. In this work two types of quantization are considered: AWQ and GPTQ.

3.8.1 Activation-aware Weight Quantization (AWQ)

The first quantization method that is applied to the Qwen2.5 models is Activation-aware Weight Quantization (AWQ). This method dynamically adjusts the precision of the weights and the activations. Which weights and activations it adjusts and to what extent is determined based on the distribution of the activations in the network layers to maximise model accuracy while minimising the size of the model [41].

3.8.2 GPTQ

The second method used in this work is GPTQ, which quantizes weights, activations and gradients to maximise the memory reduction. The extent to which the tensor types can also be determined and can differ from 1-bit per tensor type to 32-bits per tensor type. Here the 8-bit and the 4-bit variants are tested. These methods balance the computational efficiency with accuracy [40].

3.9 Performance evaluation

The performance of AI-systems can be defined in the form of throughput, latency or accuracy. However, in code generation tasks, the usefulness of a model lies in its capacity to produce correct code. A metric that aims to evaluate the functional correctness of code is the $pass@k$ metric [88].

This metric estimates the probability that at least one of the top k generated samples is correct. It is assumed that there are c correct samples in the total number of generated samples. The total number of combinations to choose k from n combinations is given by $C(n, k)$. The number of ways to draw from the incorrect samples is denoted by $C(n - c, k)$. From this it follows that the probability that a draw is incorrect is given by:

$$p = \frac{C(n-c,k)}{C(n,k)} \tag{3.35}$$

The probability that at least one of the k generated samples is correct or the expect value of the generated samples is denoted by [88]:

$$pass@k := \mathbb{E}_{problems}\left[1 - \frac{C(n-c,k)}{C(n,k)}\right] \tag{3.36}$$

Consequently, the pass@k rate is processed as well as the energy consumption in kWh to evaluate the pass@k rate per kWh achieved. The pass@kWh rate is calculated as follows for each LLM:

$$pass@kWh = \frac{pass@k}{energy\,consumption(kWh)} \tag{3.37}$$

3.10 Overview of Research Methods

The methodology has provided an overview of the methods and techniques used in the evaluation and optimisation of the energy efficiency of LLMs during code generation tasks. A variety of open-source LLMs are used for solving coding tasks. Besides, energy efficiency is incorporated in the assessment of model performance by using methods like pass@kwh, which contributes to Sustainable AI practices and addresses the research gaps identified in the previous chapter.

3.11 Ethics and Privacy

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences was conducted (see Annex A). It classified this research as low risk with no fuller ethics review or privacy assessment required.

Chapter 4

Experimental Evaluation

The previous chapters have provided essential information about measuring energy consumption, LLMs, their coding abilities and available techniques to optimise their energy efficiency. This chapter provides an experimental evaluation of the proposed methods to increase the energy efficiency of LLMs. Since coding tasks are generally very diverse, it is important to evaluate the framework on a representative benchmark. In this case the BigCodeBench was chosen, given the wide variety of coding tasks that are incorporated in the benchmark. The experimental setup section dives deeper into the data, metrics, software and hardware used during the experiment.

The experimental evaluation consists of experiments conducted with the LLaMA3.1-8B-Instruct, DeepSeek-V2-Lite-Instruct-16B models and the Qwen2.5-Coder-7B-Instruct models which are used to assess the coding performance of three of the most powerful open-source models. To further assess the effects of fine-tuning and quantization, fine-tuned and quantized versions of the Qwen2.5 models are used to generate code. Since Qwen2.5 has a wide-range of models available ranging from 0.5B to 72B models, the Qwen2.5-Coder model was selected to see how increases in model size relate to the accuracy of a model. Figure 4.11 shows the taxonomy of the Qwen2.5 models used to unravel how model size, fine-tuning and quantization are related to energy consumption.

4.1 Experimental Setup

4.1.1 Evaluation Metrics

To evaluate the abilities of LLMs to generate functional correct code, the pass@k metric is used. As stated in the Methodology section, this metric gives the probability that the top k generated samples passes the unit tests for a set of problems. In addition to the pass@k metric, the pass@kWh metric is also used to indicate how much one kWh contributes to the pass@k rate.

Other important metrics include the prefill and decode which are required for the computation of the TTFT and TPOT. The next step includes the total generation time which indicates how long VRAM is used. Based on the duration of the GPU usage the total energy consumption can be calculated by simply multiplying the duration with the power draw of the GPU. The energy consumption can be used to assess the usability of these models for code generation tasks. Finally the pass@k score is compared with the energy consumption to uncover the energy efficiency of an LLM in code generation tasks.

The total generation time can be divided into the time-to-first-token and the time per output token. Below is a hypothetical situation where the system outputs only one token. As one can see the contribution of TTFT and TPOT is roughly equal. However the next Figure shows that the largest part of the TGT is made up of token generation as the number of output tokens increases. This unsurprisingly since Equation 3.33 shows that the TPOT is multiplied by the total number of tokens generated.

Figure 4.1: *Total generation time build-up for* $output\_tokens = 1$

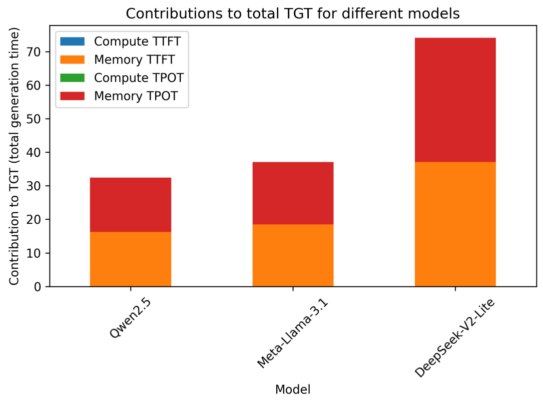
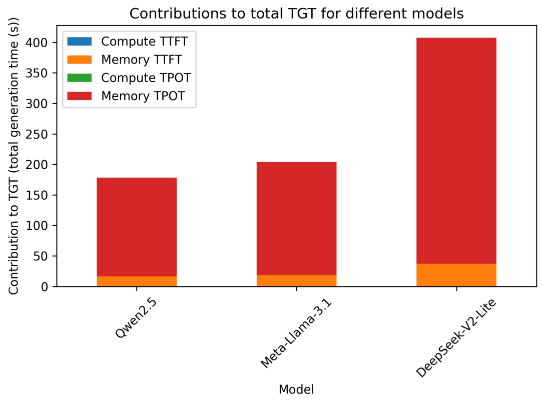


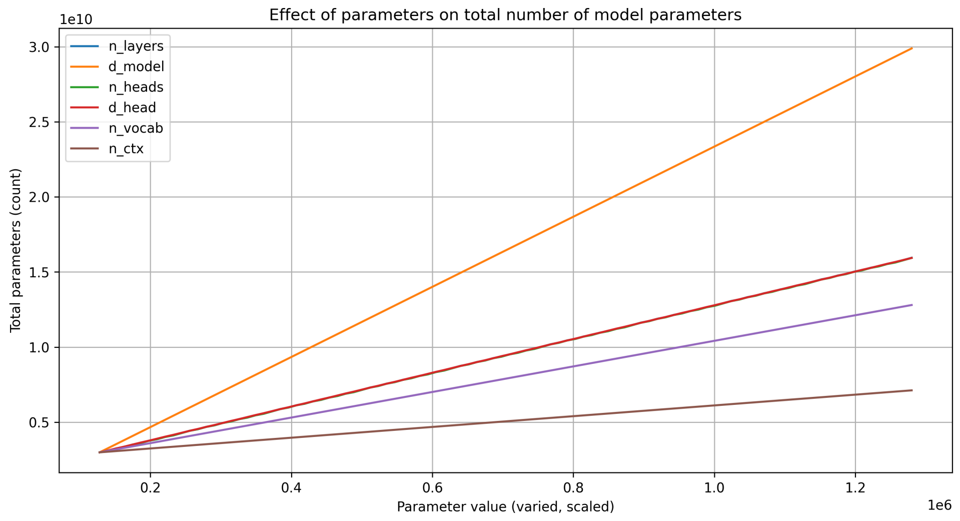
Figure 4.2: *Total generation time build-up for* $output\_tokens = 10$



4.1.2 Scaling Laws

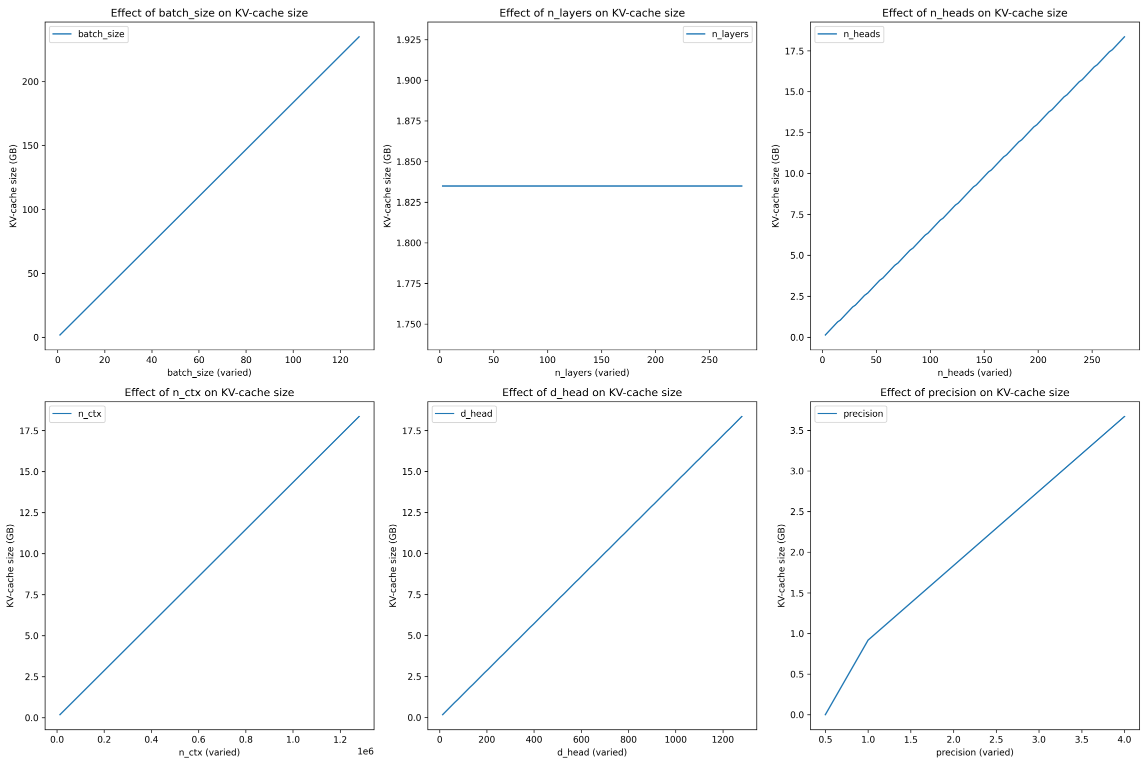
Understanding scaling laws of LLMs is crucial to see how changes in an LLM affect the required computational resources during inference. In Figure 4.3 it is shown how the number of parameters in a model scales with the number of layers (I), the hidden size (m), the number of attention heads (h), the head dimension (d), the vocabulary size (v) and the context size (c). As can be observed, all the relationships are linear relationships but the number of parameters scales faster than the other three parameters.

Figure 4.3: *Scaling relationships for parameters in an LLM*



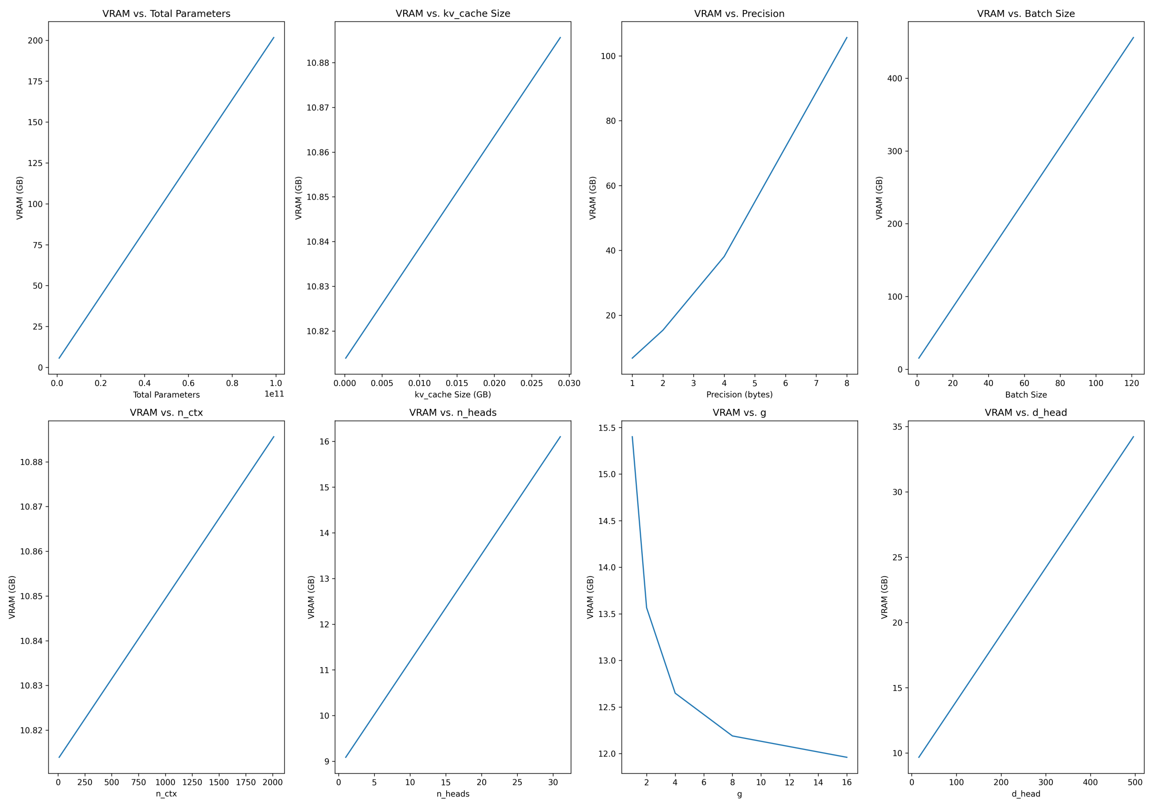
The relation between several variables and the size of the KV-cache is shown in Figure 4.4. The relationships in the KV-cache are also linear but the slope differs significantly. The batch size causes the KV-cache to increase for the Qwen2.5-Coder-7B-Instruct model with approximately 1.875 GB for every batch extra. The number of attention heads is usually determined by the attention architecture so this cannot be scaled upwards easily. However, it can be seen that for every 10 attention heads, approximately 0.67 GB extra KV-cache space is necessary.

