MSc Computer Science — Artificial Intelligence Technology

Reducing Carbon Emissions of Code Generation in Large Language Models using Line-level Completions

Thijs Nulle

April 11, 2025

Thesis Title

Reducing Carbon Emissions of Code Generation in Large Language Models with Line-level Completions

Name:	Thijs Nulle
Student Number:	4942000
Degree Program:	MSc Computer Science — Artificial Intelligence Technology
University:	Technical University Delft
Department Name:	Software Engineering Research Group
Thesis Advisor:	Arie van Deursen
Daily Supervisor:	Luís Miranda da Cruz
Committee Member:	Jie Yang
Date of Submission:	April 1, 2025
Date of Defence:	April 11, 2025

Abstract

This thesis investigates reducing carbon emissions in code generation using large language models (LLMs) by comparing function-level and line-level code completions across models of different sizes (1.5B and 9B parameters). The study utilises the BigCodeBench dataset, comprising 1,140 Python programming problems, to evaluate the energy consumption, test accuracy, and time efficiency of code completions. The models, 4-bit quantised and run on a CPU, performed 30 function-level completions and 30 line-level completions for each line, which were tested for correctness. Results indicate that, while line-level completions require slightly more energy per token, they are more efficient overall in terms of total energy consumption and token usage. The smaller model with line-level completions showed significant reductions in carbon emissions, achieving an average tenfold reduction compared to the large model with function-level completions. With the large model, line-level completions achieved a $4.5\times$ reduction in carbon emissions compared to function-level completions. Line-level completions were more token-efficient, wasting less than 1% of energy, compared to 20% for function-level completions. From a sustainability perspective, line-level completions offer a practical strategy to reduce the environmental impact of code generation tasks while maintaining strong performance. The study suggests that optimising completion strategies could help balance energy consumption, test accuracy, and time efficiency. Future research could explore a broader range of model sizes, fine-tuning models specifically for line-level completions, a performance decrease in solution length, and alternative validation metrics to assess code generation performance.

Table of Contents

Preface

Well, here it is—the culmination of seven months of challenging but rewarding work. While I have heard plenty of thesis stories from my friends, it does feel different when you do it yourself. Even though some parts of the process went more smoothly than others, I can safely say that I am incredibly proud of what I have achieved and what I put in front of you to read, understand, and hopefully make you think about something most people have never thought about—the carbon footprint of AI.

Before the thesis, it has been a long road that has taught me more about computer science, and myself, than I could have hoped for when I started. I still remember my choice to study Computer Science at TU Delft, all because I enjoyed making websites. Now I know that that is one of the things I do not want to do in the future—funny how life gave me lemons, but I refused to make lemonade. In seven years, I have learned everything from how a computer works at the lowest level to the protocols used to communicate over the internet, how programming languages actually work and the algorithms that power the modern world. With even a brief intermezzo at Forze Hydrogen Racing, I do not think I could have prepared myself better for the next steps in life.

Somewhere along this journey, I picked up reading again—something I despised in high school and vowed I would never do. But then it clicked after reading *The Diversity of Life* by *E.O. Wilson*, where he portrayed his ideas with words I had never heard, sparking a newfound interest in the world around me and making me curious to see if I could do the same. Subsequently, I followed the course *Sustainable Software Engineering*, and it just worked—the balance between sustainability and how it can be applied to software engineering. Personally, it was obvious that I wanted to pursue a thesis in this field and see what was available for me to become more knowledgeable about.

After diving into the topic of sustainability and AI—in both the academic and professional world—it has become clear that the current focus is very shortsighted. Every AI company is releasing as many products as possible to capture as much market share as possible, and it seems like we are hitting a wall with performance. The current language models are a glorified—but very good—auto-complete, and I hypothesise a significant architectural change needs to happen for the next big breakthrough.

To overcome the plateau we hit in performance caused by the training phase, we shifted to using techniques that increased the computational requirements during inference time, such as test-time compute and chain-of-thought prompting, however, this increased the energy consumption to squeeze out the last bit of performance. I do not think that we are at a point in time where marginal performance improvements will make any tangible difference for most tasks, and what I would love to see is that companies focus their efforts on achieving similar performance more efficiently —saving costs and reducing carbon emissions.

With my current understanding, I believe we are somewhere along the peak of inflated expectations in the Gartner Hype Cycle, so I wish we would focus on understanding our current technology and how we can best utilise it, while improving its efficiency along the way, and not overdo it by integrating it into every possible product you can imagine. Ultimately, we need to be open to innovation and make AI more efficient and sustainable in the process, with the goal of usable products and integrations that feel right, not rushed, like almost all current applications—I do not want another chatbot on a random website, AI-generated content that contains zero creativity, or a fake voice where it sounds like a robot.

I do want to acknowledge some people who helped me during the thesis. Thank you Luís for providing feedback along the road, and generally helping me find the path I wanted to take during the thesis. A big thanks to João, Francisco and Arie for helping me iron out the last details. And, of course, I want to thank my parents, brother, family, and close friends for being there along the way; it *really* meant a lot. Now let us dive in and explore the world of carbon emissions and code generation.

— Thijs Nulle, April 2025

Nomenclature

AI	Artificial Intelligence
CPU	Central Processing Unit, the main processor responsible for computation tasks in a computer
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit, used for parallel processing tasks in a computer
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
CO_2	Carbon dioxide
CO_2eq	Carbon dioxide equivalent emissions
°C	Degrees Celsius
Data centre	A facility used to house computer systems for large-scale processing
Fossil fuels	Non-renewable energy sources (e.g., coal, oil, and natural gas)
Function-level completions	Code completions where the model generates a function implementation
Granularity	Completion level for code generation (e.g. function-level, line-level)
Hallucination	When a model generates incorrect or fabricated information not based on input data
Inference	The process by which a trained model generates outputs based on new inputs
Line-level completions	Code completions where the model generates one line at a time
Parameters	The weights and biases in a model that are learned during the training process
Prompt	A text or query given to a language model to guide its response or generation process
Quantisation	Reducing a model's precision to lower memory needs with minimal performance loss
Renewable energy	Energy from sources that are naturally replenished (e.g., solar, wind)
Token	A unit of text, like a word or fragment, processed by a model for generation or understanding
Tokenisation	The process of breaking down text into tokens that can be processed by a model
Trigger point	A keyword or operator in code where generation begins, guiding model completions

1 Introduction

The story of climate change is often framed as a catastrophe—rising temperatures, extreme weather, melting ice caps, and dire predictions for the future. It is true that global temperatures have increased by about 1°C since pre-industrial levels and could climb by 1.5°C by 2050, potentially reaching 2–4°C by 2100 [24]. We are in the midst of the sixth mass extinction, losing biodiversity at an accelerating pace [28] and even extreme weather events are becoming more frequent and intense, directly linked to human actions [10]. These are statements we cannot ignore, but it is equally important not to overlook a critical truth: we are making unprecedented progress toward sustainability, enhancing the quality of life for more people than ever before.

We have collectively reduced the child mortality rate from 50% in industrial times to 4% in 2020 [27], where vaccines have saved 150 million children over the last 50 years and in the meantime, we even eradicated smallpox [78]. The discovery of antibiotics, such as penicillin in the 1920s [36], drastically reduced deaths from bacterial infections, and contributed to an increase in life expectancy. This, in turn, resulted in increased demand for food, which was counteracted by an almost tenfold increase in average global crop yield in the same period [91]. We now have an average life expectancy $2.5\times$ higher than in 1850, increasing from 29.3 years to a staggering 73.2 years.

However, these advancements do not paint the complete picture as carbon emissions are still a big problem. While per capita carbon emissions peaked in 2012 and have since declined [41], total emissions continue to rise due to growing populations and increasing energy demand in developing economies [89]. Where the USA and Europe accounted for 85% of global emissions in 1950, they only contribute 35% now. Europe has shown it is possible to reduce carbon emissions drastically—a 45% per capita decrease since the peak in 1979—and this will only be the precedent for the decades to come.

Currently, the energy sector emits 75% of all global emissions, with almost 30% of all emissions stemming from electricity and heat generation [114]. Whereas electricity production was dominated by fossil fuels such as coal, natural gas, and hydro, renewable energy sources are growing exponentially [52]. Over the last five years, global wind energy production doubled and solar energy production increased tenfold in the preceding decade [54]. We have reached a pivotal point in electricity production as renewable energy sources have become cheaper than fossil fuels, total investments into renewable energy are twice as large as into fossil fuels [53] and renewable energy sources have shown to claim $50\times$ less lives than fossil fuels [90].

Yet, as we accelerate toward a cleaner future, new technologies reshape our energy landscape. One of the most transformative—and energy-intensive—developments in recent years has been the rise of large language models (LLMs). These computational models are designed for natural language processing, a subfield of artificial intelligence that enables computers to analyse and generate text based on patterns in human language [87]. Widespread adoption followed the introduction of ChatGPT in 2022 [1], with applications including but not limited to translation, summarisation, transcription, and code generation.

This technology, however, comes at an environmental cost. These models are trained on expansive corpora—spanning every book ever written, countless scientific papers, the entirety of Wikipedia in multiple languages, and much more. To process this data, a lot of processing power is required, and subsequently, a lot of energy is also required. For example, training the open-source BLOOM model emitted approximately 25 tonnes CO_2eq, where the training of GPT-3 is estimated to have emitted 500 tonnes CO_2eq [70].

Castaño et al. [18] found a correlation between carbon emissions, dataset size, and model size—which is worrying as our current solution to increase performance is to increase the size of the model and dataset. The most famous model family is the GPT family by OpenAI, where model sizes have increased from 1.5B parameters for GPT-2, to 175B for GPT-3, to an astounding 1.76T parameters in GPT-4—a 1200-fold increase. Subsequently, GPT-3 was trained on 300B tokens, compared to 13T tokens of GPT-4 [15, 95]. The developments over the years have shown tremendous results and the models are better than anyone could have expected over a decade ago, however, with ever-increasing adoption it warrants the question of environmental sustainability.

However, considering training is insufficient, these models also require energy during inference—the process of responding to a prompt. At Google, inference accounted for 60% of the total energy consumption due to their large userbase [81], with estimates for the energy consumption of the inference phase consuming up to 95% for ChatGPT-like services [22]. Subsequently, recent methods for increasing performance increase inference time by utilising chain-of-thought prompting, where the model iteratively prompts itself to refine its answer [115]. It does improve performance, but it also substantially increases energy consumption because of the generated intermediary tokens.

LLMs are deployed within data centres, a large collection of computers specifically aimed to perform large numbers of computations. Data centres account for roughly 1-1.5% of all global emissions [51], and are expected to rise by 160% by 2030 [2]. This warrants research in increasing the efficiency of LLMs without compromising accuracy, such that we can sustain the benefits of the technologies, without incurring future risks regarding sustainability.

As we recognise the environmental impact of LLMs, it is crucial to examine the specific areas where these models are being applied and how their energy consumption can be reduced. One of these applications is in the software engineering field, where we can utilise LLMs to perform tasks such as code generation, bug detection, or writing documentation. Models have increasingly gotten better and better, from Codex that powers GitHub Copilot [21], Code LLaMa [93], StarCoder [67], DeepSeek-Coder [43], and Claude [9]. The Stack Overflow 2024 Developer Survey showed 63% of professional developers currently use AI in their development process, with another 14% planning to begin soon [101]—a clear indication of how integral these models have become in modern software development.

Despite extensive research on using LLMs for software engineering, the results are mixed. Some studies show improved code quality [92, 12], while others note an increase in code churn—code added and changed within two weeks—since the introduction of LLMs[40]. The overall impact remains unclear, but more research is needed to fully understand the role of LLMs in software engineering.

And yes, these models have gotten significantly better in recent years, but their performance still falls short. The SWE-bench benchmark tests models on real-world problems [56], but the top performer, Claude 2, solves only 1.96% of problems. Even with a specialized framework [120] and the upgraded Claude 3 Opus model, performance only increased to 11.6%. Despite this sixfold improvement, no company would hire an engineer who solves just 12% of basic tasks.

However, despite the significant advancements in these models, there is limited research on optimising generation strategies specifically for code generation tasks. Different methods of token generation have shown promise in reducing token costs and improving inference efficiency [116], but further exploration is needed to apply these strategies effectively in the context of software engineering.

To assess the performance of LLMs in code generation tasks, several different benchmarks have been released over the years. The first major benchmark was the HumanEval benchmark in 2021 [21], comprising 143 handwritten Python programming problems. Soon after MBPP was released [11], consisting of 974 simple programming tasks. MultiPL-E was released to assess performance across multiple programming languages [17], and finally, BigCodeBench contains problems with solutions spanning multiple domains to facilitate compositional reasoning [125]. But there is one problem, they all assess the performance of these models with the same generation technique: function-level completions.

Function-level completions involve generating code for entire functions based on input prompts, which are then promptly evaluated for correctness. However, this approach does not fully capture the complexity and nuances of real-world software development, where code is often written incrementally, with multiple smaller code snippets and logic that evolve gradually. This limitation means that while LLMs may perform well on benchmarks that focus solely on standalone functions, their ability to be utilised in a real-world context is limited, as they may struggle to handle the iterative nature of development. Additionally, one of the issues with function-level completions is the continual generation of excess tokens, which harms developer productivity and incurs a waste of computational resources [45].

To address these challenges, we explore line-level completions, where models generate code incrementally, reflecting the iterative workflow of developers. We compare this approach to function-level completions, evaluating not only performance but also environmental impact and efficiency. By considering both technical and ecological factors, we aim to optimize code generation and, in doing so, address the following key research questions:

RQ1: *How does energy consumption compare between line-level and function-level completions?*

RQ1.1: What is the quantitative impact of excess token generation on energy consumption?

RQ1.2: What is the quantitative impact of incorrect suggestions on energy consumption?

RQ2: *What reduction in carbon emissions can be achieved by substituting a smaller model for a larger model, without compromising accuracy?*

RQ2.1: How do energy per token and time per token compare?

Regarding the first research question, we aim to find differences between the energy efficiency of line-level completions and function-level completions. If a discrepancy exists, we hypothesise that it results from a difference in excess token generation and the impact of incorrect suggestions.

Subsequently, since one of the main contributors to energy consumption in LLMs is model size, we aim to determine if a large model can be substituted with a small model without compromising accuracy, potentially leading to token efficiency gains. To answer this question, we need to understand the different energy and time characteristics of various models and completion granularities.

Through our research, we have uncovered several key insights. Firstly, individual line-level completions exhibit high compilation rates—96.4% for the small model and 97.1% for the large model—and strong test pass rates—82.7% for the small model and 90.5% for the large model. Secondly, line-level completions do not outperform function-level completions with the large model in only 22% of problems with the small model, and this decreases to 13.5% for the large model.

Utilising the performance characteristics of line-level completions, we can achieve a $10\times$ reduction in carbon emissions if we substitute a large model performing function-level completions with a small model performing line-level completions. If we only change the completion granularity from function-level to line-level completions, this results in a $4.5\times$ reduction in emissions. Subsequently, carbon emission reductions stabilise after substituting more than half the lines in the reference solution.

We have found line-level completions to be more deterministic, with the average log-probability for each token generation being closer to 0 in both the large and the small model. Subsequently, the large model more often generated tokens until the token limit compared to the small model, in 25% and 21% of cases, respectively. Because of this, function-level completions exhibit more inconsistent energy characteristics, where energy measurements might differ from expectations. We finally found that incorrect function-level completions have a similar impact to incorrect line-level completions regarding wasted energy consumption. Building on this, we compare line- and function-level completions not only in terms of performance but also efficiency, energy consumption, and environmental impact, providing a list of key takeaways:

- **Comparative analysis**: We evaluated line-level vs. function-level completions, focusing on token efficiency, energy consumption, and generation accuracy.
- **Energy reduction**: Line-level completions can reduce energy consumption by $4.5\times$, and up to $10\times$ with model substitution.
- **Token efficiency**: Line-level completions achieve a nearly $50\times$ improvement in token efficiency compared to function-level completions.
- **Reproducibility**: We developed a public repository for full experiment reproducibility[1]

The *Background* section provides an overview of key concepts, including machine learning, large language models, code generation, and carbon emissions. Following an overview of the existing literature in the *Related Works* section, we cover the *Methodology* of this research. In this section, we detail

[1]https://github.com/thijsnulle/msc-experiment

the model and dataset selection, prompt design, experimental setup, energy measurement techniques, and evaluation metrics. The *Results* section presents findings on energy consumption across different completion granularities and carbon emission reductions through model substitution. The *Discussion* examines key and secondary findings, threats to validity, broader implications, future research directions, and ethical considerations. Finally, the *Conclusion* summarises key insights and emphasises the contributions to advancing sustainable AI.

2 Background

In this section, we discuss the progression of machine learning and artificial intelligence, from deep learning to large language models, followed by an examination of carbon emissions and code generation.

2.1 Machine Learning

Artificial intelligence (AI) is a field of study which enables machines to exhibit forms of intelligence using learning to achieve defined goals [94]. Machine learning (ML) is a subset of AI, and it allows machines to perform tasks that typically require human intelligence, including learning and problem solving [38]. In machine learning, intelligence is often represented through a model. This statistical framework learns patterns within data and generalises that knowledge to make predictions on new, unseen information—something we humans do frequently. Figure 1 contains a hierarchical representation of the artificial intelligence topics we cover in this section.

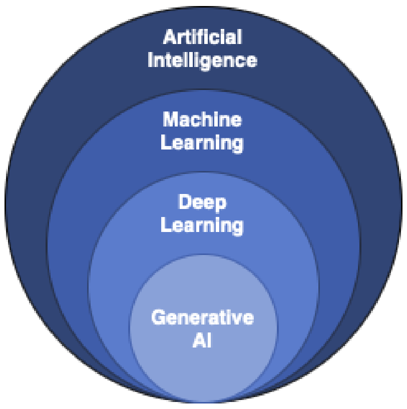

Figure 1: Hierarchical structure of AI, highlighting Machine Learning, Deep Learning, and Generative AI.

2.1.1 Concepts and Techniques

For this section, we consider a hypothetical problem where the goal is to determine the type of fruit based on three features: colour, shape and weight.

In machine learning, a model is a program that finds patterns within data and predicts outcomes based on novel information [94]. The goal of the model is often to predict a label, which is a value assigned to an example in the dataset that describes its qualities. This label categories the data points and groups instances together, e.g. apple, pear or orange. The model examines each example in the dataset and considers its features—colour, shape and weight—and aims to find the relationship between the features and the label. The number of dimensions, or dimensionality, refers to the amount of different features that are available for the model to learn; three in our case. See Figure 2 for an example of increasing dimensionality.

For a model to learn what feature values are associated with which labels, we need to iteratively teach it what is right and what is wrong—the training phase. The model can be taught with different learning paradigms, which we explain later. After giving the model many examples of fruits, including their characteristics, it might find correlations between features and labels; the colour orange is associated with oranges, an elongated shape indicates a pear or a heavy fruit corresponds to an apple.

The next step in the model development pipeline is the inference phase. Inference is the process of making predictions based on the patterns learned in the training phase. Essentially, it is the model in action. For instance, given a green fruit with an elongated shape, the model uses its learned patterns to determine that a pear is the most likely outcome.

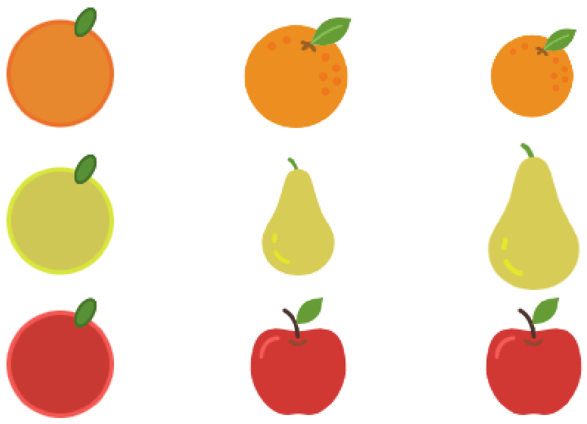
Figure 2: Illustration of increasing dimensionality in fruit classification, first based on color, then shape, and finally weight.

With the ability to get predictions for new data points, we can perform model evaluation to assess the model performance. We split the total dataset into a training and validation set, where we utilise the latter to cross-check the predicted labels with the ground truths. The two most rudimentary metrics are accuracy and precision. Accuracy refers to how often a classification is correct; it measures the proportion of fruits for which the model correctly predicted their label [74]. Precision measures the accuracy of a single prediction; it is the proportion of fruits classified as a pear that are actual pears [85]. See Figure 3 for a visual comparison between accuracy and precision.

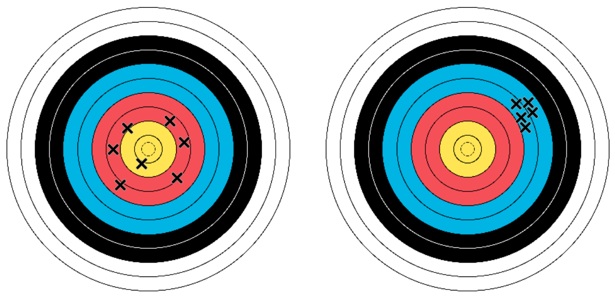
Figure 3: A visual comparison highlighting the difference between Accuracy (left) and Precision (right).

2.1.2 Learning Paradigms

We can train models in different ways, based on the amount and type of information that is available. Fundamentally, machine learning can be categorised into three main paradigms, each with distinct data characteristics and feedback mechanisms: supervised learning, unsupervised learning and reinforcement learning.

In supervised learning, the data consists of examples containing both the inputs and the desired output [76]. With a known output for a set of inputs, we can utilise this information as a feedback mechanism; if the model predicts the wrong output, use that information to tune the model for improved future predictions.

In unsupervised learning, the data consists only of a collection of inputs without an explicit output [39]. In contrast to supervised learning, where feedback is used to correct errors, unsupervised learning identifies similarities within the data. Referring back to our hypothetical machine learning problem, the model would have no knowledge of what combination of colour, shape and weight corresponds to what type of fruit. Instead, the goal shifts to clustering fruits by similar characteristics, enabling the model to determine how many different types of fruit are within the dataset.

In reinforcement learning, a model learns the best future actions based on previously taken actions [58]. The model learns by trial and error, receiving feedback as rewards or penalties, to optimise its behaviour and achieve a desired outcome. In contrast to supervised and unsupervised learning, reinforcement learning involves actively interacting with an environment to learn how to take optimal actions. An example of reinforcement learning would be a robot that needs to catch a ball; if it succeeds, it gets a reward, but if it fails, it receives a penalty. Based on that feedback, it adjusts its approach and tries again to catch the ball.

2.1.3 Neural Networks

For this section, we consider a hypothetical problem where the goal is to determine handwritten digits from 0 to 9 within an image, where each pixel is either black or white.

