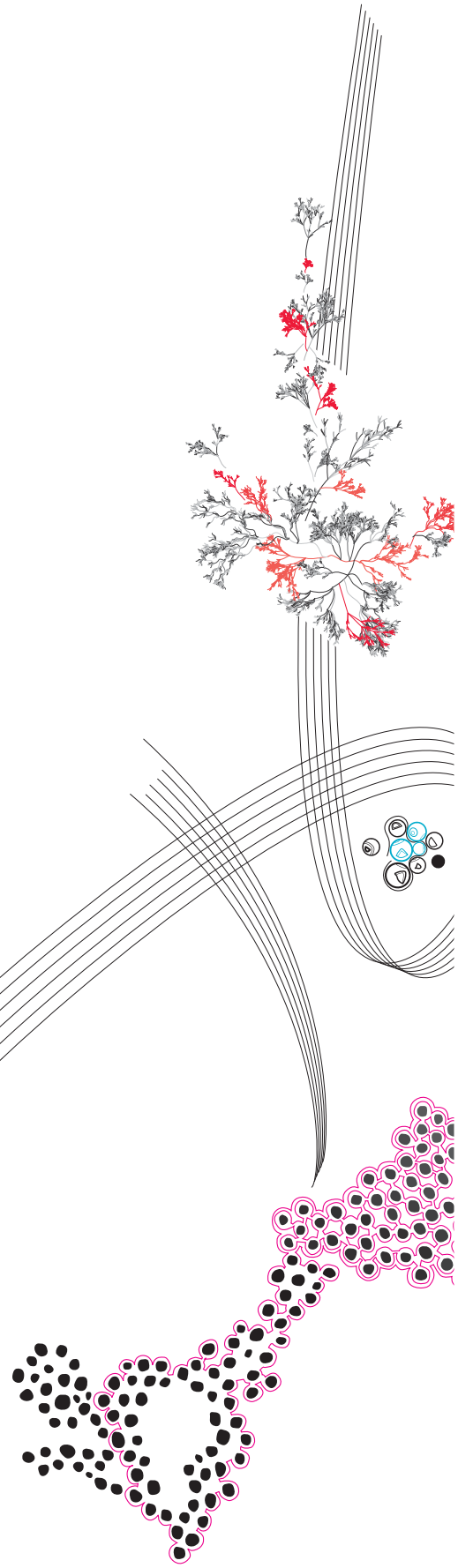


MSc Computer Science
Final Project

Code Green: Evaluating the Carbon and Energy Implications of LLM Integration in Software Development

Boris Belchev

Supervisor: Fernando J. Castor de Lima Filho & Maya Daneva

February, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Large Language Models (LLMs) have demonstrated incredible growth in their capabilities and the opportunities they provide, which has caught the public's attention. The domain of software development is naturally more inclined to bear the fruits of these advancements. However, deploying and using LLMs on a large scale comes with the burden of using more energy and releasing more carbon emissions. Although research has focused on the training costs of these models, there is a gap left uncovered and questions to be answered about the deployment phase. Therefore, this study aims to fill the gap and evaluate the use of LLMs in their inference stage. More specifically, in the context of software development. Through a series of experiments, this study quantifies the energy consumption of coding and general-purpose models across code-related tasks, such as code generation, bug fixing, documentation and testing. The findings provide insights into the trade-offs between accuracy and energy efficiency, helping guide future research and development toward more sustainable and effective LLM deployment.

Keywords: Green AI, LLM, Large Language Model(s), Inference, Software Development, Energy, Sustainability

Chapter 1

Introduction

Artificial intelligence (AI) has recently experienced a very rapid development, which has contributed to improving its adaptability and integration across industries. Particularly, Generative AI, a subset of AI specializing in generating content such as text, images, or audio from data patterns, has undergone significant growth. This growth was triggered by the introduction of the transformer model in 2017 [55]. In turn, led to the creation of the first Generative Pre-trained Transformer (GPT) models, which set the stage for OpenAI's GPT family of Large Language Models [4]. Following that, major tech companies, such as Google and Meta, released their generative models, namely BERT and LLaMa[15, 54]. However, this wave of new LLMs was not exhausted, as many new models were released afterwards, and companies were founded specialized in that area, like Mistral [37].

Generative AI extended beyond the confines of research and professional spheres closely associated with Artificial Intelligence and captured the wider public's interest. For example, ChatGPT reached 1 million users five days after its release, beating other services and technologies like Facebook and Instagram [17]. Another report about adults in the US showed that 70 % of respondents are aware of it, while almost 31 % have used Generative AI chatbots like ChatGPT, Bing and Bard. Apart from the general public becoming more aware of generative AI, businesses have indicated that they have either already used or are planning to use GenAI, with only 1 % saying that they do not intend to use it in their business [44].

From the sectors leveraging Generative AI, software development is expected to be one of the primary beneficiaries. This can be seen from the results of surveys conducted in 2023 by StackOverflow and GitHub, which revealed a significant use of AI tools among software engineers. In the first survey, 44 % of developers use AI tools, and 25 % plan to, while the second survey concludes that 92% of US developers use AI tools [2, 20]. In this domain, GenAI finds its application mainly through coding assistants powered by fine-tuned Large Language Models such as OpenAI's Codex (fine-tuned GPT-3) or Code LLaMa (fine-tuned LLaMa)[42, 35, 46]. Unlike the traditional conversational interface approach, they integrate into developers' IDEs, suggesting and completing code in real-time.

LLMs offer improved efficiency and vast potential across numerous applications in life; however, they inherently require substantial computational resources [40, 60]. For instance, training the LLaMa models, the Meta team spent 1,770,394 GPU hours using high-end NVIDIA GPU A100-80GB [1]. In terms of energy, the 7B (billion parameters) and 65B models consumed 36 MWh and 449 MWh of energy, where the former is analogous to the annual energy use of 13 households and the latter to 160 households in the Netherlands

[1]Source: "NVIDIA A100 Tensor Core GPU," available at: https://www.nvidia.com/en-us/data-center/a100/.

[11, 54]. Continuing this notion, it was derived that the carbon emitted for training the 65B parameter version is equal to 173 metric tons, equivalent to the annual emissions of 37 US cars [3]. This shows a considerable energy and emissions footprint from training these models. However, training the model is often a one-time event. At the same time, a more interesting area of exploration is the energy usage of the LLM's deployment phase, or the so-called inference. As discussed previously, software development is inherently more inclined to adopt AI tools such as GenAI for tasks within the field. Despite growing research on inference energy usage, studies have focused predominantly on general-purpose tasks such as question-answering context. There is a gap in research on software development tasks such as code generation, code completion, etc., which this study aims to fill. To address the gap in existing literature, the study will encompass a series of experiments on different proprietary Large Language Models measuring the energy used by inference for different software development tasks related to coding. The models will be selected according to criteria such as popularity, performance, etc. Because of the proprietorial nature of some of the most notable examples, such as OpenAI's GPT model, the study will not attempt to analyze them and will focus on locally downloadable open-source models. The results will provide a more comprehensive understanding of the energy implications of LLMs in software development.

This thesis is part of a broader research effort, culminating in a published study at the 22nd International Conference on Mining Software Repositories (MSR 2025). The paper, co-authored by Negar Alizadeh, Boris Belchev - the author of this thesis, Nishant Saurabh, Patricia Kelbert, and Fernando Castor, titled Language Models in Software Development Tasks: An Experimental Analysis of Energy and Accuracy, presents key methodological approaches, analysis of the results, and findings from this research [5].

1.1 Research objectives

The main objective of this study is to investigate the energy consumption and carbon emissions associated with the use of Large Language Models (LLMs) in software development. Specifically, the study will explore the following research questions:

RQ1: What energy consumption and carbon emissions are associated with using Large Language Models (LLMs) in software development?

To answer this research question, we conducted experiments that comprise deploying a sample of selected LLMs and measure the energy used during inference of tasks, such as code generation, documentation, bug fixing and test generation. The measurements will cover a large sample of LLMs from different research teams and companies. The answer to this question should provide quantitative data on a large enough sample to support the derivation of estimates on the overall usage of LLMs and their implications.

RQ2: How do coding models, fine-tuned models, and general-purpose LLMs compare in software development tasks in terms of energy usage and efficiency?

In this study we differentiate the models in three types: coding models, which are specifically designed and trained to generate programming language text; general models, which are primarily designed for generating natural language text but can also generate programming language text; and fine-tuned models, which are general models further trained or fine-tuned specifically for generating programming language text. The study will compare the energy efficiency of fine-tuned or purely coding models versus general-purpose LLMs. Its aim is to identify if there are energy and carbon savings between using specialized coding models and general-purpose models. In addition, it will indicate whether further fine-tuning of a model might enhance its efficiency.

RQ3: How does the energy consumption of LLMs vary across different software development tasks, such as code generation, bug fixing, documentation, and testing?

This research question focuses on the differences in energy consumption in software development tasks. It helped us identify existing efficiency gaps in particular tasks compared to others. By comparing the energy efficiency of the tasks, it will also provide guidance on which ones can be utilized most efficiently and for which tasks better approaches might be needed in the future.

RQ4: What characteristics of Large Language Models (LLMs) impact energy consumption and efficiency in software development tasks such as code generation, bug fixing, documentation, and testing?

This question investigates the influence of model characteristics, such as parameter count, transformer block count, embedding size, and attention head count, on energy efficiency and accuracy. By using correlation analysis, the study identifies key architectural features that affect performance, helping to optimize model design for specific tasks and operational constraints.

In summary, the research objectives of the study will seek to provide an answer to small or medium enterprises that want to preserve their privacy and choose the best solution according to their infrastructure and requirements. This involves filtering out cost-ineffective and environmentally harmful options to deploy their own locally available LLM.

Chapter 2

Background

LLMs, or Large Language Models, find their roots in Deep Neural Networks (DNNs). DNNs are a common term for a family of artificial neural networks based on their principles and working, such as Convolutional neural networks (CNNs) and Recurrent neural networks (RNNs). Deep Neural Networks consists of multiple layers in between input and output. At the same time, those layers consist of neurons, synapses, weights and biases, drawing inspiration from the structure and functioning of the human brain [8]. Fundamentally, deep neural networks (DNNs), in their basic forms, are not well suited for language modelling. Therefore, LLMs were based on the idea of Recurrent neural networks, which are more suitable for language tasks [52, 51, 27].

LLMs do not process words as directly as text as humans usually do. They use tokenization schemes to convert words to numbers, which come predefined in the tokenizer of the model. These numbers are associated with embeddings, vectors representing the word (or part of the word). Embeddings are not hard-coded or predefined, but are learned during the training process. Using these embeddings is what the model processes as input to generate an output (Figure 2.1). In addition to that, tokenizers also have special tokens, such as [END], which marks the end of a sequence, or [PAD] for padding. They serve more of an organizational purpose where the model is specifically trained to recognize them and adjust its behavior, e.g. when the model observes [PAD] in the input, it simply ignores them as it knows they are not needed for the prediction. However, there are slightly different approaches to tokenization, such as Byte Pair Encoding (BPE) and WordPiece [50, 49]. In both approaches, the word is first divided into the individual characters, but at the next step in the former, it merges the characters on the most frequently occurring together criteria while the latter directly starts forming subwords and checking for an occurrence of this subword in a predefined vocabulary.

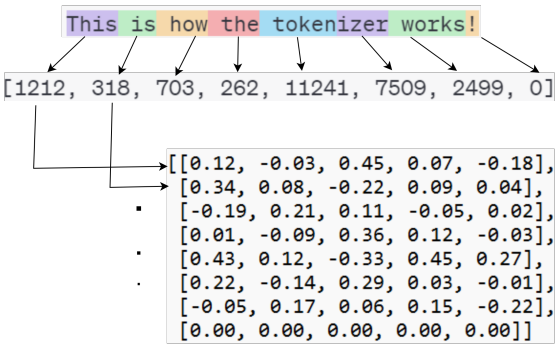


FIGURE 2.1: Transformation of input through tokenization and conversion to embedding.

As with every program and machine, LLM comes with adjustable parameters. Those hyperparameters control the behaviour during training and inference. One such important parameter is the temperature. Temperature controls the model's 'creativity' or randomness, where higher tempera-

ture gives more random and diverse responses. Lower temperature makes the model more deterministic - it chooses the most likely response. The hyperparameters do not get exhausted with temperature, but there are also top-p, sampling, maximum length, maximum new tokens, etc. The top-p, for example, limits the pool of possible answers (0.95 value means only the top 5% most likely answers participate in the pools), and the max new tokens limit the number of new tokens to be generated[1]. In contrast, max length limits the length of the response counting and including the input. Large Language Models (LLMs) vary not only in size but also in the architectures that underlie them and power their capabilities. Depending on the context of usage, several terms are used for these types of architecture.

Encoder-decoder architecture has two components: the encoder that processes the input to internal abstract representation, and the decoder utilizes that to generate target output text. Depending on the context, this architecture can also be recognized as Seq2Seq (Sequence to Sequence) whenever the focus is on translating sequences of varying lengths, where the whole sequence is encoded to decode the target output [52]. Additionally, when the model, instead of processing the entire sequence it focuses on important parts of it, it is regarded as Attention or Transformer[55].

The decoder-only architecture uses of only one component: the decoder. Models utilizing this architecture generally try to predict the subsequent tokens from the 'initial state', which can be a prompt, such as "This paper is about...". The state-of-the-art refers to this architecture with different terms, such as generative or autoregressive [45, 10].

The encoder-only architecture uses the encoder component only by capturing the essence of the input and translating it into a high-dimensional vector or abstraction to detect the underlying pattern. It is also known as Bidirectional, Feature Extractors, and Contextual Embedding, each highlighting a unique facet of this architecture [15]. Due to the foundational relationship between LLMs and DNNs, they inherit a valuable property known as transfer learning. Transfer learning is a capability of neural networks that, once pre-trained on a dataset(s), can be retrained or fine-tuned on a smaller task-specific dataset while leveraging the previous knowledge gained [23]. This allows the model to gain new knowledge in a task with fewer data and computational resources. Examples of fine-tuned code models are CodeLlama, a fine-tuned Llama2 general model, and CodeGemma, which is based on Gemma. On the other hand, a coding model can also be trained from "scratch", where such cases are StarCoder2 and Granite Code.