Figure 4.4: *Scaling relationships for parameters in the KV-cache*



Similarly, the relationships between the precision, number of parameters and the KV-cache with the VRAM requirements can be shown for the Qwen2.5-Coder-7B-Instruct model as shown 4.5. The effects of most parameters on the required VRAM are (nearly) linear. Only when g increases, the VRAM requirements decrease because the hidden size (h) is divided by the number of attention heads (g) in Equation 3.25.

Figure 4.5: *Scaling relationships for VRAM requirements*



4.2 Experiments

The previously mentioned metrics will utilized as a measure of performance for a range of experiments that will show whether there is a relationship between batch size and energy consumption. Additionally, the relationship between model size and energy consumption on the one hand and the pass@k score on the other hand is assessed to see whether a linear increase in model size, also causes a linear increase in energy consumption. The second part of the experiments focuses on the Qwen2.5 models that are adjusted with fine-tuning, AWQ and GPTQ to see what the effects of these techniques are on the energy consumption and pass@k rate of these models.

4.2.1 High-Granularity Analysis

Within the BigCodeBench [89] there are seven categories to which a task can belong based on the libraries necessary to complete each task, as shown in Table 5.1 .

Figure 4.6: *Energy consumption per Bigcodebench task*

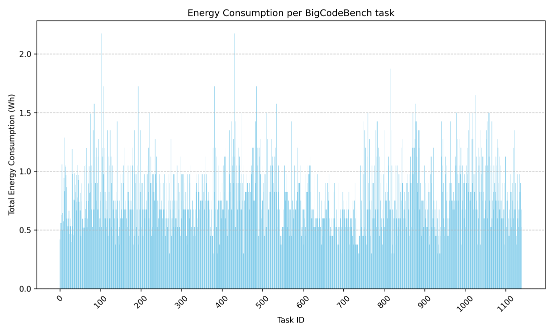


Figure 4.7: *Number of tasks per BigCodeBench domain*

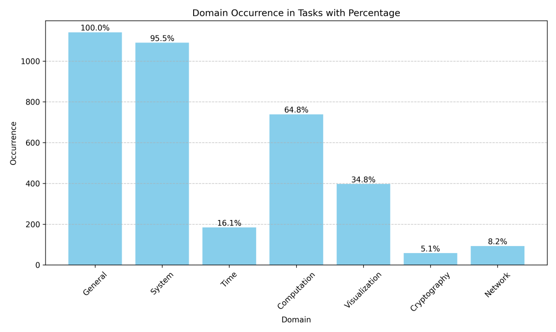


Figure 4.8: *Total energy consumption per task by domain*

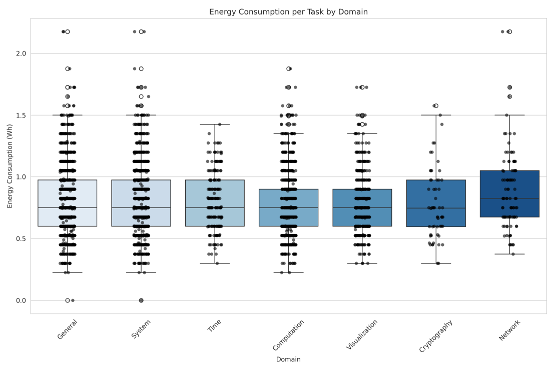
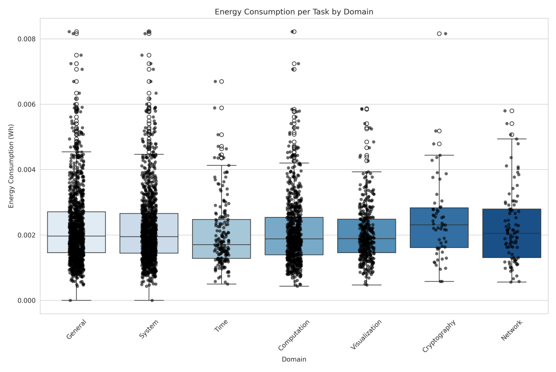


Figure 4.9: *Total energy consumption per token by domain*



Before testing for statistical differences between There are four factors that are important to assess before choosing a statistical test and performing it:

- Nature of the data
- Autocorrelation
- Normality
- Homoscedasticity

The outcome variable energy consumption is continuous which is a prerequisite for some statistical tests. Given that the tests in the BigCodeBench are executed so quickly after each other there might be autocorrelation between the tasks. In addition, the normality of the data is of importance since some tests like ANOVA or a t-test require the data to be normally distributed. Another common assumption in statistical tests is the homogeneity of variances, also called homoscedasticity. If these assumptions are violated the tests might not be suitable for this data. Several tests are available to test for autocorrelation, normality and homogeneity.

First of all, the residuals of the energy consumption data are tested for autocorrelation by using the Durbin-Watson test. The Durbin-Watson statistic can range from 0 to 4 whereby a DW-value close to 2 indicates no autocorrelation, a value lower than 2 indicates positive autocorrelation and a value higher than 2 indicates negative autocorrelation. In Table 5.2 the DW-statistics are shown for the models Qwen2.5-Coder-7B-Instruct, Meta-LLaMA 3.1-8B-Instruct and DeepSeekCoder-V2-Lite-16B and for varying batch sizes of these models. For the Qwen2.5-Coder-Instruct model the DW-values range from 0.73 for batch size 1 to 0.80 for batch size 8 which indicates strong positive autocorrelation. The Meta-LLaMA 3.1-8B-Instruct model has DW-values varying from 0.56 for batch size 1, 0.94 for batch size 4 to 0.59 for batch size 8. Also, for the Meta-LLaMA 3.1-8B-Instruct model this indicates the presence of strong positive autocorrelation. The DeepSeekCoder-V2-Lite model consistently shows extremely low DW-values ranging from 0.2693 (batch size 1), 0.2098 (batch size 4) and 0.23 (batch size 8). This indicates extreme autocorrelation across all batch sizes. Overall, all three models show strong autocorrelation for all batch sizes which indicate dependencies over time for which should be taken during model selection.

Consequently, the Shapiro-Wilk test is used to test whether the data is normally distributed. This test provides a SW-statistic and a p-value. The Shapiro-Wilk statistic gives a value between 0 and 1 whereby a value close to 1 indicates normality while a value close to 0 indicates a deviation from normality. The Qwen2.5-Coder-Instruct Shapiro-Wilk statistics are extremely low around 0.27-0.29 with corresponding p-value below 0.05 which indicates severe non-normality in the data for all batch sizes. For the Meta-LLaMA 3.1-8B-Instruct model the Shapiro-Wilk statistics are between 0.20 and 0.31 which also indicates strong non-normality of the data. The p-values all lie around 10^-54 and are therefore statistically signifcant. The Shapiro-Wilk statistics for the DeepSeekCoder-V2-Lite model range between 0.8264 for batch size 8 and 0.9478 for batch size 4 which indicates that the data is closer to normality than for the other models. However the p-values for the three batch sizes are all < 0.001 which means that the non-normality is highly significant. These results show that transformation of the data might be necessary before using the data in analysis and that parametric tests might be inappropriate.

Finally, Levene's test is used to assess whether the variances of residuals are equal over time which would indicate homoscedasticity. For the Qwen2.5-Coder-Instruct model the Levene's statistics are 0.005, 1.77 and 1.09 with corresponding p-values, 0.95, 0.18 and 0.30 respectively which indicates homoscedasticity for all three batch sizes. The Meta-LLaMA 3.1-8B-Instruct model Levene's statistics are 2.44, 0.11 and 0.80. The corresponding p-values are 0.12, 0.75 and 0.37 which are all not statistically significant hence we can conclude that these models residuals are also homoscedastic across all batch sizes. The DeepSeekCoder-V2-Lite shows different results whereby the Levene's statistics are 125.51, 179,09 and 25.62. The corresponding p-values are all < 0.001

which are all highly significant. These results showcase heteroscedasticity in the residuals of the energy consumption data of the DeepSeekCoder model. A model that accounts for heteroscedasticity must be used to control for this condition.

In summary, the outcome variable is continuous, the data exhibits autocorrelation, lacks temporal independence, deviates from normality but maintains homoscedasticity in the Qwen2.5-Coder-Instruct and Meta-LLaMA 3.1 models. However, in the DeepSeekCoder there is heteroscedasticity in the residuals. This combination of factors makes a fixed-effects OLS model with robust standard errors the most suitable model to assess potential differences in energy consumption between the domains given in 5.1. The advantage of such a model is that it is resilient to non-normality which is present in all three models. The robust standard errors ensure here that the results remain reliable even when the data is non-normally distributed. Besides, the robust standard errors address heteroscedasticity which is present in the DeepSeekCoder-V2-Lite model. Additionally, the complexity of the OLS model is relatively low, making it simple to interpret. To ensure the handling of autocorrelation, a lagged energy variable is included which partially addresses the autocorrelation. Although it does not fully model temporal dependencies as is possible in an AR structure, the lagged variable serves as a reasonably approximation for the time-related dependencies. The linmitations of this model are mainly that it does not fully account for autocorrelation and that it does not model task-specific effects. This may lead to reduced accuracy of the domain-level estimates.

In analyzing the Ordinary Least Squares (OLS) regression results for the three models, multiple factors are assessed that influence the energy consumption of coding tasks in the Bigcodebench. These models include the effects of time, number of solution tokens and the domains for each coding task. These factors collectively form the explanatory power regarding energy consumption. are shown in Table 5.5 up until Table 5.13.

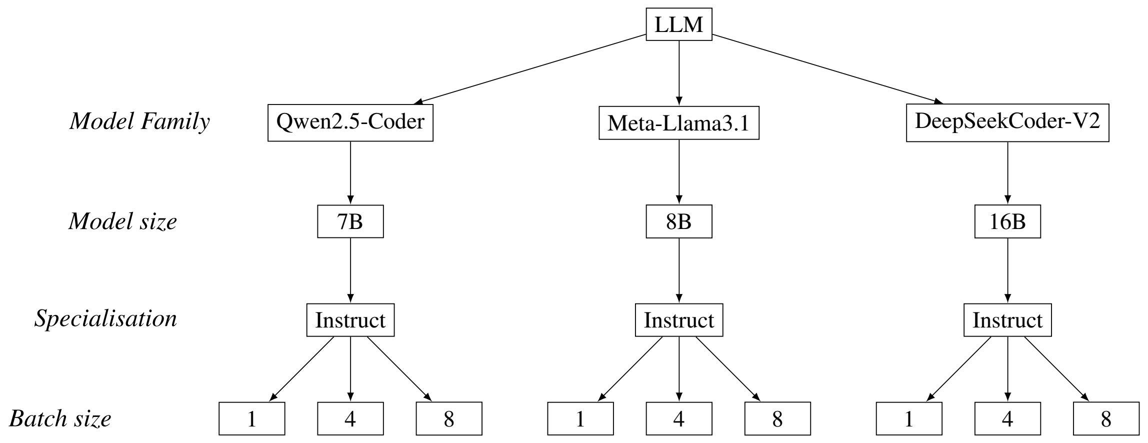
The adjusted R^2 statistic is for all models around 98% $(0.979 - 0.983)$ which indicates that almost all variance can be explained by the factors in the model. The high F-statistic values lie between 6955 and 9500 and the p-values close to zero indicate that the model is statistically significant and explain the variance of the energy consumption.

Given that the energy consumption is represented in logarithmic form, the coefficients can be interpreted as percentage changes. The coefficients of the number of solution tokens is between 0.0006-0.0008 which are all positive with p-values < 0.001. This suggests that an increase in the number of tokens increases the energy consumption of a coding task. Secondly, the log_energy_lag1 ranges from 0.09 to 0.14 which are all significant, indicating that past energy consumption has a moderately positive effect on energy consumption. Due to the autocorrelation of the energy consumption between tasks this term is added to control for it. Furthermore, the categories Computation, Cryptography, Network, System, Time and Visualization are added as dummy variables to the model. The category General is left out because every task has belongs to that category and it therefore does not add any explanatory power to the model. The coefficient for the Computation domain is positive (ranging from 0.02 to 0.05) and is highly significant $p - value < 0.05$. This means that task that use libraries within the Computation domain are associated with higher energy consumption. The 95% confidence intervals confirm the stability of these estimates since the intervals are narrow which substantiates that the usage of libraries in the Computation domain lead to higher energy consumption. The Cryptography domain coefficients are positive and significant (around 0.025 to 0.03) which indicates that cryptography tasks are energy-intensive. This is also in line with the computationally demanding nature of hashing or encryption tasks. The Network domain shows mostly insignificant coefficients across the models. This suggests Network domain libraries do not have a large impact on energy consumption. The System domain has coefficients (around 0.04 to 0.12), suggesting that system-level tasks has substantial impact on energy consumption. The coefficients of the Time domain are positive but smaller in magnitude (approximately 0.01) and significant in most models. This implies that coding tasks that utilize libraries within the Time domain have higher energy consumption. Finally, the Visualization domain shows consistently positive and highly significant coefficients (around 0.03 to 0.05), indicating that tasks involving visualization are energy-intensive. The increased energy consumption for the Visualization libraries can be explained because rendering graphics or displaying data often involves significant computational and memory resources.

The relationship between energy consumption and the use of specific libraries is consistent across all three models which suggest that coefficients of the $lagged\_log\_energy$ are robust. It can be concluded that energy consumption in the BigCodeBench is associated with higher task duration and the use of Computational, Cryptographic, and Visualization libraries significantly impact energy consumption. Given the lagged nature of the log energy variable it also indicates that energy consumption of previous tasks is a factor in energy consumption. These models effectively show how energy consumption relates to the libraries required in code generation tasks of the BigCodeBench. By understanding these relationships, organisations can better manage energy demands and optimise computational resources based on the required libraries in code.

4.2.2 Batch Size and Energy Consumption With Coding LLMs

Figure 4.10: *Models used for granularity and batch size analysis*



Based on the model characteristics given in 3.1, the number of parameters can be calculated for the models used during the experiments. First of all, the model characteristics of the base models are extracted from the config.json files in the Huggingface repository of these models (see Appendix A).