A neural network is a machine learning model inspired by the structure and function of biological neural networks in animal brains. An animal brain consists of neurons that transmit electrical information in highly complex networks via synapses. In machine learning, this concept translates to artificial neurons (neurons) connected by edges (synapses) forming a graph network. Onwards, we refer to artificial neurons as neurons for increased clarity. See Figure 4 for an illustration of a neural network.

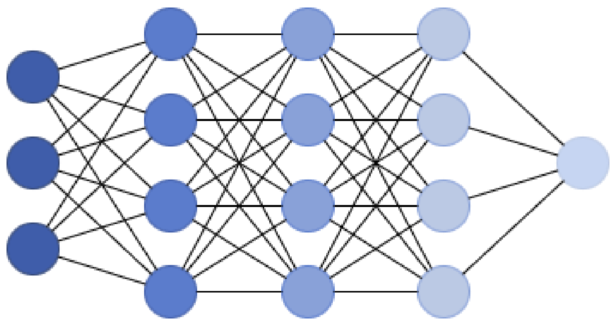

Figure 4: Illustration of a multi-layered neural network featuring 3 input neurons and 1 output neuron.

Similar to biological neural networks, neurons can receive input from multiple neurons and transmit output to multiple neurons, which are typically aggregated into layers for simplicity. Signals travel from the first layer (input layer) to the last layer (output layer), possibly passing through multiple intermediate layers (hidden layers) [14].

For our hypothetical problem, the input layer would consist of a neuron for each pixel—100 neurons for a 10 by 10-pixel image—and the output layer would consist of a neuron for each digit. Since each digit can be represented as multiple different combinations of black and white pixels, the hidden layers help model these complex relationships and aggregate pixel values in a way that leads to a more accurate prediction. After passing through one or multiple hidden layers, the signal arrives in the output layer. The output neuron with the highest value is the prediction the model has of which the handwritten digit is on the image.

Training neural networks is done through empirical risk minimisation: optimising the network parameters to minimise the difference between the predicted output and the actual target values within the training dataset [105]. Whenever a wrong result is predicted, we utilise backpropagation to modify the model weights slightly to become closer to the expected result. Essentially, backpropagation is the process of tuning the weights to enable the neural network to make more accurate predictions.

Training a neural network can be illustrated as tuning a guitar. The goal would be to achieve perfect pitch—the desired prediction. Plucking a string provides the current pitch, a new prediction. To change the pitch closer to the expected pitch, we can turn the tuning peg slightly. Similarly to neural networks' backpropagation, iterative adjustments gradually reduce errors and improve the predictive performance of the model.

2.1.4 Natural Language Processing

Natural language processing (NLP) is a subfield of artificial intelligence that enables computers to process and generate data encoded in natural language [23]. NLP covers several tasks, including language translation, text summarisation, and question answering [60, 61]. Natural language is highly unstructured and variable, due to complexities such as syntax, semantics and context.

NLP models are not inherently different than other machine learning models, under the hood, they still can only process numbers and perform mathematical operations on them. To transform text into numerical data, we perform a process called tokenisation. In tokenisation, the text is divided into smaller units, called tokens, which are then represented as numerical vectors. Tokenisation converts human-readable words into words from a specialised computer dictionary [84]. For example, the word *computer* could become two tokens—*com* and *puter*—which are then transformed into numbers.

2.1.5 Deep Learning

Determining which handwritten number is in an image is not overly complex, all the interdependencies can be modelled with a relatively small model in computing terms. In contrast, determining what movie someone might like based on their viewing history of hundreds of movies, dozens of ratings they gave, and thousands of available films already becomes a daunting task. Add millions of other users and their preferences to the same equation, and we arrive at the point where a small model will not suffice.

Taking the perspective of human ability, one needs a larger brain to be able to recommend a movie based on all those premises compared to determining what handwritten digit is within an image. Consequently, deep learning is an extension of neural networks consisting of multiple hidden layers and a larger amount of nodes within the network [65]. A handwritten digit model can be as small as 1000 neurons [80], while the first deep learning model to uproot the scientific community—AlexNet in 2012—already had 650,000 neurons; an increase of almost three orders of magnitude [63].

The advantage of deep learning found in recent years has been their ability to model increasingly complex relationships within data. This increased complexity can only be modelled with an increased model size, which has been increasing exponentially for decades [109]. Where the number of parameters of AlexNet was 60 million, the parameter count of recent models exceeds an astounding 1 trillion—five orders of magnitude difference in just 10 years.

To be able to tune the parameters of a model of increasing size, the total dataset size needs to increase as well. Training data consequently has also been growing exponentially, AlexNet was trained on roughly 1.2 million images, whereas current deep learning models are trained on datasets with trillions of data points [108].

To grasp the scale of a model with 1 trillion parameters, consider storing each parameter in a 1 cm-square within a notebook. This alone would require 14,000 football fields of space—larger than the country of San Marino. If we were to lay them one after the other, we would reach to the moon and back; thirteen times.

In essence, a deep neural network is no different compared to a neural network. However, deep learning operates at an unprecedented scale, where model and dataset sizes have become inconceivably large. Deep learning is the underlying technology most people associate artificial intelligence with and it is the basis upon which we build for the remaining sections; and thesis.

2.2 Large Language Models

A large language model (LLM) is a computational model designed for natural language processing (NLP) and is a subfield of artificial intelligence which aims to provide computers with the ability to process data encoded in natural language leveraging deep learning techniques [87]. Interfacing with an LLM involves providing an input sequence, known as a prompt, from which the model generates a sequence of tokens that can statistically follow the input prompt, utilising learned patterns in the training data; so-called

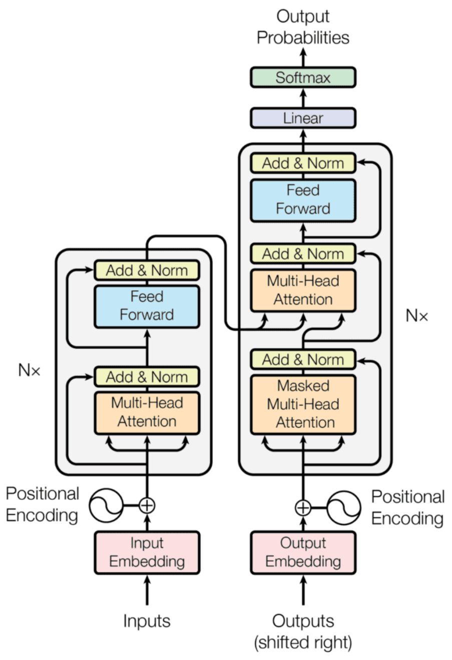

Figure 5: The Transformer model architecture, adapted from Vaswani [106].

prompting. By providing extra context within the prompt, such as a specific task like translation, the goal of the generation can be altered.

Previous architectures—namely recurrent neural networks (RNN) and long short-term memory networks (LSTM)—were designed to capture dependencies within sequences by using generated outputs as inputs for subsequent steps [48]. While these architectures allowed for capturing some long-range dependencies, their abilities to maintain relevant information decreased with longer texts.

RNNs and LSTMs require sequential processing, where each token in a sequence is processed one after the other, and each output generation depends on the previous token [48]. Due to their sequential nature, these models cannot take advantage of the parallel processing capabilities of GPUs or TPUs. As a result, training was slow and inefficient, with processing time becoming a critical bottleneck—something we cannot mitigate.

2.2.1 Transformer Architecture

To overcome the disadvantage of sequential processing, the Transformer architecture introduces a self-attention mechanism that allows the model to weigh the importance of each token in a sequence relative to other tokens, regardless of their position [106]. This self-attention mechanism is integrated within an encoder-decoder structure, where the encoder processes the input sequence and the decoder generates the output sequence while considering contextual importance between words. This architecture not only models dependencies between tokens in long sequences effectively but also enables parallel processing of the input and parallel generation of the output, serving as the driving force behind the exponential growth in model complexity. See Figure 5 for a diagram of the transformer model architecture.

2.2.2 Applications

While LLMs are still a relatively new technology, widespread adoption of language models followed the introduction of ChatGPT in 2022 [1]. Due to the generalisation capabilities of LLMs and as they are trained on a large, diverse dataset of textual data, they can be utilised for a wide array of tasks.

Question answering is among the overt applications of LLMs; it typically involves predicting the subsequent sequence of tokens given an initial input text. With a large amount of training data, the model can generalise across a diverse set of topics and generate appropriate answers for many questions.

As previously mentioned, the goal of a prompt can be changed by introducing more context. This can serve as a stepping stone for various tasks, including but not limited to translation, recommender systems, virtual assistants, text search, and transcription [57].

Subsequently, LLMs can generate code based on natural language prompts [21]. In combination with context about the programming environment, the model can generate solutions for a diverse set of problems. Beyond code generation, the model can assist in debugging existing code or explain exceptions the programmer encountered. We explain this in more detail in Section 2.3.

2.2.3 Limitations

Even though LLMs significantly improved the performance of machine learning models, and that allowed for an increased number of tasks to be tackled with them, they still have limitations. Most notably, due to the probabilistic nature of generating an output sequence, answers can be incorrect—the model hallucinates. A generated text might seem convincing and correct, however, an output sequence is bound to have mistakes at some point [46].

Generating an output sequence essentially aggregates the training dataset, thus, the data quality is vital. Everything someone writes, albeit a scientific article or a tweet, contains inherent biases. Consequently, users must remain critical of the information produced by these models, understanding that it may reflect existing prejudices and inaccuracies present within the training data.

While taking the inherent weaknesses of inaccuracy and implicit biases for granted, an arguably more impactful limitation is the lack of common sense reasoning. Mirzadeh et al. [75] found that LLMs exhibit noticeable variance when responding to different instantiations of the same question. They hypothesise that the current LLM architecture is incapable of genuine logical reasoning; instead, they replicate reasoning processes observed within training data.

2.3 Code Generation

Brief History Programming has evolved significantly, beginning with Alan Turing's theoretical concept of the *universal computing machine*, which laid the groundwork for modern computation. Early programming involved machine code, a sequence of ones and zeros inputted via punch cards. This was followed by assembly languages, which provided a more human-readable representation of machine code—arguably the first form of code generation, where machine code was translated from assembly instructions. This soon evolved into programming languages with each their unique features, another layer on top of an already existing layer of abstraction.

These programming languages are where code generation, or automatic programming, started. Some of the earliest forms of automatic programming include template-driven and rule-based code generation. Template-driven generation relies on pre-defined templates with placeholders filled dynamically based on input parameters, often used for tasks like boilerplate code generation. Rule-based generation can transform high-level specifications into executable code. A macro can be defined as a rudimentary form of rule-based code generation, however, more complex transformations exist beyond simple text replacement [64].

Pereira and Warren [83] discuss *definite clause grammars* (DCGs), an extension of context-free grammars that were used in early AI systems for code generation. DCGs were employed to generate syntactically valid code by applying probabilistic transformations—techniques that bear a striking resemblance to the methods used by current LLMs. An implementation of DCGs is a stochastic grammar, which are models trained to predict the likelihood of code token sequences.

To overcome the limitations of generating more complex patterns, models transitioned to using hidden Markov models (HMMs), which offered a practical approach for predicting token sequences based on existing code. These models were trained on code tokens and could generate likely code sequences based on an input sequence, such as an existing piece of code [47]. These, in combination with N-grams, could generate simple code, however, their ability to capture long-range dependencies and more complex patterns was still limited. While N-grams focus on local context, and HMMs model transitions between states, they struggled to effectively handle the full scope of programming language syntax and semantics.

After a brief intermezzo with genetic programming, where the goal was to evolve small programs through mutation and selection to achieve a specific objective [62], Bayesian networks emerged as a powerful tool for predicting likely program structures based on observed patterns [79]—essentially a more generalised version of HMMs. Unlike HMMs, which are primarily suited for sequential data, Bayesian networks offer greater flexibility by modelling complex relationships and conditional dependencies between variables in a more general, acyclic graph structure.

The AI field experienced periods of stagnation, known as the AI Winter, where progress slowed due to limited resources, unrealistic expectations, and lack of practical results. Research in AI, including early code generation models, faced setbacks, leading to reduced funding and interest. However, with the rise of deep learning in the 2010s, AI research was revitalised, where significant advancements made us arrive at the current state of code generation.

Code Generation with LLMs In the context of LLMs, code generation refers to using an AI model to automatically generate source code based on a given prompt or input. The input can be a description or specification of the expected behaviour in natural language or a partially written code snippet the model needs to complete. In the context of code generation, a large language model (LLM) can perform a variety of tasks, including the following:

- **Code generation**: The process of automatically creating code based on a natural language description or specific requirements.
- **Code completion**: Predicting and automatically filling in partial code snippets or functions as the developer types.
- **Code refactoring**: Improving the structure, efficiency, and readability of existing code without changing its functionality.
- **Bug detection**: Identifying potential errors, issues, or vulnerabilities in the code through analysis or testing.
- **Code translation**: Converting code written in one programming language into another language, while preserving its functionality.
- **Writing documentation**: Automatically generating documentation to explain the purpose and usage of code.

Levels of Code Generation Code generation with LLMs can occur at various levels, including token, line, and function-level completion. *Token-level* generation is the rudimentary form of code generation and forms the basis for the subsequent code generation techniques. Essentially, it generates the next most likely token for a given input sequence. *Line-level* generation performs multiple token-level generations until a newline character: `\n`. In contrast, *function-level* completion generates tokens until the end of a function implementation. However, since the full structure of a function cannot be accurately determined from tokens alone, LLMs often produce excessive token generation [45]. An example illustrating the different levels of code generation, from token-level to function-level completion, is shown in Listing 1.

```python
# Token-level generation:
def add(a, b):
  result

# Line-level generation:
def add(a, b):
  result = a + b

# Function-level generation:
def add(a, b):
  result = a + b
  return result
```

Listing 1: Examples of token-level, line-level, and function-level code generation, all based on the prompt `def add(a, b):\n`.

Challenges in Code Generation One of the primary challenges in code generation with LLMs is ensuring accuracy and reliability. Despite significant advancements, LLMs still struggle to generate code that is both syntactically correct and functionally accurate. Often, models may produce code that compiles but fails to function as intended, especially in complex scenarios involving edge cases or intricate logic [123]. Additionally, a significant challenge lies in context understanding, where LLMs may generate code without fully grasping its broader context—such as the dependencies between different functions or files. Chen et al. [20] emphasise this issue, noting that while models can generate syntactically correct snippets, they often miss crucial contextual information, leading to incomplete or incorrect code.

In addition to the challenges of accuracy and context, security risks have become an increasingly important concern. LLMs can inadvertently generate insecure code, introducing vulnerabilities due to factors such as outdated or unsafe libraries, improper input validation, or other common coding pitfalls. The risk of generating unsafe code is growing, especially when developers place trust in model suggestions without adequate review. Wang et al. [110] underscore this concern, demonstrating how LLMs may lack the understanding required to assess the security implications of the code they generate.

2.4 Carbon Emissions

Since the Industrial Revolution, humans began burning fossil fuels, and the concentration of carbon dioxide and other greenhouse gases significantly increased. When we burn fossil fuels, the reaction releases gases that stay in the atmosphere and insulate the planet from losing heat to space—the greenhouse effect. As a result, global warming of about 1.2 °C occurred since the industrial revolution, with the average surface temperature rising 0.18 °C per decade since 1981 [68]. Subsequently, measures of atmospheric carbon dioxide concentrations increased by almost 50% [7].

Carbon footprint is a calculated measure that allows for a comparison of the total amount of greenhouse gases that are added to the atmosphere. It is typically expressed in tonnes of carbon dioxide equivalent emissions (CO_2eq), which account for all greenhouse gases emitted and subsequently converts them into an equal amount of CO_2 based on their heat-trapping potential over a specific time period [19].

2.4.1 Quantifying Energy Consumption

In the context of code generation in LLMs, energy consumption is the key factor in assessing environmental impact. Energy consumption is driven by the computational demands of training, fine-tuning, and deployment of these models. To quantify energy consumption, we need to monitor the hardware that runs the model and measure power draw over time. Modern GPUs and CPUs have built-in sensors to determine power usage accurately.

With these tools, we can determine the total energy usage within a specific time frame. However, this information alone is insufficient to estimate carbon emissions. To calculate emissions, we also need the carbon intensity of the energy source, which represents the total CO_2eq emitted per kilowatt-hour (kWh) of electricity generated. Table 3 provides an overview of the carbon intensity for various technologies used in electricity generation.

Technology	**50th percentile** (g CO_2eq/kWh)
Hydroelectric	4
Wind	12
Nuclear	16
Solar Thermal	22
Geothermal	45
Biomass	230
Natural Gas	469
Coal	1001

Table 1: Lifecycle greenhouse gas emissions by electricity source, adapted from Edenhofer [31].

Due to the different energy generation characteristics of various countries, the carbon intensity varies significantly from country to country. Countries where most of their energy comes from hydroelectric have lower emissions (Norway, 30g CO_2eq/kWh), compared to countries that primarily use coal for electricity generation (China, 582g CO_2eq/kWh) [33].

2.4.2 Climate Change Mitigation

Climate change mitigation, also known as decarbonisation, aims to limit the greenhouse gases in the atmosphere that contribute to global warming. Mitigation strategies include energy conservation and the replacement of fossil fuels with alternative, cleaner energy sources, and modifying land use to enhance carbon sequestration—such as reforestation or carbon capture methods [35].

There are two primary approaches to reducing energy consumption: energy conservation and energy efficiency. Energy conservation is the effort made to reduce overall energy consumption, often by using less of an energy service, such as driving less. Increasing energy efficiency is to reduce the amount of energy required to create products or provide services, such as insulating a building to reduce heating requirements.

2.5 Summary

The background provides an overview of machine learning (ML) and artificial intelligence (AI), tracing the evolution from basic models to LLMs. Machine learning models, neural networks, and deep learning form the foundation of AI.

Deep learning underpins modern LLMs, which use transformer architectures for efficient, parallel processing of natural language tasks. LLMs excel in diverse applications, such as translation, question answering, and notably, code generation. Despite their capabilities, LLMs face challenges, including hallucinations, biases, and limitations in logical reasoning.

LLMs can perform various tasks, including code generation, translation, completion and refactoring. LLMs can complete code at different granularities (e.g. token-, line-, or function-level) during code generation.

Greenhouse gas concentrations have significantly increased, particularly carbon dioxide, since the Industrial Revolution due to the burning of fossil fuels. That has led to global warming of approximately 1.2 °C, with a rise of 0.18 °C per decade since 1981. Carbon footprint, expressed in tonnes of carbon dioxide equivalent (CO_2eq), is a comparative measure of greenhouse gas emissions by accounting for their heat-trapping potential.

In the context of LLMs, energy consumption—driven by the computational demands of training, fine-tuning, and deployment—is a critical factor in assessing environmental impact. Measuring power usage and determining the carbon intensity of energy sources (in g CO_2eq/kWh) are essential steps in quantifying emissions. The table outlines emissions across different electricity generation technologies, with renewable sources like hydroelectric and wind producing significantly less CO_2eq than fossil fuels like coal.

Various climate change mitigation strategies exist, including energy conservation, efficiency improvements, shifting to cleaner energy sources, and enhancing carbon sequestration through land use changes. These measures aim to reduce greenhouse gas emissions and limit global warming.

3 Related Works

This section reviews key research and developments in the field of large language models, with a particular focus on their environmental impact. By analysing existing literature, this section identifies trends, highlights gaps, and positions this study within the broader academic discourse.

3.1 Foundation Models for Code

Roziere et al. [93] release *Code Llama*, a family of LLMs for code based on Llama 2. It includes infilling capabilities, support for large input contexts and zero-shot instruction performance for programming tasks. They provide the foundation and instruction-following models—both including Python specialisations—with 7B, 13B, 34B, and 70B parameters each. Code Llama has scores of up to 67% and 65% on HumanEval and MBPP, respectively.

Hui et al. [50] release Qwen2.5-Coder model family, based on the Qwen2 language models [119]. They support up to 128K tokens of context, covering 92 programming languages. They achieved remarkable performance in (multi-programming) code generation, completion, and repair. Additionally, they provide instruction-tuned models with improved performance for natural language prompts.

Guo et al. [43] provide a range of open-source code models, DeepSeek-Coder, with 1.3B, 6.7B, and 33B parameters. They have been trained from scratch on 2 trillion tokens, pre-trained on a high-quality code corpus, and support code generation and infilling. DeepSeek-Coder achieves state-of-the-art performance among open-source models, but also outperforms existing closed-source models like Codex and GPT-3.5.

01.AI [6] release Yi-Coder, a series of open-source code models with 1.5B and 9B parameters. For both model sizes, a base model and an instruction-tuned model are available. The models are pre-trained on 2.4 trillion high-quality tokens over 52 programming languages, they have 128K tokens of context and outperform all similarly sized models. Additionally, the models perform better in mathematical reasoning tasks, even compared to larger models such as DeepSeek-Coder-33B.

Li et al. [67] introduce StarCoder and StarCoderBase, 15.5B-parameter code models with 8K context, infilling, and efficient inference. Trained on 1T tokens from The Stack, StarCoderBase outperforms all open multilingual Code LLMs. StarCoder, fine-tuned on 35B Python tokens, retains strong cross-language performance.

The models presented in this subsection provide the foundation for code generation with LLMs, with features ranging from code infilling, repair, and completion. These models have achieved state-of-the-art performance for multiple programming languages across varying model sizes—ranging from 1.5B to 70B parameters. Given our research aims to reduce the carbon emissions of code generation, understanding these foundational models is crucial, as they form the baseline for evaluating efficiency improvements.

3.2 Code Generation Benchmarks

Code generation benchmarks are critical for evaluating the functional correctness, reasoning capabilities, and contextual understanding of large language models in programming tasks. This subsection discusses prominent benchmarks, highlighting their unique characteristics, evaluation metrics, and contributions to advancing research in code generation.

Chen et al. [21] release the HumanEval benchmark, an evaluation set to measure functional correctness for synthesising programs from docstrings. They provided the pass@k metric, an unbiased estimator of the test pass accuracy, as highlighted in Equation 1. Additionally, they show that the BLEU score may not be a reliable indicator of functional correctness. The dataset consists of 164 hand-written programming problems, each including a function signature, docstring, reference solution, and several unit tests.

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

Lu et al. [69] introduce the CodeXGLUE benchmark, specifically focused on program understanding and code generation. It includes a collection of 10 tasks across 14 datasets and a platform for model evaluation and comparison.

Yu et al. [122] propose the CoderEval benchmark, comprising 230 Python and Java code generation tasks. In existing benchmarks, most problems ask for a standalone function as a solution, however, 70% of function popular open-source repositories are non-standalone functions. They support tasks from six levels of context-dependency, such as types, APIs, variables, and current class, file or project context. They found current models perform better on standalone functions.