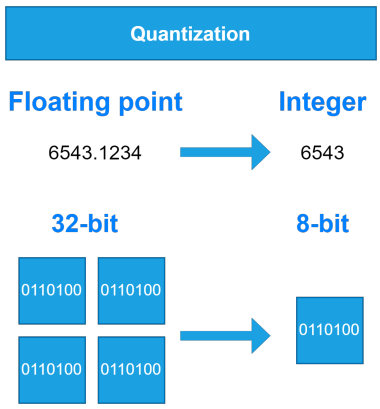


FIGURE 2.2: 32-bit floating-point to 8-bit integers.

As Large Language Models evolve, their complexity and size increase, and the demand for more computational power rises. Innovative techniques such as quantization of neural networks are employed to address this. **Quantization** reduces computational requirements by transforming the floating-point representation of weights and activations into integers (Figure 2.2). That method makes the model more accessible to a broader range of devices with less computational power, such as personal computers and notebooks. There are different approaches to performing quantization.

[1]Source: "Model Parameters and Prompting," available at: https://www.ibm.com/docs/en/watsonx/saas?topic=lab-model-parameters-prompting.

Examples include Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). Different quantization levels, e.g. 8-bit and 4-bit, offer trade-offs between performance and accuracy [39, 25].

One important aspect of large language models is their **evaluation** after training. Evaluating can cover different aspects, such as how much a model text is human-like or how well the model answers to previously unseen data. All these have associated metrics such as BLEU and perplexity and datasets with predefined tasks used to carry out the evaluations. For code generation, one such dataset is HumanEval, introduced by OpenAI for evaluating their coding model - Codex[13]. It contains prompts with Python functions and docstring, which must be completed. A newly defined metric, called pass@n, was used to measure the accuracy. There are slightly different variants of that metric, such as pass@1, pass@10 and pass@100, which differ in the number of attempts to be measured. The first is the strictest, where the model accuracy is measured from the first attempt to determine if the generated code will pass the test suites. At the same time, the rest give the model multiple attempts where one successful generation is counted as positive. Since the introduction of this way of evaluating the coding capabilities of models, the dataset has been extended through HumanEval-X to other languages such as Java, C++, etc., and through HumanEvalPack to other tasks such as bug fixing and explaining code and via HumanEvalPlus with more tests[38, 30]. The first entry of HumanEval is shown in Figure 2.3.

task_id

```
HumanEval/0
```

prompt

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

canonical_solution

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

test

```
METADATA = {
    'author': 'jt',
    'dataset': 'test'
}

def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

FIGURE 2.3: HumanEval task adapted from Yetiştiren et al. (2022) [57]

Software development is concerned with all phases of a software from the initiation to the post-deployment activities. It includes tasks such as designing, documenting, im-

plementing, testing and maintenance. It employs different methodologies such as Agile, Waterfall and etc. (Figure 2.4). These methodologies have different advantages and disadvantages according to the context of the software to be developed. Agile is based on the incremental development philosophy and it is more suitable for software that should keep up with changing customer and market requirements. On the other side, Waterfall tends to be more traditional approach ideal for software with stable requirement and regulatory obligations [24]. Regardless of the methodology, the core processes involved in software development remain consistent. LLMs integrate into the software development context in

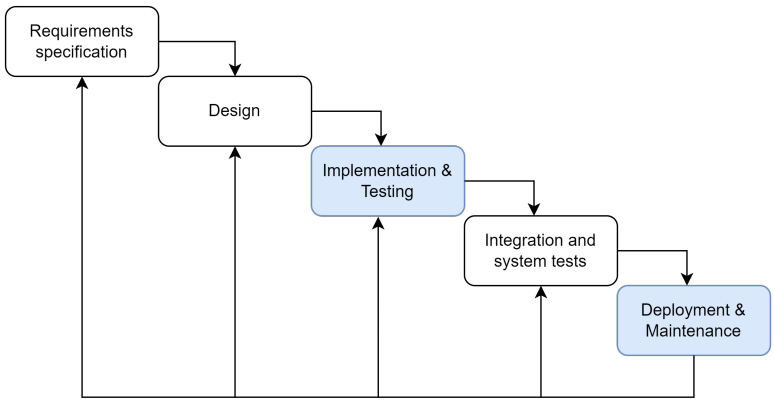
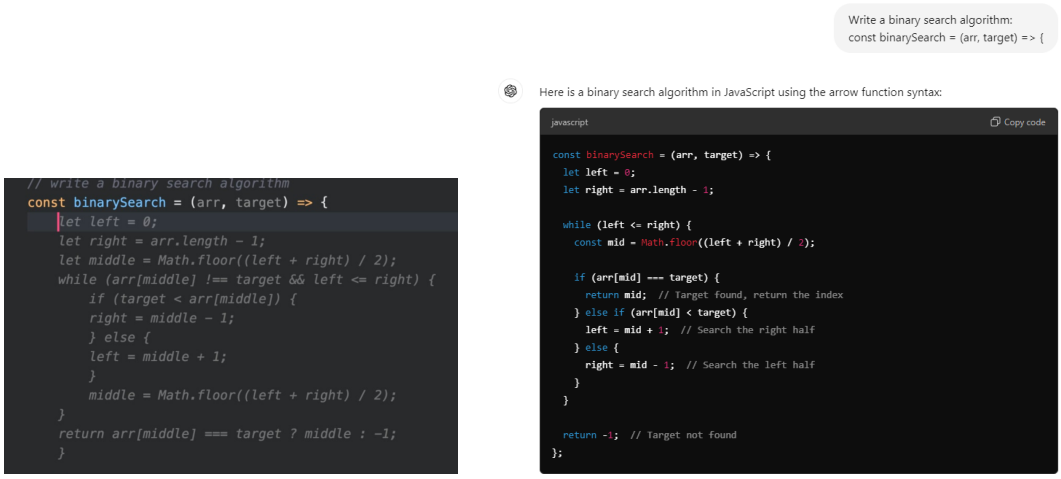


FIGURE 2.4: Waterfall model with stages of focus for this study highlighted in blue. Based on "Software engineering" 10th edition of Ian Sommerville [24]

several ways, including chat interface models such as OpenAI's GPT 3.5 and 4 and coding assistants such as GitHub CoPilot embedded in the IDE, subtly providing generations and completions. In the former case, the user should explicitly write instructions to the LLM in the provided interface with context on what to generate. In contrast, in the former, the model, without explicit prompting, suggests completion of the unfinished code, as shown in Figure 2.5.



(A) GitHub Copilot (B) ChatGPT 4

FIGURE 2.5: Binary search algorithm implemented in both approaches.

Chapter 3

Related Work

This chapter aims to describe the current state of research related to measuring energy usage of the inference phase of LLMs. One of the first examples is the work of Luccioni et al. [34], which estimates the energy in the life cycle and the carbon footprint of the BLOOM 176B model. It sets a precedent by also studying the deployment stage of the model. The model was hosted on the Google Cloud Platform, and measured the inference energy was measured over a period of 18 days. Following this first study, subsequent research delved explicitly into the deployment phase of state-of-the-art models of Meta AI. This work measured the energy consumption of three different model sizes using two datasets: one for general question-answering tasks and another for mathematical problem-solving [47]. Moreover, a study was conducted that provides a comprehensive analysis and comparison of inference energy costs across a wide range of generative systems, including both task-specific and general-purpose models [33]. The work utilized 88 models over 10 tasks and 30 datasets, including image generation, text completion, etc. The study presented key insights into the area, pointing out that generative tasks tend to produce more carbon and consume more energy than discriminative ones. Furthermore, it found that task-specific models are more energy efficient for these same tasks than the general-purpose ones. The analysis extends to comparing the efficiency of different architectures: encoder only, decoder only, and encoder-decoder. Lastly, a review-based study examined existing research on training and inference costs. The focus is on the trends of models' energy usage as performance and size increase. In the area of natural language processing, it referred to past studies that used the GLUE benchmark to estimate their compute costs. Some of the cutting-edge models under review are GPT and BERT. The findings indicate that, while larger models require more computational power energy, advances in newer models allow them to achieve the same performance using less power. Contrary to previous assumptions, it was found that the growth of energy consumption is not exponential but rather gradual due to algorithmic and hardware improvements. The study also highlights the scarcity of research for the inference phase of models compared to the training phase [14]. A recent work by Faiz et al. (2023) went further and created a large language model which estimates the carbon footprint and, with that, the energy usage by using different factors as input, such as parameter count, TFLOP, and hardware efficiency called LLMCarbon [19].

Chapter 4

Methodology

This chapter of the study describes the methodology, employed to assess the energy consumption and operational efficiency of large language models (LLMs) in software development contexts. The methodology has several different aspects: selection of models to assess (Section 4.1), selection of dataset covering the different software development activities (Section 4.2), setup, and design of the experiment that will produce results on the set problem (Sections 4.3 & 4.4) and lastly it will dicuss possible derived metrics to be used for effective comparison (Section 4.5).

4.1 Model selection

4.1.1 Criteria

The choice of different LLMs is important as it impacts the relevance and applicability of the study's findings. Therefore, the selection process adheres to a defined set of criteria:

- **Popularity**: Popular models among the community are more likely to see widespread adoption and continued support, which adds to the relevance of measuring their energy impact. Popularity can be measured by the number of downloads of the model. The data about a number of downloads is sourced from HuggingFace [1]. This will ensure that the results are applicable to the most commonly used models in real-world scenarios. For the sake of consistency, the total number of downloads is summed across every variation and version of the same model, independent of size (e.g., 3B or 7B) or type. Furthermore, only repositories provided by the model's original creators are included, excluding those modified by third parties.

- **Research presence**: Models that are included in research papers on comparing performance or evaluating the model's capabilities in different aspects etc. show a robust basis for inclusion. This will enhance the validity and the value of the study as it will position it among the existing literature.

- **Reputability of creator**: While this criterion can be arguably subjective, it is still important. Companies and research teams that have substantial funding and support from other entities in the field are perceived as more likely to develop further and support their models. They are assumed to have a larger infrastructure and more research capabilities as they strive for competitiveness with other key players in the field. For example Google and its research team made significant breakthroughs in the field and became a key figure in the field of AI. Choosing models according to that criteria increases the value of the results as small or medium-sized companies

would be more likely to contract such entities and deploy their models as they already have experience in the field of provided SaaS solutions.

- **Coding vs General LLMs**: Models with general purpose versions and fine-tuned coding versions will also be included (e.g. Code Llama vs LLaMa). This will facilitate comparison to asses whether optimizations related to the software development context have a positive energy impact. Due to limitations such as some popular and well-known models not having fine-tuned versions or otherwise, this criteria will not be of high priority (e.g. Mistral). Apart from that, an equal number of general models should be selected to serve as a baseline for the coding models. Additionally, general models might be better in certain tasks, such as documentation; therefore, this would allow for a nuanced analysis of the trade-offs between generalization and specialization.

- **Proprietary vs open-source**: Because some models are private and not available for download and local setup, such as GPT-3 and 4, those models automatically are excluded from the study. The study will focus on models that are either open source or accessible under specific licensing agreements (e.g. Meta models), requiring users to accept terms and conditions before downloading. These models can be set up locally, ensuring accessibility and reproducibility of the research findings.

4.1.2 Selected models

Using survey studies on Large Language models in the context of software engineering and coding, the study sourced 33 models specialized in code [59, 58, 22]. Certain models identified as "specialized in code" in these studies were omitted based on a more rigorous interpretation of what constitutes a large language model for coding in this research. In this study, models explicitly designed and aimed at coding tasks such as code generation and bug fixing are considered, whereas those that also serve general Q&A purposes are categorized as "general". For example, GPT-Neo by EleutherAI and Phi-2 by Microsoft were excluded because they target a wider array of tasks beyond coding, even though they were listed under coding LLMs in those studies due to their competitive performance in evaluations [9, 29]. In Figure 4.1, the ranking of models based on their download counts is displayed. To narrow down the selection, models with more than 10,000 downloads are categorized for the subsequent phase of selection. Among the models in the green box, the Phind codellama was excluded because it did not meet the reputability criteria. Details about the organization 'Phind' were unavailable. Additionally, since it is derived from an existing model by Meta, it does not constitute entirely original work. We also removed StarCoder and SantaCoder, both of the BigCode research teams. The reason for their exclusion is that Starcoder represents an older version of StarCoder2. Therefore, it is more suitable to look at the most recent iteration. SantaCoder is excluded because it is one of the oldest models from that team and lacks community activity and updates. Phi-1 is also excluded as it is the oldest iteration from Microsoft's Phi family, and since version 1.5 they hadve been focusing more on general tasks. Therefore the model purpose changed with every iteration.

The research also seeks to determine whether general-purpose models consume more or less energy compared to their coding-specific counterparts in code-related tasks. To facilitate this comparison the general-purpose equivalents must be considered. For CodeLlama, this equivalent is Llama2, and for CodeGemma, it is Gemma. During the course of the study, Meta introduced a new model dubbed Llama3, which is included to increase the study's relevance considering it is the most likely successor. The remaining models lack

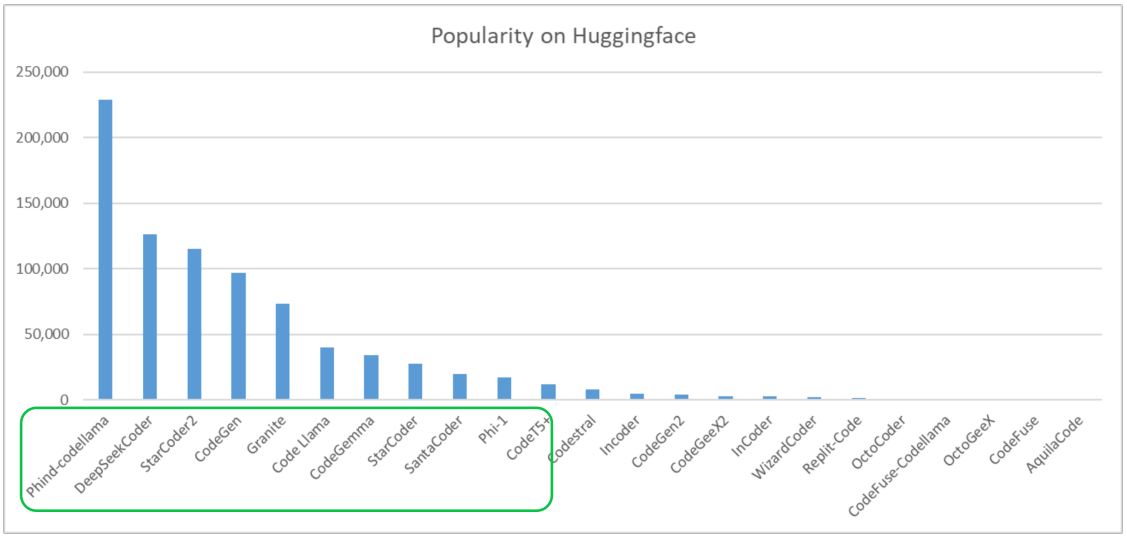


FIGURE 4.1: Popularity by the number of downloads with models above the 10,000 threshold highlighted in green (snapshot date is August 2024)

general-purpose counterparts. One of HuggingFace's most popular models, Mistral 7B, was selected to address this. Other versions, such as Mixtral, exist but cannot be deployed due to hardware limitations. In the resulting selection, there was still an unequal number of general-purpose (4) and coding models (5). To address this, Phi-3 from Microsoft was included as another general-purpose model. Phi-3 is available in a smaller variant with 3.82 billion parameters, allowing for comparisons with other 'small' coding models such as Granite 3B, CodeGemma 2B and DeepSeekCoder 1.3B. Additionally, Phi-3 extends to a larger version with 14 billion parameters, enabling comparisons with larger models like Starcoder 16B and Granite 20B. In Figure 4.2 can be seen the final population of models

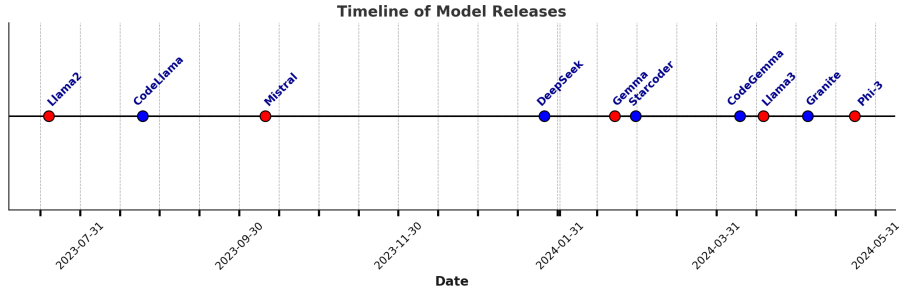


FIGURE 4.2: Timeline of release of models. Red are general-purpose models and blue is coding models.

plotted on a timeline according to their release date.

Figure 4.3 presents the selected models and their respective sizes, with the red line indicating the maximum number of parameters deployable on the available hardware for this study without quantizaton. It can also be observed that model sizes tend to cluster between the 6B and 9B marks. The complete list of models and their technical reports can be found in Appendix A

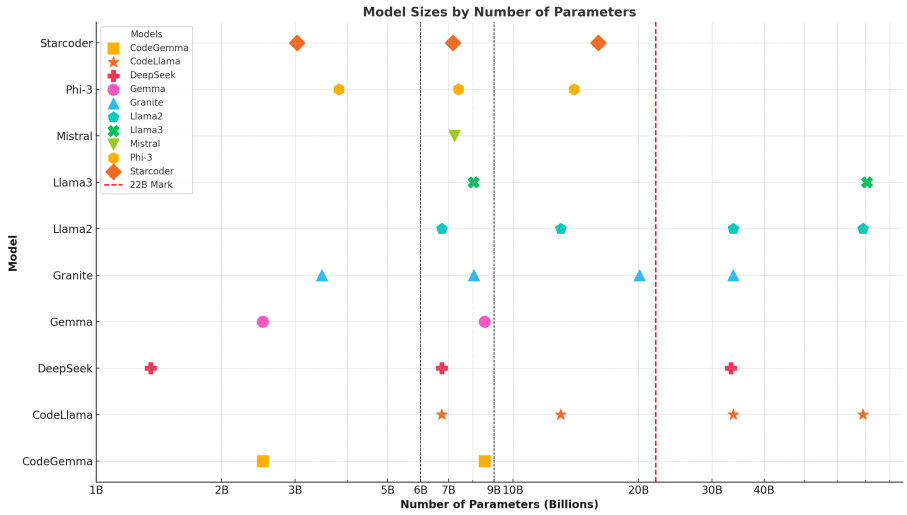


FIGURE 4.3: Models and their sizes. The red dashed line represents the demarcation line

4.2 Dataset selection

For selecting the datasets, aspects such as relevance to software development tasks, popularity as evaluation benchmarks, and the computational time required for completing inference are considered. Datasets evaluating code generation, bug fixing, and test generation capabilities will be included to mirror real-world use cases accurately. Additionally, datasets used in past research for evaluating and comparing model performance and accuracy will be considered, enhancing the credibility and contextual relevance of the results.

Selected datasets

Datasets were sourced from the technical papers of the selected models, ensuring consistency and credibility. The most popular benchmark among them was chosen based on past research and accuracy evaluations, thereby aligning with established standards and enhancing the reliability of comparisons.

Regarding **code generation** capabilities, it was found out that HumanEval from OpenAI is used to evaluate the models in every technical report [13]. Another was MBPP from Google, but results on it were not reported on Llama2 and Llama3[6]. Another reason is the size of MBPP compared to HumanEval, as it is almost 2.5 times larger, and it is not feasible for us to use it in the context of this project. An example of a HumanEval task for code generation can be seen in Figure 4.4. This is used as part of a larger prompt (the instruction part in Figure 4.7) to input into the model. To accommodate the rest of the tasks, such as bug fixing and documentation, an extended version of HumanEval is going to be used, namely HumanEvalPack, which was presented in the work of Muennighoff et al. [38].

For **bug/code fixing**, only the technical papers of Granite and StarCoder2 offered evaluation results [36, 32]. Conveniently, they included results for most of the models chosen for our study, with the exception of Llama2. We utilized the information from these papers for the other models to address this gap. In this context, HumanEvalFix from the HumanEvalPack was chosen (Figure 4.5).

For **documentation generation**, only Granite offered evaluation results. These re-