Table 4.1: *LLM characteristics*

Model name	$n_{params}(n)$	$n_{layers}(l)$	$d_{model}(m)$	$n_{heads}(h)$	$d_{head}(d)$	n_{vocab} (v)	n_{ctx} (c)
Qwen2,5-Coder	7B	28	3,584	28	128	152,064	128,000
Meta-Llama-3.1	8B	32	4,096	32	128	128,256	128,000
DeepSeek-V2-Lite	16B	27	2,048	16	128	102,400	128,000

To evaluate the abilities of LLMs to generate functional correct code, the pass@k metric is used. As stated above, three base models are tested for their coding abilities: Qwen2.5-Coder-7B, Llama3.1-8B and DeepSeek-V2-Lite-16B. Below the values of the prefill, decode, TTFT and TPOT are given. Also the required VRAM requirements for running the experiment is shown. The VRAM memory requirement are also shown in the rightmost column of 5.14.

4.2.2.1 Statistical Analysis

Following the approach in the granularity analysis, several methods are used to determine statistical differences in energy consumption related to batch size. The main methods used are an Kruskal-Wallis statistical test to see whether there are statistical differences in the energy consumption data for a model and Dunn's test that serves as a post-hoc robustness check to see where in the data these differences lie.

First of all, the outcome variable should be continuous which is the case since energy consumption is measured on a continuous scale. Second, it is important that the data for which the statistical test is performed are independent from each other. This means that the outcome of one does not affect the outcome of the other. This assumption holds given that the tasks are being processed in an isolated environment and are processed serially. In addition, all conditions are reset after each run. Thirdly, the normality of the data is examined by using a Shapiro-Wilk test for each of the three batch size experiments 5.15. Fourthly, the variance of the residuals influences the tests that can be used. If the variance of residuals does not remain constant over time, it is called heteroscedasticity. Levene's test is used to test for heteroscedasticity.

In the granularity analysis it was shown that the Shapiro-Wilk tests indicated non-normality for all three models and across all three batch sizes 5.3. Similarly, for Levene's test the p-values are well above 0.05 as stated in 5.4. This indicates homoscedasticity for each model. For the Qwen-Coder2.5-7B-Instruct model Levene's statistic is 0.1018 and the corresponding p-value is 0.9032 hence we fail to reject the nullhypothesis which means there is no significant difference in variance across the three batch sizes. which indicates that variances across the batch sizes are equal (homoscedasticity). For the Meta-LLaMA 3.1-8B-Instruct model Levene's statistic is 0.0588 and the p-value is 0.9428 which indicates no significant difference in variances across groups. The same holds for the DeepSeekCoder-V2-Lite model for which Levene's statistic is 0.0359 with p-value 0.9648, which shows that these model's variances are homoscedastic across groups as well.

Based on the fact that the outcome variable is continuous, the independence of observations, non-normality of the data and the homogeniety of variances. We can conclude that a non-parametric test is most suitable since parametric tests like ANOVA assume that the data is normally distributed. The Kruskal-Wallis test does not assume that the data is normally distributed which makes it more suitable for this experiment.

The Kruskal-Wallis test is a non parametric test to determine whether there are significant differences between the medians of multiple groups. In this case, the test evaluates if there are differences in energy consumption for different batch sizes. A higher Kruskal-Wallis test statistic indicates greater differences. The Qwen2.5-Coder-7B-Instruct the Kruskal-Wallis statistic is 21.2995 and the p-value $2.3707e - 5 < 0.05$ which indicates significant difference in energy consumption between groups. The Meta-LLaMA 3.1-8B-Instruct model has a Kruskal-Wallis statistic of 17.0849 and a p-value of $0.0002 < 0.05$ which also indicates significant differences across the batch sizes. The DeepSeekCoder model has a Kruskal-Wallis statistic of 2.5155 and a p-value of 0.2843, suggesting no significant differences in energy consumption across batch sizes.

The Kruskal-Wallis test does only test for differences between the batch sizes but it does not provide pairwise comparisons between batch sizes. A test that does provide such pairwise comparisons is the Dunn's Post-Hoc test. This test is subsequently applied to the energy consumption data. In Table 5.17, the pairwise comparisons between batch sizes are shown. The Qwen2.5-Coder-7B-Instruct model shows that there is no statistical significant difference between batch sizes 1 and 4 (p-value = 0.0801). However, the p-values for batch size 1 vs. batch size 8 and batch size 4 vs.batch size 8 are < 0.001 and 0.0494 respectively. This indicates tha there is a statistical significant difference in energy consumption between batch size 8 and the batch sizes 1 and 4. The Meta-LLaMA 3.1-8B-Instruct model has non significant p-values for batch 1 vs batch 4 and batch 1 vs batch 8. The p-value of batch 4 vs. batch 8 has a p-value of < 0.001 so this indicates a significant difference in energy consumption between batch sizes 4 and 8. The DeepSeekCoder model has a non-significant Kruskal-Wallis statistic, hence performing the Dunn-test for this model does not add value to the analysis.

The pairwise comparisons show that there is a significant difference between the energy consumption of the LLM ran with batch size 8 is significantly different from the batch size 1 and batch size 4 in Qwen2.5-Coder-Instruct. For Meta-LLaMA 3.1-8B-Instruct batch size 8 has a statistically significant higher energy consumption than batch size 4. The energy consumption of DeepSeekCoder-V2-Lite appears to be consistent across batch sizes.

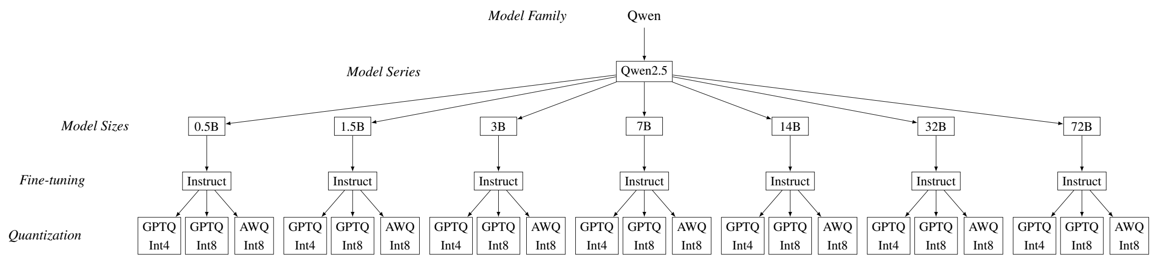
As can be seen in Table 5.18, the average energy consumption for the Qwen-coder2.5-7B-Instruct model is 0.7913 kWh, for the Meta-LLaMA 3.1-8B-Instruct model 0.9055 kWh and DeepSeek-Coder 0.3866 kWh for an entire run of the BigcodeBench. From these results it can be learnt that based on the energy consumption the energy consumption of the Meta-LLaMA 3.1-8B-Instruct model has 134.22% higher energy consumption compared to the DeepSeekCoder-V2-Lite-16B model.

The data in 5.18 provides insight in the trade-off between the energy consumption and the pass@k score of a model. It becomes clear that higher energy consumption does not result in a higher pass@k or in this case a pass@1 score. For example, the DeepSeekCoder-V2-Lite-16B model with the lowest energy consumption (0.3866 kWh for batch size 8), has a notably higher pass@1 score of 0.4947 which is notably higher than the pass@1 score of the Meta-LLaMA 3.1-8B-Instruct which is 0.2404 for batch size 8. When looking at the latency and throughput of the models it can be seen that lower latency and higher throughput indicate efficient processing. Energy efficiency in relation to accuracy can be represented as the pass@k per kWh consumed. Here also the DeepSeekCoder-V2-Lite-16B model stands out for having a high accuracy (pass@k) and low energy consumption. This model provides better accuracy per kWh than the other models. Seen the relatively small samples size per model, more batch sizes should be taken into account to reason effectively about the relationship between batch size and the pass@k rate.

4.2.3 Energy Impact of Fine-tuning, Quantization, and Model Size in Qwen2.5 Models

The effect of methods like fine-tuning, quantization and increase in model size are assessed for the Qwen2.5 model family. This model was chosen since it has a wide-range of models available ranging from 0.5B to 70B variants and Instruct and quantized models. To minimise the energy consumption of the experiments and due to computational limitations, only the Qwen 2.5 models were assessed in this work.

Figure 4.11: *Qwen2.5 model family*



4.2.3.1 Impact of Increasing Model Size on Energy Consumption

In this experiment the focus lies on the relationship between model size and energy consumption. As can be seen in 4.11, multiple model sizes are available within the Qwen2.5 Series to generate code which were used to gain insight in this relationship. In Table 5.19 estimated VRAM requirements are shown for running one task from BigCodeBench, Table 5.20 provides a general overview of distributions of the obtained energy consumption data.Tables 5.21 and 5.22 test for heteroscedasticity and differences in energy consumption between models, respectively. Together these Tables offer a insight in the energy efficiency trade-offs, memory requirements and the input processing capacities of the Qwen2.5 Base models.

The expected computational requirements shown in 5.19 reveal that the energy consumption of LLMs and VRAM requirements increases with size. The smallest model, Qwen2.5-0.5B, requires only 1GB of VRAM and as such can be ran on a consumer grade computer and has only an expected total generation time (TGT) of 0.77 seconds per BigCodeBench task. In comparison, the largest model, the 72B model in the requires 144 GB of VRAM and has a total generation time of 110.43 seconds per task. This would means that running such a model on a single state-of-the-art LLM like an NVIDIA H200 GPU which has a VRAM of 141GB is not possible, meaning that a cluster of industry-grade hardware is required to run LLMs of this size. The VRAM requirements increase linearly with model size but latency increases non-linearly and throughput decreases non-linearly with model size. These estimates suggest that larger models incur higher operational costs, as is also substantiated by the empirical results.

The Shapiro-Wilk test results for the GPU energy consumption of the models shows significant non-normality in the data, suggesting variability in energy consumption across the different model sizes. The results of Levene's test show significant differences in energy variance (heteroscedasticity). The Kruskal-Wallis test in Table 5.22 shows that there are statistically significant differences in energy consumption between model sizes. The pairwise comparisons from Dunn's test showcase the significant differences per pair of models which shows that the energy consumption for each tested model differs significantly from each other.

In conclusion, the energy consumption of Qwen2.5 models varies widely across different sizes, whereby larger models consume significantly more energy, have higher latency and reduces throughput. Otherwise, the insights in Table 5.24 show improved accuracy which is reflected in a higher pass@k rate. However, they do so at diminishing energy efficiency and processing speed. Therefore, organisations deploying LLMs must balance the trade-off between model size, energy consumption and performance requirements associated with these models.

4.2.3.2 Energy Impact of Fine-tuning

Fine-tuning Qwen2.5 models shows a significant effect on energy consumption across different models, with some models showing larger differences in energy consumption after fine-tuning than others. Additionally, fine-tuned models show slight shifts in latency, throughput and pass@k scores. The instruct LLMs are better in instruction following instructions by users to execute specific tasks. These models are trained on instruction-response pair data so that they have a sense of how a response on an instruction should be formatted. In this experiment it is investigated how fine-tuning of models affect the energy consumption of the model. Consequently, the differences in accuracy and energy consumption are investigated. In this way, insight is provided in the trade-off between energy consumption and accuracy.

Table 5.29 shows that energy consumption remains largely constant between base and fine-tuned models for smaller models (0.5B and 1.5B). However, for larger models there is a notable difference in energy consumption. For instance, the energy consumption of the 14B models drops from 1.68 kWh to 1.35 kWh after fine-tuning. At the same time, latency decreases, which is expected since energy consumption and latency are highly correlated as shown in the first experiment. Also, throughput shows improvements which indicates that fine-tuned models do not only consume less energy but also have higher token processing speed. This contrasts the smaller models where fine-tuning's effects on latency are less pronounced and throughput even decreases.

To assess the required hardware and predict the energy consumption of an LLM, the VRAM requirements for base models and fine-tuned models have been summarized in Table 5.25. Table 5.26 shows that the energy consumption data for the fine-tuned models is not normally distributed. Furthermore, the results of Levene's test in Table 5.27 show significant differences in energy consumption variance, particularly for the 14B model ($p < 0.001$) which indicates that model size is related to the energy variability after fine-tuning. The results suggest that while fine-tuning can reduce energy consumption for larger models, the degree of reduction can depend on model size.

The Kruskal-Wallis statistical test substantiates the claim that fine-tuning impacts energy consumption because all p-values for differences in energy consumption are statistically significant ($p < 0.001$) as shown in 5.28. Furthermore, the pass@k rate of a model is related to the nature of the model, For instance, the 7B-Instruct model consumes slightly less energy (0.82 kWh) than its base counterpart (0.87 kWh). At the same time the 7B-Instruct model exhibits a higher pass@k rate (0.45) than the 7B base model (0.31).

To summarise, fine-tuning has a significant effect on energy consumption, latency, throughput and the pass@k rate of Qwen2.5 models. While higher models show increased pass@k rates for fine-tuned models, smaller models do not show lower pass@k rates for fine-tuned models. This could be caused by overfitting the model before applying it to a diverse benchmark such as Bigcodebench. All in all, fine-tuning seems to be an effective approach for optimising Qwen2.5 models, particularly for larger architectures fine-tuning gains in both accuracy and energy consumption.

4.2.3.3 AWQ

AWQ of Qwen2.5 models provides significant improvements in energy efficiency, processing speed and memory usage across various model sizes. Besides energy consumption, latency, throughput and the pass@k score for each model are also evaluated. As a whole these metrics provide insight in the suitability of AWQ models for code generation.

In Table 5.30 the estimated VRAM requirements of the AWQ models are depicted which shows that the required VRAM requirements drop by 50% using a quantized model. This indicates that also the energy consumption of such a model drops by 50%. This occurs because of the reduced precision in the parameter weights.

The Shapiro-Wilk test results in 5.31 show the non-normality in all AWQ models ($p < 0.001$). Similarly, Levene's test shows the heterogeneity of variances between the Instruct and the Instruct-AWQ models which highlights the change in energy demand by AWQ. Larger models such as the 3B and 7B models show higher differences in variance than smaller models, suggesting that AWQ has the potential to increase resource efficiency for models with higher baseline resource requirements.

After applying the Kruskal-Wallis test it becomes clear that AWQ introduces statistically significant differences in energy consumption ($p < 0.001$) across all tested models. For instance, the 14B AWQ model has an energy consumption of 0.66 kWh compared to 1.35 kWh for the instruct model which is in line with the estimated energy consumption savings. It must be noted that it does not always decrease the precision of each number of weights equally which can explain the differences in energy savings across models. For example, there are almost no savings in the 0.5B model but in the 1.5B and 14B the svaings are around 50% while in the 3B and 7B models the savings are around 65%. Accordingly, the latency of the models decreases and except for the 0.5B model, throughput increases noticeably. This suggests that AWQ provides a scalable method for reducing the energy consumption of LLMs which is particulalry beneficial for high-performance applications such as chatbots and code generation tools.