Gu et al. [42] present the CRUXEval benchmark, consisting of 800 Python problems focused on code reasoning, understanding and execution evaluation. They propose a generic recipe for generating the execution benchmark to allow for future benchmark variations. Additionally, they evaluate twenty code models on the benchmark and discover discrepancies between well-performing models on different benchmarks and their respective performance on the CRUXEval benchmark, highlighting possible training data contamination.

Zhuo et al. [125] introduce the BigCodeBench benchmark, which focuses on problems that invoke function calls from multiple domains. It comprises 1,140 Python problems with function calls from 139 libraries across 7 domains. Each task contains on average 5.6 unit tests with an average branch coverage of 99%. Additionally, they provide the BigCodeBench-Instruct benchmark, where each docstring is transformed into a natural language prompt instead. They find current models are incapable of following complex instructions, with performance of up to 60%, compared to human performance of 97%.

Code generation benchmarks serve as essential tools for evaluating the performance of large language models in programming tasks, providing insights into their functional correctness, reasoning capabilities, and contextual awareness. The benchmarks discussed here—HumanEval, CodeXGLUE, CoderEval, CRUXEval, and BigCodeBench—offer diverse challenges, from relatively simple problems and problems with increasing amounts of context-dependencies, to problems focused on code reasoning and understanding. These benchmarks are particularly relevant to our research, as they establish the standard metrics for measuring efficiency and accuracy in code completion.

3.3 Carbon Footprint of AI

Luccioni, Viguier, and Ligozat [70] aim to quantify the carbon footprint of BLOOM, a 176B parameter language model, across its life cycle. They estimate the final training emitted approximately 24.7 tonnes of CO_2eq, and 50.5 tonnes if accounted for all processes, from equipment manufacturing to energy consumption during inference.

Castaño et al. [18] analyse the carbon footprint measurement of 1,417 Huggingface models, including associated datasets. They perform a repository mining study on the Huggingface Hub API. The study found correlations between carbon emissions and various attributes, such as model size, dataset size, application domains and performance metrics. The results underscore the need for improvements in energy reporting practices and promoting carbon-efficient model development.

Everman et al. [34] conduct a study on the carbon impact of various open-source LLMs, including GPT-J 6B, GPT Neo 1.3B and 2.7B, and GPT-2. They found a lack of reliable measurement tools, standard methodologies, and evaluation metrics. To compare carbon impact, they propose a quantitative framework that measures and contrasts the environmental impact of different LLMs. Subsequently, they show LLMs with high environmental impact do not necessarily provide improved performance.

Luccioni, Jernite, and Strubell [71] investigate the energy requirements of different machine learning tasks. To provide an accurate representation of the energy requirements, they perform 1,000 inferences for each task and collect the energy consumption of each model. They found *image generation* is undoubtedly the most energy inefficient by an order of magnitude with a median of 200g CO_2eq per 1,000 inferences. Text generation has a median of 5g CO_2eq per 1,000 inferences, and text classification is the most efficient with a median of 0.5g CO_2eq per 1,000 inferences.

Villalobos and Ho [108] analyse over 200 notable ML models to estimate training dataset sizes, finding that vision and language datasets have historically grown at 0.1 and 0.2 orders of magnitude per year, respectively. A shift around 2014-2015 led to significantly larger datasets and the disappearance of smaller language datasets, even though this may be due to sample size limitations. They also present trends for games, speech, and recommendation models.

Villalobos et al. [109] analyse trends in model size across notable ML systems, finding that language models grew by seven orders of magnitude from 1950 to 2018, then accelerated by another five orders in just four years up until 2022. Vision models followed a steadier trajectory, growing seven orders of magnitude by 2022. They identify a ”parameter gap” since 2020, with many models below 20B and above 70B parameters but few in between.

Understanding the carbon footprint of AI is essential for developing sustainable machine learning practices, as LLMs require significant computational resources throughout their lifecycle. The covered studies quantify the carbon emissions of multiple models during training and inference, providing deeper insights into the environmental impact of AI technologies. If we include the trend of exponentially increasing model sizes, these findings underscore the need for standardised carbon measurement tools and offer a basis for this research on how to report the energy consumption, ultimately enabling more effective strategies for reducing carbon emissions.

3.4 Optimising LLMs for Efficiency

Verdecchia, Sallou, and Cruz [107] provide a systematic review of Green AI—AI research focused on environmental sustainability—by analysing 98 primary studies. They find most studies consider monitoring AI model footprint, tuning hyperparameters to reduce energy consumption, or benchmarking models. Reported Green AI energy savings go up to 115%, with savings of 50% being rather common.

Stojkovic et al. [102] present the trade-offs of making energy efficiency the primary goal of serving LLMs under performance service-level agreements (SLAs). They show that depending on the inputs, the model and the SLAs, there are different options for the LLM inference provider to increase efficiency. Lowering GPU frequency does change the time-to-first-token, maximum throughput and power consumption, however, these do not change linearly with changes in the frequency. Subsequently, employing parallelism does increase the efficiency, but in combination with decreasing the frequency, it also shows diminishing returns.

Zhou et al. [124] survey existing literature on efficient LLM inference. They find the primary causes of inefficient LLM inference; namely large model size, the quadratic-complexity attention operation, and the auto-regressive decoding approach. Subsequently, they organise current literature in a comprehensive taxonomy into data-level, model-level, and system-level optimisations. Some optimisations include prompt pruning on a data-level, quantisation on a model-level, and batching on a system-level.

Cunningham, Archambault, and Kung [26] focus on the application of model compression, quantisation and hardware acceleration techniques for the Llama models. They find pruning and knowledge distillation methods to be effective in reducing model size, lowering training times and decreasing energy consumption. Quantisation techniques for 4- and 8-bit representations significantly decrease memory usage and improve computational speed, without substantial accuracy loss.

Chien et al. [22] provide a workload model of carbon emissions of model inference. They show that for ChatGPT-like services, inference dominates emissions, producing 25 times the emissions compared to the training phase. Subsequently, they show that carbon-aware algorithms can both maintain user experience and reduce emissions by 35%. When considering a future scenario in 2035, their proposed algorithm can reduce emissions by up to 56%.

Shi et al. [99] propose Avatar, an approach to optimise code generation of an LLM in terms of model size, inference latency and energy consumption; and thus, carbon footprint. Utilising a Satisfiability Modulo Theory solver, they find the Pareto-optimal configuration in the complete configuration space for training a model with knowledge distillation. With Avatar, they trained a model with decreased model size (160×), energy consumption (184×), carbon footprint (157×), and inference latency (76×), with only a negligible loss in performance of 1.67%.

Li et al. [66] investigate strategies for reducing the carbon footprint of LLM inference, focusing on energy-efficient model execution. They explore methods such as workload-aware scheduling and dynamic resource allocation to optimise power consumption while maintaining generation quality. Their findings highlight the potential of adaptive inference techniques to enhance sustainability. Their proposed method achieves over 40% carbon reduction in real-world tests with Llama2 and global grid data.

To reduce carbon emissions of LLMs, one of the key aspects is to improve their efficiency. The studies discussed in this section explore various strategies, including energy-efficient inference, model compression, and carbon-aware scheduling. These approaches directly relate to our research, as they provide insights into reducing computationl costs and carbon emissions of LLMs, highlighting the importance of improving both model architecture and the inference processes to increase efficiency and decrease carbon emissions.

4 Methodology

The objective of this experiment is to compare the energy consumption of code generation by a small (1.5B) and large (9B) language model. We focus on function-level and line-level completions to assess their impact on both energy usage and the correctness of generated code. To answer the research questions previously posed, we consider the following key variables:

- **Completion granularity:** The primary variable under consideration is the completion granularity—function-level and line-level. This is expected to influence both energy consumption and accuracy of generated code.
- **Model Size:** The number of parameters in the model, expected to both influence energy consumption and accuracy of generated code.
- **Energy Consumption:** A central focus of the study is understanding the energy consumption of the models during code generation, which is directly proportional to carbon emissions.
- **Test Accuracy:** Determines how well the generated code solves the problem as defined in the BigCodeBench dataset.

In this experiment, we compare the energy consumption and accuracy of code generation by testing both models with both completion granularities. A concise overview of the experimental process can be found in Figure 6. In this figure, orange blocks represent the inputs used in the experiment that were not implemented by us. The blue blocks correspond to the experimental steps, which generally align with the sections outlined here. Finally, the green blocks indicate the intermediate or final results of our experiments, which are used to address the research questions outlined in Section 4.1.

The experiment begins with dataset preprocessing, as detailed in Section 4.3. Next, we create function-level and line-level prompts, as outlined in Section 4.4. Using the models described in Section 4.2 and the energy measurement tool from Section 4.6, we proceed with the problem processing procedure detailed in Section 4.5.2. This results in function-level and line-level generations, along with their respective energy and time metrics. These metrics are then used for syntax and test validation, as described in Section 4.5.4, ultimately yielding accuracy metrics to assess the quality of the generated code.

4.1 Research Questions

In the following paragraphs, we describe the research questions to explore the energy consumption of different completion granularities—line-level and function-level completions. We will quantify energy consumption by measuring the CPU energy consumption during code completion to understand their respective efficiencies. We seek to determine the optimal completion granularity for minimising energy consumption while maintaining effective code generation.

RQ1: ***How does energy consumption compare between line-level and function-level completions?***

Given the distinct characteristics of line-level and function-level code completion, this study anticipates several key observations. Highlighting the differences between completion granularities can guide us to how we should utilise these new technologies, and possibly, uncover a code generation method that can improve developer efficiency. Specifically, this research will focus on the quantitative impact of excess token generation and the influence of inaccurate suggestions on energy consumption.

- ***What is the quantitative impact of excess token generation on energy consumption?***
- ***What is the quantitative impact of incorrect suggestions on energy consumption?***

During token generation, a model operates under two constraints: termination upon encountering a stop token sequence, or until a token limit is reached. Given the inherent simplified stop token sequence

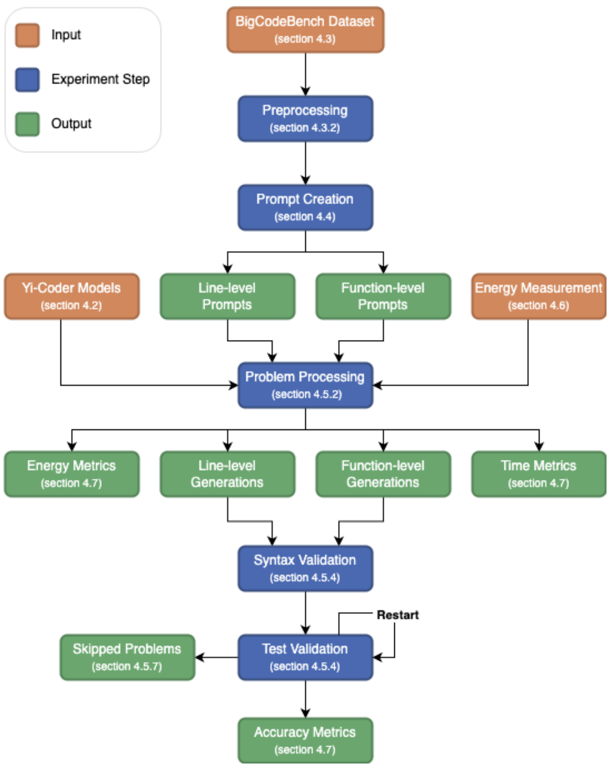
Input
Experiment Step
Output
BigCodeBench Dataset (section 4.3)
Preprocessing (section 4.3.2)
Prompt Creation (section 4.4)
Yi-Coder Models (section 4.2)
Line-level Prompts
Function-level Prompts
Energy Measurement (section 4.6)
Problem Processing (section 4.5.2)
Energy Metrics (section 4.7)
Line-level Generations
Function-level Generations
Time Metrics (section 4.7)
Syntax Validation (section 4.5.4)
Restart
Skipped Problems (section 4.5.7)
Test Validation (section 4.5.4)
Accuracy Metrics (section 4.7)

Figure 6: Overview of the Experiment Process

in line-level completions, we hypothesise a relative improvement in token effiency compared to function-level completions. Conversely, assessing the impact of incorrect suggestions is more challenging as a direct comparison between single-line suggestions and complete function implementations is infeasible. While line-level completions may exhibit higher token efficiency, this can be offset by the increased number of completions required compared to function-level completions.

RQ2: ***What reduction in carbon emissions can be achieved by substituting a smaller model for a larger model, without compromising accuracy?***

Building upon the comparitive analysis of line-level and function-level completions, we will investigate the potential energy efficiency increase possible through decreasing the model size. Specifically, by comparing function-level completions generated by a large model with line-level completions generated by a smaller model, we aim to uncover gains in energy efficiency impossible with improved generation techniques due to the computational limitations of model size. We hypothesise that, for a subset of problems, we can utilise a small model with line-level completions instead of a large model with function-level completions to attain significant energy efficiency improvements.

- ***How do energy per token and time per token compare?***

A comprehensive understanding of per-token characteristics is essential for evaluating the energy efficiency differences between completion granularities and model sizes. These statistics may prove vital in determining the best utilisation of different models and completion granularities to improve performance, efficiency, and minimise energy consumption.

4.2 Model Overview

In the experiment, we utilise the base models in the Yi-Coder family of models [6], with 1.5B and 9B parameters, respectively. The models build upon the base Yi models with an additional 2.4 trillion high-quality tokens, sourced from a repository-level code corpus on Github and code-related data filtered from CommonCrawl. This dataset contained training data for 52 major programming languages.

To lower the memory requirements and inference time of the models, we employed quantisation techniques, specifically reducing model precision to 4 bits. This approach lowers the memory requirements and speeds up inference. Additionally, we utilised the GGUF (General GPU Universal Format) to enable efficient model loading and cross-platform compatibility. When we decided on the Yi-Coder family of models, their performance for multiple benchmarks was better comparitive to others, similarly sized models. Table 2 lists the performance for the HumanEval, MBPP, CRUXEval-O and CrossCodeEval benchmarks.

Model	**HumanEval**		**MBPP (3-shot)**	**CRUXEval-O**	**CrossCodeEval**	
	Python	**Multilingual**			**Python**	**Java**
Yi-Coder (1.5B)	41.5	33.6	51.6	31.5	16.2	13.9
Yi-Coder (9B)	53.7	49.6	69.4	52.3	21.6	20.0

Table 2: Performance of Yi-Coder models on the HumanEval, MBPP, CrossCodeEval, and CRUXEval-O benchmarks.

4.3 Dataset Overview

To facilitate the experiments to measure energy consumption between line-level and function-level completions, we utilise the BigCodeBench benchmark [125]. This benchmark comprises 1,140 Python programming problems, each containing a function description (docstring), reference solution, and test suite. Whereas most code generation benchmarks focus solely on short and self-contained algorithmic tasks, the problems from the BigCodeBench benchmark utilise diverse function calls and necessitate combining multiple tools to solve them. This increased complexity ensures these problems demand compositional reasoning, requiring models to interpret the function descriptions and combine complex instructions [125]. In total, 139 libraries across 7 domains are used within the dataset, see Table 3 for an overview of the domains, including example libraries and function calls.

Domain		**Library**	**Function Calls**
Computation	(63%)	`numpy`, `pandas`, `sklearn`	`numpy.array`, `pandas.DataFrame`
General	(44%)	`random`, `re`, `collections`	`random.seed`, `re.search`
Visualisation	(31%)	`matplotlib`, `seaborn`	`matplotlib.pyplot`, `seaborn.histplot`
System	(30%)	`os`, `json`, `csv`	`os.path`, `json.loads`, `csv.writer`
Time	(10%)	`datetime`, `time`	`datetime.datetime`, `time.time`
Network	(8%)	`requests`, `bs4`, `django`	`requests.get`, `bs4.Tag`
Cryptography	(5%)	`hashlib`, `base64`, `rsa`	`hashlib.md5`

Table 3: Domains in the BigCodeBench benchmark, showcasing example libraries, function calls, and the percentage of domain usage within the dataset's problems.

Additionally, Figure 7 illustrates the distribution of the lengths of the reference solutions in the BigCodeBench dataset. The observed distribution aligns closely with a log-normal distribution, characterised by the property that the logarithm of the solution lengths follows a normal distribution. Fitting the data to this distribution yields parameters $\sigma \approx 0.496$ and $\mu \approx \ln 8.455 \approx 2.134$.

4.3.1 Structure

Each entry comprises 9 fields, for which we explain the purpose. Firstly, the model contains a unique identifier and the function name—conveniently `task_func` for each entry.

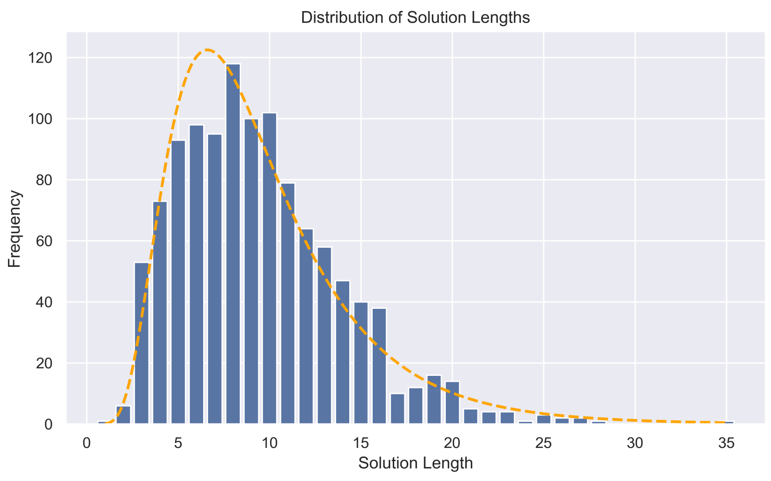

Figure 7: Distribution of reference solution lengths in the BigCodeBench dataset.

Furthermore, it contains a complete prompt and an instruction-tuned prompt for each problem. The complete prompt is defined as a docstring comprising all necessary information, whereas the instruction-tuned prompt is the same content formatted as a natural language prompt. As the models used were mainly trained on code and not natural language, only the complete prompts were used.

To highlight the features of each problem in the dataset, we examine one of the problems, as shown in Listing 2. Firstly, it contains the necessary imports for the code to execute, followed by the function definition and docstring. The docstring can have up to six parts to accurately describe the expected functionality of the function. These parts are in order:

- **Description**: concise, high-level description of the problem.
- **Parameters**: input parameters of the function, including types and concise description.
- **Returns**: return value of the function, including type and concise description.
- **Requirements**: libraries and functions required for execution.
- **Raises**: expected errors, including when they should be raised.
- **Examples**: list of examples, including inputs and expected outputs.

Besides the function definition and the docstring, each problem contains a reference solution. The solution is written by a human in collaboration with an LLM, which is iteratively prompted to refactor the implementation. To increase the correctness of the solutions, they go through a human curation pipeline that examines the code, investigates the test results and cross-checks the implementation with the docstring.

```python
import math
import numpy as np

def task_func(L):
  """
  Calculate the median of all elements
  in a nested list 'L'.

  Parameters:
  - L (list): The nested list.

  Returns:
  - median (float): The median.

  Requirements:
  - math
  - numpy

  Examples:
  >>> task_func([[1,2,3], [4,5,6]])
  3.5
  """
  def flatten(lst):
    flat = []
    for item in lst:
      if isinstance(item, list):
        flat.extend(flatten(item))
      else:
        flat.append(item)
    return flat

  flat = flatten(L)
  n = len(flat)

  sorted_flat = np.sort(flat)

  if n % 2 == 0:
    index1 = math.ceil(n / 2) - 1
    index2 = index1 + 1
    median = (sorted_flat[index1] + sorted_flat[index2]) / 2.0
  else:
    index = math.ceil(n / 2) - 1
    median = sorted_flat[index]

  return median
```

Listing 2: Example problem from the BigCodeBench benchmark, including a function description, requirements, and a Python solution.

To assess the correctness of the generated implementations, each problem contains also a test suite. For the currently highlighted problem, the test suite is in Listing 3. The inclusion of test suites for each problem ensures that the generated solutions can be objectively validated for correctness. It utilises the `unittest` library to handle test execution, which only needs a class that inherits from `unittest.TestCase` and contains a set of test cases, represented as methods. To execute the tests, the `unittest.main` function is called.

To increase the dataset usability and flexibility, it also contains separate fields for the code prompt—the complete prompt without docstring, the docstring only and the required libraries.

```python
import unittest
import numpy as np

class TestCases(unittest.TestCase):
  def test_median_single_element(self):
    result = task_func([[5]])
    self.assertEqual(result, 5.0)

  ...

  def test_median_empty_list(self):
    with self.assertRaises(ValueError):
      task_func([])

if __name__ == '__main__':
  unittest.main(failfast=True)
```

Listing 3: Example of a test suite for a problem from the BigCodeBench benchmark.

4.3.2 Preprocessing

An investigation into the dataset found inconsistencies in a small subset of the problems, and to ensure none of these influenced the final result, they were fixed. In no particular order, the following inconsistencies were fixed:

- Changed 33 single-quote docstrings (`'''`) to double-quote docstrings (`"""`).
- Added 12 missing newlines after docstring quotes.
- Removed 9 commented code lines and import statements from reference solutions.
- Changed 6 `Args:` to `Parameters:` within docstrings.

4.3.3 Limitations

We identified several significant issues with the dataset that were not feasible to address within the scope of this research. First, there were discrepancies between the import statements used in the reference solutions and the list of requirements in the docstrings. Zhuo et al. [125] provides an example of how import statements should be referenced within the docstring, including two types: `from xxx import yyy.zzz` and `import xxx`, which are referenced as `xxx.yyy.zzz` and `xxx`, respectively. Using regular expressions, we found 86 problems with incorrect references for import statements and 410 problems with incorrect references for from-import statements. Additionally, 66 problems raised an exception in the reference solution without mentioning it in the docstring.

4.4 Prompt Creation

The first step in comparing the energy consumption of LLMs between line-level and function-level code generation is the creation of prompts as input for the models. The BigCodeBench benchmark contains function-level prompts, and the reference solutions for each problem serve as a basis for line-level prompts with some processing, as highlighted later in the section.

4.4.1 Function-level Prompts

For function-level completion, the prompt can be extracted directly from the BigCodeBench benchmark as it contains a complete prompt field consisting of the imports, function definition, and a docstring.

4.4.2 Line-level Prompts

In contrast to creating the function-level promptd, creating a list of prompts for line-level completions is more complex. To create line-level completion prompts, we cannot utilise solely the complete prompt—we need context about what to solve in each line. To get a set of lines from which we can establish context, we utilise the reference solution for each problem. As we know this solution is correct, it can form the basis for the line-level completion prompts. We first split the solution into individual lines, which we will process into prompts.