```
from typing import List

def has_close_elements(numbers: List[float], threshold
    : float) -> bool:
    """ Check if in given list of numbers, are any two
        numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0,
       2.0], 0.3)
    True
    """
```

FIGURE 4.4: The first function/task from the HumanEval dataset

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
    """
    Check if in given list of numbers, are any two numbers closer
       to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = elem - elem2
                if distance < threshold:
                    return True
    return False

def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3)
       == True
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05)
        == False
    assert ...
check(has_close_elements)
# Fix bugs in has_close_elements.
```

FIGURE 4.5: Same function constructed with a buggy solution in HumanEvalFix as per Muennighoff et al. [38]

sults were used to supplement the other models that served as baselines in the same paper, excluding Llama2. Consequently, HumanEvalExplain from the HumanEvalPack was selected for this task.

For **test generation**, no results could be obtained from the technical papers of the models. Other studies try to evaluate the models for test generations using the HumanEval dataset or others [28, 48]. They propose various ways of approaching this problem, but there is currently no standard set, e.g. code generation, on evaluating the LLMs for test generation. Therefore, we decided to use the docstring and the code, including the solution to it, to prompt the LLM to produce assertions for it. For the remaining code-related tasks, such as code reasoning and understanding, code execution, code completion, vulnerability

repair and code translation, results could not be obtained neither directly nor implicitly using other technical papers of the models under research, more specifically for Gemma, Llama3, Mistral and Llama2. Therefore, these tasks are not included in the scope.

4.3 Experiment setup

There are various LLM deployment scenarios in software development. It is important to explore all of them to provide an understanding of the energy implications of different platforms. Primary deployment scenarios include platforms such as the cloud or clusters, where users access LLMs via the network. Alternatively, the models can be deployed locally on edge devices, such as notebooks or desktop machines. The availability of sufficiently capable platforms also influences the choice of models for the study, as less resourceful platforms may not effectively support larger models.

4.3.1 Environment Evaluation and Selection

There are various environments where the experiments might take place. From past research on the topic, two solutions were identified: third-party cloud environment (Google Cloud or AWS) and local institutional clusters (EEMCS-HPC CLUSTER) [34, 47]. The former was rejected due to its monetary nature and the lack of available funding. The latter option was initially accepted, but due to insufficient hardware capabilities (GPUs) and its non-interactive batch nature, it was deemed not controllable enough for the experiments. A compromise was reached by using the Jupyter Lab cloud environment provided by the University of Twente, which allows for a more interactive approach and offers access to newer, more powerful NVIDIA GPUs without incurring any financial expense.

4.3.2 Hardware

In the chosen environment, there are three different models of NVIDIA GPUs - T4, A10, A16. The A10 model was selected due to its larger VRAM capacity and was based on the technical brief from NVIDIA - "GPU Positioning for Virtualized Compute and Graphics Workloads", which outlines the performance and cost-effectiveness of each GPU for different workloads [41]. From that report, it can be extrapolated that A16 is not a benchmark for AI inference because this is not its intended workload. T4 and A10 are benchmarked among other GPU as well, and the latter shows better performance metrics for that workload. Additionally, A10 is based on the new Ampere architecture compared to the older Turing architecture of T4, which makes it more efficient in AI inference tasks due to architectural improvements, including higher tensor core performance and better memory bandwidth. It is also the same architecture of the standard for deploying LLM's - A100 NVIDIA GPU. The chosen hardware is also more financially accessible to small and medium enterprises than the more expensive higher-grade GPUs.

4.3.3 Precision Trade-offs: BFloat16/Float16 vs. FP32 in Inference

For this study, the BFloat16 or Float16 precision standard will be applied for inference instead of the single-precision FP32. This will be the baseline of the experiments for several reasons:

Consistency with training precision: The models used in the experiments are trained using the precision of FP16 or BFloat16, therefore, to preserve consistency and minimize discrepancies from precision mismatches (Figure 4.6).

Computational efficiency: Models deployed using half-precision formats have significantly fewer memory requirements. This reduction in memory usage enables the deployment of larger and more accurate models, such as Granite 20B and StarCoder2 15B, expanding the scope of results in terms of model sizes and diversifying the model population. For example, by using the 'Model Memory Calculator' provided by HuggingFace, we can calculate the VRAM needed to deploy StarCoder2 15B [18]. The float32 single-precision needs 58.95 GB, while the bfloat16 half-precision needs 29.47 GB of VRAM, which is a significant difference.

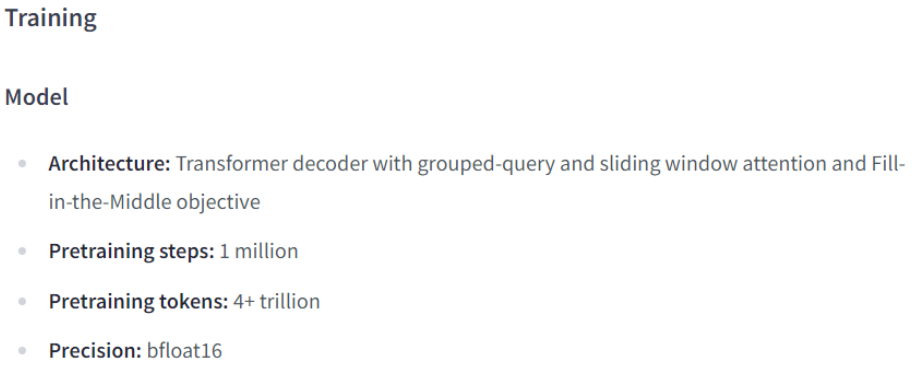


FIGURE 4.6: Training precision of StarCoder2 from its official HuggingFace page

Accuracy preservation: There is a concern that reducing the precision of the floating number operation from FP32 to FP16 or BFloat16 will affect the accuracy negatively. Studies have suggested that the difference in accuracy is negligible, and it is not considered critical [12, 26].

4.4 Experiment design

The experiment's design constitutes how the energy measurements would be taken and under what hyperparameters the models will be deployed. Given the utilized hardware of NVIDIA, `nvidia-smi` interface will be used to take measurements at a sample rate of 100ms. Hyperparameters, such as temperature and top-p, were configured to 0.1 and 0.95, respectively. This decision aligns the study with previous state-of-the-art research, including studies in the same domain, technical reports and their evaluations. Moreover, a lower temperature and high top-p result in more deterministic answers with minimal randomness, which is beneficial for coding tasks but less effective for essay writing.

To have more control over the configuration of the model, the base version is to be deployed so that it can be optimally tuned to closely represent the evaluation results on HumanEval (the pass@1 accuracy score). After initial experiments, it was found that a custom prompt would be needed to have more processable output for evaluating it using HumanEval. Initially, the output couldn't be processed to be passed on to the HumanEval automatic script for measuring pass@1. The prompts were curated according to past evaluations by research groups such as BigCode. Our prompt structure was inspired from the StarCoder2 and Code Generation LM Evaluation Harness repository, as they provided a lot of details and how they carried out the evaluation [43, 56, 7]. For each prompt, the maximum new token limit was adjusted according to the expected length of the answer, even above that limit. That was measured by doing a test run through the evaluation pipeline and measuring the average number of tokens the LLM produces for each task.

That was done because limiting the number of produced tokens would improve the post-processing of the answers to a variant ready for automatic evaluation and reduce the inference time, reducing the total time of the experiments. Those limits for code generation, bug fixing, docstring generation and test generation are 150, 150, 256, and 300 tokens. For test generation, 300 was chosen as a balance between getting complete and accurate answers and limiting the inference time, as models frequently produced many assertions. For more detailed look into the prompts used for the evaluation see Appendix B.

```
You are an exceptionally intelligent coding assistant that
   consistently delivers accurate and reliable responses to
   user instructions.

### Instruction
{instruction}

### Response
{response}
```

FIGURE 4.7: SC2_INSTRUCT_PROMPT used in our experiments.

