Évaluation Empirique de l'Impact Énergétique des Large Language Models pour la Génération et l'Optimisation de Code

Tristan COIGNION

13 Novembre 2025
Informatique et applications

Université de Lille & Inria
Laboratoire CRIStAL
École doctorale MADIS

Directeurs de la thèse:
Romain Rouvoy - Professeur à l'Université de Lille
Clément Quinton - Maître de Conférences à l'Université de Lille

Rapporteur.ices:
Anne-Laure Ligozat - Professeur au LISN et à l'ENSIIE
Laurent Lefèvre - Directeur de Recherche à l'Inria

Président du Jury:
Jean-Christophe Routier - Professeur à l'Université de Lille

Jury examinateurs:
Aurélie Bugeau - Professeur à l'Université de Bordeaux
Romain Robbes - Directeur de Recherche au LaBRI

Université de Lille

CRIStAL
Centre de Recherche en Informatique Signal et Automatique de Lille

Inria

Empirical Evaluation of the Energy Impact of Large Language Models for Code Generation and Optimization

Tristan COIGNION

13th November 2025

University of Lille & Inria center
CRIStAL laboratory
MADIS doctoral school

Supervisors:
Romain Rouvoy - Full professor at University of Lille
Clément Quinton - Associate professor at University of Lille

Reviewers:
Anne-Laure Ligozat - Full professor at LISN and at ENSIIE
Laurent Lefèvre - Senior researcher at Inria

Jury president:
Jean-Christophe Routier - Full professor at University of Lille

Jury:
Aurélie Bugeau - Full professor at University of Bordeaux
Romain Robbes - Senior researcher at LaBRI

Université de Lille

CRIStAL
Centre de Recherche en Informatique Signal et Automatique de Lille

Inria

Acknowledgements

I would like to thank my advisors, Clément Quinton and Romain Rouvoy who guided me through my PhD thesis and made it flow like a gentle breeze. Thank you, Clément and Romain for all the insightful discussions, the late night reviewing and editing, and for helping me affirm my academic ambitions. I am very thankful to both of you for letting me be the main driver of my thesis, while still providing guidance when needed. I believe that I could not have gotten better advisors for my thesis, you really taught me a lot, so thank you.

Next, I would like to thank Ivano Malavolta, Patricia Lago and Vincenzo Stoico for proposing a wonderful and enriching collaboration with me.

Then, I would like to thank Romain Robbes for the numerous invitations to give a talk at the GDR-GPL and GT-GLIA. Thank you also to Olivier Le Goaër for inviting me for a talk to the GDR-GPL.

I also thank all the members of the Spirals team, for, they are all very interesting and nice people with whom I had among the most enriching conversations these last few years. I want to thank especially Iliana Fayolle, who supported me through the hardships of the thesis and taught me a lot about a lot of things (most not related to my thesis, but I digress).

My friends and family provided me much support and relief, so, a big "Thank you!" to them also.

Last but not least, I thank my wonderful partner in life, Catherine, for without her, my life would be duller. Her unconditional support enabled me to push through numerous hardships during my PhD thesis, and allowed me to better my organization and social skills.

Abstract

The recent rise of Large Language Models (LLMs) has transformed software engineering by enabling automated code generation, documentation, testing, and optimization. While these tools promise increased productivity and potentially more efficient code, they also raise important environmental concerns due to their high energy demands, reliance on specialized hardware, and integration into energy-intensive infrastructures. This thesis investigates the trade-offs between the energy costs and benefits of using LLMs for code-related tasks, with a particular focus on their role in sustainable software development.

Three empirical studies are presented. First, we evaluate the runtime performance of LLM-generated code on over 200 Leetcode problems, revealing that while LLMs can outperform human-written solutions in some cases, their efficiency is highly sensitive to model parameters and problem familiarity. Second, we quantify the energy consumption of code assistants like GitHub Copilotthrough a user study and server instrumentation. The results show significant energy waste due to unsolicited suggestions and highlight how inference server configuration and usage modes can reduce energy consumption. Third, we analyze the net energy impact of LLM-based code optimization techniques across 118 problems. Our findings show that while optimizations can reduce energy usage, they are only environmentally profitable if the optimized code is run frequently enough to offset the cost of generating the optimization.

Together, these three contributions provide a detailed examination of the trade-offs in using LLMs for code generation and optimization, with a focus on energy consumption. By empirically assessing both the costs and potential savings of code assistants and LLM-based optimizers, this thesis highlights a critical dimension of sustainable software engineering. While LLMs can improve code efficiency and accelerate development, their environmental impact depends heavily on usage patterns, deployment conditions, and application context. As ICT's footprint grows, understanding these trade-offs is essential. This thesis offers actionable insights for aligning software development with ecological constraints.

Résumé

La montée en puissance récente des grands modèles de langage (LLMs) a transformé l'ingénierie logicielle en permettant la génération automatique de code, la documentation, les tests et l'optimisation. Bien que ces outils promettent une augmentation de la productivité et, potentiellement, un code plus efficace, ils soulèvent également d'importantes préoccupations environnementales en raison de leurs besoins énergétiques élevés, de leur dépendance à un matériel spécialisé, et de leur intégration dans des infrastructures énergivores. Cette thèse étudie les compromis entre les coûts énergétiques et les bénéfices liés à l'usage des LLMs pour les tâches liées au code, avec un accent particulier sur leur rôle dans le développement logiciel durable.

Trois études empiriques sont présentées. Premièrement, nous évaluons les performances à l'exécution du code généré par des LLMs sur plus de 200 problèmes provenant de Leetcode, révélant que, bien que les LLMs puissent surpasser des solutions humaines dans certains cas, leur efficacité reste très sensible aux paramètres du modèle et à la familiarité avec les problèmes. Deuxièmement, nous quantifions la consommation énergétique des assistants de code comme GitHub Copilotà travers une étude utilisateur et une instrumentation des serveurs. Les résultats montrent un gaspillage énergétique important dû aux suggestions non sollicitées, et soulignent comment la configuration des serveurs d'inférence et les modes d'utilisation peuvent réduire cette consommation. Troisièmement, nous analysons l'impact énergétique net des techniques d'optimisation de code basées sur les LLMs sur 118 problèmes. Nos résultats montrent que si ces optimisations peuvent réduire la consommation d'énergie, elles ne sont écologiquement rentables que si le code optimisé est exécuté suffisamment souvent pour compenser le coût de sa génération.

Ensemble, ces trois contributions offrent une analyse détaillée des compromis liés à l'usage des LLMs pour la génération et l'optimisation de code, avec un focus sur la consommation énergétique. En évaluant empiriquement à la fois les coûts et les économies potentielles liés aux assistants de code et aux optimiseurs basés sur les LLMs, cette thèse met en lumière une dimension essentielle de l'ingénierie logicielle durable. Si les LLMs peuvent améliorer l'efficacité du code et accélérer le développement, leur impact

environnemental dépend fortement des usages, des conditions de déploiement et du contexte applicatif. Alors que l'empreinte environnementale des technologies numériques continue de croître, comprendre ces compromis devient crucial. Cette thèse propose des pistes concrètes pour aligner le développement logiciel avec les contraintes écologiques.

Table of contents

List of figures

List of tables

Chapter 1

Preface

Before my PhD thesis started in October 2022, not a lot of people had heard about the Large Language Models, some form of it existed, but they were not mainstream yet. However, on the 30th of November 2022, a shiny new application named ChatGPT was launched by OpenAI and took the world by storm, quickly gaining the record of the application with the fastest growing user base [41]. Academic literature across domains soon followed suit and seized the subject of Large Language Models leading to a drastic increase of the research effort concerning this technology [60].

Nowadays, when you are working in tech, it is difficult not to hear about artificial intelligence every other day. *Large Language Model* (LLM) have become almost ubiquitous, and have integrated themselves into most white-collar domains and especially in the computer science industry. The sheer amount of automation and ease of work that is made possible by the LLM-powered tools for developers is simply astonishing. Thanks to these new technologies, developers can write code faster, with less skill required, create working tests and insightful documentation in minutes instead of hours and find and fix bugs at a much accelerated pace.

It seems too good to be true, doesn't it? "What's the catch?" I hear you say. Well, LLM technology is growing way too fast and is way too young for research to be able to reliably measure its direct impacts on the economy, environment and society, let alone its indirect impacts. Nonetheless, some of their environmental impacts have been assessed over the years, despite the non-transparency of LLM providers and GPU manufacturers. From these results, we can at least say that LLMs are rather bad for the environment as a whole, but I will go more in detail as to why later on.