Various line types are skipped during code completion to ensure consistent output. Empty lines are skipped as they provide no additional functionality. Function definitions are excluded because they must adhere to a specific structure expected in the remaining reference solution. Finally, control flow statements like `else:`, `try:`, `except:` and `finally:` are skipped as they are already complete.

The first task is to determine from what character the line generation should start. A rudimentary approach is to remove all non-whitespace characters and only keep the indentation, however, this increases the number of possible solutions exponentially. Given we know what biased information (e.g. variable names) are used in the reference solution, it only makes sense to include some context on what we expect—something a programmer would inherently do. For this, we utilise two strategies; select characters until the first trigger point or until the last variable declaration.

Trigger points are reserved keywords, operators and delimiters; which serve as an indication of the intention of the line. Listing 4 contains the list of trigger points used in the line-level prompt creation process, which took inspiration from Izadi et al. [55], who utilised trigger points to determine when to start providing suggestions generated by an LLM in a code editor extension.

```
assert raise del lambda yield return while for if else elif
global in and not or is with except . + += - -= * / % **
<< >> & | ^ = == != < > <= >= : , ( { ~ @
```

Listing 4: List of trigger points used for line-level prompt creation.

The only exception when using trigger points happens with the `for`-loop. As it contains both a trigger point and one or more variable declarations, it is important to only start generating after the last variable declaration, as the `for` keyword is situated before them. Since we utilise a reference solution, every variable declared is used later within the solution. Because of that, we need to ensure that the model does not need to generate the variable name, as a different variable name does not necessarily imply that the implementation itself is wrong. See Listing 5 for an example of a reference solution, and for each line where it would be split based on the trigger points.

```python
def task_func(input_string):
  result = ""
  lines = input_string.split('\n')
  for line in lines:
    result += line
  return result

def task_func(input_string):
  result =
  lines =
  for line
    result +=
  return
```

Listing 5: Example of a reference solution and the corresponding line-level prompts.

To create a prompt for a given line, we first split the line based on the trigger points, as previously mentioned. Then to create the context, we prepend the reference solution until the current line to the initial prompt. Listing 6 shows an example of a solution, including which line is selected, and the prompt for that line.

```python
def task_func(input_string):
  result = ""
  lines = input_string.split('\n')
  for line in lines:
    result += line
  return result

def task_func(input_string):
  result = ""
  lines = input_string.split('\n')
  for line
```

Listing 6: Example of a reference solution and the corresponding line-breaks based on the trigger points.

During each line split, we additionally store the line and character index for the prompt, such that we can substitute the generated solution in the reference solution when testing the generated solutions. Thus, when aggregating all steps of transforming a reference solution into a set of prompts, we arrive at the structure as shown in Listing 7 in YAML format.

4.5 Experimental Design

This section outlines the experimental design employed to evaluate the performance, energy consumption, and correctness of code generation with different granularities, detailing the setup, procedures, and rationale behind key methodological decisions.

4.5.1 Experimental Setup

The first step is to load the models into memory. Both models are 4-bit quantised versions of the respective base models in GGUF format, conserving memory and removing the necessity of utilising GPUs for inference. Inference happens through the `llama.cpp` library [37], which enables inference with minimal setup and state-of-the-art performance on a wide variety of hardware. Both models are loaded with

```yaml
Problem:
  id: string
  reference:
    code: string
    complete_code: string
    test_code: string
  func_prompt:
    prompt: string
    line_index: int
    char_index: int
  line_prompts:
    - prompt: string
      line_index: int
      char_index: int
```

Listing 7: Data structure of the prompts extracted from a problem in the BigCodeBench benchmark.

a context window size of 131072, 8 inference threads, a temperature of 0.25 and a token limit of 512 tokens.

Subsequently, the problems are loaded, which were processed as described in Section 4.4. Following that, the problems are shuffled randomly with the random seed `20891019142112125`. Finally, the EnergiBridge runner is instantiated and the experiment is ready to commence.

As energy consumption is highly affected by the hardware temperature, we run a CPU-intensive task before processing the problems. We calculate the Fibonacci sequence for five minutes. Afterwards, we process each problem in a stack-based order, determined by the shuffling during the experiment setup. In the following part, we explain all the steps of processing one problem.

The experiment hardware is a MacBook Pro M1 (14-inch, 2021) with a 10-core CPU and a 16-core GPU, 16GB of RAM and support for 8 threads. Subsequently, we use a batch size of 512 for data processing.

4.5.2 Problem Processing Procedure

In the next part, we explain what the procedure entails for processing one problem of the BigCodeBench dataset. Each problem contains a function-level prompt and multiple line-level prompts, for which we will generate solutions. To account for the variance of the energy measurements, we repeat the procedure multiple times ($N = 30$, which we elaborate on later).

Initially, we need to determine the current progress of the problem. All generation results are stored in a `.jsonl` file per granularity and model size, and this allows us to determine the progress by counting the number of lines within the file. For all four granularity–model size combinations, we prioritise processing the methods with the fewest results. Consequently, if for each combination the number of lines is equal to N, we can conclude the problem is finished and start the next problem. Additionally, using the file structure to track progress allows us to resume the experiment from where it left off if any issues arise.

When generating tokens with `llama.cpp`, it parses, evaluates and caches the prompt. Thus, we evaluate the function-level prompt and reference solution before generating with both models to eliminate the increased energy usage of the first generation with either model.

For each repetition, we shuffle the four combinations to reduce the bias of external conditions between executions. The code generation differs between function-level prompts and line-level prompts, so we cover these separately. But the general process of measuring the energy consumption is starting the EnergiBridge runner, performing code generation and then stopping the EnergiBridge runner to collect the energy and time measurements. For a more in-depth explanation of the energy measurement process, see Section 4.6.

For each function-level prompt, we only have one code block to generate, however, knowing when to stop generating is not trivial. Language models generate tokens for a set amount of tokens or until a stop token sequence. In a line-level prompt, the stop token is the `\n` character, indicating the end of a line. However, there is no definite stop token sequence that indicates the end of a function implementation. A rudimentary approach would consider `\n\n`, however, an empty line in a generated solution would also trigger this stopping condition—something we do not want.

To determine appropriate stop token sequences, we generated 100 solutions from randomly sampled function-level prompts in the dataset, with both the small and large models. Based on a manual inspection of the excessive token generation within these solutions, we arrived at the following stop token sequences: `\n\ndef`, `\n\nif`, `\n\nprint` and `\n\n#`. This covers function definitions, comments, print-statements and the main guard. Additionally, we use the default stop token sequence of the models: `<|endoftext|>`.

In contrast to the single function-level prompt, we have multiple line-level prompts per problem—one for each line. The process is similar to the function-level generation, with some slight differences. Firstly, the stop token in line-level generation is `\n`, alongside the `<|endoftext|>` token. Secondly, we combine the energy measurement of all line-level generations, because energy measurements with EnergiBridge become more reliable with increased sample time, and line-level completions can take insufficient time if only a small number of tokens is generated.

For each code generation—function-level and line-level—we store the generated text, including the individual tokens and their corresponding log probabilities. Additionally, we store the finish reason, which is either *stop* if the generation was terminated by the stop token sequence, or *finish* if the maximum number of tokens was generated.

```
GenerationResult:
  outputs:
    - text: string
      tokens: list[string]
      logprobs: list[float]
      finish_reason: string
  metrics:
    - energy: float
      energy_per_token: float
      time: float
      time_per_token: float
```

Listing 8: Data structure of the generation result.

We store the gathered information in the data format as shown in Listing 8. This generation result will be appended to the end of the results file of the current combination of prompt type and model size.

4.5.3 Rationale for Repetition Count

Due to the probabilistic nature of token generation with LLMs, we need to repeat the experiment multiple times to ensure we capture the variability in possible answers. Subsequently, as energy usage fluctuates over time based on external factors, we similarly need to repeat energy measurements and aggregate the results. In the following paragraphs, we first highlight the minimum repetitions for token generation, followed by the explanation for repeated energy measurements.

The best-case scenario would be that we could generate an infinite amount of line- and function-level completions, however, due to the limitations of reality this is deemed infeasible. Conversely, the best-case scenario would be a deterministic output, such that it always generates the correct answer—something also impossible for now. However, to determine how much variability exists within line-level generations, we composed a small experiment with a subset of the problems. The goal of the experiment was to see how many unique line-level generations were generated for different prompts.

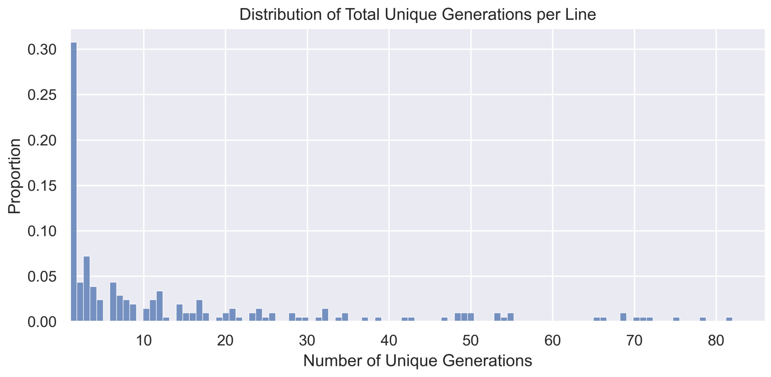

Figure 8: Distribution of the number of unique line-level generations per line.

From the dataset, we sampled eight problems comprising roughly 200 prompts. For each line, we utilised the prompts as described previously and performed 100 line-level completions. Then for each list of completions, we determine how many of them are unique. In Figure 8, one can see the distribution of the number of unique line-level completions, aggregated on a per-line basis.

Upon investigating the distribution, we can see a steep peak at the beginning—indicating the model does not deviate, or only slightly, in most line-level completions from a small set of unique completions. The distribution seems to follow a power-law distribution, as described in Equation 2. Utilising the `scipy` library, we compute the constants a and b such that they best fit the data. We arrive at $a = 0.64784$ and $b = -1.8842$.

$$f(x) = a \cdot x^b \tag{2}$$

Multiple different options can be correct for a given line, however, we want to capture as much variability between completions without generating too many suggestions. To ensure we capture the most variability of unique solutions, we aim to find the number of unique generations most lines end up with. In our case, we deem 95% to be enough to get a statistically sound sample of the unique lines that could be generated.

To arrive at 95% of the unique solutions, we first need to calculate the area of the complete curve and of a partial curve, as shown in Equation 3. Since we know the minimum and maximum possible unique generations—1 and 100—we can utilise these values as x_{min} and x_{max}.

$$\begin{aligned} A_{\text{total}} &= \int_{x_{\text{min}}}^{x_{\text{max}}} a \cdot x^b dx \\ A_{\text{partial}} &= \int_{x_{\text{min}}}^{x'} a \cdot x^b dx \end{aligned} \tag{3}$$

As we want 95% of the variability, we need to set the A_{total} to $0.95 \cdot A_{\text{partial}}$, and solve for x', as highlighted in Equation 4. Solving the equations gives $x' = 21.55986\ldots \approx 22$. Thus, the minimum number of generations per line needs to be at least 22 to get the expected variability.

$$\begin{gathered} A_{\text{total}} = 0.95 \cdot A_{\text{partial}} \\ \int_{1}^{100} a \cdot x^b dx = 0.95 \int_{1}^{x'} a \cdot x^b dx \end{gathered} \tag{4}$$

The second part involves the needed repetitions for reliable energy measurements. No exact or correct number of repetitions exists for energy measurement, as it depends on the objective of the research, available resources and environmental constraints. However, other researchers suggest 30 repetitions, as this is often enough for the data to be normally distributed [49, 72]. To ensure we capture the most variability and allow for a big enough sample size such that the data can be normally distributed, we opt for a repetition count of $N = 30$.

4.5.4 Test Setup and Verification

In the following paragraphs, we explain the setup to verify if the function-level and line-level generations are correct. We first verified syntax, then compiled and tested each solution to quickly discard incorrect ones. Subsequently, we describe how we test the solutions to arrive at an expected pass rate measure.

Let $\mathcal{P} = \{P_0, P_1, \ldots, P_{M-1}\}$ be a set of M problems, where $M = 1,140$ (the size of the BigCodeBench dataset). Each problem $P_i \in \mathcal{P}$ has an associated test suite T_i. For each problem P_i, we have a set of $N = 30$ function-level generations, denoted by $G_i^{\mathcal{F}} = \{G_{i0}^{\mathcal{F}}, G_{i1}^{\mathcal{F}}, \ldots, G_{iN-1}^{\mathcal{F}}\}$. Furthermore, for each problem P_i and each line j within that problem, we have a set of $N = 30$ line-level generations, denoted by $G_{ij}^{\mathcal{L}} = \{G_{ij0}^{\mathcal{L}}, G_{ij1}^{\mathcal{L}}, \ldots, G_{ijN-1}^{\mathcal{L}}\}$.

The first step is to verify syntactical correctness for the function- and line-level generations utilising the `ast` module. For function-level generations, we select the function implementation, discard excess tokens and generate the abstract syntax tree. For line-level generations, we substitute the generated line in the reference solution and verify the correct syntax. The function- and line-level generations that did not generate a correct abstract syntax tree, we add to $G_{\text{fail}}^{\mathcal{F}} \subseteq G^{\mathcal{F}}$ and $G_{\text{fail}}^{\mathcal{L}} \subseteq G^{\mathcal{L}}$; the sets of incorrect generations for each generation type.

New solutions are created by replacing the function implementation in the reference solution, where $S_i^{\mathcal{F}}$ is a function-level solution for problem P_i. With function-level generations, it is trivial to determine the pass rate for a specific problem. For a given solution $S_i^{\mathcal{F}}$, we only need to assess whether it passes the associated test suite T_i, as highlighted in Equation 5.

$$\text{pass}_{\mathcal{F}}(S_i^{\mathcal{F}}) = \begin{cases} 1 & S_i^{\mathcal{F}} \text{ passes } T_i \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

To arrive at a pass rate for a specific problem, we calculate it with the equation shown in Equation 6.

$$\text{PassRate}_{\mathcal{F}}(P_i) = \frac{1}{N} \sum_{n=1}^{N} \text{pass}_{\mathcal{F}}(S_{in}^{\mathcal{F}}) \tag{6}$$

However, for line-level generations, calculating a pass rate for an individual line is significantly harder. With function-level generations, we know exactly the amount of solutions we need to test—30 per problem. Conversely, with line-level generations we have 30 generations per line, and as these generations are independent of one another, the total number of unique solutions scales exponentially in the length of the reference solution.

For line-level solutions, we can substitute anywhere from 1 line to every line in the reference solution, arriving at $S_i^{\mathcal{L}} = \{S_{i0}^{\mathcal{L}}, S_{i1}^{\mathcal{L}}, \ldots, S_{ij}^{\mathcal{L}}\}$, with

$$S_{ij}^{\mathcal{L}} = \begin{cases} 1 & \text{if line } j \text{ is substituted with a} \\ & \text{generation from } G_{ij}^{\mathcal{L}} \setminus G_{\text{fail}}^{\mathcal{L}} \\ 0 & \text{otherwise} \end{cases}$$

The average length of a reference solution is 10 lines long, giving an upper limit of 30^{10} possible solutions—an impossibly large number of solutions to check. Instead, we calculate an expected pass rate with a Monte Carlo simulation (MCS). MCS relies on repeated random sampling to obtain a numerical result [77]. We determine for a subset of the solutions if they pass the test suite and with those results calculate the expected test pass rate.

During MCS, $S_i^{\mathcal{L}(k)}$ denotes a solution with $1 \leq |L^{(k)}| \leq |S_i|$ lines substituted, where $\mathcal{L}^{(k)} \subseteq \{1, 2, \ldots, |S_i|\}$ is the substituted lines in the current MCS iteration k. To determine if a line-level solution is correct, we utilise Equation 7.

$$\text{pass}_{\mathcal{L}}(S_i^{\mathcal{L}(k)}) = \begin{cases} 1 & S_i^{\mathcal{L}(k)} \text{ passes } T_i \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Utilising the pass function, we can calculate the expected pass rate for each line with the MCS, as described in Equation 8. We perform K iterations where we sample a random number of lines to substitute with a generated line to create a new solution, which we verify against a test suite for correctness. We opt for a sufficiently large value $K = 10000$.

$$\mathbb{E}[\text{PassRate}_{\mathcal{L}}(P_i)] = \frac{1}{K} \sum_{k=1}^{K} \text{pass}_{\mathcal{L}}(S_i^{\mathcal{L}(k)}) \tag{8}$$

Furthermore, we can utilise the expected pass rate for a given line to calculate the expected pass rate of a certain fraction $0 \leq \varphi \leq 1$ of lines, as highlighted in Equation 9. $\mathcal{L}_\varphi \subseteq \{1, 2, \ldots, |S_i|\}$ is the set of randomly chosen lines to substitute in iteration k for problem P_i, with $|L_\varphi| = \lceil \varphi \cdot |S_i| \rceil$.

$$\mathbb{E}[\text{PassRate}_{\Phi}(P_i, \varphi)] = \frac{1}{|\mathcal{L}_\varphi|} \sum_{j \in \mathcal{L}_\varphi} \mathbb{E}[\text{PassRate}_{\mathcal{L}}(P_i)] \tag{9}$$

With the expected pass rate for a problem and a certain fraction of lines substituted, we can aggregate this for all problems in the dataset to get an expected pass rate per fraction, as shown in Equation 10.

$$\mathbb{E}[\text{PassRate}_{\Phi}(\varphi)] = \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}[\text{PassRate}_{\Phi}(P_i, \varphi)] \tag{10}$$

After performing the substitution of line-level generations in the reference solution, and running it against a test suite, we store it in `.jsonl` files per problem. The data structure is shown in Listing 9.

```
MonteCarloSimulationResult:
  selected_lines: list[int]
  result:
    - code: string
      compilation_passed: boolean
      tests_passed: boolean
      error: optional[string]
      time: float
```

Listing 9: Data structure of the Monte Carlo simulation result.

4.5.5 Quantifying Impact of Excess Tokens

During token generation, albeit line-level or function-level completions, not all tokens are necessary for the correct functionality of the code, and thus is it important to determine the efficiency of token generation. In the next paragraphs, we highlight three types of excess tokens that can be generated, including examples.

Firstly, the most obvious type of excess token generation is subsequent tokens following the function implementation. As the goal is to solve a given problem based on a function description, any tokens that are generated after the implementation can be considered excess tokens. Secondly, we regard all

```python
def add_numbers(a, b):
  # Add both numbers
  result = a + b
  print('Result is', result)
  return result

if __name__ == '__main__':
  assert add_numbers(1, 2) == 3
```

Listing 10: Example of excess tokens, including comments, print statements, and post-implementation code.

comments as unnecessary tokens as they do not provide any additional benefit during our experiments and they do not impact functionality. Finally, we regard print- and log-statements as excess tokens as none of the problems depend on print-statements for a correct implementation.

Listing 10 is an example of the types of excess token generation, where the lines with excess tokens are highlighted. From top to bottom, the highlighted lines are: comment, print-statement, and post-implementation code. For each line- and function-level generation, we utilise Equation 11 to determine the proportion of excess tokens.

$$\rho_e = \frac{\tau_c + \tau_p + \tau_a}{\tau} \tag{11}$$

Where:

- ρ_e: Proportion of excess tokens
- τ_c: Tokens from comments
- τ_p: Tokens from print-statements
- τ_a: Tokens after function implementation
- τ: Total number of tokens

For the function in Listing 10, we can calculate the proportion of excess tokens. Let us assume the comment contains 5 tokens, the print-statement contains 10 tokens and 15 tokens after the function implementation. If the total number of generated tokens is 60, we arrive at a proportion of excess tokens of $(5 + 10 + 15)/60 = 0.5$.

4.5.6 Quantifying Impact of Incorrect Suggestions

Besides excess tokens during generation, we need to consider the possibility that the generated solutions are incorrect and the impact that has on energy consumption. To facilitate a fair comparison, we only compare function-level completions with line-level completions where each line in the reference solution is substituted. To determine the impact of incorrect suggestions on energy consumption, we utilise Equation 12.

$$E_{\text{wasted}} = \begin{cases} 0 & S \text{ passes } T \\ \tau \cdot \epsilon & \text{otherwise} \end{cases} \tag{12}$$

Where:

- E_{wasted}: Wasted energy
- S: Generated solution
- T: Test suite

- τ: Total number of tokens in S
- ϵ: Energy per token

If a generated solution S passes the test suite T, no energy is wasted. Otherwise, the wasted energy is proportional to the total number of tokens τ in S, multiplied by the energy cost per token ϵ.

4.5.7 Operational Considerations

In practice, we encountered two notable challenges during the testing process:

- **Memory Constraints**: Due to the large amount of executed test cases in combination with the interpreted nature of Python, eventually the memory usage of the process exceeds the available system memory. To combat this, we periodically check the memory usage, and if it exceeds the threshold of 99%, we restart the process.
- **Time Constraints**: Some implementations of reference solutions or unit tests are inefficient. Unit tests should execute quickly, aiming for 0.01 seconds with a maximum of 1 second in isolation [86]. Thus, we considered only the problems where unit test execution time did not exceed 1 second.

To determine which problems to skip, we executed each reference solution against its test suite multiple times, recorded the times it took to execute, and averaged them. In Figure 9, one can see the cumulative time it took to execute the tests for each problem, sorted by execution time. In total, executing all test suites took on average 480 seconds, with all the test suites that exceeded 1 second accounting for 445 seconds. Essentially, 9.3% of problems took 92.7% of the total execution time.

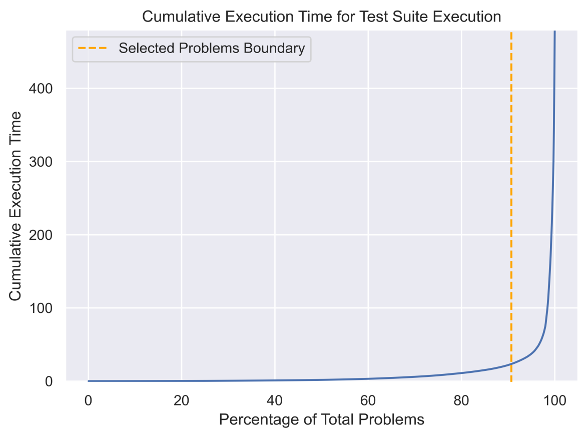

Figure 9: Percentage of test cases versus cumulative execution time, with the boundary highlighted for the selected problems.

Upon inspection of each skipped problem, we found common themes among the implementations or test suites that contributed towards the long execution time. The most prominent reasons were extensive plotting, file operations and delays (e.g. `time.sleep`). In Table 4, one can see what contributed most often towards a slower execution time. See Appendix A, Table 9 for the reasons each problem was skipped.