4.5 Multi-Objective Optimization Using Pareto Frontier

In this study, we employ the Pareto frontier (or Pareto curve) for analyzing the final results [31]. This allows us to identify optimal models that balance competing objectives—maximizing model accuracy and minimizing energy usage. A model is deemed Pareto optimal if no other model dominates in both accuracy and energy usage.

To construct the Pareto front, the models are first sorted in ascending order based on the metric to be minimized (e.g., energy consumption). If two models have the same value for this metric, they are further sorted in descending order of the metric to be maximized (e.g., accuracy). Once sorted, the models are iteratively compared, starting from the lowest value of the minimizing metric. A model is included in the Pareto front if not dominated by any previously selected model, meaning its accuracy is greater than or equal to the current maximum observed accuracy. This process ensures that only the most efficient trade-offs between metrics are retained.

Algorithm 1 Pareto Frontier Identification

Input : Set of data points $D = \{(x_i, y_i) \mid i = 1, \ldots, n\}$ where x_i is to be minimized and y_i is to be maximized.

Output: Set of Pareto-optimal points P.

Step 1: Sort Points
Sort D in ascending order of x_i. If $x_i = x_j$, sort by descending order of y_i.

Step 2: Initialize
$P \leftarrow \emptyset$
$current\_max \leftarrow -\infty$

Step 3: Iterative Comparison
foreach $(x_i, y_i) \in D$ **do**
    if $y_i \geq current\_max$ **then**
        Add (x_i, y_i) to P
        $current\_max \leftarrow y_i$

return P

This approach was chosen over statistical tests, and a custom efficiency metric was defined, such as a ratio or score produced from the summed weighted accuracy and energy usage. In the former, the population sample from each type of model is unbalanced, e.g. there are only four fine-tuned models compared to 8 coding models (counting each different size of a model as separate). This sample imbalance might lead to unreliable conclusions as it can skew the data in favour of the larger group. Significant flaws were found in the latter approach. Firstly, the ratio metric values (e.g. energy/accuracy) can produce misleading results in edge cases. For example, consider two models where the first has an extremely low accuracy of 0.5 and energy of just 1 Wh will yield a ratio of 2. In contrast, the second model, with an accuracy of 50 and energy of 100 Wh, will yield a ratio of 2 again despite the second model being much more efficient and practical than the first. That is because ratios disproportionately favour small denominators, which can skew the results. Secondly, the efficiency score metric, which would be a weighted sum of both accuracy and energy usage, is flawed as well due to the possible incorporation of subjective bias when choosing the weights. For example, choosing a higher weight in acccuracy might prioritize models with excessive energy usage, making them impractical in resource-constrained environments. Conversely, higher weight on energy usage could undervalue more practical models. Additionally, a weighted sum assumes a fixed linear trade-off between accuracy and energy, treating every unit of energy saved as equally valuable regardless of the accuracy gain or loss. The Pareto frontier addresses the abovementioned issues by skipping assumptions of linear trade-off and subjective bias towards one of the metrics. It ensures that models not dominated by both metrics are considered optimal by adapting correspondingly to the complex relationships between the objectives.

Chapter 5

Results

This chapter presents the study's results aligned with the research questions. Each section presents the specific findings for the particular research question. It also contains informative and comparative charts. For section 5.1 we use the average for the carbon intensity of the power sector for the year 2023. We present it using three regions - Netherlands (268.5 gCO_2/KWh)[1], European union (662 gCO_2/KWh)[2] and World (481 gCO_2/KWh) [3]. The results can be found in more detailed form in Appendix C.

5.1 RQ1: What are the energy and carbon impacts of LLMs in software development?

To address the first research question, we present the average results from 3 runs for each tasks for all models. In Figure 5.1 The total energy used by each model to complete the HumanEval dataset for each task examined can be seen. In total $\approx$ 5377 Watts per hour were used to complete all tasks for all models and on average a model used 256 Wh to complete all four tasks. Furthermore, this means that a model on average spend 64 Wh per task (code generation, bug fixing, etc.) which is $\approx$ 0.39 Wh per inference.

The corresponding carbon emissions depend on the electricity grid's carbon intensity. Using regional emission factors, the total energy consumption results in approximately 1444 gCO_2 (Netherlands), 3560 gCO_2 (EU), and 2586 gCO_2 (World). On average, a model emits 68.7 gCO_2 (Netherlands), 169.5 gCO_2 (EU), and 123.1 gCO_2 (World) to complete all tasks. For a single task, the emissions amount to 17.2 gCO_2 (Netherlands), 42.4 gCO_2 (EU), and 30.8 gCO_2 (World). At the inference level, a single query produces 0.10 gCO_2 (Netherlands), 0.26 gCO_2 (EU), and 0.19 gCO_2 (World).

However, the tasks require various prompts with different lengths; therefore, a direct comparison of the net energy would not be fair. On Figure 5.2 it can be seen the net energy divided by the tokens outputted by the model for each task in Joules. On average, a model used $\approx$ 3.122 J to output a token given the data from all four tasks. This results in carbon emissions of 0.00023 gCO_2 (Netherlands), 0.00057 gCO_2 (EU), and 0.00042 gCO_2 (World) per token. Furthermore, because tokenizers differ slightly, where one tokenizer can divide a word into one token and another in two, a more suitable average would be per second

[1] "Statista: Carbon intensity of the power sector in the Netherlands from 2000 to 2023", https://www.statista.com/statistics/1290441/carbon-intensity-power-sector-netherlands/.

[2] "Statista: Carbon intensity of the power sector in the European Union in 2023", https://www.statista.com/statistics/1291750/carbon-intensity-power-sector-eu-country/.

[3] "Statista: Electricity generation emission intensity worldwide", https://www.statista.com/statistics/943137/global-emissions-intensity-power-sector-by-country/.

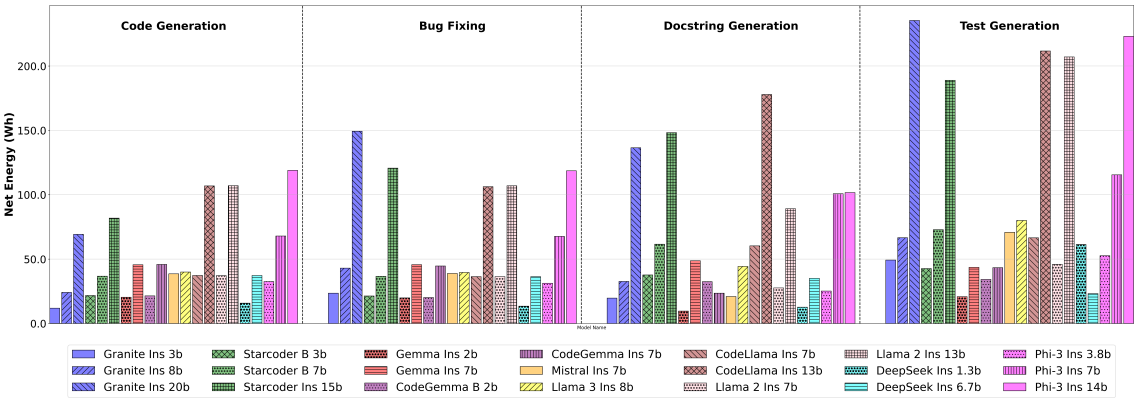


FIGURE 5.1: Net energy (Wh) for all models across the tasks examined.

of inference. On average a model used 184.50 Joules per second, leading to emissions of 0.0138 gCO_2 (Netherlands), 0.0339 gCO_2 (EU), and 0.0247 gCO_2 (World) per second.

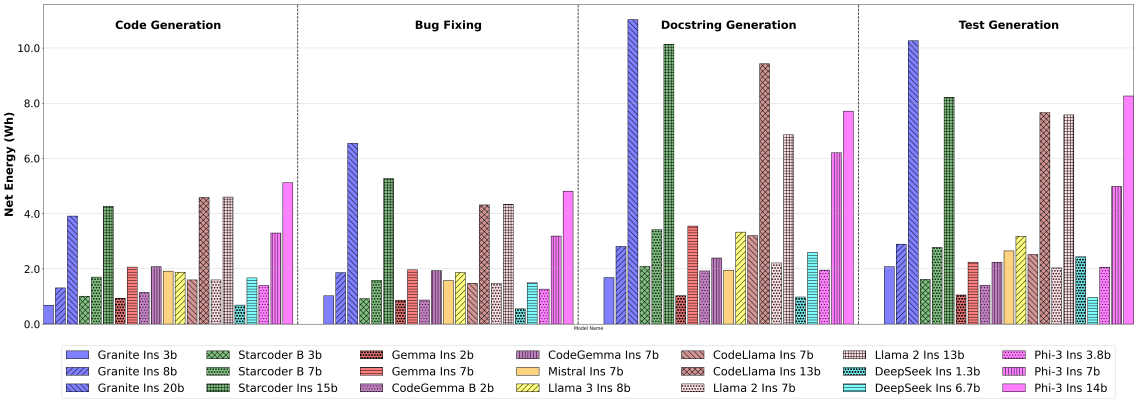


FIGURE 5.2: Energy per token in Joules for all models across the tasks examined.

5.2 RQ2: How do coding, fine-tuned, and general-purpose models compare in efficiency?

In this section, the comparison analysis results are presented using Pareto optimality. Additionally, quadrants are introduced and created from the mean energy and accuracy. This complements the Pareto analysis by offering a broader view of model performance relative to average energy and accuracy. It enables to identify models that are not Pareto optimal but still perform competitively. The models were divided into four categories: general (which includes the next category), general with twin fine-tuned, fine-tuned, and coding models. Comparing the two categories, one can determine which category dominates the rest in a software development context. The comparisons in this chapter section are coding vs. fine-tuned, fine-tuned vs. general with fine-tuned twin, fine-tuned vs. general, and general vs. coding vs. fine-tuned. The comparison will go through all four tasks: code generation, bug fixing, docstring, and test generation.

5.2.1 Coding versus fine-tuned

This subsection compares coding and a fine-tuned model using the Pareto frontier, mean energy, and accuracy to locate the most optimal quadrant. The comparisons cover the four tasks from software development examined in this study.

Code generation

The comparison of coding versus fine-tuned models using the Pareto frontier for code generation can be seen in Figure 5.3a. Coding models are solely dominate the Pareto front. Specifically, Granite Instruct 3b, DeepSeekCoder Instruct 1.3b and 6.7b. In the same figure, it can be seen that the top left quadrant, which signifies the most optimal models, is dominated by coding models. Additionally, above the average accuracy line, are two coding and one fine-tuned model, where the coding models are more accurate, but at the same time more energy intensive.

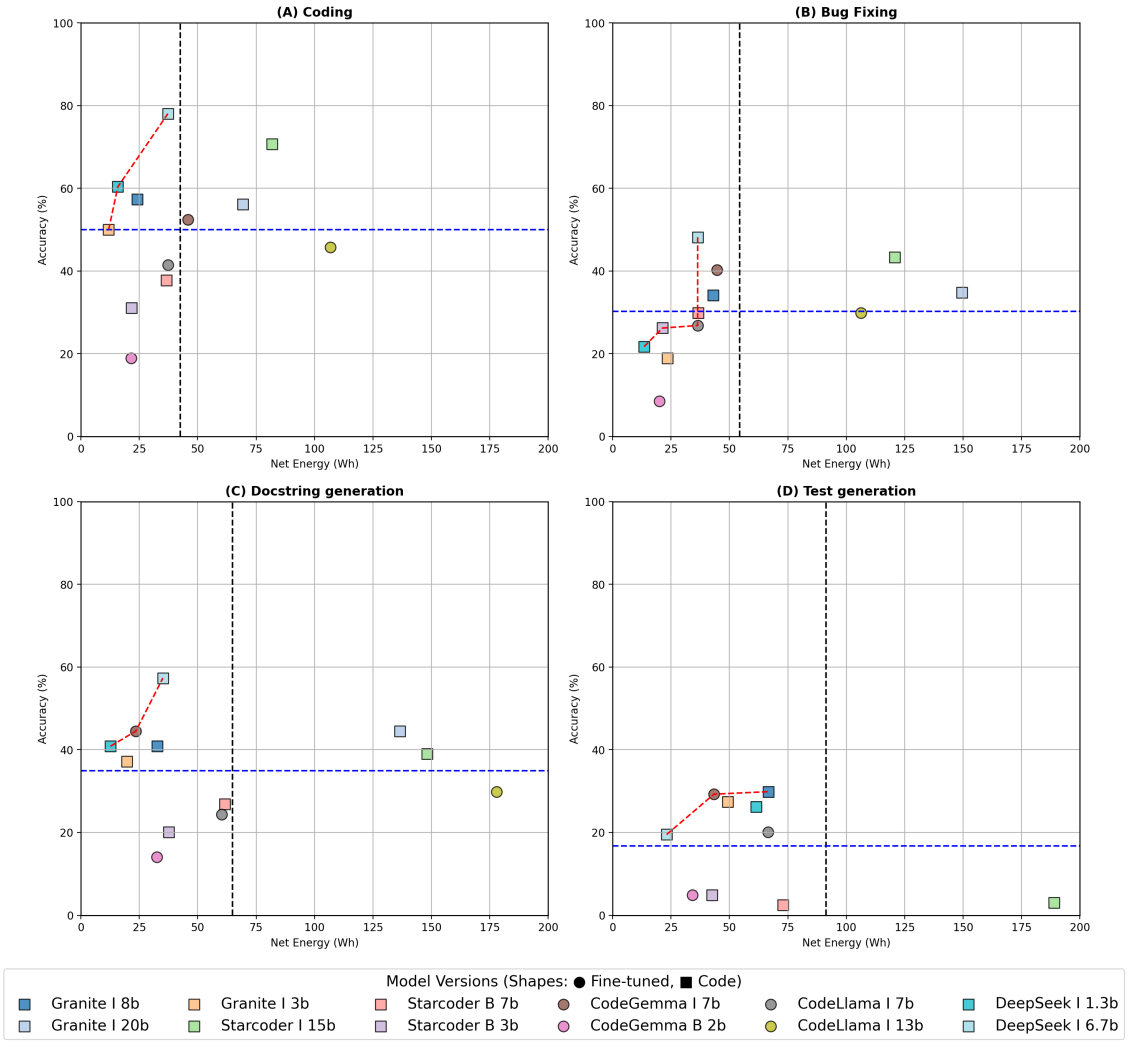


FIGURE 5.3: Coding vs fine-tuned models using Pareto optimality

Bug Fixing

Comparison of coding versus fine-tuned models for the task of bug fixing outlines three coding models (DeepSeekCoder 1.3B & 6.7B and StarCoder2 3B) on the Pareto frontier and one fine-tuned (CodeLlama 7B) on Figure 5.3b. One model falls short of the frontier, and that is StarCoder2 7B. Although it dominates over CodeLlama 7B in accuracy (29.87 vs 26.83 respectively), it uses slightly more energy (36.68 Wh compared to 36.39 Wh). Still, the Pareto frontier is mainly dominated by coding models, with 3 out of 4 from that group. Analyzing further the quadrants that we defined previously, where the top left quadrant is considered the most prominent in terms of efficiency, it can be observed that 2 out of 3 models are coding type. Starcoder2 7B is on the border of the top-left quadrant. The model with the highest accuracy is DeepSeekCoder 6.7B (coding), which is both on the optimal frontier and the top-left quadrant. At the same time, the second most accurate model is CodeGemma 7B (fine-tuned), which is not on the Pareto line but is inside the quadrant.