In this thesis, I will present my contributions to the research efforts on the environmental impacts of LLMs, more specifically in their use for code generation by developers.

I hope reading this is as insightful for you, reader, as it was for me to work on during my PhD thesis.

I wish you a good read.

Chapter 2

Introduction

2.1 Large Language Models

Deep Learning (DL) is a subfield of Machine Learning, itself a subfield of Artificial Intelligence. DL is based on neural networks, which take inspiration from neuroscience and is centered around stacking artificial neurons into layers and training them to process data. DL models are used, among other things, to perform tasks such as classification, regression and representation learning. Notably, they have been used in automatic speech recognition, image recognition as well as natural language processing and recommendation systems [85]. At its core, a *Large Language Model* (LLM) is a Deep Learning model, that can understand and write human-like text and generalize to multiple tasks. The specificity of LLMs is that they leverage transformer architectures to learn and understand the patterns present in language. The transformer architecture, as described by the 2017 foundational work "Attention is All You Need" [101], is based on self-attention mechanisms which enables the models to handle efficiently long strings of texts. Architectures that build upon transformers, such as the GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers) enable models to remember the full context of a sentence or document, which dramatically improves their performance on Natural Language Processing (NLP) tasks. As LLMs' number of parameter grew from millions to billions, researchers found them emergent capabilities, that is, abilities to solve tasks for which they were not specifically trained for. For instance, they could reason, take informed decisions, answer questions, role-play, etc., without having been trained specifically for these tasks [16]. These improvements enabled new textual applications of LLMs such as chatbots, summarizing or translating text and answering questions [16]. Nowadays, LLMs can also process other modalities of

information such as audio, images and videos, which enables more insightful conversations in multi-modal settings and expands their potential applications.

It has been shown that LLMs can dramatically improve their capabilities when prompted with examples (few-shot prompting) compared to not using examples at all (zero-shot prompting) [103]. Moreover, fine-tuning LLMs for specific tasks as well as using reinforcement learning to align them to human preference has been shown to enhance their generalization capabilities [16].

LLMs have been applied to almost all of the software engineering fields. LLMs that have been fine-tuned to perform software engineering are usually called "Code LLMs" or "Large Language Models for code" by the community. In testing, code LLMs have been used to generate unit tests [104], maintain tests [111] or even to generate test inputs [112] and inputs generators [51]. LLMs have also shown capabilities in code documentation generation [33]. LLMs have also been extensively applied to debugging and program repair. Notably, they are able to identify and locate the cause of software bugs using failing tests, error logs or even bug descriptions [104]. Once a bug has been found, LLMs have shown to be more efficient at program repair than traditional approaches [55].

More specifically, one of the areas of software engineering where LLMs have been applied the most is certainly code generation. LLMs are now widely integrated into development tools to assist programmers in building software more efficiently. The most popular instance of those tools is GitHub Copilot, an *Integrated Development Environment* (IDE) extension which main feature consists of suggesting code completions to the user. According to the StackOverflow annual developer survey [1], more than 80% of developers are using ChatGPT in their daily work, and more than 40% of developers are using a code assistant to help them in their development. As a result, a growing share of today's software is being built with the help of LLM-generated code. Given their widespread adoption, it is crucial to evaluate the reliability and safety of the code they produce. For example, Pearce *et al.* [79] found that GitHub Copilotoften generates code containing security vulnerabilities, highlighting the need for careful review of LLM-generated output.

One quality of LLM-generated code that concerned me during my thesis was its efficiency, specifically, how much time and energy resources the code uses when executed. Indeed, programming efficiency is paramount, especially when resources are scarce or programs are deployed on a large scale. In today's context where the energy consumption of software systems has become a major concern, improving software efficiency is particularly relevant since decreasing the resource consumption of a program can also lead to a reduction of its energy consumption [102, 7]. However, while users seem to put

a lot of trust in code generated by LLMs, they still have trouble reviewing it [98, 92], which can lead to slow code being shipped in production, especially if a LLM generates inefficient code.

On the other hand, LLMs have also been shown to be quite capable at code optimization, that is, rewriting programs to improve their efficiency. Recent advances in code LLMs have made it increasingly easier to generate optimized version of existing code, with shorter execution times and lower energy consumption [61]. Most of these methods rely on iterative refinement [71] which enables them to adapt and improve based on the outcomes of each optimization step. The process of manual code optimization is lengthy and intricate, requiring careful attention and a certain level of expertise, especially when searching for the best-performing algorithm, selecting the most appropriate data structure, or struggling with memory hierarchy. As such, automating this process with LLMs could have the potential to make optimizing code easier and more accessible for developers, and consequently, reduce the overall resource consumption of the ICT sector around the world.

This potential for efficiency gains is especially important given the escalating environmental crisis. Global warming is intensifying extreme weather events, accelerating sea-level rise, and provoking conflicts over increasingly scarce resources such as freshwater [30, 84]. In parallel, the extraction of minerals and metals (essential for digital infrastructure) is becoming more costly and environmentally damaging. Mining activities often lead to soil and water pollution, the destruction of ecosystems, and serious human rights concerns, including the exploitation of underpaid and underage laborers [48, 8]. According to the UN Environment Programme's International Resource Panel, global raw-material extraction tripled between 1970 and 2024, reaching 106 Gt/year. With a projected rise of 60% by 2060 ($\tilde{1}$60 Gt/year) [35]. All of these facts underscores the urgent need for actions towards reducing the overall environmental footprint of our society.

It just turns out that the Information and Communication Technology (ICT) industry has a direct role to play in these environmental and societal problems. Servers, computers, phones, IOT devices, and networks all need very specific resources to manufacture which entails a strong negative impact on the environment. For instance, modern smartphones contain over half of the periodic table, as well as 25 different metals and 17 rare earth elements, most of which are considered critical resources [5]. GPUs, which are fundamental for training and using LLMs need many conflict minerals, which have been tied to funded violence and human rights violations. GPUs also require an important quantity of rare earth elements and metals, which require polluting manufacturing processes [3].

Manufacturing is not the only impact of the ICT industry: actually using those equipment and services as well as disposing of them at their end of life also has a considerable impact. It is estimated that the ICT industry consumes about 4.7% of the world's electricity production and accounts for a growing 1.7% of global greenhouse gas emissions [108]. In France, the ICT sector is responsible for a growing 2.5% of the country's GHG emissions, which is estimated to triple by 2040 if no mitigating measures are taken [108]. Datacenters require enormous amount of fresh water to cool down, most of which is then evaporated into the atmosphere. Microsoft and Google already reported respectively, a 34% and 20% increase in water demand between 2021 and 2022. As water becomes a scarcer resource over the years, its increasing use in ICT is becoming a serious concern [67].

At the end of its life, ICT hardware needs to be disposed of, either through reusing or recycling parts of the equipment, or throwing it in landfills. According to the UN's Global E-Waste Monitor 2024, only 22% of electronic waste has been formally collected and recycled [11] while global generation of electronic waste is rising five times faster than its recycling. While recycling equipment may sound like a good idea, it is actually near-impossible to cost-effectively recycle each material in electronics, which means that most of the minerals and metals in electronics do no actually get recycled [17]. This shows that relying on the reusing and recycling part to reduce the environmental impact of ICT is at best unreliable, and the best bet is to use equipment for longer and use fewer of them.

As the impact of ICT on the environment is increasing, it is becoming ever more important to reduce its footprint and its usage. However, the rise of AI technologies in these last few years do not seem to have helped at all with the direct environmental impacts of ICT. LLMs in particular require large amount of processing power for their training and inference phase (the usage phase of the LLM), as well as specialized hardware in the form of GPUs, which require large amount of abiotic (non-renewable) materials to manufacture and consume a lot of energy when running. For instance a medium-sized LLM, such as Qwen-72B, can require a full Nvidia H100 GPU to run, which needs a most 800W of energy to run.

Some researchers have already uncovered some environmental impacts of LLMs. For example, a query to ChatGPT via the GPT-4o model is estimated to consume about 1.2Wh (the equivalent of a 10W light bulb turned on for 7 minutes), while sending the same query to OpenAI's reasoning model o3 is estimated to consume 21.4Wh (the equivalent of a 10W light bulb turned on for 2 hours) [52]. Concerning water, one paper suggests that 10–50 queries to GPT-3 consumes around half a liter of water [56]. Another

paper estimates that supporting one year of Stable Diffusion (a latent diffusion model that generates images ; not an LLM, but similar in resource use) service can generate as much as 360 tons of CO^2 equivalent [14].