Finally, Table 5.34 illustrates how AWQ impacts the energy consumption, latency, throughput and pass@k score of the Qwen2.5 models. A notable difference is the throughput of the 3B-Instruct model sees an increase from 76.24 tokens per second to 139 tokens per second when AWQ is applied. However, while AWQ usually reduces latency and energy consumption, there are sometimes trade-offs in model accuracy. Take for example the pass@k scores for the AWQ models for the 0.5B-Instruct model, which drops from 0.07 to 0.01 while keeping energy consumption approximately equal and increases latency. This is also visible for the 14B-Instruct model which energy consumption is reduced by 51% but this also results in a pass@k drop from 51% to 0.27% which suggests that AWQ conserves energy but it also impacts the accuracy in exchange for energy efficiency gains.

4.2.3.4 GPTQ

The final set of experiments with Instruct models and Instruct-GPTQ models shows significant improvements in computational efficiency, energy consumption, latency and throughput for both GPTQ-Int8 and GPTQ-Int4 models. These effects are especially visible when comparing the quantized models with the VRAM requirements for base or instruct models as can be seen in Table 5.35. Reducing the precision of the models result in significant savings in prefill and decode compute as well as prefill memory and VRAM. This emphasizes the potential of GPTQs role in LLM deployment in resource-constraint environments.

The results in 5.40 demonstrate GPTQs effect on energy consumption across all models, the INT4 reductions show the largest reductions in energy consumption which was also estimated in 5.35. While in the 0.5B model this drop in energy consumption is quite limited, this becomes more evident in larger models. In the 1.5B model the energy drop from Instruct to Instruct-GPTQ-Int8 is 33.33% it is 48% for the Instruct-GPTQ-Int4 model. These percentages further increase when model size increases as the Instruct-GPTQ-INT8 model saves 41% and the 14B-GPTQ-INT4 model saves 66% percent compared to the 14B-Instruct model. These findings stresses the usability of GPTQ models for applications that prioritise energy efficiency.

Regarding the statistical results of the GPTQ models the Shapiro-Wilk test 5.36 shows that the energy consumption data of GPTQ models is not normally distributed with p-values < 0.001. A shift to lower bit precision in GPTQ models increases the non-normality in the data. Furthermore, Levene's test statistics in Table 5.37 show statistical differences in energy consumption variance for the quantized models. This indicates that GPTQ's impact on resource utilization is larger for larger models, making it an effective method in resource-constrained environments.

The Kruskal-Wallis test results substantiates this by testing for statistical differences between Instruct models and Instruct-GPTQ models. In Table 5.38 the differences in energy consumption for all model sizes are statistically significant ($p - value < 0.001$). This suggests that there are broad efficiency benefits of GPTQ across different scales and model complexities. For example, GPTQ from Instruct to Int8 in the 7B model results in latency reduction from 191.68 ms to 109.55 ms and increased throughput from 37 tokens per second to 63.99 tokens per second. Int4 GPTQ further reduces the latency to 72.62 ms with a throughput to 97.13 tokens per second. These results show that GPTQ does not only improve energy efficiency but also speed and resource requirements, making these models more responsive and scalable.

While GPTQ models enhance efficiency there are also trade-offs in model performance, particularly in pass@k scores. The 0.5B models shows a drop from 7% to 4% for the GPTQ-Int4 model. Larger models show similar trends but the reduction in pass@k rates are limited as can be seen in Table 5.40, for the 7B model the pass@k rate drops from 45% to 44% and for the 14B model the pass@k rate drop from 51% to 49% while the GPTQ-INT8 models even show slight improvements in pass@k scores. However, the trends off improved energy efficiency at the cost of lower pass@k scores is also visible for GPTQ models.

To conclude, GPTQ proves effective for lowering energy consumption and latency for the Qwen2.5 models. The advantages are especially visible in larger models which is advantageous since the energy savings, speed improvements and memory optimisations can in absolute terms save more resources. Although there might be trade-offs between accuracy and energy consumption for GPTQ models, the effects on the pass@k rate remain minimal while the energy savings are more pronounced. This makes GPTQ methods an interesting choice when models have to maintain high performance but energy consumption or scalability is a major concern.

4.2.4 The Energy-Accuracy Trade-Off

The trade-off between energy consumption and accuracy can be visually depicted by plotting the energy consumption against the pass@k rate of an experiment. In 4.12, this trade-off is shown for the 25 experiments for the Qwen2.5 models, which shows that larger models have higher pass@kwh rates and the kWh necessary for obtaining a higher pass@k rate increases exponentially. Nonetheless, the efficient frontier is formed by the models who minimise the pass@kwh rate for a given combination of energy and pass@k or in this case the pass@1.

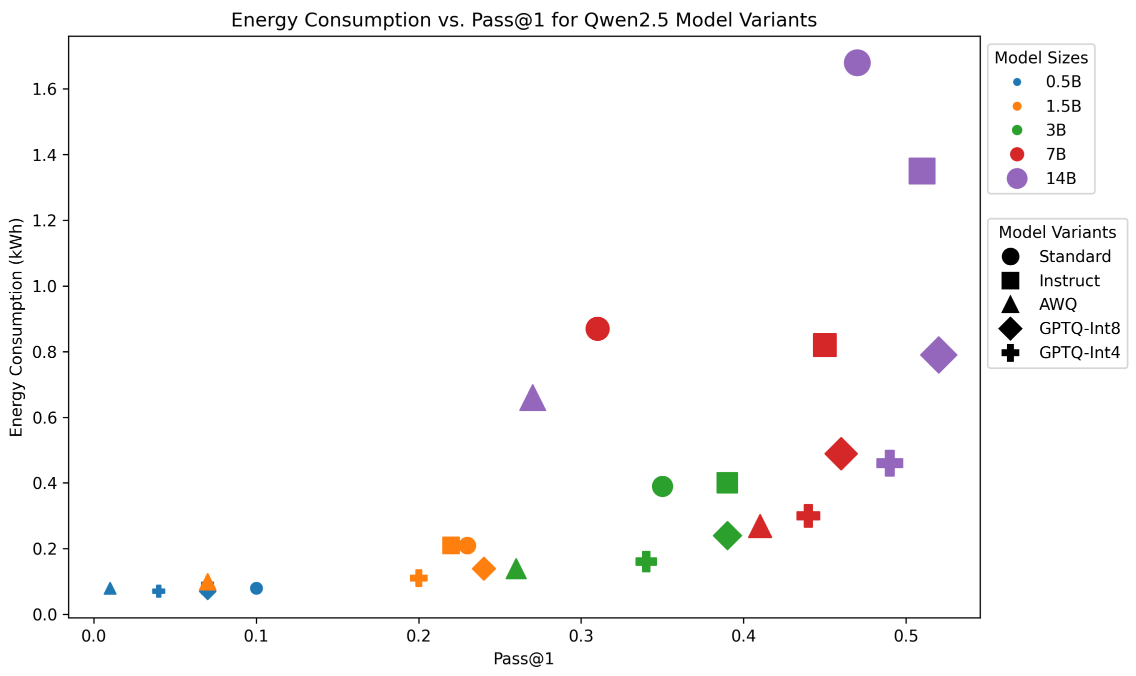


Figure 4.12: *Energy consumption for one Bigcodebench run vs Pass@1 rate for Qwen2.5 models*

The models that minimize the energy consumption for a given pass@k rate lie on the efficient frontier and are Pareto optimal in this context. As can be seen in 4.12, the GPTQ-INT4 models and the GPTQ-INT8 models regularly lie on this efficient frontier. This indicates that these models provide the optimal trade-off between energy consumption, accuracy and functionality preservation. Given the exponential nature of the efficient frontier this might indicate that exponentially more energy is required for LLMs to solve more challenging coding tasks. More research is needed to determine whether this holds for larger models and models with MoE architectures.

4.3 Discussion

This work focused on optimizing the energy efficiency of LLMs during code generation tasks in BigCodeBench. Since LLM training and inference is known for its significant contribution to carbon dioxide emissions and climate change [117], it was a logical starting point for research into the energy efficiency of AI systems. Although surprising, energy consumption is often overlooked in the performance evaluation of AI models, where speed and accuracy where most important [5]. Additionally, accurately measuring the energy consumption of LLMs during inference can be a challenge due to the variability in hardware environments and access levels to these environments.

The main objective of this work was to examine to what extent LLMs can be optimised for energy efficiency without degrading their accuracy in code generation tasks. To achieve this, we proposed a software-based energy measurement approach. During several experiments we assessed the impact of LLM optimization techniques such as changing batch size, fine-tuning, AWQ and

GPTQ on both energy consumption and pass@k rate (accuracy). The trade-off between energy consumption and accuracy was further investigated to determine optimal combinations between the two. Furthermore, we evaluated methods to ensure LLM generated code maintains its functionality and correctness.

A software-based energy measurement method that uses energy monitoring tools during LLM inference was used in this work. This method accurately measured the energy consumption of LLMs, which confirms that energy consumption can effectively be quantified with readily available open-source software. Additionally, the application of optimization techniques like fine-tuning, AWQ and GPTQ led to significant reductions in the energy consumption of LLMs, for some methods even without significant performance loss. Also, we distinguished the effectiveness of the available optimization methods for LLMs regarding energy consumption and model accuracy. This trade-off was simplified by the pass@kWh metric, which immediately shows the relationship between energy efficiency and the pass@k score. The evaluation framework of BigCodeBench [89] that includes unit testing, ensured that the correctness and functionality of LLM generated code was preserved.

These results show that LLMs can be optimised for energy efficiency with minimal accuracy and functionality loss. The research contributes to the field of AI by providing a method to effectively quantify the energy consumption of LLMs. Furthermore, a gap in the literature is addressed by providing clear energy consumption data of LLM code generation tasks which shows the currently limited ability of open-source models to solve complex programming tasks. The information on the energy consumption difference between models and model architectures shows that there are significant differences between models in their ability to solve coding tasks but also the energy that they require to do so. Besides, we provide insight in the trade-off between energy consumption and model accuracy, which is further substantiated with the introduction of the pass@kWh metric.

The implications of this study are substantial for the AI industry. It became evident that the energy consumption well-known models can be 130.77% higher during code generation tasks while achieving a lower accuracy on BigCodeBench. Optimizing LLMs for energy efficiency does not only have environmental advantages but also offers financial advantages in the form of lower energy related costs. The proposed energy measurement methodology and unit testing strategy can be used for a wide range of LLMs, which helps the broader adoption of energy-efficient practices in AI development.

Batching was shown to be crucial for managing energy efficiency across coding tasks. As demonstrated in Table 5.18, models run with higher batch sizes tend to consume slightly more energy than models with lower batch sizes. However, the differences are minimal hence future work can research a larger number of batch sizes and a greater variety of models.

The experiments showed that the energy consumption of LLMs is closely related to the model size and the computational complexity required to generate code. Larger models like Qwen2.5-

14B-Instruct demand more energy but generally achieve higher pass@k scores, as substantiated by Table 5.24. For instance, unquantized models such as Qwen2.5-14B consume approximately 1.35 kWh per BigCodeBench run, while smaller models like Qwen2.5-0.5B-Instruct consume 0.08 kWh for comparable tasks. However, this comes at the cost of model accuracy which is reflected in lower pass@k rates for smaller models. While smaller models may be suitable for basic coding tasks with limited complexity, larger models appear to be necessary to solve more complex tasks. A more complex task is here more computationally complex and might need a larger number of parameters to 'grasp' the relations necessary to solve a task. This finding aligns with previous research indicating that model size correlates with performance in complex tasks [10].

Fine-tuning models for specific tasks such as code generation has shown to improve performance without or with limited effects for the functional correctness of generated code. For example, Qwen2.5-14B-Instruct shows decreased energy consumption after fine-tuning. An explanation for this result could be that fine-tuned models require fewer computational resources during inference, due to a decreased number of parameter adjustments.

Quantization techniques like AWQ and GPTQ have shown to significantly decrease the energy consumption of LLMs. Quantization methods reduce the precision of weights, parameters or both and thereby reduce the required VRAM and energy demand to run a model. For example, the energy consumption of the Qwen2.5-14B model decreased by 66% from its Instruct variant to the GPTQ-INT4 variant, and the required VRAM decreased from 28 GB to 7 GB, which is shown in Table 5.35. While quantization may cause decreased accuracy, it democratizes access to these models by decreasing the hardware requirements which makes it available to a larger public.

This work shows the importance of both energy efficiency and model performance in the deployment of LLMs for code generation tasks. The trade-offs between energy consumption, model accuracy, and computational resources indicates that model choice is highly dependent on context and model requirements. In sectors like healthcare or finance, where accuracy is top priority, accuracy loss because of quantization may be unacceptable regardless the energy savings. On the other hand, quantization offers a scalable approach to reduce energy consumption and VRAM requirements in less strict environments such as day-to-day use of chatbots.

4.4 Limitations

Despite the energy consumption insights uncovered by this study, there are also limitations. For instance, the energy consumption measurement was only conducted on NVIDIA L40S GPUs. Since the energy consumption for hardware differs from machine to machine, the findings may not generalize to all computing environments.

Also, the experiments were conducted on a selection of open-source LLMs, which for the granularity and batch experiments were models from different families while for the model size, fine-tuning and quantization experiments this were models from the Qwen2.5 model family. While the Qwen2.5 has a wide range of models available, these results might not generalise to other model architectures. In the batch size analysis there were no significant differences in energy consumption. Since DeepSeekCoder-V2-Lite-Instruct-16B is a MoE model this might indicate that batch size does not impact energy consumption in MoE models. Besides, it might indicate that the results from the second part of the experiments regarding the effects of fine-tuning and quantization also cannot directly be generalized to MoE models.

Due to computational limitations, the experiments used the pass@1 metric. Higher pass@k rates, such as pass@5 and pass@10 were not explored, which might provide more insight in the trade-off between energy consumption and accuracy.

The energy consumption of the GPU has been collected every 10 seconds, which might have influenced the accuracy of the measurements for the granularity analysis. Models with short runtimes might therefore have less accurate estimates of energy consumption than models with longer runtimes.

The experiments focused on the energy consumption of different LLMs from the moment that they were trained. However, the cumulative energy consumption from scratch were not taken into account. Incorporating this data might impact model choice since fine-tuning and quantization also costs energy.

4.5 Future Work

Based on the limitations of this study, future research could focus on evaluating a broader range of models by incorporating multiple architectures such as multiple encoder-decoder, decoder-only and MoE models, to increase the generalizability of the findings. Furthermore, it is encouraged to repeat these measurements across different hardware platforms such as different GPU models, CPUs and TPUs to understand how hardware impacts energy efficiency and model performance. Using higher pass@k scores offer another opportunity to gain insight in the energy-accuracy trade-off. Incorporating the energy costs of the training phase would give a more complete view on the sustainability of LLMs and provides insight on a larger part of a models life-cycle. By focusing on these aspects in future studies, the fundament laid in this study to gain insight in the capacities of LLMs to efficiently generate code can further be advanced.