Docstring generation

In the context of docstring generation, where a model is tasked to produce a docstring for a given Python function and then use that docstring to generate the function again from the result of the previous step and measure the accuracy, there are three models on the optimal Pareto front (Figure 5.3c). From these models, 2 out of 3 are coding models, namely DeepSeekCoder 1.3B & 6.7B, while from the fine-tuned category, only CodeGemma 7B. Looking at the top-left quadrant, most models are coding models, while only one is fine-tuned.

Test generation

In Figure 5.3d, the results can be seen by comparing fine-tuned and coding models in the context of test generation. On average, the models are less accurate and generally use more energy. On the frontier, there are two-to-one coding and general models (DeepSeekCoder 6.7B & Granite 8B, CodeGemma 7B). The top-left quadrant is dominated by coding models, where two are also close to the efficient Pareto frontier - Granite 3B and DeepSeekCoder 1.3B.

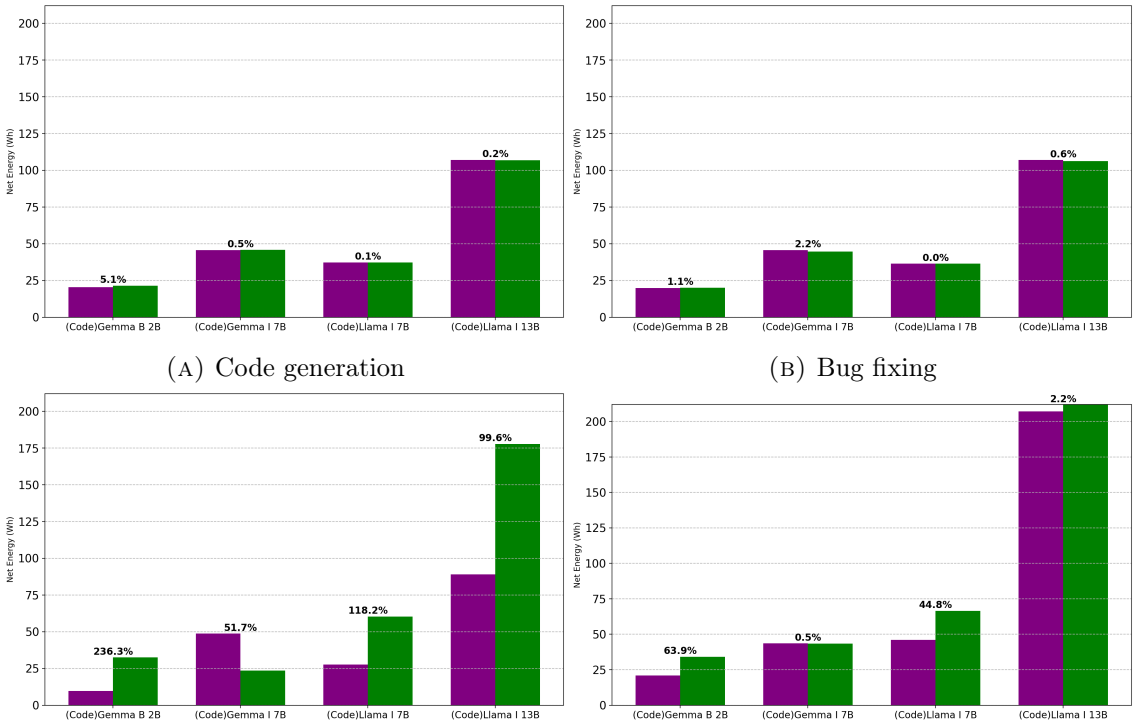
5.2.2 General with fine-tuned twin versus fine-tuned twin

This section will compare the models from the fine-tuned category, the general model with their fine-tuned "twin". The former is based on the latter, with the difference that it has been fine-tuned additionally for code. Instead of multi-objective optimization with Pareto frontier, a direct one-to-one comparison was conducted, as both categories have the same number of data points and consist of related model pairs.

Net energy comparison

The direct one-to-one comparison did not indicate an improvement in efficiency between the fine-tuned and the general version of a model in the code generation task (Figure 5.4a). The difference between net energy usage is minimal, below 1%, except for Gemma and CodeGemma 2b, where the latter used 5.1% more energy. However, this can be considered insignificant because of the small magnitude of the difference, which falls within the expected variability. Similarly, bug fixing in Figure 5.4b has equivalent insignificant

differences. In contrast to the trend, the generation of docstrings shows significant differences in net energy usage (Figure 5.4c). For example, for CodeGemma and Gemma 2B, the fine-tuned version has more than triple the energy impact. Significant differences can be observed in the CodeLlama and Llama models, where the code models again used double the amounts of energy. One outlier from the group is CodeGemma and Gemma 7B, where the general model uses slightly more than 50 % compared to its fine-tuned counterpart. For test generation in Figure 5.4d it can be seen that the larger models of the population (Code)Gemma 7B and (Code)Llama 13B tend to not have significant discrepancies between the fine-tuned and general models. On the other hand, the smaller versions 2B and 7B have respectively ≈ 64 % and 45 % difference, where the fine-tuned uses more energy for that task.



(A) Code generation

(B) Bug fixing

(C) Docstring generation

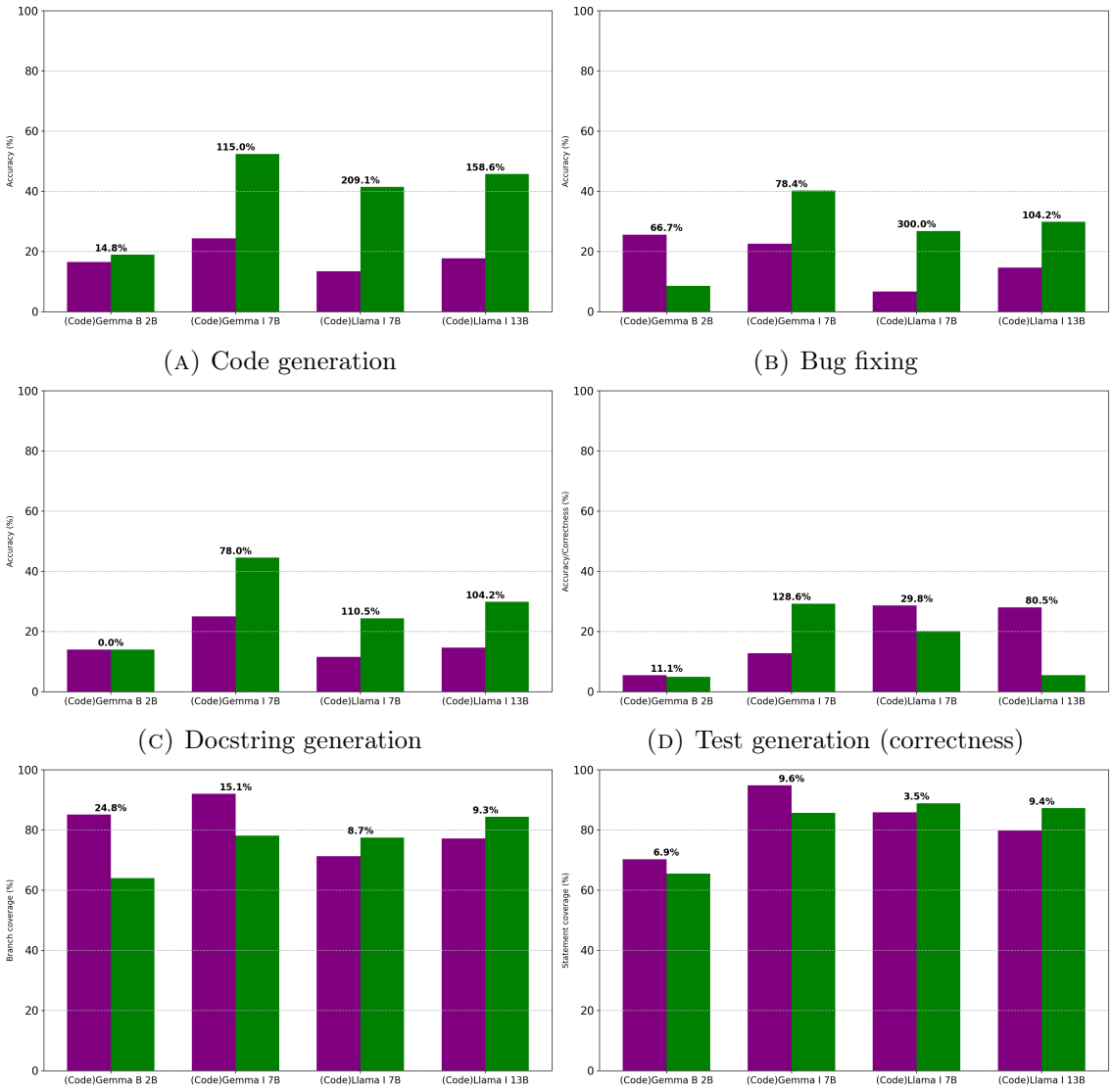
(D) Test generation

FIGURE 5.4: One-to-one comparison of general (**purple**) and their fine-tuned (**green**) version using net energy.

Accuracy comparison

However, when comparing the accuracies for code generation using a one-to-one comparison again, it can be observed for 3 of 4 models (Figure 5.5a). The fine-tuned models almost double or even triple the accuracy. Again, except for Gemma and CodeGemma 2B, the improvement was only with ≈ 15%. Again, this trend repeats for bug fixing and docstring where, in some cases, the accuracy even quadrupled (CodeLlama 7B and Llama 7B in bug fixing). The exception to this is with CodeGemma and Gemma 2B, where the accuracy improved in favour of the general model or remained equivalent (Figure 5.5b & 5.5c). The test generation results are represented using the correctness (how many are correct), branch and statement coverage of the produced test. Figure 5.5d shows that the Llama 2 family general models tend to be more accurate in generating tests. In contrast, for Gemma, the

fine-tuned model is outperforming only for the 7B version case. In the case of branch and statement coverage that can be observed in Figures 5.5e & 5.5f, it can be seen that the Gemma family general models outperform significantly (24.8 % & 15.1 %) in case of the former and with not so much significance (6.9 % & 9.6 %) in the case of the latter while for the Llama 2 family is the opposite way (fine-tuned model outperform) but with not that large of a difference - 8-9 %.



(A) Code generation

(B) Bug fixing

(C) Docstring generation

(D) Test generation (correctness)

(E) Test generation (branch coverage)

(F) Test generation (statement coverage)

FIGURE 5.5: One-to-one comparison of general (**purple**) and their fine-tuned (**green**) versions using accuracy/correctness

5.2.3 Fine-tuned versus general (all)

This subsection compares the fine-tuned models against all of the general models, including their general twin versions.

Code generation

Comparison of fine-tuned versus all general models outlines a broad Pareto frontier from the lowest < 20 % accuracy to the highest energy usage ≈ 120 Wh (Figure 5.6a). On the optimal front, there is only one fine-tuned model and five general, with one pair of models being a fine-tuned and general twin. Analysing further, the top left quadrant is shared equally between two fine-tuned and two general models, but the latter dominates accuracy and energy. Suppose we exclude impractical cases like Gemma and CodeGemma 2B from the Pareto frontier (due to their low accuracy). In that case, we can conclude that General models that are not twins to the fine-tuned dominate over fine-tuned.