Some people might argue that AI has a net positive impact on the environment, because it can allow us to produce code faster and cheaper (thus requiring less human labor), and produce code that is more resource-efficient. As such, it is tempting to see code assistants like GitHub Copilotand LLM-based code optimizers as a straightforward win for sustainability, right? Well, as we just said, LLMs are very resource-intensive and require powerful inference servers for deployment and operation. Thus, any potential savings achieved by producing code faster and optimizing code must be weighed against the substantial energy cost of using LLMs in the first place.

In light of the environmental impact of LLMs and their potential to alleviate the resource consumption of ICT, I ask the following question that will guide the rest of this thesis: **"What is the negative and positive energy impact of code LLMs?"**.

While I cannot fully answer the question in this thesis alone, I've aimed at answering parts of this question during my thesis. Notably, I've explored three components of this question, which I will describe below:

(I) **How efficient are the solutions generated by LLMs?** First and foremost, I wanted to know how efficient was the code generated by LLMs out-of-the-box and how it compared to human-written code. As the usage of LLM-powered tools increase, it is important to understand and assess the actual quality (here, the efficiency) of the code that they produce.

(II) **How much energy is used when using code assistants like GitHub Copilot?** As code assistants usually make their inference (the text generation) on a remote server, the users are not aware of their real energy consumption when they use such a tool, and may believe it to be negligible. With this question, I aim to measure how much energy is really consumed when using a code assistant like GitHub Copilotas well as find levers to reduce this energy consumption. Thanks to this question, I hope that users will be more aware of their energy impact when using such tools.

(III) **What is the real energy cost of using LLM-powered code optimizers? Is the performance and energy gain of optimized code sufficient to justify the energy cost of optimization?** As LLMs become increasingly used to optimize source code, the question of their actual effectiveness at reducing the energy consumption remains. With this question, I would like to investigate the cost of optimizing code with LLMs and compare it to the actual energy savings that were enabled thanks to the optimization.

This thesis presents my answers to these three smaller questions via three contributions that I present in the next section.

2.2 Contributions

In this section, I present an overview of the contributions realized during my thesis. As stated before, the goal of my research is to study the energy impact of using LLM for code generation and code optimization. The main contributions of my work are summarized as follows.

A study of the performance of LLM-generated code on Leetcode. In this first contribution, we studied the performance of code generated by 18 LLMs on problems from Leetcode, a competitive programming website, and compared them to human-written solutions. We considered factors such as model temperature (a parameter regulating the randomness of the model) and success rate, and their impact on code performance. To perform our research, we extracted old and new problems from Leetcode. Old problems were problems that were old enough to have been present in the training dataset of the tested LLMs. In total, we had 300 old problems and 204 new ones. We made the LLMs generate 10 solutions to each of those problems, and evaluated the validity and performance of those solutions with tests extracted from Leetcode.

Using a novel method for measuring and comparing the speed of LLM-generated code, we observed that LLMs produce code with comparable performance, irrespective of the adopted model. Additionally, using a higher temperature parameter led to more frequent inefficient and slow generated code with also an increased chance of infinite loops being generated or code with an exponential complexity. We also found that LLMs are capable of generating code that is, on average, more efficient than the code written by humans. Our contribution highlights the risks of using Leetcode as a dataset for LLM evaluation as there is important data contamination issues concerning older problems.

A study on the key factors affecting the energy consumption of code assistants. In this second contribution, we studied the energy consumption of code assistants with a user study of developers using GitHub Copilot. First, we created AssistantTraces, a dataset of development traces collected thanks to 20 participants who developed a Java project for one hour while using GitHub Copilot. This dataset enabled us to easily replay the developer's interactions with GitHub Copilot, this time using our own inference server. This setup allowed us to study the impact of some of the inference server's and assistant's configuration options on their energy consumption and performance.

Our results revealed that a considerable portion of energy was wasted due to suggestions being cancelled by GitHub Copilotor not wanted by the users. We concluded that if the automatic suggestion trigger mechanism was replaced by a manual one, we could greatly minimize unnecessary requests, thus saving a lot of energy. Additionally, the number of concurrent developers using a single inference server greatly affected its energy efficiency, as having more concurrent developers better used the resources of the server. Lastly, we found that the configuration of the inference server played a significant role in its energy consumption. For instance, (i) smaller models tended to use less electricity ; (ii) reducing the number of GPUs decreased the energy consumption at the cost of a higher latency ; and (iii) some quantization techniques positively impacted both the energy consumption and latency of the code assistant. In the end, we estimated that on average, a developer using a code assistant like GitHub Copilotwould consume about 20W of electricity when actively coding. Our results highlighted the need for raising awareness of the users on their code assistant's energy consumption.

A study on the cost of LLM-based code optimization. In this last contribution, we systematically evaluated the energy gains of 8 LLM-based code optimization methods on 118 tasks from the EvalPerf benchmark. We assessed the trade-offs between code performance, correctness, and energy consumption of multiple optimization methods across multiple families of LLMs. To help with our analysis, we introduced the *Break-Even Point* (BEP) as a key metric to quantify the number of executions required for an optimized program to outweigh the energy consumed when generating the optimization itself.

Our results showed that, while most LLM-based optimization strategies were capable of reducing the energy consumption of code, the energy savings they delivered did not always offset their own energy costs. In some cases, the optimized code had to be executed hundreds of thousands of times before the energy spent on the optimization was compensated by the energy saved. For example, a large model like DeepSeek-R1-14B combined with a powerful optimization method could reduce energy usage by up to 49% on average, but only became energetically beneficial after approximately 200,000 executions of the optimized programs. In contrast, smaller models like Qwen2.5-Coder-7B achieved worthwhile savings with far fewer executions, making them more suitable for lightly used code samples or one-off scripts. Notably, in some high-consumption scenarios, we observed energy reductions from 318 J to just 2.6 J, which offset the optimization cost in under 100 executions. Moreover, the optimization process often yielded incorrect or less efficient code. On average, the code was correct only 62% of the time and only 76% of the remaining correct code samples were more efficient than their "unoptimized" parent.

Importantly, we identified a weak negative correlation between performance metrics gains and actual energy savings, challenging assumptions that faster code automatically equates to a smaller energy footprint.

These findings demonstrated that while LLM-based optimization can yield substantial energy savings, its actual profitability is highly context-dependent and must be carefully evaluated based on how frequently the optimized code is expected to run.

Taken together, these three contributions offer a comprehensive examination of the trade-offs involved in using LLMs for code generation and optimization, with a specific focus on their energy consumption. By empirically analyzing both the energy cost and the energy-saving potential of code assistants and LLM-based optimization methods, this thesis sheds light on a critical dimension of sustainable software engineering. The results highlight that while LLMs can, in some cases, produce more efficient code or accelerate development workflows, their benefits must be critically assessed in light of their substantial resource requirements. This work demonstrates that there is no straightforward sustainability gain from the use of LLMs: their environmental impact is nuanced and strongly dependent on deployment conditions, usage patterns, and application contexts. In an era where ICT's environmental footprint continues to grow and the urgency of climate action increases, understanding and quantifying these trade-offs is essential. By addressing each facet of the main research question, this thesis provides actionable insights for researchers, tool designers, and practitioners aiming to align software development practices with planetary boundaries.

2.2.1 Digression on the rebound effect

Recent work by Luccioni et al. [67] has shed important light on a point that's often overlooked in discussions around AI and sustainability: efficiency gains do not always translate into environmental benefits. In fact, they can sometimes make things worse. This phenomenon is known as Jevons' Paradox, and it plays a central role in what's called rebound effects. The idea is simple: when something becomes more efficient or cheaper to use, people tend to use it more, sometimes a lot more, so much so that the total resource use actually increases instead of decreasing.

In the case of AI, this means that even if we build models that are more efficient, or data centers that use less energy per operation, we might still end up with a larger overall footprint. Luccioni et al. describe three types of rebound effects that are particularly relevant here: material, economic, and behavioral. Material rebound effects happen when efficiency gains lead to more hardware being produced and deployed like more GPUs

or larger data centers. Economic rebound effects emerge when AI becomes cheap and convenient enough to integrate everywhere, driving up demand for AI-enabled services. And behavioral rebound effects are about us, the users, how we tend to interact more with AI systems when they are fast, available, and easy to use, sometimes in ways that are not necessary or useful.