Chapter 5

Conclusion

This work has explored the opportunities to decrease the energy consumption of LLMs while preserving the model accuracy. Recognizing the significant energy consumption of LLMs and their environmental impacts, this study aimed to close the gap in obtaining and reporting detailed energy consumption information for code generation tasks during inference. With the eventual goal of producing functionally correct code with minimal energy consumption.

Through experiments with Qwen2.5-Coder-7B-Instruct, Meta-LLaMA3.1-8B-Instruct, DeepSeekCoder-V2-Lite-16B, energy consumption was measured for individual coding tasks in BigCodeBench, which showed that categories like Visualization and Cryptography consumed significantly more energy than General or Network tasks. A significant difference in energy consumption was found across batch sizes for the Meta and Qwen2.5 models. A noticeable difference was that the Llama3.1-8B-Instruct model had an energy consumption of 130.77% higher than the DeepSeekCoder-V2-Lite-16B model on the BigCodeBench run while achieving a lower pass@1 score. Experiments on model size showed that the energy consumption increases nearly linear with model size. For the fine-tuning, AWQ, and GPTQ with 8-bit and 4-bit precision led to decreases in energy consumption of up to 19.65%, 51.11%, 41.48% and 63.41%, respectively. Larger models tend to show larger relative reductions in energy consumption compared to smaller models.

The introduction of the pass@kWh metric showed a novel way to gain insight in the performance of a model both on accuracy and pass@k rate, balancing both factors in model performance assessment.

This work shows the feasibility of software-based energy consumption monitoring during LLM inference on tasks such as code generation. Energy consumption can be effectively measured without additional hardware which increases the opportunities to map the energy efficiency and environmental impact by a larger group of researchers and developers. Furthermore, the effectiveness of optimisation tasks such as fine-tuning, AWQ and GPTQ has been shown to achieve energy savings, in some cases without significant decreases in model performance. Fine-tuned models not only increase the accuracy of models but also tend to lower the energy consumption of a model. In addition, optimal combinations of energy consumption and accuracy were defined which guides model choice dependent on accuracy and energy consumption requirements. The pass@kWh metric captures the trade-off between these two information sources. This provides a more nuanced view on the concepts sustainability and model performance. Finally, this work showed that in the context of efficient computing, unit testing ensured the correctness and functionality preservation of LLM-generated code.

Despite, there are also limitations regarding this approach. The energy measurements were performed on one GPU type and due to hardware configurations, the results can be different for other computing environments. Besides, the scope of the LLMs used was limited due to computational constraints, so generalizing the effects of the optimisation techniques to other model architectures requires additional research. Evaluating additional models, hardware, and BigCodeBench configurations is a future area of research. As well as the development of adaptive systems that balance energy consumption with accuracy.

In conclusion, this work shows that optimising LLMs is both feasible and beneficial with regard to energy efficiency and model performance. By integrating the optimisation techniques and evaluation measures, it is possible to generate correct and functional code with significantly less energy. The methodology and results contribute to the advancement of sustainable AI knowledge and practices. With the rapid adoption of AI in many industries, the focus on energy efficiency is required to minimise environmental impact and ensure responsible technological progress.

Appendix

Model: Qwen2.5-7B

The following describes the architecture and hyperparameters for the Qwen2.5-7B model.

Listing 5.1: *Model Configuration Qwen2.5-7B*

```
{
  "architectures": [
    "Qwen2ForCausalLM"
  ],
  "attention_dropout": 0.0,
  "bos_token_id": 151643,
  "eos_token_id": 151643,
  "hidden_act": "silu",
  "hidden_size": 3584,
  "initializer_range": 0.02,
  "intermediate_size": 18944,
  "max_position_embeddings": 131072,
  "max_window_layers": 28,
  "model_type": "qwen2",
  "num_attention_heads": 28,
  "num_hidden_layers": 28,
  "num_key_value_heads": 4,
  "rms_norm_eps": 1e-06,
  "rope_theta": 1000000.0,
  "sliding_window": 131072,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.40.1",
  "use_cache": true,
  "use_mrope": false,
  "use_sliding_window": false,
  "vocab_size": 152064
}
```

Model: Llama-3.1-8B

The following describes the architecture and hyperparameters for the Llama-3.1-8B model.

Listing 5.2: *LLaMA3.1-8B-Instruct Model configuration*

```
{
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 128000,
  "eos_token_id": 128001,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 14336,
  "max_position_embeddings": 131072,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 8,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_scaling": {
    "factor": 8.0,
    "low_freq_factor": 1.0,
    "high_freq_factor": 4.0,
    "original_max_position_embeddings": 8192,
    "rope_type": "llama3"
  },
  "rope_theta": 500000.0,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.43.0.dev0",
  "use_cache": true,
  "vocab_size": 128256
}
```

Model: DeepseekV2-Lite-16B

The following describes the architecture and hyperparameters for the DeepseekV2-Lite-16B model.

Listing 5.3: *DeepSeekCoder-V2-Lite-16B Model Configuration JSON*

```
{
  "architectures": [
    "DeepseekV2ForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "auto_map": {
    "AutoConfig": "configuration_deepseek.DeepseekV2Config",
    "AutoModel": "modeling_deepseek.DeepseekV2Model",
    "AutoModelForCausalLM": "modeling_deepseek.DeepseekV2ForCausalLM"
  },
  "aux_loss_alpha": 0.001,
  "bos_token_id": 100000,
  "eos_token_id": 100001,
  "first_k_dense_replace": 1,
  "hidden_act": "silu",
  "hidden_size": 2048,
  "initializer_range": 0.02,
  "intermediate_size": 10944,
  "kv_lora_rank": 512,
  "max_position_embeddings": 163840,
  "model_type": "deepseek_v2",
  "moe_intermediate_size": 1408,
  "moe_layer_freq": 1,
  "n_group": 1,
  "n_routed_experts": 64,
  "n_shared_experts": 2,
  "norm_topk_prob": false,
  "num_attention_heads": 16,
  "num_experts_per_tok": 6,
  "num_hidden_layers": 27,
  "num_key_value_heads": 16,
  "pretraining_tp": 1,
  "q_lora_rank": null,
  "qk_nope_head_dim": 128,
  "qk_rope_head_dim": 64,
  "rms_norm_eps": 1e-06,
  "rope_scaling": {
    "beta_fast": 32,
    "beta_slow": 1,
    "factor": 40,
    "mscale": 0.707,
    "mscale_all_dim": 0.707,
    "original_max_position_embeddings": 4096,
    "type": "yarn"
  },
  "rope_theta": 10000,
  "routed_scaling_factor": 1.0,
  "scoring_func": "softmax",
  "seq_aux": true,
  "tie_word_embeddings": false,
  "topk_group": 1,
  "topk_method": "greedy",
  "torch_dtype": "bfloat16",
  "transformers_version": "4.33.1",
  "use_cache": true,
  "v_head_dim": 128,
  "vocab_size": 102400
}
```

High granularity analysis

Table 5.1: *Occurrence percentages of categories in BigCodeBench based on library presence in the solutions of coding tasks*

Domain	Usage (%)	Library	Function Call
Computation	64.8%	pandas, numpy, sklearn, scipy, math, nltk, statistics, cv2, statsmodels, tensorflow, sympy, textblob, skimage	pandas.DataFrame, numpy.random, numpy.random.seed, numpy.array, numpy.mean, pandas.read_csv, numpy.random.randint, pandas.Series
General	100%	random, re, collections, itertools, string, operator, heapq, ast, functools, regex, bisect, inspect, unicodedata	collections.Counter, random.seed, random.randint, random.choice, re.sub, re.findall, itertools.chain
Visualization	34.8%	matplotlib, seaborn, PIL, folium, wordcloud, turtle, mpl_toolkits	matplotlib.pyplot, matplotlib.pyplot.subplots, matplotlib.pyplot.figure
System	95.5%	os, json, csv, shutil, glob, subprocess, pathlib, sqlite3, io, zipfile, sys, logging, pickle, struct, psutil	os.path, os.path.join, os.path.exists, os.makedirs, glob.glob, os.listdir, json.load, csv.writer, shutil.move
Time	16.1%	datetime, time, pytz, dateutil, holidays, calendar	datetime.datetime, datetime.datetime.now, time.time, time.sleep, datetime.datetime.strptime
Network	8.2%	requests, urllib, bs4, socket, django, flask, ipaddress, smtplib, http, flask_mail, cgi, ssl, email, mechanize	urllib.parse.urlparse, django.http.HttpResponse, ipaddress.IPv4Network, smtplib.SMTP, requests.post, socket.gaierror
Cryptography	5.1%	hashlib, base64, binascii, codecs, rsa, cryptography, hmac, blake3, secrets, Crypto	cryptography.fernet.Fernet.generate_key, cryptography.hazmat.primitives.padding, cryptography.hazmat.primitives.padding.PKCS7

Table 5.2: *Durbin-Watson Test for Autocorrelation*

Model Name and Batch Size	**Durbin-Watson Statistic**
Qwen-Coder2.5-7B-Instruct	
Batch size 1	0.73
Batch size 4	0.77
Batch size 8	0.80
Meta Llama3.1-8B-Instruct	
Batch size 1	0.56
Batch size 4	0.94
Batch size 8	0.59
DeepSeekCoder-V2-Lite-16B	
Batch size 1	0.27
Batch size 4	0.21
Batch size 8	0.23

Table 5.3: *Shapiro-Wilk Normality Test Results*

Model	**Shapiro-Wilk Statistic**	**p-value**
Qwen-Coder2.5-7B-Instruct		
Batch size 1	0.28	< 0.001
Batch size 4	0.29	< 0.001
Batch size 8	0.27	< 0.001
Meta Llama3.1-8B-Instruct		
Batch size 1	0.29	< 0.001
Batch size 4	0.21	< 0.001
Batch size 8	0.31	< 0.001
DeepSeekCoder-V2-Lite-16B		
Batch size 1	0.93	< 0.001
Batch size 4	0.95	< 0.001
Batch size 8	0.83	< 0.001

Table 5.4: *Levene's Test for Homogeneity of Variance (Homoscedasticity)*

Model	**Levene's Statistic**	**p-value**
Qwen-Coder2.5-7B-Instruct		
Batch size 1	0.00	0.95
Batch size 4	1.77	0.18
Batch size 8	1.09	0.30
Meta Llama3.1-8B-Instruct		
Batch size 1	2.44	0.12
Batch size 4	0.11	0.75
Batch size 8	0.80	0.37
DeepSeekCoder-V2-Lite-16B		
Batch size 1	125.51	< 0.001
Batch size 4	179.09	< 0.001
Batch size 8	25.62	< 0.001

Table 5.5: *Qwen-Coder OLS regression results (batch size 1)*

Variable	**Coef.**	**Std. Err.**	**z**	**p$>|z|$**	**[0.025**	**0.975]**
solution tokens	0.0009	2.71e-05	33.030	0.000	0.001	0.001
log_energy_lag1	0.1336	0.013	9.931	0.000	0.107	0.160
Computation	0.0443	0.005	8.180	0.000	0.034	0.055
Cryptography	0.0545	0.008	6.557	0.000	0.038	0.071
Network	-0.0029	0.009	-0.320	0.749	-0.020	0.015
System	0.0773	0.008	9.179	0.000	0.061	0.094
Time	0.0199	0.006	3.204	0.001	0.008	0.032
Visualization	0.0303	0.005	6.375	0.000	0.021	0.040
Model Statistics						
R-squared (uncentered)	0.982				**F-statistic**	9037
Adj. R-squared (uncentered)	0.982				**Prob (F-statistic)**	0.00
Log-Likelihood	1404.1				**AIC**	-2792
BIC	-2752				**Durbin-Watson**	2.071
Omnibus	61.352				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	182.627				**Skew**	0.202
Kurtosis	4.919				**Cond. No.**	2460

Table 5.6: *Qwen-Coder OLS regression results (batch size 4)*

Variable	**Coef.**	**Std. Err.**	**z**	**p> $|z|$**	**[0.025**	**0.975]**
solution tokens	0.0009	2.7e-05	33.705	0.000	0.001	0.001
log_energy_lag1	0.1208	0.013	9.077	0.000	0.095	0.147
Computation	0.0452	0.005	8.216	0.000	0.034	0.056
Cryptography	0.0517	0.008	6.590	0.000	0.036	0.067
Network	-0.0022	0.009	-0.238	0.812	-0.020	0.016
System	0.0767	0.009	8.715	0.000	0.059	0.094
Time	0.0231	0.006	3.719	0.000	0.011	0.035
Visualization	0.0309	0.005	6.276	0.000	0.021	0.041
Model Statistics						
R-squared (uncentered)	0.982				**F-statistic**	8736
Adj. R-squared (uncentered)	0.982				**Prob (F-statistic)**	0.00
Log-Likelihood	1383.0				**AIC**	-2750
BIC	-2710				**Durbin-Watson**	2.093
Omnibus	52.664				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	153.116				**Skew**	0.138
Kurtosis	4.774				**Cond. No.**	2450

Table 5.7: *Qwen-Coder OLS regression results (batch size 8)*

Variable	**Coef.**	**Std. Err.**	**z**	**p$>|z|$**	**[0.025**	**0.975]**
solution tokens	0.0009	2.78e-05	32.293	0.000	0.001	0.001
log_energy_lag1	0.1285	0.013	9.713	0.000	0.103	0.154
Computation	0.0453	0.005	8.240	0.000	0.035	0.056
Cryptography	0.0554	0.008	6.785	0.000	0.039	0.071
Network	-0.0031	0.009	-0.346	0.730	-0.021	0.015
System	0.0774	0.008	9.262	0.000	0.061	0.094
Time	0.0203	0.006	3.266	0.001	0.008	0.032
Visualization	0.0322	0.005	6.759	0.000	0.023	0.042

Model Statistics			
R-squared (uncentered)	0.982	**F-statistic**	9099
Adj. R-squared (uncentered)	0.982	**Prob (F-statistic)**	0.00
Log-Likelihood	1395.7	**AIC**	-2775
BIC	-2735	**Durbin-Watson**	2.074
Omnibus	63.502	**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	209.692	**Skew**	0.161
Kurtosis	5.076	**Cond. No.**	2460