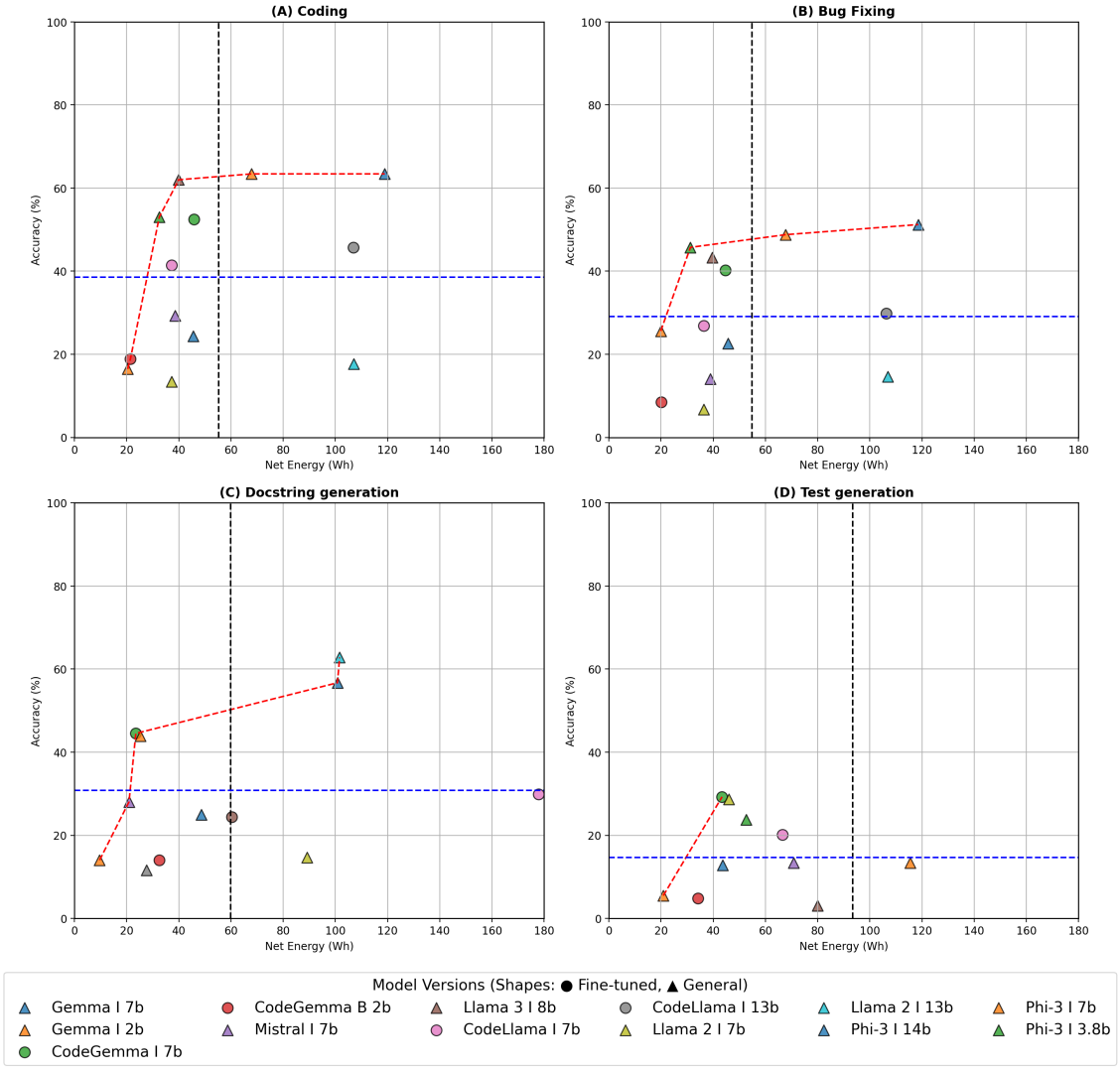


FIGURE 5.6: Fine-tuned vs general model using Pareto optimality

Bug Fixing

The comparison for bug fixing shown in Figure 5.6b indicates that general models are the most optimal for the Pareto frontier. From the Pareto models, 3 out of 4 are from one family, namely the Phi-3 family, and one model is Gemma 2B, which has a fine-tuned version. The top left quadrant contains most general models, with only one out of three

models being fine-tuned. Thus, for bug fixing, general models are the most efficient ones, whereas the majority of those models do not have fine-tuned versions.

Docstring generation

For docstring generation, the Pareto front is dominated by general models (4 out of 5 models), which can be seen in Figure 5.6c. However, two of those general models are below the mean accuracy threshold line, so if we exclude them due to their impracticality, the general models still dominate the frontier with 2 out of 3. Looking at the top-left quadrant, there are only two models - one general and one fine-tuned. Interestingly, these models have almost equivalent energy usage and accuracy (CodeGemma 7B & Phi-3 3.8B). Still, it can be concluded that general models without fine-tuned versions are the most optimal for docstring generation.

Test generation

In Figure 5.6d, it can be seen that for test generation, the Pareto frontier is shared between one general (Gemma 2B) and one fine-tuned (CodeGemma 7B). However, the former has very low test correctness, while the latter has the highest scores from the whole population. In the top-left quadrant, the space is divided again equally with one general model, namely Llama 2 7B, being second and almost overlapping with CodeGemma 7B (the highest accuracy). The next model in that quadrant is a general one followed by a fine-tuned one if we order them by accuracy.

5.2.4 Coding versus fine-tuned and general

In this subsection a comparison is made between all types of model: coding, fine-tuned and general.

Code Generation

Figure 5.7a shows the results from the comparison for code generation. There are only coding models on the Pareto line, namely Granite 3B, DeepSeekCoder 1.3B & 6.7B. In the top-left quadrant, 4 out of 7 models are coding (Pareto models plus Granite 8B), while two are general, namely Phi-3 3.8B and Llama 3 8B. Only one model in that same quadrant is fine-tuned - CodeGemma 7B. Another notable model that is neither on the Pareto line nor in the top left quadrant is StarCoder 15B (code model), which had the second highest accuracy.

Bug Fixing

Figure 5.7b is the analysis for bug fixing. Most models on the frontier are general ones, with 4 out of 7, while the rest are coding models. Three of the models, namely DeepSeek 1.3B (coding), Gemma 2B (general) & Starcoder2 3B (coding), are below the mean accuracy (30 %). The top left quadrant has two coding, two general and one fine-tuned model. One model that barely crosses into that same quadrant is StarCoder2 7B (coding). An outlier here, not in the quadrant but on the Pareto line, is Phi-3 14B, with the highest accuracy. The highest accuracies have two general along with one coding model, followed in second place with two general and one coding.

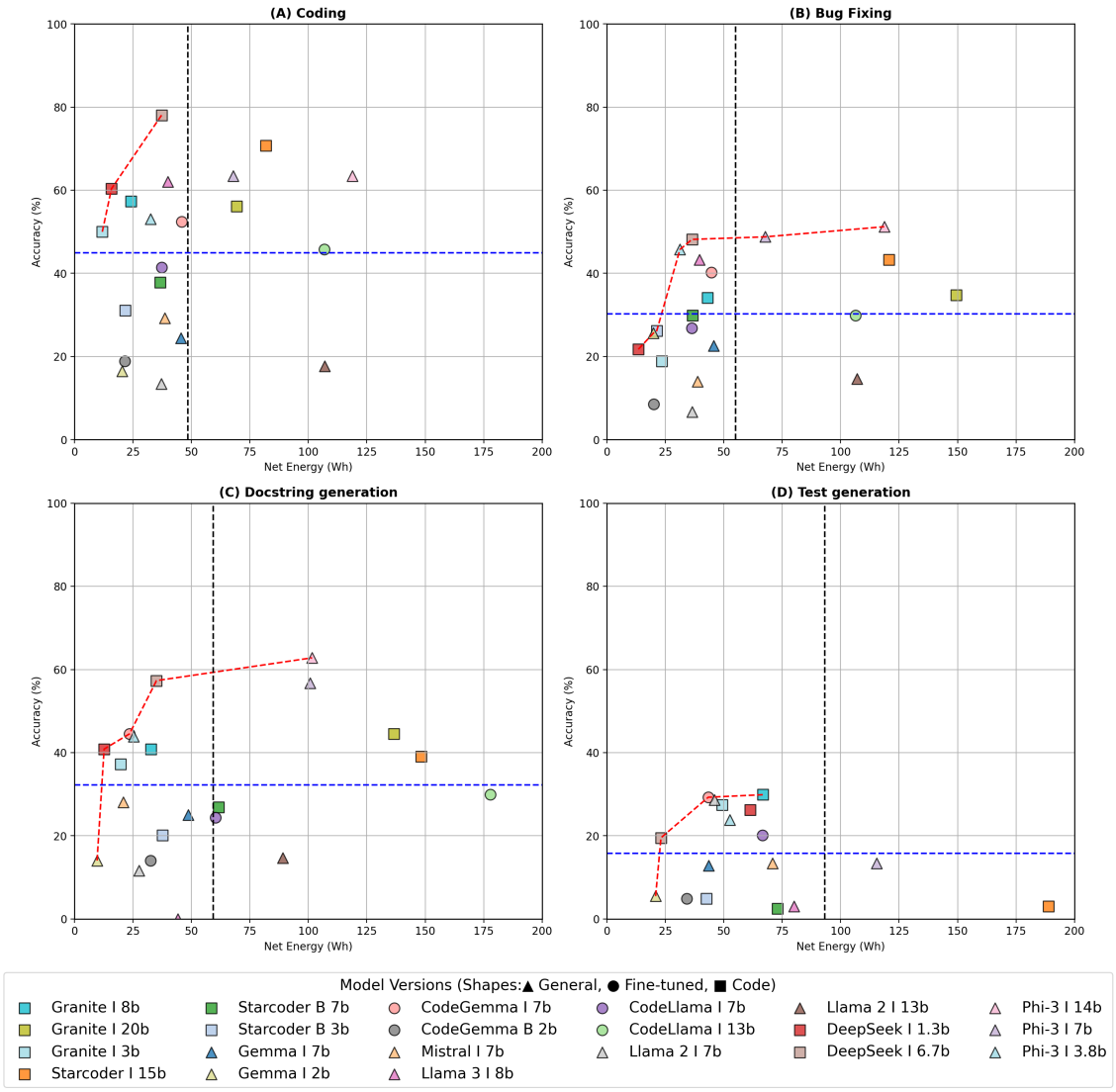


FIGURE 5.7: Coding vs fine-tuned vs general models using Pareto optimality

Docstring generation

Figure 5.7c presents the results for docstring generation. In this case, the Pareto line is slightly extended on the lower end, including a model below 20 % accuracy. On the Pareto front, there are two coding and two general models where only one is fine-tuned. One of the general models is on the lower end of the frontier, namely, Gemma 2B. One model from the general models (Phi-3 3.8B) almost overlaps with the Pareto fine-tuned model (CodeGemma 7B). On the other hand, in the top-left quadrant, we can identify six models, of which four belong to the coding category and one from the rest fine-tuned and general (the beforementioned models). One notable model not in that quadrant but on the Pareto frontier and has the highest accuracy is Phi-3 14B but with higher energy usage.

Test generation

Figure 5.7d shows the results for test generation tasks. On the Pareto frontier, there are two coding models (DeepSeekCoder 6.7B, Granite 8B), one general and one fine-tuned

(respectively Gemma 2B and CodeGemma 7B). However, the general one can be considered an impractical model as it is well below the mean accuracy for this task by only 5% correct answers. In the top-left quadrant, close to the Pareto front, Llama 2 7B (general) and two coding models, Granite 3B and DeepSeekCoder 1.3B, are situated. The rest of the models are general (Mistral 7B) and fine-tuned (CodeLlama 7B). Models which are neither Pareto nor in the 'efficient' quadrant but are one of the most accurate are Llama2 13B, which does not differ in accuracy significantly from it is smaller 7B variant and Granite 20B, which has slightly lower accuracy than it is smaller but more efficient variants 3B and 8B.

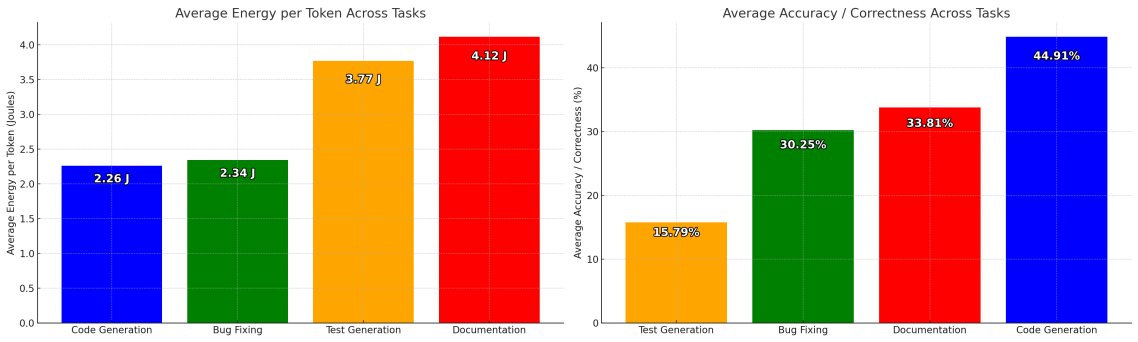
5.3 RQ3: How does energy consumption vary across different software development tasks?

For the third research question, the study will examine and compare the averages of the four tasks—code generation, bug fixing, docstring generation, and test generation. The averages include the energy per token in Joules and the tasks' accuracy or correctness (in the case of test generation). In this research question, we use energy per token because the tasks require prompts and outputs of different lengths (in tokens). In the first subsection, tasks will compared overall, including all the results. The following subsection breakdown in categories will be analysed: coding, general, and fine-tuned models.

5.3.1 Overall comparison

Comparing the tasks in energy per token and accuracy/correctness[4] can be seen in Figure 5.8. For energy per token in Figure 5.8a, it becomes clear that code generation and bug fixing are the least resourceful tasks, while docstring and test generation are almost twice as energy-intensive. In Figure 5.8b, code generation is the clear leader in accuracy, while bug fixing and docstring generation are next with similar results. Test generation has the lowest accuracy.

In order to find the most efficient task where accuracy is maximized and energy per token is minimized, a ranking approach is used. The task with the lowest energy is assigned the rank of 1, while the one with the highest is assigned 4. The opposite is done for ranking the tasks in accuracy, where the highest accuracy is assigned 1 and the lowest 4. The sum of the ranks provided in Table 5.1 indicates which task is the most efficient (lowest rank): code generation followed by bug fixing, docstring generation and test generation.



(A) Average nergy per token

(B) Average accuracy/correctness

FIGURE 5.8: One-to-one comparison of the tasks

[4]For test generation here the correctness are only taken into account.