These insights have direct consequences for the findings in this thesis. Yes, LLMs can generate code that runs faster and uses less energy. And yes, they can help developers work more quickly. But these gains do not exist in a vacuum. For instance, we saw that using Copilot triggers many suggestions that are never accepted, but still consume energy. In the case of code optimization, we showed that while LLMs can reduce energy usage in many cases, the benefit only becomes real if the optimized code runs often enough to pay back the cost of generating it. Without some care, these tools could end up being used indiscriminately: optimizing code that barely runs, or pushing suggestions for every keystroke, ultimately wasting more energy than they save.

Another important rebound effect comes from the fact that LLMs make it easier and faster to produce software. This might seem like a good thing, but it can also lead to developers creating more features, more tools, and more systems than they otherwise would. When the friction to build something drops, the volume of software tends to rise. But every additional line of code has a lifecycle: it needs to be tested, deployed, maintained, and executed, often repeatedly, sometimes indefinitely. So by speeding up development, LLMs might end up increasing the overall amount of software running in the world, which in turn means more servers, more energy, more data transfers, and more emissions. This kind of rebound effect reinforces the idea that efficiency needs to be paired with restraint and sobriety if we want to avoid turning short-term gains into long-term environmental costs.

This brings us to a broader point: efficiency is not a substitute for sobriety. Rebound effects remind us that solving environmental problems is not just about doing the same things more efficiently, it is also about questioning whether we need to do all those things in the first place. LLMs are powerful tools, but their environmental impact will ultimately depend on how we choose to use them. If they enable developers to offload cognitive effort without reflection, or to optimize low-impact scripts by default, their efficiency may mask a broader pattern of excess. On the other hand, if they are used carefully (triggered manually, aimed at meaningful bottlenecks, or deployed only where the long-term payoff justifies the cost) they can be part of a more sober and sustainable development process.

The real challenge, then, is not just technical. It is about promoting a mindset of moderation in a field that has historically been driven by speed, scale, and abstraction. We need to resist the temptation to equate automation with improvement, and to remember that in a world facing ecological limits, less can often be better. Rebound effects are not inevitable, but avoiding them requires intention. It requires design choices that encourage responsible usage, and social norms that value sufficiency over-abundance. If we want LLMs to contribute to a sustainable digital future, we must learn not only how to use them well, but when not to use them at all.

2.3 Structure of this thesis

In this section, I describe the structure of the rest of this thesis. The second chapter of this thesis pertains to the state of the art. Third, fourth and fifth chapters present the contributions of this thesis, and the sixth and last chapter concludes this thesis while providing perspectives on the contributions. The chapters are as follows:

Chapter 3 - State of the art. This chapter aims at providing the reader with an overview of the domain of LLM code generation, LLM-based optimization as well as the sustainability of LLMs in general. This chapter helps define the various concepts and terminology that will be used in the later chapters of this thesis.

Chapter 4 - A Performance Study of LLM-Generated Code on Leetcode. In this chapter, I present a study on the performance of LLM-generated code where 18 LLMs have been evaluated on problems extracted from Leetcode. Notably, this study shows that the code generated by LLM is not always efficient and that using Leetcode as a benchmark presents important risks.

Chapter 5 - Green My LLM: Studying the Key Factors Affecting the Energy Consumption of Code Assistants. This chapter revolves around a study on the energy consumption of code assistants like GitHub Copilot. The study estimates how much energy is used when a developer uses GitHub Copilotand highlights some energy inefficiencies of GitHub Copilot.

Chapter 6 - When Faster Isn't Greener: the Hidden Costs of LLM-Based Code Optimization. In this chapter, I present my latest study which is about the potential energy gains of optimizing code with LLMs and the costs of the LLM-based optimization process. The results highlight that while LLMs are able to optimize the code efficiency a majority of the time, most of the optimized programs would need to be executed thousands of time in order to turn an energy profit.

Chapter 7 - Conclusion and Perspectives. With this last chapter, I conclude the work presented in this thesis, summarize my contributions and discuss its implications and limitations. I also present research ideas and future directions as short-term and long-term perspectives.

2.4 Publications

During the development of the contributions presented in this thesis, I co-wrote three research papers. The first of the three papers got accepted in an international conference, whereas the other two are still under review by a journal and conference. These papers are as follows:

1. Tristan Coignion, Clément Quinton, Romain Rouvoy. *A Performance Study of LLM-Generated Code on Leetcode.*. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 79-89) (EASE). June 2024. https://doi.org/10.1145/3661167.3661221.

2. Tristan Coignion, Clément Quinton, Romain Rouvoy. *Green My LLM: Studying the Key Factors Affecting the Energy Consumption of Code Assistants*. Submitted at the Journal of Systems and Software (JSS). Under major revision.

3. Tristan Coignion, Clément Quinton, Romain Rouvoy. *When Faster Isn't Greener: the Hidden Costs of LLM-Based Code Optimization*. In Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). November 2025

Chapter 3

State of the art

In this chapter, I review the growing body of literature that investigates the intersection of LLMs and software engineering, with a specific focus on their applications in code generation, code optimization, and software sustainability. I begin by examining how LLMs are evaluated for their ability to generate syntactically and functionally correct code, before transitioning to studies concerned with the performance and efficiency of LLM-generated solutions. Following this, I detail approaches that leverage LLMs for improving the quality of both auto-generated and human-written code, and conclude by reviewing recent research on the environmental impact of both LLM-generated software and LLM inference processes themselves.

3.1 LLMs for code generation

LLMs' code generation capabilities have been extensively investigated in academic literature. Most of the time, their capacity to generate code is tested using code evaluation datasets and programming contest datasets. Code evaluation datasets such as HumanEval [20] and MBPP [10] usually contain functions signatures and their documentation (which both act as the prompt for the LLM) as well as relevant unit tests. HumanEval's goal is to measure the functional correctness of programs synthesized from docstrings. It consists of 164 original Python programming problems that assess language comprehension, algorithm writing and simple mathematics. MBPP is a similar dataset, that contains around 1,000 crowd-sourced Python programming problems, designed to be solvable by entry level programmers. On such datasets, the correctness of the LLM-generated solutions is measured by `pass@k`, which is the probability of getting at least one valid generation (*i.e.*, which passes all tests) out of k generations.

While HumanEval and MBPP come bundled with unit tests, it has been shown that such tests are too few in numbers, use relatively simple inputs and outputs and do not fully explore the functionality of the code or corner cases. As such, some generations from LLMs may pass the dataset's tests and appear correct when there are in fact incorrect. To solve this problem, Liu *et al.* [63] proposed EvalPlus, a method to synthesize large amounts of new test inputs using LLM-based and mutation-based input generators. EvalPlus allowed them to extend HumanEval by building HumanEval+ with 81x additionally generated tests. They found out that when using HumanEval+, the measured correctness of the LLMs dropped by 15.1%. Additionally, thanks to their more exhaustive tests, some ground truth solutions from HumanEval were found to be in fact incorrect.

Later on, Qiu *et al.* [87] used expert-written test generators on HumanEval and found that incorrect ground truth solutions were still present in HumanEval+. Moreover, they also observed that many of the solutions in both the original and augmented HumanEval were inefficient and could be improved.

Apart from the traditional LLM evaluation datasets, competitive programming websites are also often used in programming LLM evaluations. Their abundance of problems, their multiple programming languages support, automated judge system as well as their database of user-submitted solutions make them a very good fit for easily evaluating LLMs. They usually contain natural language descriptions of algorithmic problems (*e.g.*, sorting, data structures problems, etc.) as well as the signature. One prominent competitive programming dataset built for ML is CodeNet [86], a large collection of code samples used for training LLMs. However, among the programming contest websites, Leetcode is the one that appears the most across literature [24, 76, 97, 83, 45, 100, 29].

However, most LLM evaluation datasets are susceptible to data contamination. That is, the data on which LLMs are evaluated may also be present in their training data, which would lead to the model performing better than usual on the leaked test data [72, 27]. For instance, it has been shown multiple times that some LLMs such as ChatGPT might have been contaminated by HumanEval's data [31] or may have been overfitted specifically on HumanEval [51]. To prevent this from happening, model makers specifically filter out popular evaluation datasets like HumanEval and MBPP from their training data (competitive programming datasets however are not filtered out). However, slight lexicographic changes in the function signatures or documentation can bypass this filtering while still significantly contaminating the model, which might explain the apparent contamination of ChatGPT by HumanEval [89].

While many studies have explored the potential data contamination of HumanEval or other code datasets, the data contamination of Leetcode and the rest of the competitive programming websites remain, as far as I know, underexplored in academic literature. The potentiality of data contamination of Leetcode is especially important since Leetcode is used in many LLM evaluations papers all the while many instances of user-submitted solutions to their problems exist on GitHub, a major source of training data for the LLMs. I explore in my first contribution (in chapter 4) the data contamination of multiple LLMs on Leetcode problems.