Table 5.8: *Meta Llama3.1 OLS regression results (batch size 1)*

Variable	**Coef.**	**Std. Err.**	**z**	**p>** $\|z\|$	**[0.025**	**0.975]**
solution tokens	0.0008	2.56e-05	30.912	0.000	0.001	0.001
log_energy_lag1	0.1405	0.016	8.737	0.000	0.109	0.172
Computation	0.0510	0.006	8.789	0.000	0.040	0.062
Cryptography	0.0310	0.012	2.694	0.007	0.008	0.054
Network	-0.0140	0.010	-1.364	0.173	-0.034	0.006
System	0.1216	0.011	11.456	0.000	0.101	0.142
Time	0.0110	0.007	1.539	0.124	-0.003	0.025
Visualization	0.0514	0.005	9.629	0.000	0.041	0.062
Model Statistics						
R-squared (uncentered)	0.983				**F-statistic**	9578
Adj. R-squared (uncentered)	0.983				**Prob (F-statistic)**	0.00
Log-Likelihood	1306.9				**AIC**	-2598
BIC	-2557				**Durbin-Watson**	2.091
Omnibus	32.536				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	76.258				**Skew**	0.017
Kurtosis	4.267				**Cond. No.**	2770

Table 5.9: *Meta Llama3.1 OLS regression results (batch size 4)*

Variable	**Coef.**	**Std. Err.**	**z**	**p> $\|z\|$**	**[0.025**	**0.975]**
solution tokens	0.0008	2.53e-05	31.100	0.000	0.001	0.001
log_energy_lag1	0.1415	0.016	8.863	0.000	0.110	0.173
Computation	0.0508	0.006	8.968	0.000	0.040	0.062
Cryptography	0.0323	0.011	2.828	0.005	0.010	0.055
Network	-0.0132	0.010	-1.336	0.182	-0.033	0.006
System	0.1209	0.011	11.451	0.000	0.100	0.142
Time	0.0099	0.007	1.435	0.151	-0.004	0.023
Visualization	0.0514	0.005	9.729	0.000	0.041	0.062
Model Statistics						
R-squared (uncentered)	0.983				**F-statistic**	9507
Adj. R-squared (uncentered)	0.983				**Prob (F-statistic)**	0.00
Log-Likelihood	1320.7				**AIC**	-2625
BIC	-2585				**Durbin-Watson**	2.094
Omnibus	28.093				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	60.744				**Skew**	-0.026
Kurtosis	4.130				**Cond. No.**	2780

Table 5.10: *Meta Llama3.1 OLS regression results (batch size 8)*

Variable	**Coef.**	**Std. Err.**	**z**	**p$>|z|$**	**[0.025**	**0.975]**
solution tokens	0.0008	2.56e-05	31.043	0.000	0.001	0.001
log_energy_lag1	0.1412	0.016	8.943	0.000	0.110	0.172
Computation	0.0525	0.006	9.111	0.000	0.041	0.064
Cryptography	0.0298	0.011	2.643	0.008	0.008	0.052
Network	-0.0141	0.010	-1.409	0.159	-0.034	0.005
System	0.1203	0.011	11.398	0.000	0.100	0.141
Time	0.0118	0.007	1.710	0.087	-0.002	0.025
Visualization	0.0500	0.005	9.332	0.000	0.039	0.060

Model Statistics

R-squared (uncentered)	0.983	**F-statistic**	9500
Adj. R-squared (uncentered)	0.983	**Prob (F-statistic)**	0.00
Log-Likelihood	1311.8	**AIC**	-2608
BIC	-2567	**Durbin-Watson**	2.103
Omnibus	31.080	**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	71.083	**Skew**	-0.015
Kurtosis	4.223	**Cond. No.**	2760

Table 5.11: *DeepSeekCoder-V2-Lite OLS regression results (batch size 1)*

Variable	Coef.	Std. Err.	z	p> $\|z\|$	[0.025	0.975]
solution tokens	0.0006	1.77e-05	32.117	0.000	0.001	0.001
log_energy_lag1	0.0897	0.013	6.746	0.000	0.064	0.116
Computation	0.0191	0.003	6.086	0.000	0.013	0.025
Cryptography	0.0265	0.005	4.987	0.000	0.016	0.037
Network	-0.0036	0.005	-0.733	0.464	-0.013	0.006
System	0.0395	0.005	7.925	0.000	0.030	0.049
Time	0.0111	0.004	2.939	0.003	0.004	0.018
Visualization	0.0254	0.003	8.272	0.000	0.019	0.031
Model Statistics						
R-squared (uncentered)	0.980				**F-statistic**	6959
Adj. R-squared (uncentered)	0.980				**Prob (F-statistic)**	0.00
Log-Likelihood	1973.7				**AIC**	-3931
BIC	-3891				**Durbin-Watson**	2.184
Omnibus	46.894				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	119.746				**Skew**	-0.161
Kurtosis	4.555				**Cond. No.**	3720

Table 5.12: *DeepSeekCoder-V2-Lite OLS regression results (batch size 4)*

Variable	Coef.	Std. Err.	z	p> $\|z\|$	[0.025	0.975]
solution tokens	0.0006	1.77e-05	31.832	0.000	0.001	0.001
log_energy_lag1	0.0922	0.013	7.043	0.000	0.067	0.118
Computation	0.0189	0.003	6.193	0.000	0.013	0.025
Cryptography	0.0255	0.006	4.490	0.000	0.014	0.037
Network	-0.0034	0.005	-0.694	0.487	-0.013	0.006
System	0.0404	0.005	8.357	0.000	0.031	0.050
Time	0.0135	0.004	3.752	0.000	0.006	0.020
Visualization	0.0264	0.003	8.778	0.000	0.021	0.032
Model Statistics						
R-squared (uncentered)	0.980				**F-statistic**	7191
Adj. R-squared (uncentered)	0.980				**Prob (F-statistic)**	0.00
Log-Likelihood	1985.3				**AIC**	-3955

Table 5.13: *DeepSeekCoder-V2-Lite OLS regression results (batch size 8)*

Variable	**Coef.**	**Std. Err.**	**z**	**p>** $|z|$	**[0.025**	**0.975]**
solution tokens	0.0006	1.73e-05	32.364	0.000	0.001	0.001
log_energy_lag1	0.0928	0.013	7.092	0.000	0.067	0.118
Computation	0.0186	0.003	5.910	0.000	0.012	0.025
Cryptography	0.0269	0.005	4.957	0.000	0.016	0.037
Network	0.0006	0.005	0.112	0.911	-0.009	0.010
System	0.0408	0.005	8.294	0.000	0.031	0.050
Time	0.0138	0.004	3.790	0.000	0.007	0.021
Visualization	0.0281	0.003	8.918	0.000	0.022	0.034
Model Statistics						
R-squared (uncentered)	0.980				**F-statistic**	6955
Adj. R-squared (uncentered)	0.979				**Prob (F-statistic)**	0.00
Log-Likelihood	1968.7				**AIC**	-3921
BIC	-3881				**Durbin-Watson**	2.235
Omnibus	44.848				**Prob (Omnibus)**	0.000
Jarque-Bera (JB)	128.355				**Skew**	0.004
Kurtosis	4.644				**Cond. No.**	3780

5.1 Batch size

Table 5.14: *LLM Computational Requirements*

Model	**Prefill Compute**	**Decode Compute**	**Prefill & Decode Memory**	**TTFT (ms)**	**TPOT (ms)**	**TGT (s)**	**VRAM (GB)**
Qwen2.5-Coder-7B	3.58×10^{12}	14.00	28.00	36.00	16.24	4.17	16.80
Meta-Llama-3.1-8B	4.10×10^{12}	16.00	32.00	41.14	18.56	4.76	19.20
DeepSeek-V2-Lite-16B	8.09×10^{12}	31.60	63.20	81.26	36.66	9.40	37.92

Table 5.15: *Levene's Test for Homogeneity of Variance (Homoscedasticity)*

Model	**Levene's Statistic**	**p-value**
Qwen-Coder2.5-7B-Instruct	0.10	0.90
Meta Llama3.1-8B-Instruct	0.06	0.94
DeepSeekCoder-V2-Lite-16B	0.04	0.96

Table 5.16: *Kruskal-Wallis Test for Differences in Energy Consumption*

Model	**Kruskal-Wallis Test Statistic**	**p-value**
Qwen2.5-Coder-7B	21.30	< 0.001
Meta-Llama-3.1-8B	17.08	< 0.001
DeepSeek-V2-Lite-16B	2.52	0.28

Table 5.17: *Dunn's Post-Hoc Test with Adjusted p-values*

Comparison	**1 (p-value)**	**4 (p-value)**	**8 (p-value)**
Qwen-Coder2.5-7B-Instruct			
Batch size 1	1.00	0.08	< 0.001
Batch size 4	0.08	1.00	0.05
Batch size 8	< 0.001	0.05	1.00
Meta Llama3.1-8B-Instruct			
Batch size 1	1.00	0.15	0.09
Batch size 4	0.15	1.00	< 0.001
Batch size 8	0.09	< 0.001	1.00

Table 5.18: *Energy Consumption, Latency, Throughput, and Pass@1 for various batch sizes*

Model and Batch Size	Energy Consumption (kWh)	Latency (m)	Throughput (tokens/s)	Pass@1
Qwen-Coder2.5-7B-Instruct				
Batch size 1	0.79	174.67	39.43	0.32
Batch size 4	0.79	174.62	39.44	0.21
Batch size 8	0.80	174.62	39.44	0.21
Meta Llama3.1-8B-Instruct				
Batch size 1	0.90	201.23	38.14	0.24
Batch size 4	0.90	201.20	38.14	0.24
Batch size 8	0.91	201.03	38.17	0.24
DeepSeekCoder-V2-Lite-16B				
Batch size 1	0.39	108.25	61.52	0.29
Batch size 4	0.39	108.33	61.47	0.29
Batch size 8	0.39	108.45	61.41	0.49

Energy impact of increasing model size

Table 5.19: *Estimated Computational Requirements for Different Qwen2.5 Base Models*

Qwen2.5 Model	Prefill Compute	Decode Compute	Prefill Memory	Decode Memory	TTFT (ms)	TPOT (ms)	TGT (s)	VRAM (GB)
0.5B	0.66	0.00	0.00	0.00	0.00	0.00	0.77	1.00
1.5B	1.98	0.00	0.00	0.00	0.01	0.00	2.30	3.00
3B	3.96	0.01	0.01	0.01	0.02	0.01	4.60	6.00
7B	9.24	0.01	0.01	0.01	0.04	0.02	10.74	14.00
14B	18.48	0.03	0.03	0.03	0.08	0.03	21.47	28.00

Table 5.20: *Shapiro-Wilk Test Statistics for Different Model Sizes*

Qwen2.5 Model	**Shapiro-Wilk Statistic**	**p-value**
0.5B	0.75	< 0.001
1.5B	0.85	< 0.001
3B	0.92	< 0.001
7B	0.93	< 0.001
14B	0.87	< 0.001

Table 5.21: *Levene's Test for Differences in Energy Consumption by Model Size*

Qwen2.5 Model Sizes	**Levene's Test Statistic**	**p-value**
0.5B, 1.5B, 3B, 7B, 14B	591.13	$< .001$

Table 5.22: *Kruskal-Wallis Test for Differences in Energy Consumption by Model Size*

Qwen2.5 Model Sizes	**Kruskal-Wallis Test Statistic**	**p-value**
0.5B, 1.5B, 3B, 7B, 14B	5143.80	$< .001$

Table 5.23: *Dunn's Test for Differences Among Model Sizes*

Comparison of Qwen2.5 Models	**Qwen2.5-0.5B (p-value)**	**Qwen2.5-1.5B (p-value)**	**Qwen2.5-3B (p-value)**	**Qwen2.5-7B (p-value)**	**Qwen2.5-14B (p-value)**
0.5B	1.00	< 0.001	< 0.001	< 0.001	< 0.001
1.5B	< 0.001	1.00	< 0.001	< 0.001	< 0.001
3B	< 0.001	< 0.001	1.00	< 0.001	< 0.001
7B	< 0.001	< 0.001	< 0.001	1.00	< 0.001
14B	< 0.001	< 0.001	< 0.001	< 0.001	1.00

Table 5.24: *Energy Consumption, Latency, Throughput, Pass@1, and Pass@kWh for Five Qwen2.5 Base Models*

Qwen2.5 Model	**Energy Consumption (kWh)**	**Latency (m)**	**Throughput (tokens/s)**	**Pass@1**	**Pass@kWh**
0.5B	0.08	28.77	253.00	0.10	1.25
1.5B	0.21	56.52	126.62	0.23	1.10
3B	0.39	92.15	76.24	0.35	0.90
7B	0.87	191.68	37.00	0.31	0.36
14B	1.68	369.58	19.18	0.47	0.28

Energy Impact of Fine-Tuning

Table 5.25: *Expected Energy Impact of Fine-Tuning*

Qwen2.5 Model	**Prefill Compute**	**Decode Compute**	**Prefill Memory**	**Decode Memory**	**TTFT (ms)**	**TPOT (ms)**	**TGT (s)**	**VRAM (GB)**
0.5B	0.66	0.001	0.001	0.001	0.003	0.001	0.77	1.00
0.5B-Instruct	0.66	0.001	0.001	0.001	0.003	0.001	0.77	1.00
1.5B	1.98	0.003	0.003	0.003	0.009	0.004	2.30	3.00
1.5B-Instruct	1.98	0.003	0.003	0.003	0.009	0.004	2.30	3.00
3B	3.96	0.006	0.006	0.006	0.018	0.007	4.60	6.00
3B-Instruct	3.96	0.006	0.006	0.006	0.018	0.007	4.60	6.00
7B	9.24	0.014	0.014	0.014	0.042	0.016	10.74	14.00
7B-Instruct	9.24	0.014	0.014	0.014	0.042	0.016	10.74	14.00
14B	18.48	0.028	0.028	0.028	0.084	0.033	21.47	28.00
14B-Instruct	18.48	0.028	0.028	0.028	0.084	0.033	21.47	28.00

Table 5.26: *Shapiro-Wilk Test for Normality of Fine-Tuned Models*

Qwen2.5 Model	**Shapiro-Wilk Statistic**	**p-value**
0.5B	0.75	< 0.001
0.5B-Instruct	0.75	< 0.001
1.5B	0.85	< 0.001
1.5B-Instruct	0.89	< 0.001
3B	0.92	< 0.001
3B-Instruct	0.94	< 0.001
7B	0.93	< 0.001
7B-Instruct	0.93	< 0.001
14B	0.87	< 0.001
14B-Instruct	0.97	< 0.001

Table 5.27: *Levene's Test for Differences in Energy Consumption for Fine-Tuned Models*

Qwen2.5 Model	**Levene's Test Statistic**	**p-value**
0.5B	2.19	0.14
1.5B	0.65	0.42
3B	0.26	0.61
7B	4.37	0.04
14B	57.78	< 0.001