4.6 Energy Measurement

Energy measurement plays a crucial role in evaluating the efficiency of our experimental methods, enabling us to quantify the energy consumption during execution. To facilitate energy measurements, we utilized the EnergiBridge library [30], a Rust-based tool compatible with Linux, Mac OS, and Windows,

Reason	Amount
plotting	46
file operations	18
delays	14
train ML model	11
large input size	10
network operations	7
image manipulation	5
math operations	5
cryptography	3
compilation	1

Table 4: Breakdown of factors contributing to test suite slowness, ranked by frequency of occurrence.

```python
from pyEnergiBridge.api import EnergiBridgeRunner

runner = EnergiBridgeRunner()
runner.start()
# Measures energy for the code here
energy, time = runner.stop()
```

Listing 11: Example code of how to use the PyEnergiBridge wrapper.

and supports Intel, AMD, and Apple ARM CPU architectures. EnergiBridge collects CPU and GPU power usage in intervals for a specified time frame and returns, either a summary of energy consumption, or prints the results to a CSV file. We utilised an interval of 100 milliseconds.

Given that our experiments were implemented in Python, we used PyEnergiBridge [25], a Python wrapper for EnergiBridge. PyEnergiBridge provides an interface to initiate energy measurement processes, manage their execution, and retrieve results upon completion. This integration ensured seamless compatibility between our Python-based experiments and the measurement capabilities of EnergiBridge. See Listing 11 for an example of the PyEnergiBridge wrapper usage.

While the setup described above provides a solid foundation for the methodology, potential issues in energy measurements must be considered and mitigated before the experiment begins. Given the unreliable nature of energy measurements, we aimed for consistent settings throughout the experiment. These mitigations included closing all applications, terminating unnecessary background processes, disabling WiFi, and reducing screen brightness to its minimum. Additionally, as previously mentioned, we performed a five-minute CPU-intensive task before starting the experiment to decrease the variability of the measurements.

One issue with our approach is the tail energy consumption associated with energy measurements, for which the mitigation is a pause between measurements. The recommended wait time is approximately one minute. However, due to the large number of measurements and their significantly shorter duration—one second versus one minute—we decided not to include a pause between executions.

4.7 Evaluation Metrics

In this subsection, we provide a clear description of the evaluation metrics used to assess the models' performance in terms of energy consumption and code correctness.

4.7.1 Energy Efficiency

Energy efficiency is a critical measure of performance, particularly for large-scale models. To evaluate energy consumption, we utilise the EnergiBridge tool, which measures both the total energy used during the code generation process as well as the energy consumed per token generated. These metrics are defined as:

- **Total Energy**: The total energy consumed during the generation, measured in Joules (J).
- **Energy per Token**: The average energy consumed per generated token, measured in Joules (J).

4.7.2 Time Efficiency

In addition to energy, the time required for code generation is important for assessing efficiency. We measure the time taken to generate the function-level and line-level completions with the following key metrics:

- **Total Time**: The total time taken during the generation, measured in seconds (s).
- **Time per Token**: The average time taken per generated token, measured in seconds (s).

4.7.3 Correctness

Correctness is evaluated by testing the generated code against predefined test suites for each problem. For both function-level and line-level generations, correctness is assessed through the test pass rate, which is the percentage of generated solutions that pass the corresponding test suite. For an in-depth explanation of how the test pass rate for function-level and line-level completions is determined, see Section 4.5.4.

4.8 Summary

The methodology of this experiment is designed to compare the energy consumption of code generation using two LLMs from the Yi-Coder family with two different model sizes (1.5B and 9B), focusing on function-level and line-level code completions. Key variables include prompt type, model size, energy consumption, and test accuracy. The experiment uses the BigCodeBench dataset, comprising 1,140 Python programming problems. Code completion is evaluated using test suites, ensuring the correctness of the generated code.

For prompt creation, function-level prompts are derived directly from the BigCodeBench dataset, while line-level prompts require splitting the reference solution into individual lines, using trigger points and context to guide generation.

The models used are 4-bit quantised versions of the base models, loaded in GGUF format. This reduces memory usage and removes the need for GPUs. Inference is performed using the `llama.cpp` library. The models are set with a context window size of 131072, 8 inference threads, a temperature of 0.25, and a token limit of 512. The problems are processed after being shuffled randomly, with a CPU-intensive task (calculating the Fibonacci sequence) run beforehand to normalise hardware temperature. The experiment runs on a MacBook Pro M1 with a 10-core CPU, 16-core GPU, and 16GB of RAM, and processes data in batches of 512.

Each problem consists of a function-level and multiple line-level prompts, for which solutions are generated. The energy consumption and time are measured using EnergiBridge, and the Python wrapper PyEnergiBridge. For function-level prompts, one solution is generated, whereas line-level prompts generate one solution per line. The generation is repeated 30 times to account for variability. Tokens are parsed, evaluated, and cached during generation to eliminate energy bias from the first generation. The energy consumption is measured for each generation.

Repetition is necessary due to the probabilistic nature of token generation, which leads to variability in answers. To ensure reliable energy measurements, the experiment is repeated 30 times, which is standard practice to attain a normal distribution of energy data. Additionally, variability in line-level generations was studied by sampling 8 problems and performing 100 line-level completions per problem. From this, the number of unique line-level generations needed to achieve 95% of the possible variations was calculated to be 22 repetitions per line minimum.

To verify the correctness of the generated code, initial syntax checks are followed by compiling the solutions and running them against the test suite. Function-level generations are directly evaluated against the test suite, while line-level generations require a Monte Carlo simulation (MCS) to handle the large number of possible solutions generated by different combinations of line substitutions. The MCS iterates 10,000 times to estimate the expected pass rate for line-level generations.

Challenges include memory and time constraints. Memory usage is monitored, and processes are restarted if they exceed 99% of usage. Time constraints are addressed by only considering problems where test execution times do not exceed 1 second, as longer tests would consume disproportionate resources.

To evaluate model performance, we utilise energy consumption, time, and correctness as evaluation metrics. Key energy metrics include total energy and energy per token. Time efficiency is measured by total time and time per token. Correctness is assessed by testing generated code against predefined test suites, with pass rates calculated for both function-level and line-level generations, the latter using a Monte Carlo simulation.

5 Results

This section presents the results from the experiment designed to compare the energy consumption of code generation using Yi-Coder models. Utilising the BigCodeBench dataset we created prompts to perform line-level and function-level generations. Key variables considered include prompt type, model size, energy consumption, and test accuracy. The experiment, conducted over 17 days, resulted in 3.8 kg CO_2eq emissions, equivalent to running an LED light bulb for 500 hours [16].

We begin by summarising the characteristics of the collected data before addressing each research question individually. As summarised in Table 5, the small and large models differ notably in their compilation and test pass rates.

Model Size	Total Solutions	Compilation (Passed/Failed)	Tests (Passed/Failed)
Small — 1.5B	26596	25636 (96.4%) / 960 (3.6%)	21190 (82.7%) / 4446 (17.3%)
Large — 9B	26344	25587 (97.1%) / 757 (2.9%)	23145 (90.5%) / 2442 (9.5%)

Table 5: Performance statistics of line-level code generation for small (1.5B) and large (9B) models, including compilation and test pass/fail rates.

To assess the difference in quality between the line-level completions of the small and large models, each generation was compiled and tested individually. Table 5 presents the statistics for the compilation and test results. The small model exhibited a 3.6% compilation failure rate, slightly higher than the 2.9% of the large model. Among the successfully compiled generations, the small model had a 17.3% test case failure rate, considerably higher than the 9.5% failure rate observed for the large model. This suggests that the larger model produces higher-quality line-level solutions, outperforming the smaller model in compilation and test pass rates.

After analysing the compilation and test pass rates, we observed a difference in the log-probabilities of token generation across model sizes and completion granularities. For the small model, the average log-probability was -0.2 for line-level completions and -0.32 for function-level completions. For the large model, line-level completions had an average log-probability of -0.14, while function-level completions showed a value of -0.24. A value closer to 0 indicates a higher likelihood.

Additionally, the distribution of unique generations per line follows a similar distribution as highlighted in Section 4.5 as expected. Figure 10 shows the distributions, which both follow a power-law distribution.

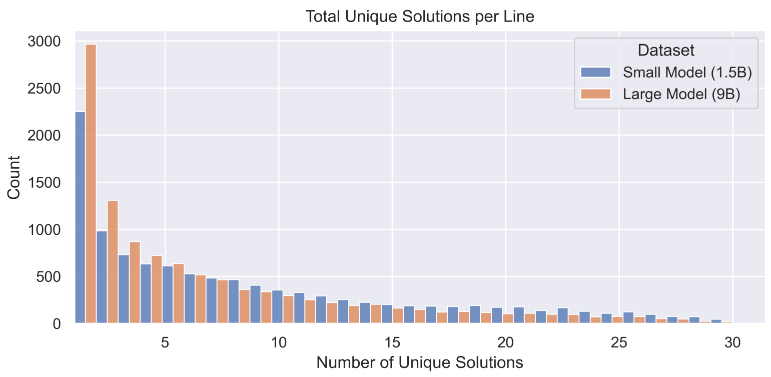

Figure 10: Distribution of unique generations per line for the small (1.5B) and large (9B) models.

5.1 Energy Consumption for Different Completion Granularities

In this subsection, we address the first research question, focused on the difference in energy consumption between line-level and function-level completions. Line-level completions generate tokens until a `\n` token, and function-level completions generate tokens until the end of a function implementation—a less precisely defined boundary. The different granularities might exhibit different characteristics regarding the total generated tokens, thus influencing energy consumption.

Research Question 1: How does energy consumption compare between line-level and function-level completions?

Table 6 contains an overview of the experiment statistics for each model size–granularity combination. For both model sizes, function-level completions generated significantly more tokens. The small model generated 42% more tokens, compared to a 51% increase with the large model. The total energy consumption is higher, but not as drastic, with an increase of 14% for both model sizes, respectively. Similarly, the total generation time is similar for both granularities, with an increase of 6% for the small model and 4% for the large model. When energy consumption is converted to CO_2eq emissions, the small model produced 478g and 417g CO_2eq, for function-level and line-level completions, respectively. The large model generated 1593g for function-level completions and 1393g CO_2eq for line-level completions.

Model Size	**Granularity**	**Tokens**	**Energy** (MJ)	**Time** (s)	**Carbon Emissions** (g CO_2eq)
Small — 1.5B	Function-level	5391274	6.410575	244299	478
Small — 1.5B	Line-level	3798085	5.592815	230859	417
Large — 9B	Function-level	5808864	21.370993	518476	1593
Large — 9B	Line-level	3848681	18.679031	499250	1393

Table 6: Results of experiments with energy, time, and carbon emissions statistics. Carbon emissions are calculated using the carbon intensity of the Netherlands in 2023 [32].

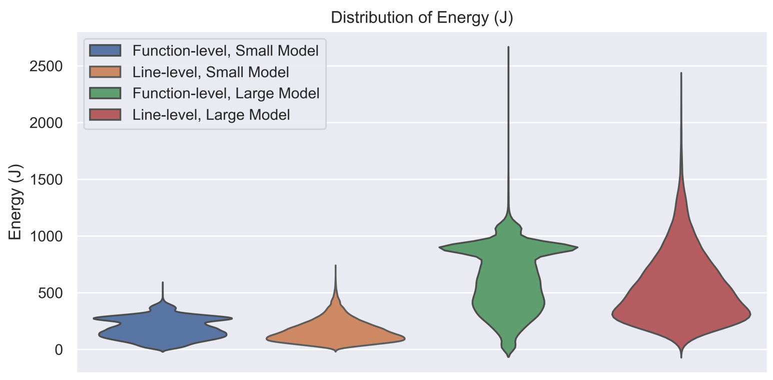

Figure 11: Distribution of total energy per generation for line-level and function-level completions across both model sizes.

To delve deeper into the energy consumption characteristics of the experiments, Figure 11 shows the distribution of the energy consumption per generation. Note that the line-level generations for one problem are grouped per iteration. For both model sizes, each granularity follows a similar distribution. The function-level distribution is roughly normal with a skew toward lower values. Secondary peaks appear at higher values, accompanied by a long tail, suggesting asymmetry and a possible bimodal distribution. The line-level distribution appears roughly normal, with a skew toward lower values and a pronounced long tail on the higher end, suggesting asymmetry for larger values.

Additionally, not all function-level completions terminated due to generating a stop token sequence. For the small model, 21% of the completions generated tokens until the token limit, compared to 25% with the large model. This might explain the secondary peaks in Figure 11 within the upper quartiles.

Subsequently, we can consider a cumulative distribution function of the normalised energy consumption per generation, as shown in Figure 12. This highlights the difference between function-level and line-level completions, where in line-level generations the average energy consumption is significantly less than the peak energy consumption. Roughly 80% of generations consumed less than 20% of the maximum with line-level completions, compared to 10% of the maximum with function-level completions. Similarly, 95% of line-level completions consumed less than 40% of the maximum, compared to only 40% of function-level completions.

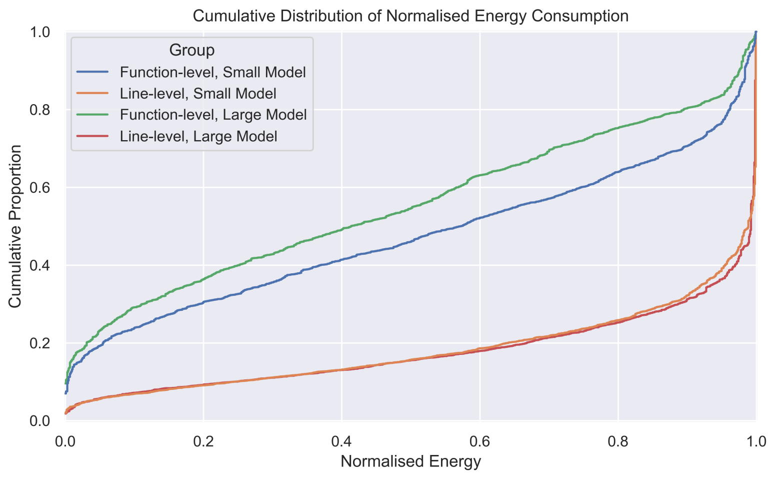

Figure 12: Cumulative distribution function of normalised energy consumption per generation, for line-level and function-level completions.

5.1.1 Quantitative Impact of Excess Token Generation

In the following paragraphs, we consider the impact of excess token generation on energy consumption. Excess token generation can be defined concisely as any tokens generated that do not influence the functionality of the code; e.g. tokens generated after the function implementation, comments and print-statements. For an in-depth explanation of excess token generation, see Section 4.5.

Research Question 1.1: What is the quantitative impact of excess token generation on energy consumption?

The excess token generation calculation results, as shown in Figure 13, show a large difference between the token efficiency of function-level and line-level completions. For function-level completions with the large model, the lower quartile exhibited a maximum excess token percentage of 12%, 17% for the median and 24% for the upper quartile. In comparison, the function-level generations with the small model had higher token efficiency. The lower quartile range started at 9%, the median at 13% with an upper quartile at 18%.

Comparatively, both models exhibit significantly higher token efficiencies when utilising the line-level granularity. For the large model, the lower quartile ranged from 0.3% to a median of 0.6%, with an upper quartile at 1.2%. Similarly, the small model exhibited high token efficiency with a lower quartile ranging from 0.3% to a median of 0.5%, and an upper quartile range of 0.9%.

To assess the statistical significance, we utilise either a T-test or a U-test. For a T-test, the data needs to be normally distributed, whereas the U-test is a nonparametric statistical test. We utilised the Shapiro–Wilk test [98] to test for normality. See Table 7 for an overview of the P-values associated with each model size and granularity. A P-value greater than $\alpha = 0.05$ indicates the data is most likely normally

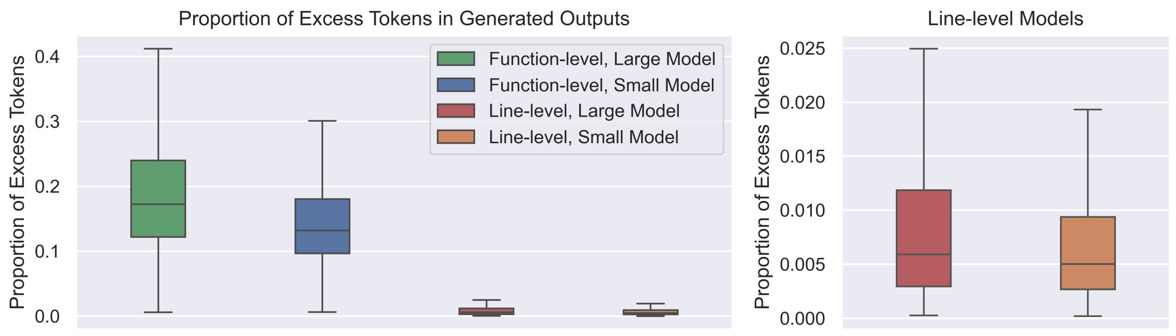

Figure 13: Proportion of excess tokens generated for different model sizes and granularities.

Model Size	**Granularity**	**P-value**
Small — 1.5B	Function-level	3×10^{-33}
Small — 1.5B	Line-level	2×10^{-28}
Large — 9B	Function-level	5×10^{-40}
Large — 9B	Line-level	9×10^{-40}

Table 7: Shapiro-Wilk test p-values for normality assessment of model size and granularity combinations in excess token calculation.

distributed, and as none of the values exceed that α, we cannot utilise the T-test to determine statistical significance.

After determining all energy consumption distributions were not normally distributed, we opted for the Mann–Whitney U test to assess if the differences between the distributions were statistically significant. The Mann–Whitney U test is a nonparametric statistical test that checks if, for two randomly selected values X and Y from two populations, the chance of X being greater than Y is the same as the chance of Y being greater than X [73]. It returns a P-value, where if that is smaller than 0.05, the difference is statistically significant.

To assess the statistical significance between the results, we calculate the P-value for each unique combination where either model size or granularity differs. The most significant differences exist with the comparison between line-level and function-level completions. For the small model, the P-value is 1×10^{-256}, and the large model exhibits a P-value of 2×10^{-263}.

For the difference between model sizes with the same granularity, the results are also statistically significant. The P-value for function-level completions is 3×10^{-27} and for line-level completions, it is 0.012. This indicates all differences are statistically significant, however, the largest differences exist in the differences between granularities.

5.1.2 Quantitative Impact of Incorrect Suggestions

In the following paragraphs, we show the impact of incorrect suggestions of different model sizes and granularities, regarding wasted resources during token generation. For an in-depth explanation of the methodology, see Section 4.5.

Research Question 1.2: What is the quantitative impact of incorrect suggestions on energy consumption?

Figure 14 shows the distribution of wasted energy consumption of incorrect solutions per problem. For function-level completions, this is an aggregation of the energy consumption per solution, whereas for line-level completions the solutions are taken from the Monte Carlo Simulation where each line is substituted in the reference solution.

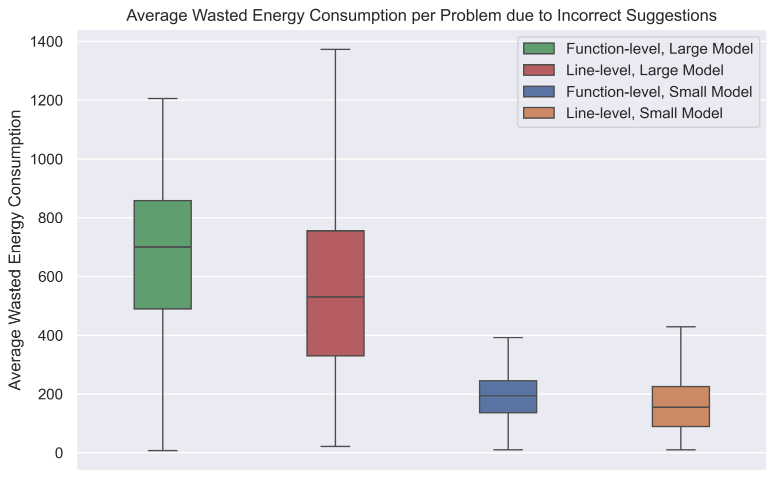

Figure 14: Wasted energy consumption due to incorrect suggestions for different model sizes and granularities.

For both model sizes, the function-level completions exhibit lower wasted energy due to an increased test pass accuracy. For the function-level completions with the large model, the lower quartile range starts at 480J with a median of 700J. For line-level completions, however, the lower quartile range sits at 340J with a median of 530J. Conversely, for the small model, the function-level completions have a lower quartile range of 130J with a median of 200J. Subsequently, the line-level completions exhibit a lower quartile range at 100J and a median of 160J.

Model Size	**Granularity**	**P-value**
Small — 1.5B	Function-level	4×10^{-4}
Small — 1.5B	Line-level	4×10^{-13}
Large — 9B	Function-level	1×10^{-25}
Large — 9B	Line-level	6×10^{-28}

Table 8: Shapiro-Wilk test p-values for normality assessment of model size and granularity combinations in wasted energy consumption due to incorrect suggestions.

Similarly, as with the excess token calculations, it is vital to determine the statistical significance between the observations. Table 8 shows the P-values for the Shapiro–Wilk normality tests. All P-values do not exceed the threshold of $\alpha = 0.05$, and thus, are all not normally distributed.

We employ the Mann–Whitney U test to assess if the differences between results are statistically significant. As the proportion of wasted energy consumption for incorrect suggestions is directly linked to test pass accuracy, we opt to only compare the different granularities with the same model. Comparing the line- and function-level completions with the small model had a P-value of 4×10^{-20}, whereas the large model had a P-value of 9×10^{-23}. Both sets of observations are statistically significant.

To summarise the findings for the first research question, function-level completions consume more energy than line-level completions. The lower bound of function-level completions is comparable to line-level completions, but function-level completions contain a secondary peak of higher energy consumption due to continued token generation until the token limit. Line-level completions exhibit significantly higher token efficiency—approximately 30× higher. Furthermore, we found no substantial difference in the impact of incorrect suggestions between function-level and line-level completions.

5.2 Carbon Emission Reductions Through Model Substitution

In this subsection, we address the second research question, focused on carbon emission reduction if we substitute a large model performing function-level completions with a small model performing line-level completions. Comparing both models involves calculating a test pass accuracy for the different granularities, including for different fractions of line-level completions substituted in the reference solution.

> Research Question 2: What reduction in carbon emissions can be achieved by substituting a smaller model for a larger model, without compromising accuracy?

Before we delve deeper into the carbon emission reductions, we highlight the test pass accuracy of the different models and granularities. Figure 15 compares function-level completions with line-level completions substituted in each line of the reference solution. Upon manual investigation, we identified a discrepancy in test pass accuracy of shorter and longer solutions.

We calculated the Spearman correlation coefficient ρ between the number of lines substituted and the test results. The Spearman correlation coefficient measures the strength of a monotonic relationship based on ranks, not values [100]. For the small model, $\rho = -0.76$ and for the large model, $\rho = -0.74$. For both model sizes, the correlation coefficients indicate there is a strong, but not perfect, inverse relationship between the number of substituted lines and the test pass accuracy.