However, recently, Jain *et al.* [51] created LiveCodeBench–a Leetcode-based comprehensive and contamination-free benchmark for LLMs. To avoid data contamination, it continuously collects new coding problems from platforms like Leetcode, AtCoder, and Codeforces.

To improve LLMs code generation capabilities, researchers have built bigger and better models, with higher quality training datasets [47, 57, 65]. If training a new model from scratch is too costly, another option to improve quality is to fine-tune the models with specialized or high-quality code [65, 103]. However, there also exists methods to improve LLMs without needing to retrain or fine-tune them. Many of these techniques are categorized as prompt engineering techniques, that is, techniques which modify the prompt given to the LLMs in order to produce better results. These techniques can be as simple like asking the model to "Think step-by-step." (Chain-Of-Thought) [106] and showing the LLMs examples of task and their solutions in the prompt (Few-Shot Prompting). Prompt engineering techniques are often paired with iterative refinement strategies, that is, repeatedly refining the output of an LLM using feedback [71].

Iterative refinement offers great promises in terms of enhancing the capabilities of LLMs without needing retraining. For instance, it can increase the rate of correct code production and increase the trust in the LLM's output. For example using the LLM to give feedback to itself using the Self-Refine method improves the average task performance by 20% [71]. Using a more complex methodology, Reflexion [93] manages to achieve a 91% accuracy on HumanEval surpassing GPT-4 that achieves 80%.

3.2 Evolving algorithms using LLMs

LLMs have also been used as operators in evolutionary algorithms, more specifically, genetic programming. Notably, Taherkhani *et al.* [96] present *Evolutionary Prompt Engineering for Code* (EPiC) which uses an LLM to mutate a prompt to produce better results. The LLM mutates the prompt of a HumanEval or MBPP task, which then helps

a second LLM to generate a better program. Their method is able to attain slightly better results than the others tested methods on HumanEval and MBPP with GPT-4o while minimizing the amount of LLM-ressources used when compared to other evolutionary methods. Similarly, Chen *et al.* [19] used an LLM to evolutionarily improve a neural architecture search prompt.

The *Self-Taught Optimizer* (STOP), presented by Zelikman *et al.* [114] uses an LLM as a meta-optimizer to find the best way to improve its code generation. The strategies used by the LLM to improve on itself included genetic algorithms, decomposing and improving parts, multi-armed prompt bandit, varying temperature, simulated-annealing based search and beam search.

Lastly, in Evolution of Heuristics by Liu *et al.* [62], the LLM does not evolve code directly, but evolves heuristics: ideas describing how the algorithm works, which are then translated into code. The authors evaluate their method on optimization problems such as the traveling salesman problem and find that their method surpasses other evolutionary methods for this application.

While LLMs show great promises as mutators in evolutionary algorithms, to my knowledge, their potential as mutators has not yet been applied to optimizing the efficiency of everyday code such as with the tasks of the HumanEval and MBPP datasets. In my third contribution in chapter 6, I use Evolution of Heuristics to improve code efficiency.

3.3 LLMs for code optimization

Code optimization is the act of improving certain qualities of a program other than its correctness. A growing body of research focuses on producing and improving LLM-based code optimization methods. Many of these methods leverage iterative refinement and prompt engineering to optimize code and improve its runtime performance or energy efficiency. Parallel to this, there has been a lot of efforts towards measuring the efficiency of the code generated by LLMs. In this section, I present the benchmarks that were created to measure the efficiency of LLM-generated code. Next, I will explain the results from the literature on the efficiency of LLM-generated code as well as on the optimization capabilities of LLMs.

3.3.1 Benchmarks to measure the efficiency of LLM-generated code

As the need for efficient code generation grows, a number of recent works have developed specialized benchmarks to evaluate not only the correctness, but also the runtime, memory, and energy efficiency of code produced by LLMs. Traditional benchmarks such as HumanEval and MBPP primarily focus on functional correctness, often using small or non-representative inputs. In contrast, the benchmarks discussed in this subsection are specifically designed to stress test the generated code, assess its performance characteristics, and measure its alignment with expert-crafted or canonical implementations. These benchmarks vary in scope, methodology, and evaluation metrics, offering a diverse landscape for the systematic assessment of LLM-generated code efficiency.

COFFE. Peng *et al.* [82] proposed COFFE, a code generation benchmark for evaluating the time efficiency of LLM-generated solutions. COFFE contains 398 and 358 problems for function-level and file-level code generation respectively. Namely, they proposed a new metric, named efficient@k that considers both correctness and time efficiency based on CPU instruction count measurements. Along with their benchmark, the authors developed a stressful test generator, STGen, which uses an LLM across a multi-step process to create generators of large test inputs while verifying that the test cases are correct.

EffiBench. EffiBench [45] is a benchmark with 1,000 coding problems from Leetcode. As Leetcode does not make its performance-stressing tests public, the authors of EffiBench created their own test case generator using LLMs. To measure the efficiency of the code, EffiBench uses the normalized execution time and memory usage of the solutions on their generated tests. Some caveats of this benchmark is that it uses Leetcode without any effort to avoid the effects of the data contamination of Leetcode, which may bias its results. Moreover, the authors created test cases generators using LLMs, but they do not verify the validity of the generated test cases.

ENAMEL. ENAMEL is also a benchmark for evaluating the efficiency of LLM-generated code [87]. It is based on a modified version of HumanEval. Namely, the authors removed the problems with a time complexity of $\Theta(1)$, and used expert-written test case generators to generate the inputs. They proposed a novel efficiency metric called `eff@k` which generalizes the `pass@k` metric from correctness to efficiency. One novel feature of ENAMEL is its handling of right-censored execution time and its ability to precisely characterize the efficiency of programs under different sample sizes.

Mercury. Mercury [32] is a code efficiency benchmark comprising 1,889 Python tasks extracted from Leetcode problems. They created test cases generators which

they validated using Leetcode's Online Judge system. Moreover, the authors introduce Beyond, a novel metric which computes a runtime-percentile-weighted Pass score to reflect functional correctness and code efficiency simultaneously. Akin to EffiBench, the authors did not try to mitigate the important data contamination of Leetcode.

ECCO. ECCO is a benchmark built by Waghjale *et al.* [103] that aims to evaluate the efficiency of LLM-generated programs. It can evaluate programs generated from Natural Language (NL) descriptions of tasks or from history-based code editing. ECCO is based on the CodeNet dataset [86]. It uses CodeNet's test cases as well as the test cases generated by the AlphaCode project [59].

EvalPerf. The authors of EvalPlus and HumanEval+ devised Differential Performance Evaluation (DPE), a framework to evaluate LLMs for efficient code generation [64]. DPE curates efficiency datasets by selecting efficiency-demanding tasks from existing benchmarks (HumanEval+ and MBPP+ in their case) and then generates performance-exercising tests for those datasets. Then, to evaluate the efficiency of a program, DPE profiles and compares it against a set of reference solutions that are clustered together by efficiency level. Their metric, DPS, ensures that comparisons on a given problem are based on how fast a solution is *compared* to the other existing solutions for this problem, which greatly facilitates unbiased comparison across problems of different complexities. Efficiency here is measured based on CPU instructions rather than execution time which improves the reliability of the measures. Using DPE, the authors created the EvalPerf benchmark from HumanEval+ and MBPP+. It contains 118 peformance-exercising tasks in Python.

We summarize the benchmarks presented in this chapter in Table 3.1. When realizing the contribution presented in chapter 6, we used EvalPerf as our dataset because of its extensive performance-stressing tests cases and its near absence of data contamination– especially when compared to Leetcode-based datasets. While ENAMEL, and ECCO were also valid choices, we ultimately decided to use EvalPerf thanks to its ease of use, its robust measuring process and adaptability (we had to use the benchmark with a variety of code optimization methods).

These benchmarks provide the tools needed to assess LLM-generated code beyond correctness, enabling systematic evaluation of runtime, memory, and other efficiency aspects. The following section builds on these resources to examine how LLMs perform in the absence of explicit optimization.