TABLE 5.1: Task Performance and Energy Usage with Ranks

Task	Average Accuracy (%)	Average Energy (J)	Energy Rank	Accuracy Rank	Total Rank
Test Generation	15.79	3.77	3	4	7
Bug Fixing	30.25	2.34	2	3	5
Documentation	33.81	4.12	4	2	6
Code Generation	**44.91**	**2.26**	1	1	**2**

5.3.2 Breakdown on types: coding, general, finetuned

Table 5.2 presents the average breakdown into the model type for each task. From there, we can observe that general models are the most efficient for test and code generation (equal ranks), followed by bug fixing and docstring generation. On the other hand, coding models are the most efficient in code generation (rank of 2), while they are much less efficient in tasks such as bug fixing, docstring, and test generation. Finally, finetuned models fare best with code generation, while the score of bug fixing is almost equivalent, followed by docstring and test generation.

Model Type	Task	Avg. Energy per Token (J)	Avg. Accuracy (%)	Energy Rank	Accuracy Rank	Total Rank
General	Code Generation	3.10	15.55	2	2	**4**
	Bug Fixing	2.91	10.67	1	4	5
	Docstring Generation	4.54	13.11	4	3	7
	Test Generation	4.81	28.35	3	1	**4**
Coding	Code Generation	2.15	50.51	1	1	**2**
	Bug Fixing	2.87	31.20	2	3	5
	Docstring Generation	5.19	34.76	4	2	6
	Test Generation	4.64	15.85	3	4	7
Finetuned	Code Generation	2.35	39.63	2	1	**3**
	Bug Fixing	2.15	26.37	1	3	4
	Docstring Generation	4.24	28.20	4	2	6
	Test Generation	3.46	14.93	3	4	7

TABLE 5.2: Energy and Accuracy Metrics with Total Ranks for Different Model Types and Tasks

5.4 RQ4: What characteristics of the model influence energy use and efficiency?

Figure 5.9 a correlation matrix between five model characteristics and five performance metrics. The numbers in each cell represent the Spearman correlation; for example, -0.41 is the correlation between transformer blocks and GPU memory utilization. The matrix shows only coefficients with statistically significant p-values. To control the family-wise error rate, Bonferroni correction was applied by dividing the default alpha value (α=0.05) by the number of conducted tests n(n=25). The figure shows that accuracy is not significantly correlated to any of the model characteristics, while the feed-forward network size was not correlated with any of the performance metrics. Parameter count, or in other words, the model's size, does not correlate with the accuracy and memory utilization of the GPU. In contrast, it correlates positively with the total energy used (net energy(Wh)), elapsed time and Joules per token. Embedding size exhibits a similar pattern, showing no significant correlation with accuracy and GPU memory utilization while positively correlating with total energy consumption (net energy [Wh]), elapsed time, and energy per token (Joules per token). Transformer blocks (count) have a positive correlation with net energy (Wh), elapsed time (s), and Joules per token, while they are negatively correlated with GPU memory utilization. Attention Heads (count) show the same pattern of correlation.

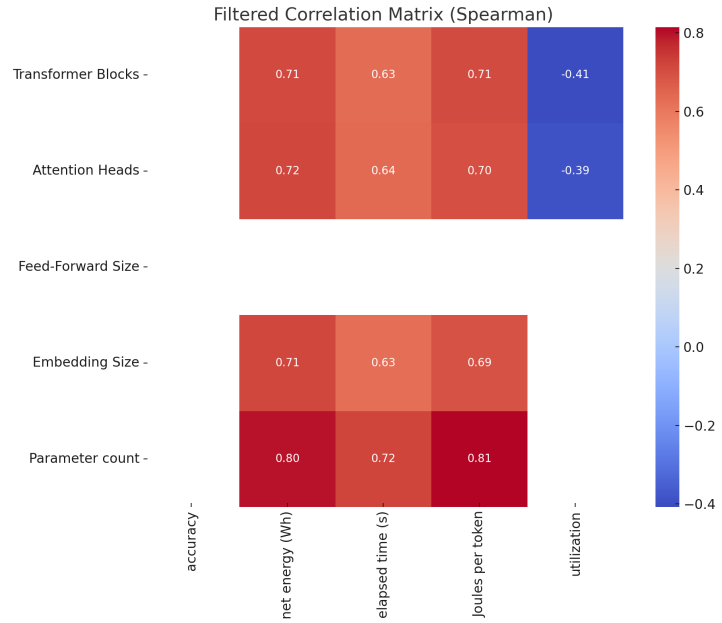


FIGURE 5.9: Spearman's correlation matrix for all models across all tasks

Chapter 6

Discussion

6.1 Energy in Action: Scaling LLMs for Real-World Code

The findings show that the energy usage of LLMs varies across different tasks and models. They provide a foundation for estimating the energy usage of LLMs at scale. Without considering the optimality or efficiency of the other models in this study, we can try to assess the implications in the real-world use case. Assuming a project of 100,000 LOC translates to 7,900,000 characters in total if we use PEP8[1] as a guideline for writing Python code and with a rough estimation of the total tokens that translates to (different models have slightly different tokenizers), that would be a 1,975,000 tokens project. Deploying an LLM solely for this project would end up using 6,165,950 Joules or ≈ 1,712.76 108 Wh, of course, assuming the ideal case of 100 % accuracy, where in reality, the average accuracy of models is much lower. If we take that into account as well, then developing this project might require 5,491.3 Wh, which is equivalent to ≈ 31.38 km driving with a Tesla Model S[2]. If we continue to scale further with a real-world project like the social media Instagram, estimated to have around 1 million LOC, it will take 54,913 Wh, which is 313.8 km with our Tesla Model S example. In carbon emissions, this would mean the release of 36.35 $kgCO_2$ in the environment.

6.2 Not All Models Are Created Equal

Even though that provides a rough baseline on what to expect from the average Large Language Model, it does not consider the full complexity of model performance. A deeper examination into the intricacies and trade-offs of different model types, coding, fine-tuned and general, reveals that the choice of model might significantly improve inference efficiency.

When comparing the coding and fine-tuned types, model types are inherently expected to excel in software development tasks. Coding models dominated over the fine-tuned for all four tasks. For code generation, code models dominated the whole spectrum by being the only ones on the Pareto frontier, top-left quadrant and most models with accuracy higher than the mean (50 %). Coding models maintain their dominance for bug fixing, docstring and test generation, with some fine-tuned models becoming more competitive, like CodeGemma 7B. These findings emphasize that coding models, designed specifically for software tasks, generally outperform fine-tuned models.

[1] PEP 8 – Style Guide for Python Code, https://peps.python.org/pep-0008/#maximum-line-length.
[2] Tesla. *Tesla Support*, https://www.tesla.com/en_ie/support/power-consumption.

However, in the following comparisons between the fine-tuned models and their general counterparts, the former showed that in code generation and bug fixing, it uses almost the same energy as the former but most of the time with significant improvement in accuracy. Docstring was the only task in which the fine-tuned versions used significantly more energy than the general, even though they still provided improvement in accuracy. Test generation provided mixed results, wherein only one case did a fine-tuned model provide meaningful improvement and for the rest, the general models outperformed. The additional training proved to improve model performance in accuracy while not increasing the energy usage except for docstring generation, where there is an obvious trade-off of energy for accuracy in the task.

On the other hand, fine-tuned models proved to be less performant when compared to other general models. General models consistently outperform those models on both the Pareto frontier and top-left quadrant and have the highest accuracy. Interestingly, in all cases, models from the Phi-3 family are the reason for that result. Another model, Llama 3 8B, also outperforms fine-tuned models in code generation and bug fixing. Both model families are newer based on previous iterations, such as Phi-2.5 and Llama 2. However, test generation is an exception, where a fine-tuned model (CodeGemma 7B) achieves the highest accuracy, suggesting that fine-tuning may still be beneficial in certain scenarios despite its overall inefficiency. Yet, Llama 7B scores closely to these results, suggesting that fine-tuning is not quite outcompeting general models again.

Finally, the analysis, including all types of models: coding, fine-tuned, and general, confirmed the leading spot in efficiency for coding models in the task of code generation, followed by the aforementioned top performers from the general models. Coding models are becoming less efficient for tasks such as bug fixing than general models. The general models are more performant in that task. In docstring generation, coding models dominated again, even though some general models from the Phi-3 family showed competitive performance, with Phi-3 14B being the most accurate for that task. This challenges the assumption that coding models might be generally better, but it is also worth considering that the predecessors of Phi-3 were focused on code, so it might be that the same datasets are included in its training. For test generation, coding models generally outperformed other models, but there were still outliers from other groups competing closely with them. The results highlight that assuming that one type of model is generally superior is incorrect; on the contrary, for each task, there is a suitable model or one that is at least as good as the other.

The pairwise comparison showed that while highly effective and efficient for code generation, coding models may not be the optimal choice when addressing a broader range of software development tasks. They provide ready-to-use solutions in use cases where code generation is the focus but are not versatile enough if the range of tasks expands. General models can be a better choice for cases where the scope of tasks is more extensive, offering strong baseline performance. Furthermore, in such cases, fine-tuning can improve these baseline capabilities significantly without trade-offs in energy usage most of the time.

6.3 Context matters: Tasks aren't equal as well

Still, choosing the right type of model does not exhaust the tale of efficiency. The context in which the model is deployed also matters, which, for this study, means one of the four tasks.

Code generation emerged as the most efficient task, using the least average energy and having the highest accuracy. This can be attributed to the straightforward nature

and predictability of coding patterns, which likely reduce the computational complexity. Another reason might be the availability of high-quality datasets for code generation and focus during training on that specific task, as this is the most prominent task in software development.

Next was bug fixing, a task in which the model should generate code in a different context. Almost the same energy was used, similar to code generation, but the output was less accurate. This can be attributed to the models being focused mainly on completing or predicting the next token. Bug fixing, however, involves additional complexity: identifying the location of the bug, repairing the problematic code, while preserving the surrounding prefix and suffix, and reasoning about correct code execution with potential inputs and outputs (at least that is how it works in the human case).

Docstring generation is slightly higher in accuracy than bug fixing but twice as energy-intensive compared to bug fixing and code generation. Still, evaluating the docstring has limitations, such as being limited by the understanding of the underlying model and its coding capabilities ("eat your own food" approach). The best way to evaluate it is by human experts. Nevertheless, this shows that models struggle with generating sensible docstrings and use significantly more energy than the other tasks. Lastly, test generation proved to be the least efficient task due to its low accuracy/correctness. This was assumed to be due to the lack of high-quality datasets that contain training data related to producing tests.

When analyzing the breakdown by model type, code generation consistently emerges as the most efficient task (alongside test generation for general models), with coding models exhibiting the lowest energy usage per token and the highest accuracy. Similarly, docstring generation consistently stands out as the most energy-intensive task, achieving higher accuracy than bug fixing but ranking among the least efficient tasks alongside test generation, except in the case of general models, where bug fixing and docstring generation are the least efficient.

These findings underscore the need for creating high-volume, high-quality datasets to improve models' capabilities for tasks such as docstring generation and test generation and address their current limitations. Another implication is when it comes to deploying such models for real-world use cases, where they would be more valuable and efficient for code generation and bug fixing.

Finally, if we consider the findings in the context of developing an application like Instagram again and taking into account the average accuracy of coding models (50.51 %) in code generation, this would result in 23,590.28 Wh of energy, almost two and a half times (2.32 times exactly) less than 54,913 Wh or equivalently ≈135 km with the Tesla Model S.

6.4 Breaking the Myth: Bigger Isn't Always Better

Analysing further using the Spearman correlation also gave us interesting insights into what model characteristics might affect the performance metrics. Accuracy, one of the most important metrics in LLM, proved to be uncorrelated with any of the characteristics, which means that no matter the size (parameter count) or depth of the model (transformer blocks), it can improve the accuracy of answers. We must assume that this is purely a training and dataset problem. Therefore, a larger or deeper model does not guarantee superior performance. Increasing the size of the models decreases their energy efficiency, while utilization is unaffected. This indicates that larger models consume more energy without effectively leveraging additional resources for improved performance. Similarly,

increasing the embedding size, which would allow a model to capture more nuanced and complex patterns, does not affect accuracy or GPU memory utilization. However, it increases the inference time and energy consumption. Transformer blocks and attention heads count exhibit similar correlations as they are interconnected, with the former representing the depth of the model and the latter indicating the breadth of each layer. More attention heads mean additional focus points, resulting in greater computational work at each layer. Increasing the depth and breadth of a model does not increase its accuracy, and it affects energy efficiency negatively by increasing energy per token and energy in total. Additionally, it reduces GPU memory utilization, making it less resource-efficient.

In summary, larger and deeper models inherently require more energy and computational time, while accuracy does not necessarily improve. This shows the importance of training and quality datasets over the size of the model.

Chapter 7

Threats, Conclusions & Future Work

7.1 Threats to validity

A potential threat to the validity of the results is using uniform prompts across all models. This might result in differences from the evaluation in the models' technical reports. The chosen prompt might be suboptimal for some models, but it was done to facilitate a fair comparison between models. Moreover, the outputs from the models required post-processing steps, which were dependent on the structure of the answer. In some cases, a model might not end their answer properly with the expected keywords, which could result in incorrect evaluation and negatively affect accuracy. To bypass that, a semi-manual inspection was carried out on the results to confirm missed function implementations or docstrings further. Another potential limitation of the study is the usage of one particular dataset - HumanEval. HumanEval might not represent the complexity of real-world software development tasks to the best of its ability or fully grasp their versatility. This, along with the fact that the study was carried out in the context of one programming language, Python, might impact the generalizability of the results. Additionally, the transformers library provided by HuggingFace was used for all models deployment. It configurations, optimizations, etc. might not fully represent the most efficient deployment for some models. Variability in library implementations might affect the generalizability of the results. Moreover, the underlying hardware may have influenced model performance, as more advanced hardware could enable more optimal outcomes. Finally, during the course of this study, many new models and versions of models, which were included in the scope. However, adding these new models would significantly expand the scope. Therefore, this study provides a snapshot of a specific period. For example, model selection ended in August 2024, and the last experiments were conducted in September 2024.

7.2 Conclusions

This work explored the energy consumption, performance, and efficiency of large language models in software development, specifically the four core tasks of successfully producing a working software product: code generation, bug fixing, docstring generation, and test generation. It provided insights into the trade-offs between the different types of models and tasks. It explored the relationship between model size and architectural characteristics and their impact on energy efficiency and accuracy.