Due to the discrepancy between short and long solutions, we opted to divide the reference solutions into two groups: those with a length less than 10 and those with a length greater than or equal to 10. Based on the distribution of reference solution lengths discussed in Section 4.3, this threshold provides an approximately even split.

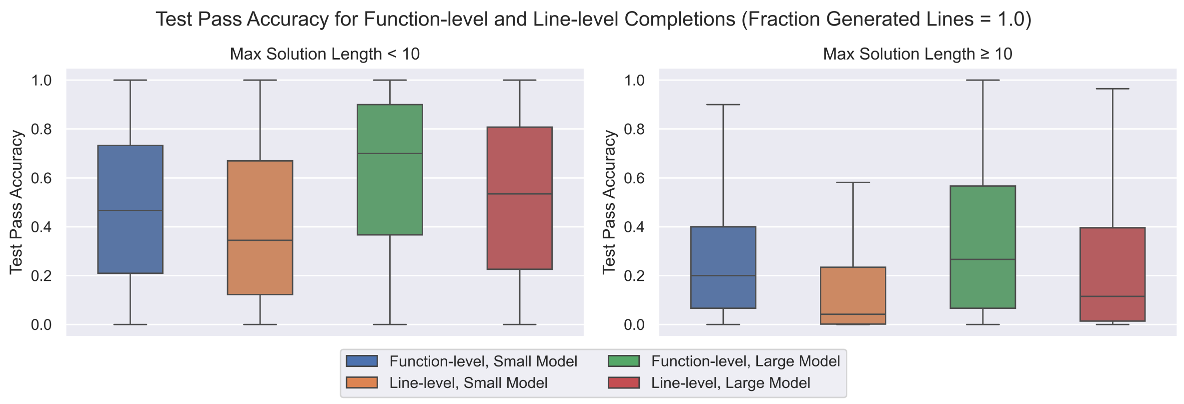

Figure 15: Comparison of test pass accuracy between function-level completions and multiple line-level completions, where each line in the reference solution is substituted.

For the problem with a short reference solution, the test pass accuracy is higher for the function-level completions for both model sizes. For the small model, the median of the function-level completions is 46%, compared to 35% with line-level completions. For the large model, the median of the function-level completions is 70%, whereas the median of the line-level completions is 54%. When considering the long solutions, for the small model, the median of the function-level completions is 20%, and 5% for line-level completions. For the large model, the median is 26% for function-level completions and 12% for line-level completions.

Figure 16 shows the test pass accuracy for each fraction of line-level completions. Overall, a downward trend in test pass accuracy is observed as the fraction of substituted lines increases, which aligns with our expectations. Both splits—one with a reference solution length of less than 10 and the other with more than 10—show that the large model consistently outperforms the small model.

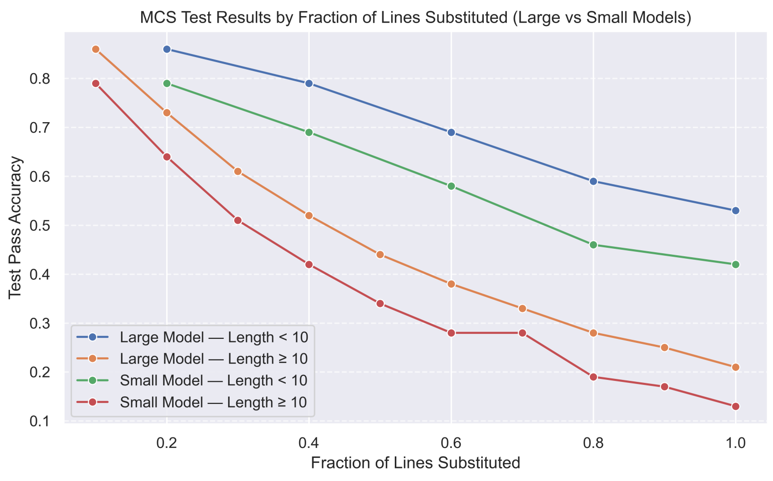

Figure 16: Test pass accuracy for different fractions of line-level completions for both model sizes, split by solutions with a maximum length of 10.

To facilitate a comparison between line-level completions and function-level completions, we needed to assess from which fraction of lines substituted within the reference solution line-level completions outperform function-level completions. Figure 17 shows at which fraction both models with line-level completions outperform the large model with function-level completion. A fraction of 1 indicates all lines can be substituted, and it still outperforms the function-level solutions. Conversely, a fraction of 0 indicates the line-level solutions never outperform the function-level solutions. Ideally, we want to be able to substitute each line with a line-level completion and still outperform the large model, thus values closer to 1 are better.

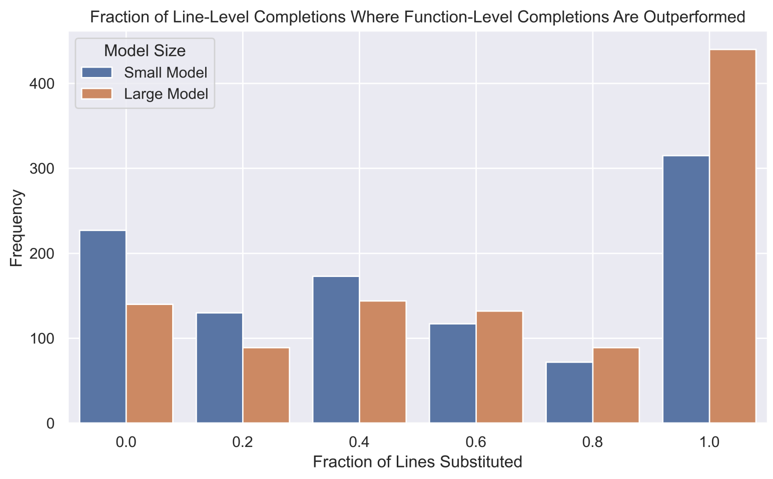

Figure 17: Distribution of increased accuracy of line-level completions compared to function-level completions, for both model sizes.

For the large model, in approximately 40% of problems, all lines can be substituted with line-level completions and the performance is increased. Subsequently, only 15% of the line-level completions never outperform the function-level completions. Following the small model, it performs worse but is not far behind. In around 30% of problems, the small model outperforms the large model with function-level completions if all lines are substituted and in only 22% of cases, it never outperforms the large model with function-level completions.

5.2.1 Comparison of Energy and Time Per Token

To allow for a deeper comparison between different granularities, we delve into the characteristics of individual token generation in the following paragraphs.

> Research Question 2.1: How do energy per token and time per token compare?

Firstly, it is important to consider the energy per token distribution, indicating how much energy each token takes to generate for different granularities and model sizes. Upon inspection of the distributions in Figure 18, one can see that the same granularity exhibits the same pattern, albeit with the small or large model. Similarly, as with the energy per function-level completion, the distribution has a peak beside the mean, which is an indication of non-normality during generation. This is most likely explained by the inconsistency of generating until the end of the function definition and continuing to generate until the token limit. The energy per token for line-level completions is similar to that of function-level completions, however, the mean is shifted towards higher energy consumption with extended tails towards the extremes.

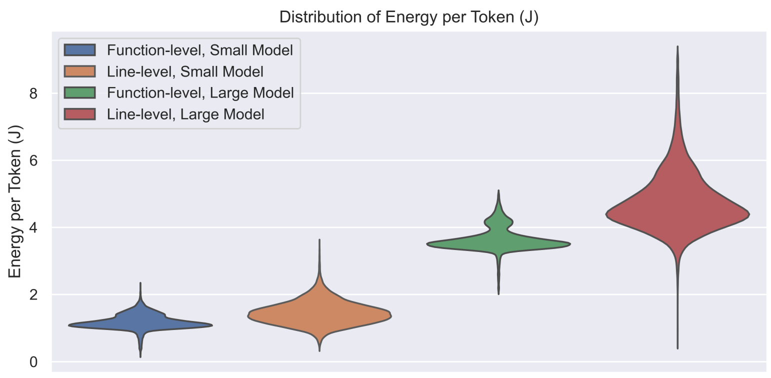

Figure 18: Energy per token violin plot for all model sizes and granularities.

When considering the time per token instead, as illustrated in Figure 19, the distributions exhibit the same patterns as with the energy per token distributions. For function-level completions, one peak exists at a lower value, and one smaller peak exists at a higher time per token. Subsequently, for line-level completions, it follows the same distributions as with energy per token; a normal distribution with extended tails towards the extreme values.

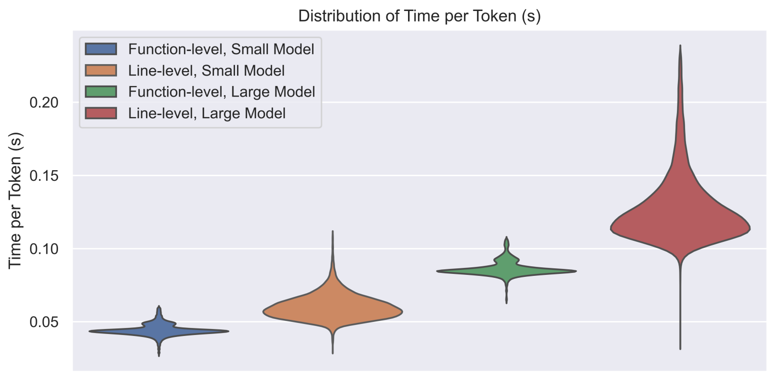

Figure 19: Time per token violin plot for all model sizes and granularities.

5.2.2 Carbon Emissions Reduction Overview

Now that we have uncovered when line-level completions outperform function-level completions, we can consider the carbon emissions reductions possible by switching from function-level completions to line-level completions. As the small model can still often outperform the large model, as shown in Figure 17, we can consider the carbon reductions at the first fraction of line-level completions where the small model outperforms the large model.

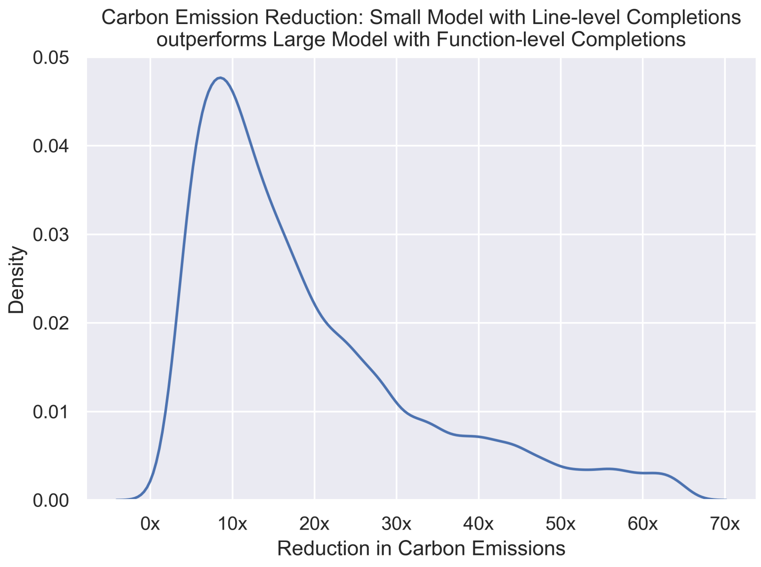

Figure 20: Carbon emissions reduction when the small model with line-level completions outperforms the large model with function-level completions.

Figure 20 shows the reduction in carbon emissions when the small model with line-level completions outperforms the large model with function-level completions. The figure contains a kernel density estimate plot; a method for visualising the distribution of observations in a dataset, analogous to a histogram. Outliers were removed if they exceeded the interquartile range by more than 50%. The distribution shows a primary peak at a reduction of $9\times$, with a long tail extending to approximately $70\times$.

By considering both the line-level and function-level completions of the small and large models, we can compare the carbon emission reductions across different completion granularities. Figure 21 shows the carbon emission reductions where line-level completions outperform function-level completions for a given problem. For the small model, the carbon emissions are reduced by a factor of $9\times$ in the lower quartile, $16\times$ at the median, and $31\times$ in the upper quartile. For the large model, the reductions are $2.5\times$ in the lower quartile, $4.5\times$ at the median, and $9\times$ in the upper quartile.

In Figure 22, one can see the potential decrease in carbon emissions per fraction of lines completed, disregarding if the small model with line-level completions outperforms the large model with function-level completions. When considering the fraction $[0.6, \ldots, 1.0]$, we can see the interquartile ranges do not deviate much. Lower quartile ranges start at a $4\times$ decrease up to a $6\times$ times decrease, with almost half of all problems expected a reduction of around $10\times$.

In summary, switching from function-level to line-level completions has the potential to achieve significant reductions in carbon emissions, particularly when we can utilise the small model with line-level completions as a substitution for the large model with function-level completions. Carbon emission reductions range from $4.5\times$ without model substitution to as much as $9\times$ with model substitution.

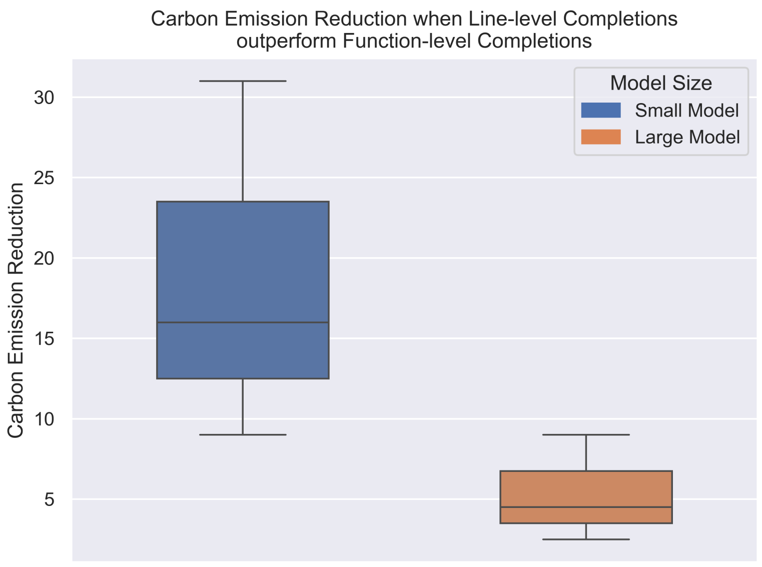

Figure 21: Comparison of carbon emission reduction when line-level completions outperform function-level completions, between the small and large models.

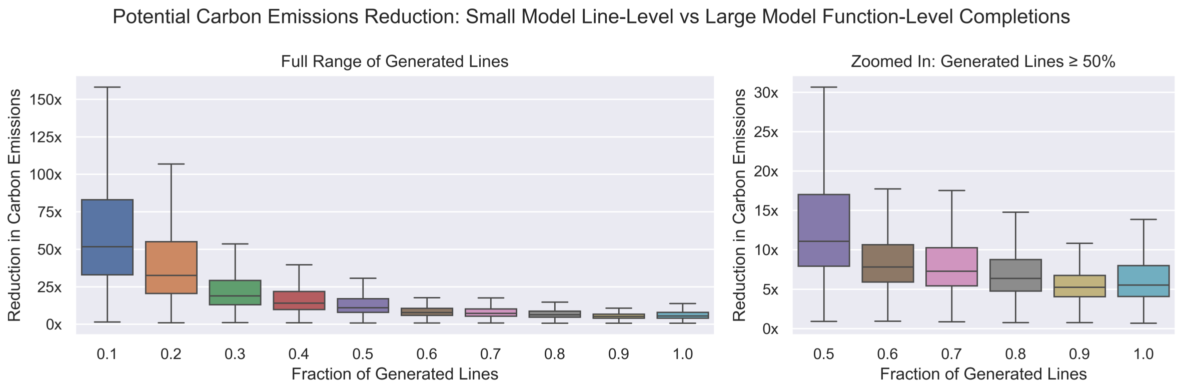

Figure 22: Potential carbon emissions reduction across different fractions of line-level completions.

5.3 Summary

This section presents results regarding an energy consumption comparison in LLM code generation. The study, conducted over 17 days with the BigCodeBench dataset, examined line-level and function-level completions across small (1.5B) and large (9B) models. The results revealed several key findings. The larger model outperformed the smaller model in terms of both compilation success and test pass rates.

However, when comparing energy consumption between the two granularities, function-level completions were found to generate more tokens and consume more energy than line-level completions. Specifically, there was an increase of 14% in energy consumption for both sizes when performing function-level generation. Despite this, the small model exhibited better token efficiency, generating fewer excess tokens.

Statistical analysis confirmed that the differences in energy consumption and efficiency between the granularities were significant, further supporting the observed trends. One interesting finding was that incorrect suggestions resulted in higher energy waste for line-level completions. Conversely, function-level generations demonstrated more efficient energy usage in that regard.

While function-level generation typically consumed more energy, line-level generation proved to be more efficient regarding token usage and energy consumption, highlighting its potential as a more sustainable approach in code generation tasks.

Furthermore, we explored the potential for carbon emission reductions by substituting a larger model performing function-level completions with a smaller model executing line-level completions, while keeping accuracy as close to the reference solution as possible. We analyse test pass accuracy, energy consumption, and time per token to highlight the trade-offs involved and assess the reduction in carbon emissions at different fractions of line-level completions substituted.

The comparison between function-level and line-level completions shows an observable trend where the test pass accuracy drops as the fraction of line-level completions increases, especially for solutions with more than 10 lines. However, in many cases, substituting line-level completions with the small model still yields good performance.

For both model sizes and granularities, the line-level completions follow a similar trend when increasing the fraction of lines substituted in the reference solution. There is a gradual decrease in test pass accuracy for short solutions, whereas the longer solutions have a steeper decline for test pass accuracy. In general, the large model performs better across both solution lengths, however, there are specific cases where line-level completions from the small model perform as well as function-level completions from the large model.

The energy per token distribution shows a shift towards higher energy consumption for line-level completions compared to function-level completions. However, these variations are relatively minor compared to the potential savings in the number of tokens processed. Similar to energy consumption, the time per token for line-level completions is slightly higher than for function-level completions, with extended tails in the distribution.

The key insight from the carbon emissions analysis comes when comparing the performance of the small model with line-level completions to the large model with function-level completions. When the small model with line-level completions outperforms the large model with function-level completions, the reductions in carbon emissions are striking. The lower quartile sees a 9-fold reduction, while the median is a 16-fold reduction, and the upper quartile experiences up to a 32-fold reduction in carbon emissions.

6 Discussion

The findings of this study provide valuable insights into the carbon emissions of code generation in LLMs, contributing to a deeper understanding of how different completion granularities impact energy consumption. The results indicate that line-level completions can significantly reduce carbon emissions, despite not compromising accuracy. This section will interpret the significance of these findings, compare them with existing literature, and discuss potential implications. Additionally, the limitations of the study will be acknowledged, and recommendations for future research will be proposed.

6.1 Key Findings

In the following paragraphs, we present the key insights uncovered through our research. We examine the quality of line-level completions, including a comparison with function-level completions. Additionally, we highlight the sustainability benefits of line-level completions, including the stabilisation of carbon emission reductions. Finally, we cover the discrepancy in token efficiencies across different granularities.

Line-level Completion Performance Upon first investigating the results, we found that individual line-level completions have high compilation and test pass rates. When considering each line-level completions individually and substituting it in the reference solution, we found a compilation rate of 96.4% for the small model and 97.1% for the large model, respectively. This possibly indicates diminishing returns when increasing model size, as almost all line-level completions compile already.

Subsequently, the test pass rate does exhibit a larger difference between the small and the large model. The individual line-level completions of the small model have a test pass rate of 82.7%, and the large model has a test pass rate of 90.5%. This indicates the correctness of line-level completions does increase significantly with model size, but warrants more thorough research with different model sizes to determine the relation between model size and test pass rate. Following the scaling laws outlined by Kaplan et al. [59], which demonstrate an exponential relationship between model size and training time, it is likely that compilation and test pass rates for line-level completions exhibit a similar trend.

Comparison Completion Granularities Building on the high test pass rate of individual line-level completions, we can compare them with function-level completions to assess whether and when line-level completions can serve as a substitute or complement. To assess this possibility, we looked at different fractions of lines substituted within the reference solution and compared the test pass rate with that of the large model with function-level completions.

For the small model, we found that in only 22% of problems, it never outperformed the large model for any fraction. However, in 30.5% of problems it outperformed the large model when substituting every line in the reference solution. When comparing line-level and function-level completions using the large model, we found that line-level completions never outperformed at any fraction in only 13.5% of problems. However, when substituting every line, they outperformed function-level completions in 42.5% of problems.

Chen et al. [21] demonstrate that the test pass rate declines as more individual components are chained within a prompt, a trend that can similarly apply to chaining multiple line-level completions. Despite this, our results show that line-level completions often outperform function-level completions, highlighting their potential as a viable alternative.

Sustainability Benefits of Line-level Completions The results demonstrate that in cases where line-level completions outperform function-level completions, there is a notable reduction in carbon emissions without compromising test pass rates. This trend is evident for both the small and large models, with substantial differences in emission reductions across quartiles. The small model exhibits a median carbon emission $10\times$ reduction, while the large model has a $4.5\times$ reduction. These findings highlight the efficiency gains of line-level completions, reinforcing their potential for more sustainable code generation.

Carbon Emission Reduction Stabilisation If we consider all problems instead, a notable finding is the relative stabilisation of carbon emission reduction for fractions $\varphi > 0.5$. This indicates relatively minor differences in emission reductions with large fractions of line substitutions, suggesting that beyond a certain point, the impact of additional line-level completions on emission reduction becomes less significant. Consequently, to achieve significant reductions, function-level completions do not have to be entirely replaced by line-level completions. Instead, partial responsibility should be returned to the developer, with only a subset of lines completed in the final solution which still would achieve a significant reduction in carbon emissions.

Token Efficiency Zooming out to compare line-level and function-level completions more broadly, we observe key differences in token efficiency between the two granularities. Within this research, we looked at two instances that affect token efficiency; the proportion of excess tokens and the impact of incorrect suggestions. Whereas the impact of incorrect suggestions is relatively similar for both granularities, the difference between the proportion of excess tokens is remarkable—on average 20% of tokens is wasted with function-level completions, compared to less than 1% with line-level completions.

6.2 Supplementary Findings

In this section, we present findings that, while not central to our primary research objectives, provide valuable insights and merit discussion. We show the increase in determinism of line-level completions and the varying frequency of reaching the token limit with different model sizes. Additionally, we explore the energy characteristics of both completion granularities, including the wasted energy consumption associated with incorrect suggestions.