Benchmark	Based On	Goal	Metrics Used
HumanEval [20]	Manually written problems	Evaluate functional correctness of Python code	`pass@k`
MBPP [10]	Crowd-sourced Python problems	Test beginner-level code generation	`pass@k`
HumanEval+ [63]	HumanEval with extended tests	Robust correctness evaluation	`pass@k`
EvalPerf [64]	HumanEval+ and MBPP+	Compare runtime efficiency using reference clustering	DPS (CPU instruction comparison)
ENAMEL [87]	Modified HumanEval	Fine-grained efficiency evaluation with censoring	`eff@k`
COFFE [82]	HumanEval, MBPP, CodeContests and APPS	Joint correctness and efficiency (CPU instructions)	`efficient@k`
EffiBench [45]	Leetcode	Runtime and memory profiling	Normalized runtime and memory usage
Mercury [32]	Leetcode	Combined correctness and efficiency benchmarking	`Beyond` (runtime-weighted `pass@k`)
ECCO [103]	CodeNet + AlphaCode tests	NL-to-code and code editing evaluation	Execution time
LiveCodeBench [51]	Recent Leetcode problems (Leetcode, AtCoder, Codeforces)	Contamination-free correctness evaluation	`pass@k`, etc.
CodeNet [86]	Aizu & AtCoderP	Large-scale dataset for training and evaluation	Pass/fail via judge systems

Table 3.1: Summary of existing benchmarks for LLM-generated code evaluation

3.3.2 Efficiency of LLM-generated code without optimization

While LLMs have demonstrated impressive capabilities in generating functionally correct code, a growing body of empirical studies highlights their consistent shortcomings in producing efficient implementations when no explicit optimization strategies are employed. Across a wide range of benchmarks and evaluation settings, LLM-generated code is observed to be systematically less performant than human-written counterparts in terms of execution time, memory usage, and energy consumption.

Several benchmarks, including COFFE, EffiBench, ENAMEL, and Mercury, converge on the observation that state-of-the-art models such as GPT-4 generate code that is significantly more computationally expensive than canonical human solutions [82, 45, 32, 87]. For example, code generated by GPT-4 may exhibit, on average, over three times the execution time of human-written solutions, with extreme cases reaching up to thirteen times slower execution and forty times higher memory consumption [45].

These results demonstrate that correctness alone is not a reliable indicator of runtime performance. Indeed, even when models achieve high functional correctness (*e.g.*, `pass@1` scores exceeding 80%), their efficiency (*e.g.*, `eff@1`) remains markedly lower.

Importantly, increasing model size does not appear to mitigate these efficiency deficits. Larger models tend to improve correctness, but not efficiency [82]. This suggests that the number of parameters alone is insufficient for giving models the ability to reason effectively about performance, algorithmic complexity, or low-level implementation details. Moreover, performance degradation becomes even more pronounced when LLMs are tasked with generating larger code units, such as file-level or multi-function programs, compared to isolated function-level snippets [82]. This indicates limitations in the models' ability to manage performance trade-offs over extended, context-rich code.

The root causes of these inefficiencies are both algorithmic and syntactic. Analyses of generated code reveal recurring antipatterns, including the use of suboptimal algorithms, inefficient function calls, poorly structured loops, and underutilization of language-specific performance features [58]. These shortcomings reflect the models' limited awareness of performance-sensitive design principles and highlight a gap between generating semantically correct and computationally efficient code.

Finally, it is worth noting that in rare cases where LLMs produce efficient solutions, these are often attributable to memorization of known problems and widespread answers, rather than genuine algorithmic reasoning [97]. Such cases, while potentially beneficial in narrow contexts, do not generalize to novel or complex programming tasks, further emphasizing the models' lack of robust performance-aware generalization.

Taken together, these findings establish a clear baseline: without targeted optimization strategies, LLM-generated code is reliably correct but computationally inefficient. This finding is further explored in chapter 4. In the next section, I present solutions to this issue with the various LLM-powered methods that are used to improve the efficiency of code.

3.3.3 Improving the efficiency of LLM-generated and human-written code with LLMs

Recent advances have demonstrated a broad spectrum of techniques for enhancing the efficiency of both LLM-generated and human-written code using large language models. These techniques can be categorized based on whether they are applied during code generation, to already existing code, or in a hybrid manner across both stages. The underlying strategies span from lightweight prompt-based interventions to full-fledged

model fine-tuning guided by execution feedback or historical data. I present a summary of these methods in Table 3.2

A first family of methods focuses on prompt-based optimizations, where efficiency is sought directly through how the model is queried. In-context learning allows the LLM to observe examples of efficient code transformations and imitate such patterns in new generations [103]. Similarly, Chain-Of-Thought prompting structures the model's reasoning process, leading to more deliberate and optimized code outputs. General prompt engineering, particularly when guided by observed inefficiencies in the code, has also been found to improve runtime and memory performance [103]. In some cases, simply requesting energy-efficient or optimized code, as tested in environments like Matlab or Leetcode, can lead to measurable gains, although results remain inconsistent [88, 97].

Going beyond prompt design, iterative refinement methods introduce feedback loops into the generation process. These approaches typically rely on either natural language critiques or execution-based feedback such as test outcomes, runtime measurements, or overhead resource usage profiles. Tools like PerfCodeGen [81] and EffiLearner [44] employ this strategy, using runtime signals to revise and improve the initially generated code across multiple iterations. Execution information plays a crucial role here, acting as a grounding signal that helps LLMs balance functional correctness with performance improvements.

A growing body of work explores retrieval-augmented approaches, where external sources of performance knowledge are leveraged to guide LLM behavior. For instance, RAPGen [40] retrieves prompt instructions from a curated knowledge base of prior performance bug fixes to guide the generation of fixes for similar inefficiencies in new code. More advanced systems integrate retrieval into the optimization process through search-based frameworks, combining LLM generation with evolutionary algorithms or heuristic search. These systems, such as SBLLM [38], not only evaluate and select candidate solutions based on execution metrics but also integrate optimization patterns and chain-of-thought prompting to iteratively improve them.

When it comes to optimizing already existing human-written code, LLMs benefit significantly from exposure to datasets that record performance-focused edits. Fine-tuning on such data, such as the PIE dataset of performance-improving edits [94], allows the model to learn patterns of transformation that developers apply to improve efficiency. Complementary strategies include fine-tuning on execution traces or refactoring logs, thereby enabling models to capture the relationship between code changes and performance outcomes [44]. Some systems also leverage dictionaries of antipatterns, mined from large-scale code repositories, to identify and repair inefficient constructs at

scale. For example, ECO [61] applies this approach to billions of lines of production code in Google's monorepo, achieving massive performance savings with high correctness guarantees.

Finally, recent frameworks have proposed integrated generation pipelines that explicitly separate the logical design of algorithms from their implementation. LLM4EFFI [112] exemplifies this paradigm by first generating multiple algorithmic strategies, optimizing their theoretical performance, and only then implementing and refining them to ensure correctness. This two-stage approach mirrors how expert developers approach efficiency, enabling the model to achieve state-of-the-art performance across multiple efficiency benchmarks. Similarly, SwiftCoder [46] focuses on fine-tuning models using only correct and efficient code examples, and selecting the most performant variant from multiple candidates based on local execution profiling.

While these methods demonstrate the potential of LLMs to produce more efficient code, the question remains: at what (energetic) cost? My third contribution addresses this issue in chapter 6, quantifying when LLM-based optimization is actually worth it. In this contribution, we evaluated some of these optimization methods (LLM4EFFI, ECCO and other prompt-based methods) on their capacity to reduce the energy consumption of code, while also measuring their own energy cost.

Overall, improving code efficiency with LLMs requires a multi-faceted toolkit. While prompt engineering provides accessible and sometimes effective baselines, substantial and reliable efficiency gains increasingly depend on execution-informed iteration, retrieval of optimization knowledge, and model fine-tuning grounded in empirical performance data. As these methods mature, they offer the potential not only to generate optimized code from scratch, but also to scale optimization practices across vast, real-world software repositories. LLM-based optimization may help to build more sustainable software. In the next section, I will present the complicated relationship between sustainability and LLMs.

3.4 Sustainable software and LLMs

The growing use of LLMs in software development and digital services has raised important questions regarding their environmental impact. Recent research has approached this issue from two complementary angles: the sustainability of the code produced by LLMs and the resource consumption of LLMs themselves during training and inference. This section surveys both lines of inquiry to provide a broader understanding of how LLMs intersect with software sustainability.