The findings show that Large Language models have substantial energy demands when deployed to real-word use cases. As usage and deployment scale, it becomes important to be able to choose energy-efficient models and define optimization strategies to mitigate exces-

sive energy consumption and carbon footprint. Additionally, it was found that the model's energy usage varies significantly across tasks and models, underscoring the importance of selecting the models according to the context in which they are deployed. Moreover, the findings showed that coding models outperform general models only for code, while the latter showed more versatility in expanding the scope of software tasks. Furthermore, fine-tuning a general model significantly improves the efficacy across all tasks, which shows promising research and development for future models. Finally, we found that model accuracy does not increase with a model's overall size or depth. This results in a poor trade-off where the accuracy increase is not proportional to the energy increase. Therefore, smaller models are more efficient and a better choice than large ones.

In conclusion, improving the efficiency of LLMs requires selecting the right model for the right task, curating high-quality datasets, that encompass a broader range of software development activities, and optimizing model architectures for better energy efficiency and performance.

7.3 Future work

Future work could involve cross-validating the model's performance across different libraries, such as transformers, Ollama, Triton, ONNX, etc. This would help identify variations in different deployment scenarios where a library/framework might provide further optimizations or limitations to the model performance.

Another potential venue involves deploying the models on different GPUs and architectures, such as NVIDIA A100, L40, etc., or AMD alternatives to investigate the differences and trade-offs in accuracy and energy usage. This can provide valuable insights into how model-hardware interplays and optimize further model deployments.

Finally, a promising area of research is to deploy the models in a real-world scenario. This can involve selecting the already identified performant models and deploying them in an organization for employees to use. This can solve the limitations of using datasets like HumanEval, which does not fully grasp the prompts that could be served to an LLM in a software development context. Real-world deployment would provide a more comprehensive evaluation of the models' effectiveness, usability, and energy efficiency under practical conditions, offering insights beyond controlled experimental settings.

Bibliography

[1] Hugging face – the ai community building the future. https://huggingface.co/.

[2] Developer sentiment around ai/ml, March 2023. URL: https://stackoverflow.co/labs/developer-sentiment-ai-ml/.

[3] Greenhouse gas emissions from a typical passenger vehicle. https://www.epa.gov/greenvehicles/greenhouse-gas-emissions-typical-passenger-vehicle, 2023.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[5] Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. Language models in software development tasks: An experimental analysis of energy and accuracy. *arXiv preprint arXiv:2412.00329*, 2024.

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[7] Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

[8] Y. Bengio. Learning deep architectures for ai. *Foundations*, 2:1–55, 01 2009. doi:10.1561/2200000006.

[9] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Centraal Bureau voor de Statistiek. Energy consumption private dwellings; type of dwelling and regions. https://www.cbs.nl/en-gb/figures/detail/81528ENG, 2023.

[12] Aditya Chakravarty. Deep learning models in speech recognition: Measuring gpu energy consumption, impact of noise and model quantization for edge deployment. *arXiv preprint arXiv:2405.01004*, 2024.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[14] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. Trends in ai inference energy consumption: Beyond the performance-vs-parameter laws of deep learning. *Sustainable Computing: Informatics and Systems*, 38:100857, 2023.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[17] Exponential View. Adoption rate for major milestone internet-of-things services and technology in 2022, in days [graph]. https://www.statista.com/statistics/1360613/adoption-rate-of-major-iot-tech/, 2022. Accessed: December 12, 2022.

[18] Hugging Face. Model memory utility. https://huggingface.co/spaces/hf-accelerate/model-memory-usage, 2024.

[19] Ahmad Faiz, Sotaro Kaneda, Ruhan Wang, Rita Osi, Prateek Sharma, Fan Chen, and Lei Jiang. Llmcarbon: Modeling the end-to-end carbon footprint of large language models. *arXiv preprint arXiv:2309.14393*, 2023.

[20] GitHub. Survey reveals ai's impact on the developer experience, June 2023. URL: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/.

[21] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[22] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024. arXiv:2308.10620.

[23] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. pages 328–339, 01 2018. doi:10.18653/v1/P18-1031.

[24] Sommerville Ian. Software engineering tenth edition, 2016.

[25] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[26] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[28] Kefan Li and Yuan Yuan. Large language models as test case generators: Performance evaluation and enhancement. *arXiv preprint arXiv:2404.13340*, 2024.

[29] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: **phi-1.5** technical report. *arXiv preprint arXiv:2309.05463*, 2023.

[30] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[31] Alexander V Lotov and Kaisa Miettinen. Visualizing the pareto frontier. In *Multiobjective optimization: interactive and evolutionary approaches*, pages 213–243. Springer, 2008.

[32] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[33] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment? *arXiv preprint arXiv:2311.16863*, 2023.

[34] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of Machine Learning Research*, 24(253):1–15, 2023.

[35] Meta. Introducing code llama, a state-of-the-art large language model for coding, 2023. URL: https://ai.meta.com/blog/code-llama-large-language-model-coding/.

[36] Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*, 2024.

[37] Mistral AI. Mistral ai homepage, 2023. URL: https://mistral.ai/.

[38] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

[39] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[40] Shakked Noy and Whitney Zhang. Experimental evidence on the productivity effects of generative artificial intelligence. *Science*, 381(6654):187–192, 2023. URL: https://www.science.org/doi/abs/10.1126/science.adh2586, arXiv:https://www.science.org/doi/pdf/10.1126/science.adh2586, doi:10.1126/science.adh2586.

[41] NVIDIA Corporation. Gpu positioning for virtualized compute and graphics workloads: Selecting the right gpu for your virtualized workload. Technical brief, NVIDIA Corporation, 2023. URL: https://images.nvidia.com/data-center/technical-brief-gpu-positioning-virtualized-compute-and-graphics-workloads.pdf.

[42] OpenAI. Openai codex, 2021. URL: https://openai.com/blog/openai-codex/.

[43] BigCode project. StarCoder2-Instruct: Fully Transparent and Permissive Self-Alignment for Code Generation. https://github.com/bigcode-project/starcoder2-self-align, 2024.

[44] Rackspace. Stage of generative artificial intelligence (ai) implementation within businesses in 2023 [graph], 2023. URL: https://www.statista.com/statistics/1447897/stages-of-generative-ai-implementation-in-business/.

[45] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[46] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[47] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2023.

[48] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[49] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[50] Rico Sennrich. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[51] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1017–1024, 2011.

[52] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[53] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal

Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[55] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[56] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct, 2024. URL: https://arxiv.org/abs/2312.02120, arXiv:2312.02120.

[57] Burak Yetiştiren, Eray Tüzün, and Işık Özsoy. Assessing the quality of github copilot's code generation. 11 2022. doi:10.1145/3558489.3559072.

[58] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*, 2023.

[59] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2023.

[60] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 21–29, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3520312.3534864.

Appendix A

Models

Name	Size (in B)	Version	Repository ID on HF	Technical Report
StarCoder 2	3.0	2.0	bigcode/starcoder2-3b	[32]
-	7.0	-	bigcode/starcoder2-7b	-
-	15.0	-	bigcode/starcoder2-15b-instruct-v0.1	-
DeepSeek Coder	1.3	1.0	deepseek-coder-1.3b-instruct	[21]
-	6.7	-	deepseek-coder-1.3b-instruct	-
Gemma	2.0	1.0	google/gemma-2b	[53]
-	7.0	-	google/gemma-7b-it	-
CodeGemma	2.0	1.0	google/codegemma-2b-it	-
-	7.0	-	google/codegemma-7b-it	-
Granite Code	3.0	1.0	ibm-granite/granite-3b-code-instruct	[36]
-	8.0	-	ibm-granite/granite-8b-code-instruct	
-	20.0	-	ibm-granite/granite-20b-code-instruct	-
Llama 3	8.0	3.0	meta-llama/Meta-Llama-3-8B-Instruct	[16]
Llama 2	7.0	2.0	meta-llama/Llama-2-7b-chat-hf	[54]
-	13.0	-	meta-llama/Llama-2-13b-chat-hf	-
CodeLlama	7.0	1.0	meta-llama/CodeLlama-7b-Instruct-hf	[46]
-	13.0	-	meta-llama/CodeLlama-13b-Instruct-hf	-
Phi-3	3.8	3.0	microsoft/Phi-3-mini-4k-instruct	[29]
-	7.0	-	microsoft/Phi-3-small-8k-instruct	-
-	14.0	-	microsoft/Phi-3-medium-4k-instruct	-
Mistral	7.0	0.1	mistralai/Mistral-7B-Instruct-v0.1	[37]

TABLE A.1: Models used in the study

Appendix B

Prompts

```
"""You are an exceptionally intelligent coding assistant that
   consistently delivers accurate and reliable responses to user
   instructions.
### Instruction
Write a Python function to solve the given task:"""
```python

from typing import List
def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
 """ Check if in given list of numbers, are any two numbers closer
    to each other than given threshold. >>> has_close_elements
    ([1.0, 2.0, 3.0], 0.5)    False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)   True
   """
```
"""### Response"""
```python
from typing import List
def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
 """ Check if in given list of numbers, are any two numbers closer
    to each other than given threshold. >>> has_close_elements
    ([1.0, 2.0, 3.0], 0.5)    False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)   True
   """
```

FIGURE B.1: Prompt used for code generation
```

```
"""You are an exceptionally intelligent coding assistant that
   consistently delivers accurate and reliable responses to user
   instructions.
###Instruction
Fix the bugs in the given task:"""
```python
from typing import List
def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
      for idx, elem in enumerate(numbers):
          for idx2, elem2 in enumerate(numbers):
              if idx != idx2:
                  distance = elem - elem2
                  if distance < threshold:
                      return True
       return False
def check(has_close_elements):
       assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2],
          0.3) == True
       assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2],
          0.05) == False
       assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95)
          == True
       assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) ==
           False
       assert has_close_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0],
          0.1) == True
       assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) ==
           True
       assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) ==
           False
check(has_close_elements)
```
"""###Response"""
```python
from typing import List
def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
```
```

FIGURE B.2: Prompt used for bug fixing

```
"""You are an exceptionally intelligent coding assistant that
   consistently delivers accurate and reliable responses to user
   instructions.
### Instruction
Provide a concise natural language description of the function
   using at most 213 characters. No code, only docstring."""
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
      for idx, elem in enumerate(numbers):
          for idx2, elem2 in enumerate(numbers):
              if idx != idx2:
                distance = abs(elem - elem2)
                 if distance < threshold:
                      return True
        return False
```
"""### Response
Sure, here is the docstring of the function:
\"\"\"
"""
```

FIGURE B.3: Prompt used for docstring generation

```
"""You are an exceptionally intelligent coding assistant that
   consistently delivers accurate and reliable responses to user
   instructions.
### Instruction
Generate a number of assertions for the following Python function:
   """
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) ->
   bool:
    """ Check if in given list of numbers, are any two numbers
       closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
       True """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False
```
"""### Response
Sure, here are the assertions:"""
assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) ==
   True
assert
```

FIGURE B.4: Prompt used for test generation

Appendix C

Data

Model version	Net Energy (Wh)				Accuracy (0-100)			
	code generation	bug fixing	docstring generation	test generation	code generation	bug fixing	docstring generation	test generation
Granite Instruct 8b	24.17489512	43.0093067	32.63000863	66.663	57.31707317	34.14634146	40.85365854	29.87
Granite Instruct 20b	69.28621455	149.4153794	136.5452015	235.4269392	56.09756098	34.75609756	44.51219512	27.43
Granite Instruct 3b	11.89454785	23.52750703	19.71109012	49.234	50	18.90243902	37.19512195	27.43
Starcoder Instruct 15b	81.87624397	120.7194031	148.2261201	188.9208369	70.73170732	43.29268293	39.02439024	3.04
Starcoder Base 7b	36.69989645	36.67828591	61.60805651	72.909	37.8	29.87804878	26.82926829	2.43
Starcoder Base 3b	21.72962423	21.36081562	37.65765126	42.621	31.1	26.2195122	20.12195122	4.87
Gemma Instruct 7b	45.59114942	45.67050693	48.70008147	43.607	24.3902439	22.56097561	25	12.8
Gemma Instruct 2b	20.40975039	19.88996541	9.655239393	20.861	16.46341463	25.6097561	14.02439024	5.48
CodeGemma Instruct 7b	45.82735338	44.673628	23.5271141	43.378	52.43902439	40.24390244	44.51219512	29.26
CodeGemma Base 2b	21.46032723	20.10983619	32.47232163	34.196	18.9	8.536585366	14.02439024	4.87
Mistral Instruct 7b	38.60648174	38.81854665	20.91772614	70.787	29.26829268	14.02439024	28.04878049	13.41
Llama 3 Instruct 8b	39.97664413	39.59375871	44.27539593	80.016	62	43.29268293	-	3.04
CodeLlama Instruct 7b	37.25617721	36.39265213	60.33147283	66.551	41.46341463	26.82926829	24.3902439	20.12
CodeLlama Instruct 13b	106.8386744	106.3009683	177.844116	211.6721158	45.73170732	29.87804878	29.87804878	5.48
Llama 2 Instruct 7b	37.20401372	36.40957056	27.6471476	45.95	13.41463415	6.707317073	11.58536585	28.65
Llama 2 Instruct 13b	107.0205267	106.9420922	89.11395357	207.0458329	17.68292683	14.63414634	14.63414634	28.04
DeepSeek Instruct 1.3b	15.82529399	13.48869557	12.67345212	61.358	60.36585366	21.73170732	40.85365854	26.21
DeepSeek Instruct 6.7b	37.24320432	36.40892399	35.07584102	23.148	78.04878049	48.17073171	57.31707317	19.51
Phi-3 Instruct 14b	118.9238602	118.5994719	101.6319049	223.0481285	63.4146	51.2195122	62.80487805	2.43
Phi-3 Instruct 7b	67.90343695	67.68818819	100.7889995	115.46	63.4146	48.7804878	56.70731707	13.41
Phi-3 Instruct 3.8b	32.53580813	31.2501674	25.22170093	52.626	53.0487	45.73170732	43.90243902	23.78

Table C.1: The raw data of the results