More Deterministic Line-level Completions Upon investigating the log-probabilities during token generation across different model sizes and granularities, we observed a notable difference. The log-probabilities for the large model are closer to 0, indicating the model is more confident the generated token is correct. For function-level completions, the large model has an average log-probability of -0.24, compared to -0.32 for the small model. For line-level completions, the large model shows an average log-probability of -0.14, while the small model has an average of -0.2. This highlights that for both model sizes, with line-level completions the model is, on average, more confident about its completions.

Token Generation until Token Limit Frequency We observed a discrepancy in the percentage of function-level completions reaching the token limit between the small and large models. For the small model, 21% of completions reached until the token limit, compared to 25% for the large model. While the absolute difference may seem small, it represents a 20% relative increase from the small model to the large model. This warrants further investigation to see if larger model generally generate tokens more often until the token limit.

Function-level Completion Energy Characteristics In Figure 11, we compare the energy consumption of function-level and line-level completions. Line-level completions show a skewed normal distribution, while function-level completions exhibit a bimodal distribution. The larger second peak in the bimodal distribution reflects the influence of generating tokens up to the token limit, as opposed to the simplified stop token used in line-level completions. Based on the energy consumption distributions, we conclude that the median energy consumption is lower for function-level completions. However, the secondary peaks introduce measurement inconsistencies, which may not be ideal for certain use cases.

Wasted Energy Consumption of Incorrect Suggestions While the impact of excess token generation is significantly larger for function-level completions, the wasted total energy consumption from incorrect suggestions is similar between line-level and function-level completions, as shown in Figure 14. The median wasted energy is lower for line-level completions, but this comes with higher variance. Additionally, wasted energy is significantly lower for the small model compared to the large model, which is expected due to the differences in energy per token, as shown in Figure 18.

Negative Correlation Solution Length and Test Accuracy We calculated the Spearman correlation coefficient, denoted as ρ, to assess the relationship between the number of lines substituted and the test pass accuracy. For the small model, $\rho = -0.76$, and for the large model, $\rho = -0.74$. These values indicate a strong negative correlation between the length of the reference solution and the test pass accuracy, suggesting that as more lines are substituted, test accuracy tends to decrease. This inverse relationship is consistent with the probabilistic nature of token generation in language models, where increasing the number of tokens introduces greater variability and uncertainty, in turn affecting performance.

6.3 Threats to Validity

This subsection addresses the potential threats to the validity of the study's findings, highlighting factors that may influence the generalisability and accuracy of the results. First, we will address two threats to validity related to token generation, followed by potential mistakes during the experiment.

Challenges in Line-level Generation A key challenge in line-level generation is establishing the context for each prompt. Currently, we use the lines from the reference solution that precede the target line for generating line-level completions. However, ideally, we would incorporate previous completions into the context. To determine a test pass rate per fraction of lines substituted and to establish trends between different fractions, we require solutions where each line is either substituted or left unchanged. As we can substitute each previous line within the context with a line-level completion, the number of possible contexts scales exponentially in the length of the reference solution. Consequently, line-level completions could not be reused, as they were bound to a specific context rather than the reference solution. Finally, incorrect previous completions could invalidate all subsequent generations, leading to wasted energy and time.

Challenges in Function-level Generation For function-level generations, there is a risk that the token limit is reached before the function implementation is fully generated. Before starting the experiment, we set an appropriate token limit based on reference solution lengths. However, during the investigation of the results, we found that in function-level completions, the proportion of excess tokens was significantly higher than expected, and, thus, this might have impacted some of the generated solutions. A manual inspection of randomly sampled solutions did not reveal any cases where the token limit was exceeded before completing the function implementation. However, this does not rule out the possibility that such cases exist within any function-level completion. To prevent this issue in future experiments, increasing the token limit would ensure full function generation, however, due to time constraints, this was not feasible for our experiment.

Dataset Inconsistencies During an inspection of the BigCodeBench dataset, we found a set of inconsistencies regarding the docstrings of several problems, as highlighted in Section 4.3. In all likelihood, the inconsistencies found do not have a significant impact on the results of the experiment as they did not change the intention of the docstring. However, the omission of the expected exceptions to be raised during execution might have influenced the results. However, we found these inconsistencies with regular expressions and thus could be mitigated beforehand.

Conversely, function implementations lack a consistent structure, making it difficult to automatically verify their correspondence with their respective docstrings. Additionally, each problem includes a test suite, which may either be overly lenient or excessively strict—incorrectly marking correct solutions as incorrect or vice versa. To mitigate the possibility of discrepancies between docstrings, reference solutions, and test suites, we randomly sampled a set of problems and manually checked them according to the review process described within the paper. While we did not identify any issues, we cannot rule out the presence of inconsistencies in the full dataset.

Monte Carlo Simulation for Test Pass Rate To calculate the test pass rate for different fractions of line substitutions using line-level completions, we employ a Monte Carlo Simulation, which relies on random sampling. Since MCS relies on random sampling, its results may deviate from the true test pass rate that would be obtained by evaluating all possible solution combinations. We were aware of MCS's probabilistic nature but chose to use it, as evaluating all possible line-level completions was infeasible. To mitigate associated risks with random sampling and to reduce result variability, we set the number of iterations to $K = 10,000$.

Study Limitations and Scope Considerations Beyond methodological validity concerns, our study also has inherent limitations due to its chosen scope. Firstly, we could utilise different programming languages besides Python to facilitate more robust results and to determine the generalisability across other languages. Additionally, we relied solely on the CPU for token generation, but larger models typically require GPUs due to high inference times, and thus, more verification is needed that these results still are correct when performing the token generation in a way that more closely resembles a cloud environment—the setting in which most users interact with LLMs. Similarly, our use of the quantised models might unexpectedly influence the results.

Inconsistent Energy Measurement Setup To ensure accurate energy measurements, we addressed potential sources of inconsistency, as detailed in Section 4.6. However, due to the inherent variability of energy measurements, some threats to the validity still exist. The experiment spanned 17 days, leading to temperature fluctuations and other factors that could affect energy consumption. Additionally, it was not feasible to incorporate the recommended one-minute pause between measurements, as the generations typically took much less time, and we needed to perform a large number of them.

Potential Implementation Bugs Finally, during the programming of the experiment and result analysis, we cannot ensure the absence of implementation bugs or mistakes made. Despite thorough testing of the core components and the modular development of the experiment pipeline, the complexity of the system and the interdependence of various modules increases the risk of errors being introduced unintentionally. While significant efforts were made to ensure the correctness of the implementation, it is impossible to rule out the presence of such issues entirely. Therefore, there remains a possibility that unnoticed bugs could have impacted the validity of the findings.

Trigger Point Distribution We generated prompts from reference solutions using a set of trigger points, as described in Section 4.4. The set aligns roughly with Python keywords and operators, but the distribution is skewed. The top six statements account for 89.5 percent of all generation starts, while six keywords are not used within any reference solution (`>>`, `<<`, `~`, `assert`, `global`, `yield`). These keywords and operators are valid Python constructs, but we cannot accurately assess whether the model generates correct code with them, potentially affecting result robustness. See Appendix B for the frequency of trigger points within the prompts.

Validation Metrics We used test pass accuracy to compare the performance of different models, though this metric may be limiting due to its binary nature. Wang and Zhu [111] propose a new approach for validating generated code through metamorphic prompt testing, a technique that detects code flaws by generating multiple prompt variations and using cross-validation to ensure semantic consistency across the generated code. This approach could be particularly beneficial for improving code generation performance in longer solutions, where maintaining coherence throughout is essential. This might explain the discrepancy in performance between short and long solutions, highlighting a potential threat to validity, as traditional test pass accuracy may not fully capture errors in longer solutions that require stricter semantic consistency.

6.4 Implications

This section examines the broader impact of the study findings from the perspectives of four key groups: researchers, software engineers, academic institutions, and AI coding tool companies. By considering each group individually, we highlight the potential significance of this study in shaping future research, industry practices, education, and tool development.

Researchers For researchers within the field of AI, specifically LLMs, this research has multiple implications, with the most important implication following directly from the reduction in carbon emissions when utilising line-level completions over function-level completions. Since line-level completions are essentially a different method of code generation, it highlights the possibility more efficient methods exist for generating code. Whereas function-level completions might be the simplest and most effective method to compare the performance of different models, the software engineering landscape is more varied and less constrained than the sandbox setting of benchmarks may make it seem.

Subsequently, we have shown that line-level completions exhibit different results for metrics such as inference time, energy consumption, and accuracy. The metric used might differ per problem, and thus, we should focus on the generation method most suitable for the issue at hand. More research is needed to determine how different generation processes influence which metrics, and how we can optimise for any specific metric more effectively.

We also found that accuracy drops and the number of generated tokens are negatively correlated. However, line-level completions are a form of incremental code generation, where we do not generate all tokens at once, but rather in small steps. Splitting a larger generation into multiple steps reduces the complexity of individual prompts, and thus, decreases the negative correlation. This method of incremental generation might be extrapolated to different domains besides code generation, such as text summarisation, named entity recognition or multilingual translation. Utilising incremental token generation can therefore possibly increase correctness when generating a sufficiently large number of tokens.

Finally, we showed that line-level completions exhibit superior token efficiency compared to function-level completions. Regarding code generation, researchers should investigate the level of verbosity of different models and generation methods to increase token efficiency and reduce excess token generation. Similarly to incremental token generation, it can generalise to other domains; the goal should be correctness, but also low verbosity and high token efficiency.

Software Engineers Second is the group of software engineers that utilise code generation tools. The Stack Overflow 2024 Developer Survey showed 63% of professional developers currently use AI in their development process, with another 14% planning to begin soon [101]. Even though this is not a completely accurate representation of the software engineering field, the possible implications of this research can impact a large group of developers.

Our results show the generation quality depends on the model size, which subsequently increases the inference time for each generation. When developers utilise function-level completions within their workflow, the slower inference time can negatively influence productivity. Line-level completions are inherently less complex prompts, possibly allowing for a switch to a smaller model. They also lower the total number of tokens that need to be generated, decrease inference time, and allow for a more seamless integration within developer workflows.

Subsequently, the task of writing code and reviewing code is different. These two tasks necessitate different approaches to be solved, and, thus, a programmer needs to switch mental contexts when switching tasks. Since the introduction of LLMs, programmers review more code, as they need to verify the correctness of the suggestions. Subsequently, code churn—code added and changed within two weeks—has doubled in two years [40], an indication this constant switching is challenging. With line-level completions, this task becomes less daunting as a single line suggestion is more easily verified than a complete code block.

Conversely, research on code quality in GitHub Copilot-generated code suggests an overall improvement [92, 12], contradicting the concerns raised by increased code churn. However, their research focused on readability, reliability, maintainability, and conciseness—not correctness. While these factors contribute to software quality, they do not necessarily ensure functional correctness. Additionally, much of the focus was on efficiency and coding speed, however, trends in open-source repositories suggest that increased speed does not always translate into long-term benefits.

The downwards trend in code quality can be explained in a multitude of ways, one of which aligns with current biases in language models—the anchoring effect. The anchoring effect is a psychological phenomenon in which the decision of an individual is influenced by a reference point or *anchor* [104]. Just as an unreasonably priced item can make another seem like a bargain, the first solution generated by an LLM can serve as a mental anchor, making it difficult—if not impossible—for developers to consider alternative approaches. When a solution is presented on a silver platter, your first instinct is not to disregard it, but rather refine the solution. This can lead to situations only suboptimal solutions are considered—the best solution is simply not the most likely one.

Utilising line-level completions instead of function-level completions might have the secondary benefit of programmers better understanding the code they write. These small increases in understanding individually might not account for much, however, over time this morphs into a more thorough understanding of the complete system, possibly increasing the longevity of complex software projects.

Academic Institutions Finally, our research implicates academic institutions in one primary way: how students interact with AI. Takerngsaksiri et al. [103] show that AI code completion improved students' productivity, however, the over-reliance on it may lead to a surface-level understanding of programming concepts, diminishing problem-solving skills and restricting creativity. Imagine never taking the training wheels off your bicycle; technically, it is like riding a bicycle, but you do not want to rely on them always being there.

With line-level completions, the solution does not appear completely, you need to stop and think about what you want to do next, and incrementally arrive at a correct solution—you are in the driving seat. Copying code may provide an efficiency gain in the short term, but being able to write the code yourself increases your future efficiency instead. Line-level completions might prove the perfect balance to improving student productivity while ensuring they do not teach themselves learned helplessness—where repeated reliance on automated suggestions leads to a perceived inability to solve problems independently [96].

If students rely too heavily on AI-generated code and lose the ability to write code independently, we may experience a case of shifting baseline syndrome. This phenomenon refers to the gradual acceptance of lower standards as the new norm due to a lack of awareness of previous, higher-quality benchmarks [82]. While the previous case of learned helplessness already highlights a concerning issue, this scenario would be even more detrimental—students might not only accept AI-assisted coding as the norm but also lose the fundamental ability to code without it altogether.

Universities also have a substantial impact on the future of the professional software world. If universities utilise line-level completions, albeit in combination with function-level completions, students are more informed about the potential impact different code generation methods have. Over time, this knowledge will end up within the professional world, slowly changing the perspective of larger companies and the ways we utilise code generation tools.

AI Coding Tool Companies Companies that develop AI coding tools must investigate the usage of line-level completions. As their primary costs during inference are energy consumption, utilising line-level completions might reduce their overall costs, with reducing carbon emissions as collateral. The integration process is relatively simple as only the stop token sequence needs to be changed, indicating a relatively minor change while providing significant potential upsides.

Subsequently, little research has been conducted on the energy consumption of code generation in large language models, and thus, not many people are aware of the energy consumption of AI tools. Our research highlights the significant difference in energy consumption between different generation methods, and thus, the companies must give insights to their users towards energy consumption.

6.5 Future Research

While this study provides insights into reducing carbon emissions in code generation, further research is needed to refine and expand upon these findings. Exploring broader model configurations, alternative approaches, and additional evaluation methods could offer deeper understanding and improved efficiency. The following subsections outline key directions for future research opportunities.

Increased Model Sizes Following as a direct extension of this research, there are multiple future research opportunities. Firstly, we can consider different model sizes compared to the 1.5B and 9B models used, and subsequently, larger families of models. For example, one can consider the DeepSeek Coder model family (1.3B, 5.7B, 6.7B, and 33B parameters) [43], or the Qwen2.5-Coder model family (0.5B, 3B, 14B, and 32B parameters) [50]. DeepSeek Coder would facilitate a comparison between similar model sizes, 5.7B and 6.7B, whereas Qwen2.5-Coder has a wider range of model sizes to determine the robustness of line-level completions across larger model size differences.

Instruction-tuned Dataset Future research could explore using line-level completions with instruction prompts, specifically designed for instruction-tuned models. These prompts convey the same information as complete prompts but in natural language. On the BigCodeBench leaderboard, the performance of models for the instruct prompts is lower compared to the complete prompts. For instance, the instruction-tuned YiCoder-9B-Chat model scores 17.6 on complete prompts but only 11.5 on instruction prompts [13]. Since most users interact with LLMs via natural language, it is crucial to determine whether this discrepancy affects real-world performance.

Excess Token Discrepancy The discrepancy between excess token generation in line-level and function-level generation suggests a key area for improvement. An LLM designed to minimise excess tokens is crucial, but training one from scratch may be too costly. Fine-tuning offers a more efficient alternative, requiring fewer computational resources [118]. Current models struggle to determine when a function block ends, whereas detecting the end of single statements—spanning one or multiple lines—may be easier. Fine-tuning does require a large dataset, however, it should be trivial to automatically generate examples by scraping open-source repositories, extracting code, and transforming it into various prompts.

Similarly, reinforcement learning could be used to encourage the model to generate shorter code blocks, improving token efficiency by avoiding unnecessary generation up to the token limit. Guo et al. [44] apply reinforcement learning in their DeepSeek R1 model to develop a chain-of-thought process. A similar approach could be used to guide the model toward generating shorter snippets—not by explicitly instructing it but by rewarding higher token efficiencies, thereby steering it in the right direction.

Impact of Token Generation Length on Accuracy Test pass accuracies per fraction of lines substituted in the reference solution revealed a discrepancy between short (< 10 lines) and long (≥ 10 lines) solutions in test pass accuracy. Due to the probabilistic nature of code generation, accuracy decreases for longer solutions, however, instead of simply accepting this drop, we should explore restructuring long solutions into multiple shorter ones. For instance, we could explore the performance difference between a single function and the same function split into smaller, separate functions.

Solving coding problems resembles general problem-solving, where the sequential, left-to-right nature of token generation can lead to failures due to the compounding effects of early decisions. Yao et al. [121] introduce Tree of Thoughts, a framework that enables language models to explore multiple reasoning paths, allowing for more deliberate decision-making—an approach that can be adapted for programming tasks. Additionally, integrating execution-based verification allows generated code to be executed and refined iteratively, improving accuracy through feedback incorporation.

6.6 Ethical Considerations

Alongside the findings of this research and their implications, we also address several ethical considerations, blending both objective analysis and personal perspectives. In the following paragraphs, we explore various ethical angles, drawing on not just the results but also on personal experiences, usage of LLMs, and current knowledge of the technology. We begin by considering the transparency of the AI sector regarding carbon emissions, followed by the discussions on efficiency and performance. Subsequently, we cover the current availability of models and how they impact the democratisation of AI. Finally, we cover the biases existent within current LLMs for code generation.

Future Environmental Impact To acknowledge the pressing issue at hand: the future environmental impact of AI. Where the capabilities have become increasingly more impressive in recent years, model sizes have also increased exponentially to sizes never before thought possible. From our work and others [34, 124], one can find an energy consumption increase in model size, due to the increased number of computations within the network. Sevilla et al. [97] found that the required compute for AI models increased, on average, $1.4\times$ each year, however, since 2010, it increased roughly $4\times$ each year—an exorbitant difference. Interestingly, this happens in all domains of AI, not only the domains where LLMs shine.

Currently, the energy consumption of all data centres is estimated to be around 1-1.5% of global energy consumption [51]—which in itself accounts for over 40% of all emissions [3]. Where AI is expected to increase the data centre power demand by 160% by 2030 [2], it is vital to focus on carbon neutrality and mitigation of climate change risks in the near future. Despite positive goals for big tech companies regarding reducing carbon emissions and positive reports, the emissions reported by big tech companies for their carbon emissions for data centres can be roughly $7.5\times$ higher than they report themselves as they do not account for location-based emissions [4]. This worrying number warrants an increase in the transparency of carbon emissions surrounding AI.

Transparency The transparency regarding the environmental impact of AI is lacking. Castaño et al. [18] investigated the carbon emissions associated with different models on HuggingFace, and they found surprising results regarding transparency. Within a time frame of 1.5 years, on average, only 0.9% of released models report their carbon emissions. They subsequently found a carbon emissions increase concerning model size and dataset size.

In the current AI landscape, the most widely used models are those developed by large technology companies. These models are exceptionally large in scale and trained on extensive datasets, suggesting that their carbon emissions are substantial. However, there is insufficient reporting on this matter. While all individuals and organisations need to consider the environmental impact of the AI models they train and deploy, the cumulative effect of models trained by individuals is negligible compared to the massive user base of AI services operated by major technology companies. Therefore, the primary focus should be on these companies reporting their carbon emissions to provide a clearer understanding of the true environmental impact of AI.

Efficiency Given the current lack of transparency regarding the carbon emissions of most AI models developed by major technology companies, a secondary focus should be on improving the efficiency of these models. Our research demonstrates that a fivefold reduction in emissions is achievable through alternative completion granularities. In contrast, a tenfold reduction is possible by substituting a large model with a smaller one, albeit with a marginal decline in performance. While the assumption that lower performance is acceptable may be flawed, the primary short-term focus remains on maximising performance. However, if we eventually get diminishing returns, it may become necessary to shift priorities toward efficiency and sustainability.

As competition in AI development intensifies, big tech companies must develop the most advanced models for their consumers. However, if all models eventually converge in performance, this competitive perspective may shift. Since efficiency is directly tied to energy consumption—determined by various factors—we may, in the future, see models with slightly lower performance that remain adequate for practical use while significantly reducing energy consumption and, consequently, operational costs.

One important consideration is Jevons' paradox, which suggests that increasing the efficiency of resource use often leads to greater overall consumption rather than a reduction [8]. This presents a potential risk, as improvements in AI model efficiency could inadvertently drive increased usage, ultimately resulting in higher carbon emissions over time. If AI continues to be integrated into everyday life, mitigating its overall environmental impact may become increasingly challenging, if not unavoidable.

We should also consider whether language models are appropriate for all the currently researched tasks. Luccioni, Jernite, and Strubell [71] demonstrate that there is a significant discrepancy in efficiency between different machine learning tasks when performed by specialised models. For instance, text classification is an order of magnitude more efficient than text generation, yet, it remains an area of active research [112, 113]. A similar trend is observed with extractive question answering [88]. This suggests a potential mismatch between the tasks we are deploying these models for and the need to optimise energy efficiency and performance.

Availability A notable characteristic of the current generation of AI models is their limited availability to the public due to the substantial GPU resources required for deployment. However, the recently released DeepSeek R1 model challenges this norm by being significantly more cost-effective to train and run inference on. Notably, on the Aider LLM leaderboard, a benchmark designed to evaluate code editing performance [5], DeepSeek R1 outperformed the newly released GPT-4.5 while maintaining a dramatically lower cost—33$\times$ cheaper ($5.42 compared to $183.18). This shift emphasises the potential for more efficient models to achieve state-of-the-art performance at a fraction of the computational expense.

A significant reduction in the cost of training and inference enhances the accessibility and democratisation of AI models, allowing entities beyond big tech companies to shape the future of AI. The decreasing cost of state-of-the-art models, combined with the adoption of alternative generation methods to improve efficiency and reduce energy consumption, could play a crucial role in the widespread adoption of future AI technologies.

Dataset Bias A significant challenge in current AI models is the presence of bias within the datasets on which they are trained. For LLMs used in natural language processing, these biases may include racial, historical, or labelling biases. While such biases may not directly translate to code generation, other forms of bias still exist.

One critical aspect of software development is that code is written for specific versions of a programming language, which continuously evolve—Java code from 2000, for instance, differs significantly from code written in 2020. This presents a challenge for AI models, as they must accurately recognise and differentiate between programming language versions and, based on the provided code, determine which version is being used.

Since most programming languages have multiple versions, LLMs must not only understand the language itself but also comprehend each of its iterations. This presents a challenge, as certain concepts may become obsolete in newer versions, best practices for using language constructs evolve, and overall coding preferences shift over time. To ensure that an LLM can effectively assist with all language versions, its training dataset must include a representative set of examples from each version, allowing it to recognise and adapt to these differences accurately.

Similarly, the largest models are trained on vast amounts of data sourced from the web, which includes a significant amount of low-quality code. Many example projects are unfinished, contain poor coding practices, are incorrect, or use language constructs in objectively suboptimal ways. Despite this, these examples are still included in the training data. Since the model does not inherently understand the code it generates, we are reliant on the ability of the model to distinguish between high and low-quality code, which can lead to unpredictable or subpar results.