Method / Concept	**Application Type**	**Broad Category**	**LLM Dependency Type**
In-context learning [103]	Both	Prompt-based	Prompting
Chain-of-thought prompting [97]	Both	Prompt-based	Prompting
Prompt engineering based on inefficiencies [58]	Both	Prompt-based	Prompting
Simply asking for efficient code [88, 99, 97]	New code	Prompt-based	Prompting
Iterative refinement with execution or NL feedback [103, 81, 44, 80, 71]	Both	Iterative refinement	Prompting
Retrieval-Augmented Generation [40]	Existing code	Retrieval-based	Prompting
Search-based LLMs + evolutionary methods [38]	Both	Search-based	Prompting
Dictionary of anti-patterns + fine-tuned LLM [61]	Existing code	Pattern mining + fine-tuning	Fine-tuning
Algorithm exploration before implementation [112]	New code	Multi-stage generation	Prompting
Fine-tuning on execution and edit history [103]	Existing code	Fine-tuning	Fine-tuning
Fine-tuning on performance-improving edits [94]	Existing code	Fine-tuning	Fine-tuning
Fine-tuning on correct and efficient code only [46]	New code	Fine-tuning	Fine-tuning

Table 3.2: Summary of methods to improve code efficiency using LLMs.

Many studies have investigated the capacity of LLMs to produce less energy-intensive code. Notably, Vartziotis *et al.* [99] evaluated the sustainability of auto-generated code produced by GitHub Copilot, ChatGPT and CodeWhisperer on 6 of the most popular Leetcode problems. Their experiment was run under two generation conditions: in the first generation condition, they only asked the LLM to solve the problem, while in the second condition, they asked it "Give me a runtime-optimized solution for this problem". The authors found that their LLMs produced a less environmentally conscious code than the optimized human solutions taken from Leetcode. However, despite its novel analysis method, this study presents one big caveat: the authors use only 6 problems from Leetcode, which, as I previously said, are among the most popular problems. This means that there may be a strong data contamination of the tested models by these problems, consequently biasing the results of the experiment.

A study on CodeLLaMa led by Cursaru *et al.* [24] observed the energy efficiency of LLM-generated code with respect to human-written source code on three well-known programming problems each in C++, JavaScript and Python. They found that the capacity of CodeLLaMa to generate green code was strongly mediated by the programming

language and specific problem at hand, while human solutions were generally more efficient. They also found that temperature seemed to have no effect on the efficiency of the code.

There is also a growing interest in examining the resource consumption of the LLMs themselves. Luccioni, Jernite and Strubell [68] studied the cost of inference using 88 task-specific and general-purpose ML models (including LLMs) across 10 tasks and 30 datasets. The tasks covered many applications of ML such as image classification, question answering or text-generation. They found, among other things, that generative tasks cost more energy than discriminative tasks, and that similarly, generalist models cost more energy than task-specific models for the same tasks.

Samsi *et al.* [91] were among the firsts to measure the inference energy usage of LLMs, particularly LLaMa models of various sizes, on the Alpaca and GSM8K datasets. They were able to measure the energy usage of the inference with different perspectives: energy per second, energy token, energy per response. The authors also varied multiple experimental parameters to study their effect such as the batch size, the number of shards, the size of the model and the type of GPU used.

While most studies focus on measuring the energy consumption of the inference phase, Berthelot *et al.* [14] used an LCA-based methodology to estimate the environmental impact of training and hosting a Stable Diffusion model for a whole year. While Stable Diffusion model is a latent diffusion model that generates images, *i.e.*, not an LLM, its characteristics are very similar to that of an LLM. The impacts studied were not only the electricity usage, but also the greenhouse gases emissions as well as the abiotic depletion for minerals and metals. Moreover, they not only studied the impact of the service provider, but also the impacts of the network and user terminals which use the service. The authors report that supporting one year of Stable Diffusion service can generate as much as 360 tons of CO^2 equivalent.

More recently, Jegham *et al.* [52] estimated the energy consumption, water usage, and carbon emissions of 30 closed and open-source LLMs, including GPT-4o or DeepSeek-R1 as they would be deployed in commercial data centers. They found that reasoning models such as DeepSeek-R1 were among the most energy-intensive models, consuming over 33Wh per long prompt while a single short GPT-4o query consumes 0.43Wh. The authors illustrated that although AI is becoming cheaper and faster, its total environmental impact keeps increasing.

Taken together, these studies reveal the multifaceted and still poorly understood environmental footprint of large language models. While some efforts have begun to evaluate the sustainability of LLM-generated code or the energy costs of inference, much of this work remains fragmented–either limited to small-scale examples, or disconnected

from real-world developer practices. Moreover, only few studies address the embodied impacts of integrating LLMs directly into everyday software development workflows.

This is precisely the gap addressed by our second contribution (in chapter 5), which focuses on the energy consumption of LLM-powered code assistants as they are used in realistic developer scenarios. By measuring how different usage patterns and configurations influence energy consumption at inference time, our study seeks to provide insight into the actual sustainability of using code assistants in software engineering.

3.5 Conclusion

This growing body of work confirms that while LLMs are increasingly integrated into software development pipelines, their implications for software performance and sustainability remain only partially understood. Many studies evaluate code correctness or speed, but few explicitly measure energy consumption. Others explore LLM-driven optimization, but without considering the cost of the optimization itself. Additionally, the broader environmental impacts of LLM-based tooling–especially rebound effects, such as increased tool usage–are rarely quantified. These gaps motivate the three contributions of this thesis: (i) a performance-focused comparison of LLM-generated code, (ii) a study of the energy consumption of code assistants, and (iii) an analysis of the energy trade-offs of LLM-based code optimization.

Chapter 4

A performance study of LLM-generated code on leetcode

Chapter summary. This first contribution focuses on how efficient the code generated by LLMs really is, especially when they're just asked to produce working solutions to algorithmic problems. We tested 18 different models on a wide set of Leetcode tasks, and compared their output to human-written code. Our findings show that most LLMs produce code with similar performance, regardless of their size or training data. Interestingly, they often generate code that runs faster than human solutions–though higher generation temperatures tend to lead to buggier or less efficient outputs. We also point out that while Leetcode is a useful testbed, it has major issues with data contamination and measure reliability, which could skew evaluation results if not handled carefully. This study helps paint a clearer picture of how well LLMs perform out of the box, and why it's important to measure not just correctness, but also efficiency.

4.1 Introduction

Since LLMs have increased in popularity, they have been integrated in many developer tools, like GitHub Copilot, a tool that generates code for the developers. There has also been a significant amount of research dedicated to evaluate the LLMs and their limits. Notably, many researchers have assessed various qualities of the code generated by LLM, with a particular emphasis on code correctness [20, 10, 113, 42, 105] or the presence of vulnerabilities in the code [79, 92, 83, 53]. Some studies delve deeper into the assessment of the code correctness by observing the impact of changes in the configuration (*e.g.*, temperature parameter) or context of the generation (*e.g.*, modifying the prompt) [105, 29].

However, to the best of my knowledge, when this contribution was first made, there was no research work evaluating the performance of the code generated by LLMs on normal code generation tasks. Yet, having code that runs faster is an often sought-after characteristic of a program. As explained in chapter 2, in today's context the energy consumption of software systems has become a major concern. Thus, improving software efficiency is particularly relevant since increasing the performance of a program can also lead to energy consumption reduction [102, 7].

Yet, the process of code optimization is lengthy and intricate, requiring careful attention and a certain level of expertise, especially when searching for the best-performing algorithm, selecting the most appropriate data structure, or struggling with memory hierarchy. However, this process is necessary to identify opportunities for improvement that may result in minor reductions in execution time. LLMs can be used as a way to make this process easier, *e.g.*, by generating performance-improving code edits [70, 39, 21]. Garg *et al.* [40] presented RAPGen, a method for generating zero-shot prompts to enhance performance. However, while users seem to put a lot of trust in code generated by LLMs, they still have trouble reviewing it [98, 92], which can lead to slow code being shipped in production, especially if a LLM generates inefficient code.

This is why in this contribution, we study the performance of LLM-generated code, when the LLM is simply tasked to generate correct code. To that end, we will source problems from Leetcode, a programming contest website which hosts a myriad of programming problems. We describe in detail the reason for choosing Leetcode in section 4.2. To guide our study, we come up with the following research questions:

RQ1: *Can Leetcode be used as a reliable dataset and a benchmark platform for evaluating LLMs?* Leetcode can serve as both a dataset of problems and a tool to evaluate and measure solutions to the problems. Particularly, we study if the dataset is subject to data contamination and if the measures provided by Leetcode are reliable.

RQ2: *Are there notable differences between the performance of the code generated by different LLMs?* LLMs differ greatly in terms of generating correct code, so we want to know if they also differ in terms of generating efficient code.