Table 5.28: *Kruskal-Wallis Test for Differences in Energy Consumption for Fine-Tuned Models*

Qwen2.5 Model	**Kruskal-Wallis Test Statistic**	**p-value**
0.5B	285.22	< 0.001
1.5B	87.38	< 0.001
3B	6.95	0.01
7B	19.23	< 0.001
14B	166.75	< 0.001

Table 5.29: *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for Fine-Tuned Models*

Qwen2.5 Model	**Energy Consumption (kWh)**	**Latency (m)**	**Throughput (tokens / s)**	**Pass@1**	**Pass@kWh**
0.5B	0.08	28.77	253.00	0.10	1.25
0.5B-Instruct	0.08	27.97	241.32	0.07	0.88
1.5B	0.21	56.52	126.62	0.23	1.10
1.5B-Instruct	0.21	57.72	124.58	0.22	1.05
3B	0.39	92.15	76.24	0.35	0.90
3B-Instruct	0.40	93.73	76.63	0.39	0.98
7B	0.87	191.68	37.00	0.31	0.36
7B-Instruct	0.82	180.82	38.95	0.45	0.55
14B	1.68	369.58	19.18	0.47	0.28
14B-Instruct	1.35	290.87	21.18	0.51	0.38

Energy impact of Activation-aware Weight Quantization

Table 5.30: *Estimated Computational Requirements for Activation-Aware Weight Quantization*

Qwen2.5 Model	**Prefill Compute**	**Decode Compute**	**Prefill Memory**	**Decode Memory**	**TTFT (ms)**	**TPOT (ms)**	**TGT (s)**	**VRAM (GB)**
0.5B-Instruct	0.66	0.001	0.001	0.001	0.003	0.001	0.77	1.00
0.5B-Instruct-AWQ	0.33	0.001	0.001	0.001	0.001	0.001	0.38	0.50
1.5B-Instruct	1.98	0.003	0.003	0.003	0.009	0.004	2.30	3.00
1.5B-Instruct-AWQ	0.99	0.002	0.002	0.002	0.003	0.002	1.15	1.50
3B-Instruct	3.96	0.006	0.006	0.006	0.018	0.007	4.60	6.00
3B-Instruct-AWQ	1.98	0.003	0.003	0.003	0.006	0.004	2.30	3.00
7B-Instruct	9.24	0.014	0.014	0.014	0.042	0.016	10.74	14.00
7B-Instruct-AWQ	4.62	0.007	0.007	0.007	0.014	0.008	5.36	7.00
14B-Instruct	18.48	0.028	0.028	0.028	0.084	0.033	21.47	28.00
14B-Instruct-AWQ	9.24	0.014	0.014	0.014	0.029	0.016	10.72	14.00

Table 5.31: *Shapiro-Wilk Test for Normality*

Qwen2.5 Model	**Shapiro-Wilk Statistic**	**p-value**
0.5B-Instruct	0.75	< 0.001
0.5B-Instruct-AWQ	0.61	< 0.001
1.5B-Instruct	0.89	< 0.001
1.5B-Instruct-AWQ	0.64	< 0.001
3B-Instruct	0.94	< 0.001
3B-Instruct-AWQ	0.79	< 0.001
7B-Instruct	0.93	< 0.001
7B-Instruct-AWQ	0.91	< 0.001
14B-Instruct	0.97	< 0.001
14B-Instruct-AWQ	0.74	< 0.001

Table 5.32: *Levene's Test for Differences in Energy Consumption for Model Sizes*

Qwen2.5 Model	**Levene's Test Statistic**	**p-value**
0.5B	33.98	< 0.001
1.5B	34.82	< 0.001
3B	375.75	< 0.001
7B	428.93	< 0.001
14B	4.46	0.03

Table 5.33: *Kruskal-Wallis Test for Differences in Energy Consumption for Model Sizes*

Qwen2.5 Model	**Kruskal-Wallis Test Statistic**	**p-value**
0.5B	312.53	< 0.001
1.5B	1123.57	< 0.001
3B	1455.62	< 0.001
7B	1643.12	< 0.001
14B	742.95	< 0.001

Table 5.34: *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for LLMs with and without AWQ*

Qwen2.5 Model	**Energy Consumption (kWh)**	**Latency (m)**	**Throughput (tokens / s)**	**Pass@1**	**Pass@kWh**
0.5B-Instruct	0.08	28.77	253.00	0.07	0.88
0.5B-Instruct-AWQ	0.08	35.50	185.27	0.01	0.13
1.5B-Instruct	0.21	56.52	126.62	0.22	1.05
1.5B-Instruct-AWQ	0.10	33.92	163.23	0.07	0.70
3B-Instruct	0.40	92.15	76.24	0.39	0.98
3B-Instruct-AWQ	0.14	43.18	139.00	0.26	1.86
7B-Instruct	0.82	191.68	37.00	0.45	0.55
7B-Instruct-AWQ	0.27	68.87	94.99	0.41	1.52
14B-Instruct	1.35	369.58	19.18	0.51	0.38
14B-Instruct-AWQ	0.66	141.25	49.66	0.27	0.41

Energy impact of GPTQ

Table 5.35: *GPTQ*

Qwen2.5 Model	**Prefill Compute**	**Decode Compute**	**Prefill Memory**	**Decode Memory**	**TTFT (ms)**	**TPOT (ms)**	**TGT (s)**	**VRAM (GB)**
0.5B-Instruct	0.66	0.001	0.001	0.001	0.003	0.001	0.77	1.00
0.5B-Instruct-GPTQ-Int4	0.17	0.0003	0.0003	0.0003	0.0005	0.0003	0.19	0.25
0.5B-Instruct-GPTQ-Int8	0.33	0.0005	0.0005	0.0005	0.0010	0.0006	0.38	0.50
1.5B-Instruct	1.98	0.003	0.003	0.003	0.009	0.004	2.30	3.00
1.5B-Instruct-GPTQ-Int4	0.50	0.0008	0.0008	0.0008	0.0015	0.0009	0.57	0.75
1.5B-Instruct-GPTQ-Int8	0.99	0.0015	0.0015	0.0015	0.0031	0.0017	1.15	1.50
3B-Instruct	3.96	0.006	0.006	0.006	0.018	0.007	4.60	6.00
3B-Instruct-GPTQ-Int4	0.99	0.0015	0.0015	0.0015	0.0031	0.0017	1.15	1.50
3B-Instruct-GPTQ-Int8	1.98	0.003	0.003	0.003	0.0062	0.0035	2.30	3.00
7B-Instruct	9.24	0.014	0.014	0.014	0.0417	0.0162	10.74	14.00
7B-Instruct-GPTQ-Int4	2.31	0.0035	0.0035	0.0035	0.0072	0.0041	2.68	3.50
7B-Instruct-GPTQ-Int8	4.62	0.0070	0.0070	0.0070	0.0144	0.0081	5.36	7.00
14B-Instruct	18.48	0.028	0.028	0.028	0.0835	0.0325	21.47	28.00
14B-Instruct-GPTQ-Int4	4.62	0.007	0.007	0.007	0.0144	0.0081	5.36	7.00
14B-Instruct-GPTQ-Int8	9.24	0.014	0.014	0.014	0.0288	0.0162	10.72	14.00

Table 5.36: *Shapiro-Wilk Test for Normality*

Qwen2.5 Model	Shapiro-Wilk Statistic	p-value
0.5B-Instruct	0.75	< 0.001
0.5B-Instruct-GPTQ-Int8	0.67	< 0.001
0.5B-Instruct-GPTQ-Int4	0.59	< 0.001
1.5B-Instruct	0.89	< 0.001
1.5B-Instruct-GPTQ-Int8	0.85	< 0.001
1.5B-Instruct-GPTQ-Int4	0.79	< 0.001
3B-Instruct	0.94	< 0.001
3B-Instruct-GPTQ-Int8	0.92	< 0.001
3B-Instruct-GPTQ-Int4	0.87	< 0.001
7B-Instruct	0.93	< 0.001
7B-Instruct-GPTQ-Int8	0.93	< 0.001
7B-Instruct-GPTQ-Int4	0.91	< 0.001
14B-Instruct	0.97	< 0.001
14B-Instruct-GPTQ-Int8	0.94	< 0.001
14B-Instruct-GPTQ-Int4	0.96	< 0.001

Table 5.37: *Levene's Test for Differences in Energy Consumption for Model Sizes*

Qwen2.5 Model	Levene's Test Statistic	p-value
0.5B	1.61	0.20
1.5B	82.73	< 0.001
3B	196.52	< 0.001
7B	244.59	< 0.001
14B	355.67	< 0.001

Table 5.38: *Kruskal-Wallis Test for Differences in Energy Consumption for Model Sizes*

Qwen2.5 Model	**Kruskal-Wallis Test Statistic**	**p-value**
0.5B	794.81	< 0.001
1.5B	1415.38	< 0.001
3B	1890.48	< 0.001
7B	2236.10	< 0.001
14B	2354.20	< 0.001

Table 5.39: *Dunn's Post-Hoc Test with Adjusted p-values for GPTQ Models*

Comparison of Qwen2.5 models	**Instruct (p-value)**	**Instruct-GPTQ-Int8 (p-value)**	**Instruct-GPTQ-Int4 (p-value)**
0.5B-Instruct	1.00	< 0.001	< 0.001
0.5B-Instruct-GPTQ-Int8	< 0.001	1.00	< 0.001
0.5B-Instruct-GPTQ-Int4	< 0.001	< 0.001	1.00
1.5B-Instruct	1.00	< 0.001	< 0.001
1.5B-Instruct-GPTQ-Int8	< 0.001	1.00	< 0.001
1.5B-Instruct-GPTQ-Int4	< 0.001	< 0.001	1.00
3B-Instruct	1.00	< 0.001	< 0.001
3B-Instruct-GPTQ-Int8	< 0.001	1.00	< 0.001
3B-Instruct-GPTQ-Int4	< 0.001	< 0.001	1.00
7B-Instruct	1.00	< 0.001	< 0.001
7B-Instruct-GPTQ-Int8	< 0.001	1.00	< 0.001
7B-Instruct-GPTQ-Int4	< 0.001	< 0.001	1.00
14B-Instruct	1.00	< 0.001	< 0.001
14B-Instruct-GPTQ-Int8	< 0.001	1.00	< 0.001
14B-Instruct-GPTQ-Int4	< 0.001	< 0.001	1.00

Table 5.40: *Energy Consumption (kWh), Latency, Throughput, Pass@1, and Pass@kWh for GPTQ-Int8 and GPTQ-Int4*

Qwen2.5 Model and Quantization Type	**Energy Consumption (kWh)**	**Latency (m)**	**Throughput (tokens / s)**	**Pass@1**	**Pass@kWh**
0.5B-Instruct	0.08	28.77	253.00	0.07	0.88
0.5B-Instruct-GPTQ-Int8	0.07	28.72	232.46	0.07	1.00
0.5B-Instruct-GPTQ-Int4	0.07	30.77	211.77	0.04	0.57
1.5B-Instruct	0.21	56.52	126.62	0.22	1.05
1.5B-Instruct-GPTQ-Int8	0.14	44.58	162.31	0.24	1.71
1.5B-Instruct-GPTQ-Int4	0.11	37.17	196.85	0.20	1.82
3B-Instruct	0.40	92.15	76.24	0.39	0.98
3B-Instruct-GPTQ-Int8	0.24	65.80	109.43	0.39	1.63
3B-Instruct-GPTQ-Int4	0.16	49.72	146.09	0.34	2.13
7B-Instruct	0.82	191.68	37.00	0.45	0.55
7B-Instruct-GPTQ-Int8	0.49	109.55	63.99	0.46	0.94
7B-Instruct-GPTQ-Int4	0.30	72.62	97.13	0.44	1.47
14B-Instruct	1.35	369.58	19.18	0.51	0.38
14B-Instruct-GPTQ-Int8	0.79	168.13	36.69	0.52	0.66
14B-Instruct-GPTQ-Int4	0.46	102.23	58.47	0.49	1.07

Bibliography

[1] Nadine Amsel et al. “Toward sustainable software engineering (nier track)”. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 976–979.

[2] Sara S Mahmoud and Imtiaz Ahmad. “A green model for sustainable software engineering”. In: *International Journal of Software Engineering and Its Applications* 7.4 (2013), pp. 55–74.

[3] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. “Analyzing Programming Languages’ Energy Consumption: An Empirical Study”. In: *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. PCI ’17. New York, NY, USA: Association for Computing Machinery, Sept. 2017, pp. 1–6. ISBN: 978-1-4503-5355-7. DOI: 10.1145/3139367.3139418. URL: https://doi.org/10.1145/3139367.3139418 (visited on 05/31/2024).

[4] Daniele D’Agostino et al. “Hardware and Software Solutions for Energy-Efficient Computing in Scientific Programming”. In: *Scientific Programming* 2021.1 (2021), p. 5514284.

[5] Payal Dhar. “The carbon impact of artificial intelligence.” In: *Nat. Mach. Intell.* 2.8 (2020), pp. 423–425.

[6] Eric Masanet et al. “Recalibrating global data center energy-use estimates”. In: *Science* 367.6481 (2020), pp. 984–986.

[7] Tom B Brown. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[8] Nishith Reddy Mannuru et al. “Artificial intelligence in developing countries: The impact of generative artificial intelligence (AI) technologies for development”. In: *Information Development* (2023), p. 02666669231200628.

[9] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: https://aclanthology.org/P19-1355.

[10] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[11] Jens Malmodin et al. “ICT sector electricity consumption and greenhouse gas emissions–2020 outcome”. In: *Telecommunications Policy* (2024), p. 102701.

[12] Steffen Lange, Johanna Pohl, and Tilman Santarius. “Digitalization and energy consumption. Does ICT reduce energy demand?” In: *Ecological economics* 176 (2020), p. 106760.

[13] Guglielmo Tamburrini. “The AI carbon footprint and responsibilities of AI scientists”. In: *Philosophies* 7.1 (2022), p. 4.

[14] Pablo Villalobos et al. “Machine learning model sizes and the parameter gap”. In: *arXiv preprint arXiv:2207.02852* (2022).

[15] Hannah Ritchie and Pablo Rosado. “Electricity Mix”. In: *Our World in Data* (2020). https://ourworldindata.org/el
mix.

[16] Charlotte Freitag et al. “The climate impact of ICT: A review of estimates, trends and regulations”. In: *arXiv preprint arXiv:2102.02622* (2021).

[17] Guosai Wang, Lifei Zhang, and Wei Xu. “What can we learn from four years of data center hardware failures?” In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, pp. 25–36.