As previously mentioned, the anchoring effect—a psychological phenomenon in which an initial reference point influences subsequent decisions—can impact the chosen solutions by developers working with LLMs. When the initial code generation is accepted without critical evaluation, it can limit the range of solutions a developer considers, reinforcing the biases already present in the training data. This process speeds up model collapse, where the performance of a model degrades if it is trained on synthetic data, often its outputs [117, 29]. Over time, this feedback loop can reduce the ability to produce diverse and unbiased results, further exacerbating the problem of model degradation.

Conclusion

This study has examined the difference between line-level and function-level completions based on their energy and performance characteristics, highlighting several key findings. We found that line-level completions consistently achieve high compilation and test pass rates, with the small model achieving a test pass rate of 82.7%, and the large model 90.5%. Notably, line-level completions often outperform function-level completions with the large model, where the small model is not better in only 22% of problems, and the large model in only 13.5%. It suggests that line-level completions can be a viable alternative to function-level completions in many situations.

Regarding sustainability, our research shows that utilising line-level completions over function-level completions can significantly reduce carbon emissions, with a $10\times$ reduction with the small model and a $4.5\times$ reduction with the large model. However, reductions stabilise after half of the code is substituted in the reference solution. Additionally, we observed that line-level completions exhibit greater token efficiency, with excess tokens being reduced by over 99% compared to function-level completions. Despite some trade-offs, such as higher variance in energy per token, the overall results suggest that line-level completions can offer clear advantages in terms of both performance and energy efficiency.

We observed several possible threats to the validity of our research, including context-building challenges in line-level generations, token limits in function-level completions, and dataset inconsistencies. Function completions may be truncated, and errors in function implementations or test suites could affect correctness, though manual verification and Monte Carlo Simulation help mitigate uncertainty. Scope limitations include only using Python, CPU-based inference, and quantised models. Energy measurements can be variable due to the long experiment duration, and consequently, the changing environmental factors. Despite rigorous testing, implementation bugs remain a potential risk.

These findings have implications for several stakeholders. For *researchers*, the increased token efficiency of line-level completions—and thus reduced environmental impact—offer new insights into sustainable AI development. For *software engineers*, adopting line-level completions can improve productivity due to the decreased inference time and lesser overhead of mental context switching. In *academic institutions*, line-level completions may provide students to be more engaged, and consequently improve their problem-solving skills instead of the student becoming dependent on the tool. Finally, for *AI coding tool companies*, the integration of line-level completions can reduce energy consumption, offering both environmental and economic benefits.

Ultimately, line-level code completions show the potential to mitigate the environmental impact of code generation in LLMs. Shifting from function-level to line-level completions resulted in a carbon emission reduction of $4.5\times$, whereas if we include model substitution, the reduction increases to $10\times$. With code generation seemingly only growing more prevalent within the software industry, we must question if our technologies are utilised to the fullest extent, and highlight the consequences our usage of the technology has. As Yuval Noah Harari said, *"Humans were always far better at inventing tools than using them wisely."* This statement is more relevant than ever as we embrace new and exciting technologies, ensuring we remain mindful of their far-reaching consequences as we shape the future.

References

[1] Nov. 2022. URL: https://openai.com/index/chatgpt.

[2] May 2024. URL: https://www.goldmansachs.com/insights/articles/AI-Mayoised-to-drive-160-increase-in-power-demand.

[3] en. Sept. 2024. URL: https://world-nuclear.org/information-library/energy-and-the-environment/carbon-dioxide-emissions-from-electricity.

[4] Sept. 2024. URL: https://www.theguardian.com/technology/2024/sep/15/data-center-gas-emissions-tech.

[5] en-US. Dec. 2024. URL: https://aider.chat/2024/12/21/polyglot.html.

[6] 01.AI. *Meet Yi-Coder: A Small but Mighty LLM for Code*. Sept. 2024. URL: https://01-ai.github.io/blog.html?post=en/2024-09-05-A-Small-but-Mighty-LLM-for-Code.md.

[7] AGGI. *Annual Greenhouse Gas Index (AGGI) - NOAA Global Monitoring Laboratory*. URL: https://gml.noaa.gov/aggi/.

[8] Blake Alcott. “Jevons’ paradox”. In: *Ecological economics* 54.1 (2005), pp. 9–21.

[9] Anthropic. *Claude 3.7 Sonnet and Claude Code*. en. URL: https://www.anthropic.com/news/claude-3-7-sonnet.

[10] Paola Arias et al. “Climate Change 2021: the physical science basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change; technical summary”. In: (2021).

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

[12] Jared Bauer. *Does github copilot improve code quality? here’s what the Data says*. Feb. 2025. URL: https://github.blog/news-insights/research/does-github-copilot-improve-code-quality-heres-what-the-data-says/.

[13] *BigCodeBench Leaderboard - a Hugging Face Space by bigcode*. URL: https://huggingface.co/spaces/bigcode/bigcodebench-leaderboard.

[14] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[15] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[16] Clever Carbon. *Find out the carbon footprint of common items: Clever carbon*. URL: https://clevercarbon.io/carbon-footprint-of-common-items.

[17] Federico Cassano et al. “Multipl-e: A scalable and extensible approach to benchmarking neural code generation”. In: *arXiv preprint arXiv:2208.08227* (2022).

[18] Joel Castaño et al. “Exploring the Carbon Footprint of Hugging Face’s ML Models: A Repository Mining Study”. In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2023, pp. 1–12.

[19] Intergovernmental Panel On Climate Change. “Climate change 2007: The physical science basis”. In: *Agenda* 6.07 (2007), p. 333.

[20] Liguo Chen et al. “A survey on evaluating large language models in code generation tasks”. In: *arXiv preprint arXiv:2408.16498* (2024).

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

[22] Andrew A Chien et al. “Reducing the Carbon Impact of Generative AI Inference (today and in 2035)”. In: *Proceedings of the 2nd workshop on sustainable computer systems*. 2023, pp. 1–7.

[23] KR1442 Chowdhary and KR Chowdhary. “Natural language processing”. In: *Fundamentals of artificial intelligence* (2020), pp. 603–649.

[24] Matthew Collins et al. “Long-term climate change: projections, commitments and irreversibility”. In: (2013).

[25] Luís Cruz. *PyEnergiBridge*. `https://github.com/luiscruz/pyEnergiBridge`. 2025.

[26] Sophia R Cunningham, Dominique Archambault, and Austin Kung. “Efficient training and inference: Techniques for large language models using llama”. In: *Authorea Preprints* (2024).

[27] Saloni Dattani et al. “Child and Infant Mortality”. In: *Our World in Data* (2023). https://ourworldindata.org/child-mortality.

[28] Sandra Myrna Díaz et al. “The global assessment report on biodiversity and ecosystem services: Summary for policy makers”. In: (2019).

[29] Elvis Dohmatob et al. “Strong model collapse”. In: *arXiv preprint arXiv:2410.04840* (2024).

[30] Thomas Durieux. *EnergiBridge*. `https://github.com/tdurieux/EnergiBridge`. 2025.

[31] Ottmar Edenhofer. *Renewable energy sources and climate change mitigation: Special report of the intergovernmental panel on climate change*. Cambridge University Press, 2011.

[32] Ember. *Carbon intensity of the power sector in the Netherlands from 2000 to 2023*. July 2024. URL: `https://www.statista.com/statistics/1290441/carbon-intensity-power-sector-netherlands/`.

[33] Ember and Energy Institute. *Carbon intensity of electricity generation – Ember and Energy Institute*. Dataset. With major processing by Our World in Data. ”Yearly Electricity Data” and ”Statistical Review of World Energy” [original data]. 2024. URL: `https://ourworldindata.org/grapher/carbon-intensity-electricity` (visited on 01/07/2025).

[34] Brad Everman et al. “Evaluating the carbon impact of large language models at the inference stage”. In: *2023 IEEE international performance, computing, and communications conference (IPCCC)*. IEEE. 2023, pp. 150–157.

[35] Samer Fawzy et al. “Strategies for mitigation of climate change: a review”. In: *Environmental Chemistry Letters* 18 (2020), pp. 2069–2094.

[36] Alexander Fleming. “On the antibacterial action of cultures of a penicillium, with special reference to their use in the isolation of B. influenzae”. In: *British journal of experimental pathology* 10.3 (1929), p. 226.

[37] Georgi Gerganov. *Ggerganov/llama.cpp: LLM Inference in C/C++*. URL: `https://github.com/ggerganov/llama.cpp`.

[38] John S Gero and Fay Sudweeks. *Artificial Intelligence in Design’96*. Springer Science & Business Media, 2012, pp. 151–170.

[39] Zoubin Ghahramani. “Unsupervised learning”. In: *Summer school on machine learning*. Springer, 2003, pp. 72–112.

[40] GitClear. *AI Copilot Code Quality — Evaluating 2024’s Increased Defect Rate via Code Quality Metrics*. Feb. 2025.

[41] Global Carbon Budget (2024); Population based on various sources (2024). *Per capita CO2 emissions – GCB [dataset]*. Major processing by Our World in Data. Original data from ”Global Carbon Budget” (Global Carbon Project) and ”Population” (Various sources). Retrieved March 11, 2025. 2024. URL: `https://ourworldindata.org/grapher/co-emissions-per-capita`.

[42] Alex Gu et al. “Cruxeval: A benchmark for code reasoning, understanding and execution”. In: *arXiv preprint arXiv:2401.03065* (2024).

[43] Daya Guo et al. “DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence”. In: *arXiv preprint arXiv:2401.14196* (2024).

[44] Daya Guo et al. “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning”. In: *arXiv preprint arXiv:2501.12948* (2025).

[45] Lianghong Guo et al. “When to stop? towards efficient code generation in llms with excess token prevention”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024, pp. 1073–1085.

[46] Muhammad Usman Hadi et al. “A survey on large language models: Applications, challenges, limitations, and practical usage”. In: *Authorea Preprints* (2023).

[47] Abram Hindle et al. “On the naturalness of software”. In: *Communications of the ACM* 59.5 (2016), pp. 122–131.

[48] S Hochreiter. “Long Short-term Memory”. In: *Neural Computation MIT-Press* (1997).

[49] Robert V Hogg and Elliot A Tanis. *Probability and statistical inference*. Vol. 13. Prentice Hall Upper Saddle River, NJ, 2001.

[50] Binyuan Hui et al. “Qwen2. 5-Coder Technical Report”. In: *arXiv preprint arXiv:2409.12186* (2024).

[51] IEA. *Tracking Data Centres and Data Transmission Networks*. 2024. URL: `https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks`.

[52] *IEA | Energy Mix World*. en-GB. URL: `https://www.iea.org/world/energy-mix#how-is-energy-used-globally`.

[53] International Energy Agency (IEA). *World Energy Investment 2024*. Licence: CC BY 4.0. 2024. URL: `https://www.iea.org/reports/world-energy-investment-2024`.

[54] IRENA. *Installed capacity for different renewable technologies [dataset]*. Processed by Our World in Data. Original data from ”Renewable Capacity Statistics”. Retrieved March 11, 2025. 2024. URL: `https://ourworldindata.org/grapher/installed-global-renewable-energy-capacity-by-technology`.

[55] Maliheh Izadi et al. “Language models for code completion: A practical evaluation”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[56] Carlos E Jimenez et al. “Swe-bench: Can language models resolve real-world github issues?” In: *arXiv preprint arXiv:2310.06770* (2023).

[57] Jean Kaddour et al. “Challenges and applications of large language models”. In: *arXiv preprint arXiv:2307.10169* (2023).

[58] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

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

[60] Diksha Khurana et al. “Natural language processing: state of the art, current trends and challenges”. In: *Multimedia tools and applications* 82.3 (2023), pp. 3713–3744.

[61] Mikhail V Koroteev. “BERT: a review of applications in natural language processing and understanding”. In: *arXiv preprint arXiv:2103.11943* (2021).

[62] John R Koza. “Genetic programming: On the programming of computers by means of natural selection (complex adaptive systems)”. In: *A Bradford Book* 1 (1993), p. 18.

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[64] Czarnecki Krzysztof and Ulrich W Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[65] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[66] Baolin Li et al. “Toward sustainable genai using generation directives for carbon-friendly large language model inference”. In: *arXiv preprint arXiv:2403.12900* (2024).

[67] Raymond Li et al. “Starcoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).

[68] Rebecca Lindsey and Luann Dahlman. *Climate change: Global temperature*. Jan. 2024. URL: `https://www.climate.gov/news-features/understanding-climate/climate-change-global-temperature`.

[69] Shuai Lu et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation”. In: *arXiv preprint arXiv:2102.04664* (2021).

[70] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. “Estimating the carbon footprint of bloom, a 176b parameter language model”. In: *Journal of Machine Learning Research* 24.253 (2023), pp. 1–15.

[71] Sasha Luccioni, Yacine Jernite, and Emma Strubell. “Power hungry processing: Watts driving the cost of AI deployment?” In: *The 2024 ACM Conference on Fairness, Accountability, and Transparency*. 2024, pp. 85–99.

[72] 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.

[73] Henry B Mann and Donald R Whitney. “On a test of whether one of two random variables is stochastically larger than the other”. In: *The annals of mathematical statistics* (1947), pp. 50–60.

[74] Charles E Metz. “Basic principles of ROC analysis”. In: *Seminars in nuclear medicine*. Vol. 8. 4. Elsevier. 1978, pp. 283–298.

[75] Iman Mirzadeh et al. *GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models*. 2024. URL: `https://arxiv.org/abs/2410.05229`.

[76] Mehryar Mohri. *Foundations of machine learning*. 2018.

[77] Christopher Z Mooney. *Monte carlo simulation*. 116. Sage, 1997.

[78] Zack S Moore, Jane F Seward, and J Michael Lane. “Smallpox”. In: *The Lancet* 367.9508 (2006), pp. 425–435.

[79] Kevin Patrick Murphy. *Dynamic bayesian networks: representation, inference and learning*. University of California, Berkeley, 2002.

[80] Michael Nielsen. “Using neural nets to recognize handwritten digits”. In: *Neural Networks and Deep Learning* (2015), pp. 1–75.

[81] David Patterson et al. “The carbon footprint of machine learning training will plateau, then shrink”. In: *Computer* 55.7 (2022), pp. 18–28.

[82] Daniel Pauly et al. “Anecdotes and the shifting baseline syndrome of fisheries”. In: *Trends in ecology and evolution* 10.10 (1995), p. 430.

[83] Fernando CN Pereira and David HD Warren. “Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks”. In: *Artificial intelligence* 13.3 (1980), pp. 231–278.

[84] Aleksandar Petrov et al. “Language model tokenizers introduce unfairness between languages”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[85] David MW Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *arXiv preprint arXiv:2010.16061* (2020).

[86] Jake Prickett. *How long should it take to run unit tests?* Mar. 2020. URL: `https://medium.com/@jakeprickett/how-long-should-it-take-to-run-unit-tests-5decd79679c5`.

[87] Alec Radford et al. “Better language models and their implications”. In: *OpenAI blog* 1.2 (2019).

[88] Zafaryab Rasool et al. “Evaluating LLMs on document-based QA: Exact answer selection and numerical extraction using CogTale dataset”. In: *Natural Language Processing Journal* 8 (2024), p. 100083.

[89] Hannah Ritchie. “CO2 emissions dataset: our sources and methods”. In: *Our World in Data* (2022). https://ourworldindata.org/co2-dataset-sources.

[90] Hannah Ritchie. “What are the safest and cleanest sources of energy?” In: *Our World in Data* (2020). https://ourworldindata.org/safest-sources-of-energy.

[91] Hannah Ritchie, Pablo Rosado, and Max Roser. “Crop Yields”. In: *Our World in Data* (2022). https://ourworldindata.org/crop-yields.

[92] Mario Rodriguez. *Research: Quantifying github copilot’s impact on code quality*. Oct. 2023. URL: `https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-code-quality/`.

[93] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).

[94] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[95] Maximilian Schreiner. “GPT-4 architecture, datasets, costs and more leaked”. en-US. In: *THE DECODER* (July 2023). URL: `https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/`.

[96] Martin EP Seligman. “Learned helplessness”. In: *Annual review of medicine* 23.1 (1972), pp. 407–412.

[97] Jaime Sevilla et al. “Compute Trends Across Three Eras of Machine Learning”. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022, pp. 1–8. DOI: `10.1109/IJCNN55064.2022.9891914`.

[98] Samuel Sanford Shapiro and Martin B Wilk. “An analysis of variance test for normality (complete samples)”. In: *Biometrika* 52.3-4 (1965), pp. 591–611.

[99] Jieke Shi et al. “Greening large language models of code”. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*. 2024, pp. 142–153.

[100] Charles Spearman. “The proof and measurement of association between two things.” In: (1961).

[101] Stack Exchange. *Stack Overflow Developer Survey 2024*. 2024. URL: `https://survey.stackoverflow.co/2024/`.

[102] Jovan Stojkovic et al. “Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference”. In: *arXiv preprint arXiv:2403.20306* (2024).

[103] Wannita Takerngsaksiri et al. “Students’ Perspectives on AI Code Completion: Benefits and Challenges”. In: *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2024, pp. 1606–1611.

[104] Amos Tversky and Daniel Kahneman. “Judgment under Uncertainty: Heuristics and Biases: Biases in judgments reveal some heuristics of thinking under uncertainty.” In: *science* 185.4157 (1974), pp. 1124–1131.

[105] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

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

[107] Roberto Verdecchia, June Sallou, and Luís Cruz. “A systematic review of Green AI”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 13.4 (2023), e1507.

[108] Pablo Villalobos and Anson Ho. *Trends in Training Dataset Sizes*. 2022. URL: `https://epochai.org/blog/trends-in-training-dataset-sizes`.

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

[110] Jiexin Wang et al. “Is Your AI-Generated Code Really Secure? Evaluating Large Language Models on Secure Code Generation with CodeSecEval”. In: *arXiv e-prints* (2024), arXiv–2407.

[111] Xiaoyin Wang and Dakai Zhu. “Validating LLM-Generated Programs with Metamorphic Prompt Testing”. In: *arXiv preprint arXiv:2406.06864* (2024).

[112] Zhiqiang Wang, Yiran Pang, and Yanbin Lin. “Smart Expert System: Large Language Models as Text Classifiers”. In: *arXiv e-prints* (2024), arXiv–2405.

[113] Zhiqiang Wang et al. “Adaptable and Reliable Text Classification using Large Language Models”. In: *arXiv preprint arXiv:2405.10523* (2024).

[114] Climate Watch. *Historical GHG Emissions*. 2021. URL: `https://www.climatewatchdata.org/ghg-emissions`.

[115] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[116] Sean Welleck et al. “From decoding to meta-generation: Inference-time algorithms for large language models”. In: *arXiv preprint arXiv:2406.16838* (2024).

[117] Robert Wu and Vardan Papyan. “Linguistic collapse: Neural collapse in (large) language models”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 137432–137473.

[118] Yuchen Xia et al. “Understanding the performance and estimating the cost of llm fine-tuning”. In: *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2024, pp. 210–223.

[119] An Yang et al. “Qwen2 technical report”. In: *arXiv preprint arXiv:2407.10671* (2024).

[120] John Yang et al. “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering”. In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. 2024. URL: `https://arxiv.org/abs/2405.15793`.

[121] Shunyu Yao et al. “Tree of thoughts: Deliberate problem solving with large language models”. In: *Advances in neural information processing systems* 36 (2023), pp. 11809–11822.

[122] Hao Yu et al. “Codereval: A benchmark of pragmatic code generation with generative pre-trained models”. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–12.

[123] Jiasheng Zheng et al. “Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models”. In: *arXiv preprint arXiv:2407.11470* (2024).

[124] Zixuan Zhou et al. “A survey on efficient inference for large language models”. In: *arXiv preprint arXiv:2404.14294* (2024).

[125] Terry Yue Zhuo et al. “Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions”. In: *arXiv preprint arXiv:2406.15877* (2024).

A Problem Selection

ID	Reasons
0	large input size
17	delays
20	plotting
31	plotting
35	plotting
37	train ML model
43	plotting
46	plotting
57	plotting, file operations
70	plotting, file operations
71	plotting, file operations
76	network operations
78	file operations
82	file operations
85	plotting
99	plotting
102	plotting
105	plotting
147	network operations
156	plotting
180	image manipulation
181	delays
187	plotting
195	delays, network operations
224	plotting, math operations
227	file operations
237	plotting, file operations
241	plotting
242	image manipulation
278	math operations
289	train ML model
314	network operations
324	delays
337	plotting
346	delays
347	large input size
348	delays
363	multi-processing
372	file operations
377	delays
396	plotting
417	train ML model
418	train ML model
419	train ML model
421	delays, network operations
424	image manipulation
425	image manipulation
428	plotting
437	file operations
443	train ML model
444	plotting
449	plotting
451	train ML model
456	plotting
459	delays
460	plotting, file operations
461	delays
486	plotting
489	large input size
502	plotting
527	plotting, file operations
534	cryptography
547	cryptography
579	plotting, file operations
580	large input size
582	plotting
583	cryptography
596	plotting
604	compilation
608	plotting
609	large input size
610	plotting
614	plotting
618	plotting
622	plotting
639	plotting
640	plotting
655	math operations
719	file operations
774	train ML model
791	large input size
819	delays
821	delays
823	delays
845	math operations
857	delays, file operations
868	large input size
878	train ML model
908	plotting, file operations
917	train ML model
953	plotting, file operations
979	train ML model
980	plotting
983	plotting
995	plotting, file operations
1003	network operations
1016	image manipulation
1032	plotting, large input size
1033	plotting, large input size
1034	plotting
1040	delays, network operations
1058	plotting, large input size
1064	plotting
1069	plotting
1104	delays, file operations
1105	delays, file operations

Table 9: Breakdown of reasons contributing to test suite slowness per problem.

B Trigger Points

Trigger Point	Count
`=`	3796
`.`	1840
`return`	1237
`if`	880
`raise`	461
`for`	452
`,`	291
`with`	196
`:`	182
`except`	138
`(`	124
`+=`	54
`elif`	28
`while`	10
`else`	9
`+`	9
`not`	5
`or`	5
`and`	3
`lambda`	2
`*`	2
`<`	2
`@`	2
`>`	1
`del`	1
`-`	1
`in`	1
`-=`	1
`assert`	0
`global`	0
`yield`	0
`is`	0
`/`	0
`%`	0
`**`	0
`<<`	0
`>>`	0
`&`	0
`|`	0
`^`	0
`==`	0
`!=`	0
`<=`	0
`>=`	0
`{`	0
`_`	0

Table 10: Frequency of trigger points from which token generation starts in line-level completions. Underlined trigger points are keywords or operators that are not used in any reference solution.