[18] Barbara Krumay and Roman Brandtweiner. “Measuring the environmental impact of ICT hardware”. In: *Environmental & Economic Impact on Sustainable Development (2016)* 238 (2016).

[19] Luiz André Barroso and Urs Hölzle. “The case for energy-proportional computing”. In: *Computer* 40.12 (2007), pp. 33–37.

[20] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. “Energy efficiency in cloud computing data centers: a survey on software technologies”. In: *Cluster Computing* 26.3 (2023), pp. 1845–1875.

[21] Krishna V Palem. “Energy aware algorithm design via probabilistic computing: From algorithms and models to Moore’s law and novel (semiconductor) devices”. In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 2003, pp. 113–116.

[22] SO Akinola. “Energy consumption, cyclomatic and time complexities of variants of quicksort algorithm: A comparative study”. In: *Journal of Science Research* 15.1 (2016), pp. 8–8.

[23] Kristina Carter et al. “Energy Complexity for Sorting Algorithms in Java”. In: *arXiv preprint arXiv:2311.07298* (2023).

[24] Daniel Nichols et al. “Performance-aligned llms for generating fast code”. In: *arXiv preprint arXiv:2404.18864* (2024).

[25] Burak Yetiştiren et al. "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt". In: *arXiv preprint arXiv:2304.10778* (2023).

[26] Jovan Stojkovic et al. *Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference*. arXiv:2403.20306 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2403.20306. URL: http://arxiv.org/abs/2403.20306 (visited on 05/30/2024).

[27] Michael Chui et al. "Technology trends outlook 2023". In: (2023).

[28] Adam Dingle and Martin Kruliš. "Tackling Students' Coding Assignments with LLMs". In: (2024).

[29] Biao Zhang et al. "When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method". In: *arXiv preprint arXiv:2402.17193* (2024).

[30] Robert R Harmon and Nora Auseklis. "Sustainable IT services: Assessing the impact of green computing practices". In: *PICMET'09-2009 Portland International Conference on Management of Engineering & Technology*. IEEE. 2009, pp. 1707–1717.

[31] Vyacheslav Kharchenko et al. "Green computing and communications in critical application domains: Challenges and solutions". In: *The International Conference on Digital Technologies 2013*. IEEE. 2013, pp. 191–197.

[32] Tajana Simunic et al. "Source code optimization and profiling of energy consumption in embedded systems". In: *Proceedings 13th International Symposium on System Synthesis*. IEEE. 2000, pp. 193–198.

[33] David A. Ortiz and Nayda G. Santiago. "Impact of source code optimizations on power consumption of embedded systems". In: *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*. June 2008, pp. 133–136. DOI: 10.1109/NEWCAS.2008.4606339. URL: https://ieeexplore.ieee.org/abstract/document/4606339 (visited on 05/30/2024).

[34] Constanze Fetting. "The European green deal". In: *ESDN report* 53 (2020).

[35] Carl-Friedrich Schleussner et al. "Science and policy characteristics of the Paris Agreement temperature goal". In: *Nature Climate Change* 6.9 (2016), pp. 827–835.

[36] Haoye Tian et al. "Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–13.

[37] Gerhard Fettweis and Ernesto Zimmermann. "ICT energy consumption-trends and challenges". In: *Proceedings of the 11th international symposium on wireless personal multimedia communications*. Vol. 2. 4. (Lapland Finland. 2008, p. 6.

[38] Emilia Hansson and Oliwer Ellréus. *Code Correctness and Quality in the Era of AI Code Generation: Examining ChatGPT and GitHub Copilot*. 2023.

[39] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". en. In: *Advances in Neural Information Processing Systems* 36 (Dec. 2023), pp. 21558–21572. URL: `https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html` (visited on 05/30/2024).

[40] Elias Frantar et al. "Gptq: Accurate post-training quantization for generative pre-trained transformers". In: *arXiv preprint arXiv:2210.17323* (2022).

[41] Ji Lin et al. "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems* 6 (2024), pp. 87–100.

[42] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "A review of energy measurement approaches". In: *ACM SIGOPS Operating Systems Review* 47.3 (2013), pp. 42–49.

[43] Achim Guldner et al. "Energy consumption and hardware utilization of standard software: Methods and measurements for software sustainability". In: *From Science to Society: New Trends in Environmental Informatics*. Springer. 2018, pp. 251–261.

[44] Philipp Hurni et al. "On the accuracy of software-based energy estimation techniques". In: *Wireless Sensor Networks: 8th European Conference, EWSN 2011, Bonn, Germany, February 23-25, 2011. Proceedings 8*. Springer. 2011, pp. 49–64.

[45] Qingqing Cao, Aruna Balasubramanian, and Niranjan Balasubramanian. "Towards accurate and reliable energy measurement of NLP models". In: *arXiv preprint arXiv:2010.05248* (2020).

[46] Tina Vartziotis et al. *Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation*. arXiv:2403.03344 [cs]. Mar. 2024. DOI: `10.48550/arXiv.2403.03344`. URL: `http://arxiv.org/abs/2403.03344` (visited on 05/30/2024).

[47] Vlad-Andrei Cursaru et al. "A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama". In: *arXiv preprint arXiv:2405.03616* (2024).

[48] Alycia Lee, Brando Miranda, and Sanmi Koyejo. "Beyond scale: the diversity coefficient as a data quality metric demonstrates llms are pre-trained on formally diverse data". In: *arXiv preprint arXiv:2306.13840* (2023).

[49] Russell A Poldrack, Thomas Lu, and Gašper Beguš. "AI-assisted coding: Experiments with GPT-4". In: *arXiv preprint arXiv:2304.13187* (2023).

[50] Javier Mancebo, Felix Garcia, and Coral Calero. "A process for analysing the energy efficiency of software". In: *Information and Software Technology* 134 (2021), p. 106560.

[51] Andreas Schuler and Gabriele Kotsis. "A systematic review on techniques and approaches to estimate mobile software energy consumption". In: *Sustainable Computing: Informatics and Systems* (2023), p. 100919.

[52] Martin Burtscher, Ivan Zecena, and Ziliang Zong. “Measuring GPU power with the K20 built-in sensor”. In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2014, pp. 28–36.

[53] Xinxin Mei, Qiang Wang, and Xiaowen Chu. “A survey and measurement study of GPU DVFS on energy conservation”. In: *Digital Communications and Networks* 3.2 (2017), pp. 89–100.

[54] Marcus Hähnel et al. “Measuring energy consumption for short code paths using RAPL”. In: *ACM SIGMETRICS Performance Evaluation Review* 40.3 (2012), pp. 13–17.

[55] Yehia Arafa et al. “Verified instruction-level energy consumption measurement for nvidia gpus”. In: *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 2020, pp. 60–70.

[56] Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tozun. “Profiling and Monitoring Deep Learning Training Tasks”. In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 2023, pp. 18–25.

[57] Nupur Kothari and Arka Bhattacharya. “Joulemeter: Virtual machine power measurement and management”. In: *MSR Tech Report* (2009).

[58] Aurélien Bourdon et al. “Powerapi: A software library to monitor the energy consumed at the process-level”. In: *ERCIM News* 92 (2013), pp. 43–44.

[59] Nathan Binkert et al. “The gem5 simulator”. In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[60] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. California, USA. 2005, pp. 10–5555.

[61] Jaeseok Yun et al. “Monitoring and control of energy consumption using smart sockets and smartphones”. In: *Computer Applications for Security, Control and System Engineering: International Conferences, SecTech, CA, CES 3 2012, Held in Conjunction with GST 2012, Jeju Island, Korea, November 28-December 2, 2012. Proceedings*. Springer. 2012, pp. 284–290.

[62] Takeshi Kamiyama, Hiroshi Inamura, and Ken Ohta. “A model-based energy profiler using online logging for Android applications”. In: *2014 Seventh International Conference on Mobile Computing and Ubiquitous Networking (ICMU)*. IEEE. 2014, pp. 7–13.

[63] Victor Schmidt et al. “CodeCarbon: estimate and track carbon emissions from machine learning computing”. In: *Cited on* 20 (2021).

[64] Omar Shaikh et al. “EnergyVis: interactively tracking and exploring energy consumption for ML models”. In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–7.

[65] Prakhar Ganesh et al. "Compressing large-scale transformer-based models: A case study on bert". In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 1061–1080.

[66] Venkatesh Balavadhani Parthasarathy et al. "The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities". In: *arXiv preprint arXiv:2408.13296* (2024).

[67] Zhaozhuo Xu et al. "Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt". In: *arXiv preprint arXiv:2305.11186* (2023).

[68] Torsten Hoefler et al. "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks". In: *Journal of Machine Learning Research* 22.241 (2021), pp. 1–124.

[69] V Sanh. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[70] A Vaswani. "Attention is all you need". In: *Advances in Neural Information Processing Systems* (2017).

[71] Mingi Ryu. "[RE] ALBERT: A Lite BERT for Self-supervised Learning of Language Representations". In: (2021).

[72] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The efficient transformer". In: *arXiv preprint arXiv:2001.04451* (2020).

[73] Robert A Jacobs et al. "Adaptive mixtures of local experts". In: *Neural computation* 3.1 (1991), pp. 79–87.

[74] Noam Shazeer et al. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer". In: *arXiv preprint arXiv:1701.06538* (2017).

[75] Jiaao He et al. "Fastmoe: A fast mixture-of-expert training system". In: *arXiv preprint arXiv:2103.13262* (2021).

[76] Rewon Child et al. "Generating long sequences with sparse transformers". In: *arXiv preprint arXiv:1904.10509* (2019).

[77] Iz Beltagy, Matthew E Peters, and Arman Cohan. "Longformer: The long-document transformer". In: *arXiv preprint arXiv:2004.05150* (2020).

[78] Deepak Narayanan et al. "Efficient large-scale language model training on gpu clusters using megatron-lm". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[79] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[80] Roy Schwartz et al. "Green ai". In: *Communications of the ACM* 63.12 (2020), pp. 54–63.

[81] Alexandre Lacoste et al. “Quantifying the carbon emissions of machine learning”. In: *arXiv preprint arXiv:1910.09700* (2019).

[82] Liyuan Liu et al. “Understanding the difficulty of training transformers”. In: *arXiv preprint arXiv:2004.08249* (2020).

[83] James Bergstra et al. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems* 24 (2011).

[84] Lu Lu et al. “Physics-informed neural networks with hard constraints for inverse design”. In: *SIAM Journal on Scientific Computing* 43.6 (2021), B1105–B1132.

[85] Imrus Salehin et al. “AutoML: A systematic review on automated machine learning with neural architecture search”. In: *Journal of Information and Intelligence* 2.1 (2024), pp. 52–81.

[86] Tim Dettmers et al. “Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 30318–30332.

[87] Geoffrey Hinton. “Distilling the Knowledge in a Neural Network”. In: *arXiv preprint arXiv:1503.02531* (2015).

[88] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[89] Terry Yue Zhuo et al. “BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions”. In: *arXiv preprint arXiv:2406.15877* (2024).

[90] Sida Peng et al. “The impact of ai on developer productivity: Evidence from github copilot”. In: *arXiv preprint arXiv:2302.06590* (2023).

[91] Katie Kang et al. “Unfamiliar finetuning examples control how language models hallucinate”. In: *arXiv preprint arXiv:2403.05612* (2024).

[92] Sarah Fakhoury et al. “LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation”. In: *arXiv preprint arXiv:2404.10100* (2024).

[93] Mohammed Latif Siddiq and Joanna CS Santos. “Generate and pray: Using sallms to evaluate the security of llm generated code”. In: *arXiv preprint arXiv:2311.00889* (2023).

[94] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[95] Michele Tufano et al. “Unit test case generation with transformers and focal context”. In: *arXiv preprint arXiv:2009.05617* (2020).

[96] Wanqin Ma, Chenyang Yang, and Christian Kästner. “(Why) Is My Prompt Getting Worse? Rethinking Regression Testing for Evolving LLM APIs”. In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 2024, pp. 166–171.

[97] Brittany Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[98] Diego Marcilio et al. “Are static analysis violations really fixed? a closer look at realistic usage of sonarqube”. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 209–219.

[99] Atanas Rountev, Scott Kagan, and Michael Gibas. “Static and dynamic analysis of call chains in Java”. In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. 2004, pp. 1–11.

[100] Aryaz Eghbali and Michael Pradel. “DynaPyt: a dynamic analysis framework for Python”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 760–771.

[101] Heejae Chon et al. “Is Functional Correctness Enough to Evaluate Code Language Models? Exploring Diversity of Generated Codes”. In: *arXiv preprint arXiv:2408.14504* (2024).

[102] Jacob Austin et al. “Program synthesis with large language models”. In: *arXiv preprint arXiv:2108.07732* (2021).

[103] William Murphy et al. “Combining LLM Code Generation with Formal Specifications and Reactive Program Synthesis”. In: *arXiv preprint arXiv:2410.19736* (2024).

[104] Pratyush Patel et al. “Characterizing Power Management Opportunities for LLMs in the Cloud”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 207–222.

[105] Aimee van Wynsberghe. “Sustainable AI: AI for sustainability and the sustainability of AI”. en. In: *AI and Ethics* 1.3 (Aug. 2021), pp. 213–218. ISSN: 2730-5961. DOI: `10.1007/s43681-021-00043-6`. URL: `https://doi.org/10.1007/s43681-021-00043-6` (visited on 04/25/2024).

[106] Q Vera Liao and Jennifer Wortman Vaughan. “Ai transparency in the age of llms: A human-centered research roadmap”. In: *arXiv preprint arXiv:2306.01941* (2023), pp. 5368–5393.

[107] Binyuan Hui et al. “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186* (2024).

[108] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[109] Qihao Zhu et al. “DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence”. In: *arXiv preprint arXiv:2406.11931* (2024).

[110] *Metrics*. en. URL: `https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html` (visited on 11/01/2024).

[111] Alec Radford. “Improving language understanding by generative pre-training”. In: (2018).

[112] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *ArXiv e-prints* (2016), arXiv–1607.

[113] Zhengyan Zhang et al. “Moefication: Transformer feed-forward layers are mixtures of experts”. In: *arXiv preprint arXiv:2110.01786* (2021).

[114] Shuowei Jin et al. “Compute Or Load KV Cache? Why Not Both?” In: *arXiv preprint arXiv:2410.03065* (2024).

[115] Noam Shazeer. “Fast transformer decoding: One write-head is all you need”. In: *arXiv preprint arXiv:1911.02150* (2019).

[116] Tim Dettmers et al. “Qlora: Efficient finetuning of quantized llms”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[117] David Patterson et al. “Carbon emissions and large neural network training”. In: *arXiv preprint arXiv:2104.10350* (2021).