RQ3: *Is there an effect of the success rate and the temperature of the LLM on the code's performance?* Having a higher temperature decreases the capacity of the LLMs to generate valid code, so we aim to study if this also applies to the performance of the code. In the same way, we also study if an LLM that is very good at generating valid code on one problem, is going to generate efficient solutions to this problem.

RQ4: *How efficient are the solutions generated by the LLMs compared to human solutions?* Comparing the LLMs to a set of human-authored solutions can provide insights into their position relative to humans in terms of code performance.

The key contributions presented in this chapter are as follows: (i) We study the performance of the code generated by 18 LLMs on 204 problems and investigate performance differences across models using a novel method for measuring and comparing the performance of LLM-generated code. (ii) We compare the performance of the code generated by LLMs to the code written by humans. (iii) Incidentally, we evaluate the usability of Leetcode,[1] a public repository of algorithmic problems that we use as a dataset.

Main findings & implications. Our results show that LLMs produce code with comparable performance, irrespective of the adopted model. Additionally, using a higher temperature parameter let to more frequent inefficient and slow code. We also found that LLMs are capable of generating that is, on average, more efficient than the code written by humans. Lastly, we uncover strong risks associated with the use of Leetcode as a dataset for LLM evaluation as there is important data contamination issues concerning older problems.

From section 4.2 to section 4.4, we describe the tasks dataset and the models we selected, outline our experiment setup, and explain the methodology we followed to analyze the obtained results, respectively. We report in section 4.5 on the results of our evaluation and provide a critical discussion in section 4.6. Finally, section 4.8 concludes the chapter.

4.2 Dataset

In this section, we describe our dataset and explain our reasons for choosing it.

Task selection. The input tasks—*i.e.*, problems specified by a prompt—we consider to generate code from various LLMs has to meet the following requirements:

- A given problem, should offer multiple candidate solutions, whose generated code performs differently. This ensures one can observe differences across the various LLMs;
- Generated solutions should exhibit variable execution times. Given more complex inputs, one should differentiate $\mathcal{O}(n)$ from $\mathcal{O}(2n)$ and $\mathcal{O}(n^2)$ algorithms.

[1] https://leetcode.com/

As a result, task datasets, such as HumanEval or *Mostly Basic Python Programming* (MBPP), which are classically used when evaluating LLMs for code assessments [20, 77, 4, 9], cannot be considered for our purpose. Indeed, while they do provide unit tests to drive the generations, the size of the inputs in these unit tests remains small and fails to scale to appreciate performance issues. Moreover, the solutions that need to be generated are often very short, which would lead to fewer possible variations between implementations. Also, the fact that many problems are not algorithmic by nature makes them less prone to inefficient practices and performance variation. To face such issues, we used input prompts that were built from Leetcode problems. Leetcode is an online judge platform that suggests programming problems to registered users. It addresses the above limitations as it provides algorithmic problems with varying levels of difficulty and test cases with large input sizes. Leetcode also exposes a GraphQL API[2] to fetch relevant metadata on the problems, such as exercise instructions, code snippets containing the signature of the function to generate, as well as the difficulty and topics of the problem.

We followed an experimental design similar to the one used by Döderlein *et al.* [29], while using a different set of Leetcode questions. To avoid data contamination, which happens when an LLM is tested on data it was trained on,[3] we only considered problems that were published after January 1^{st}, 2023. As all the LLMs (except GitHub Copilot) we evaluated were trained using datasets older than these problems, we avoid any data contamination. As these problems are published by Leetcode in the context of programming competitions, they are always original. However, GitHub Copilotbeing an online closed-source tool, one cannot tell whether it underwent training with the problem set we employed. Our set was composed of 204 problems—labeled as 56 easy problems, 104 medium problems, and 44 hard problems, as classified by Leetcode upon the problem's publication.

To answer RQ1, we also performed our experiment a second time on the set of questions used by Döderlein *et al.* [29], which is composed of 300 problems (95 easy, 105 medium, and 100 hard) from the most liked problems of Leetcode. We will refer to this dataset as the "old" dataset, and to our dataset of problems published during 2023 as the "new" dataset. Code generation was performed in Python for its ease of use and because of the prevalence of Python-written datasets for evaluating LLMs [20, 10].

Input prompts. The instructions given by Leetcode for each programming problem contain *(i)* the description of the problem, *(ii)* examples of inputs and outputs, and *(iii)* constraints on the input data. To build the prompts inputted to the LLMs, we

[2] https://pypi.org/project/python-leetcode/
[3] You can find more details on data contamination here.

chose to only include the description of the problem. This choice aimed to maintain the prompt's conciseness and ensure compliance with potential LLMs' length restrictions. Indeed, LLMs are constrained by a context size, imposing the maximum number of tokens they can process at once, encompassing both the prompt and the answer. In our study, the majority of LLMs impose a context window of 1024 tokens. Figure 4.1 illustrates a descriptive prompt (enclosed within triple quotes) along with a solution generated using this prompt and the corresponding benchmark instructions. Leetcode occasionally includes an additional comment in the prompt, indicating the available methods of the elements passed to the solution (*e.g.*, binary trees). While adding examples in the prompt could potentially increase the likelihood of generating correct solutions [29], we chose not to include them due to the complexity of representing Leetcode's data structures (such as arrays, graphs, linked lists, trees, etc.) in textual form. On Leetcode's website, data

```python
# Start of the input prompt
"""
Given an integer array nums, return all the triplets `[nums[i], nums[j],nums[k]]`
↪  such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] ==
↪  0`.

Notice that the solution set must not contain duplicate triplets.
"""
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # End of the input prompt

        # Start of the generated code
        nums.sort()
        res = []
        for i in range(len(nums) - 2):
            ...
        # End of the generated code

# Start of the benchmarking code
def check():
    Solution().threeSum([82597, -9243, 83030, ...])
    Solution().threeSum([0, 0, 0, 0, ...])
    Solution().threeSum([0, 0, -1, -1, ...])

import pytest
@pytest.mark.benchmark(group="3sum")
def test_3sum_generated_1(benchmark):
    benchmark(check)
# End of the benchmarking code
```

Figure 4.1: Example of problem's input prompt, generated code, and benchmarking code.

structures are textually represented using an array notation format with brackets (*e.g.*, "`[1, 2, 3]`"), which might be misleading for an LLM working with data structures that differ from arrays or lists.

Canonical solutions. Each problem was matched with a single valid solution, written by a human and fetched from various sources. These solutions, referred to as "canonical solutions", are considered as baselines during the benchmarking process, although they may not represent the entirety of human-written solutions. They were used to assess the stability of the measuring process. Most of the canonical solutions were fetched from the WalkCC repository of Leetcode solutions.[4] When one question did not come up with any solution in Python from this repository, we selected one solution proposed by the Leetcode community among the most upvoted ones (starting from the one with the most upvotes), which are also publicly available on the Leetcode website. These modifications only included changes to the function name, variable names, and type hints except for one specific case, where a recursive solution provided by WalkCC was replaced by an iterative one from the Leetcode community because of the stack limit on our local setup. We ended up with one Python-based canonical solution for each problem of our dataset.

Test cases. Testing generated solutions is a twofold process, as both the correctness and scalability (performance) of each solution must be checked. First, to ensure that generated solutions are correct, we provide such solutions to the Leetcode online judge system, which in turn validates them based on its test suites. Second, we execute the valid solutions with input data to assess their performance. To retrieve such input data, we crawled through the problems' instructions and extracted two to three examples of inputs and expected outputs provided by Leetcode. However, such inputs were too small to exhibit significant performance differences, *e.g.*, arrays containing only two to five elements. To execute the generated solutions with larger input data, we took advantage of Leetcode's judge system that returns the inputs and expected output of the first failed test of a test suite. We noticed that Leetcode tests if the submitted solution is inefficient by using a timeout system. Since all selected problems are algorithmic by nature, one simple way to put a heavier load on their implementations is to increase the size of the inputs. For every problem, we thus submitted a modified version of a canonical solution that failed only when the size of the first parameter exceeded a certain threshold. We then set this threshold manually multiple times for every problem to extract three different inputs for each problem, resulting in more than 150 MB of fetched input data with most inputs having over 10^5 elements.

[4] https://github.com/walkccc/LeetCode

LLM Model	Model family	Size	RQ1
GitHub Copilot	Codex	11	✓
CodeGen-Mono 6B	CodeGen	6	✓
CodeGen-Mono 2B	CodeGen	2	✓
CodeGen-Mono 350M	CodeGen	0.35	✓
CodeGen2.5-7B-mono	CodeGen2.5	7	
CodeGen2.5-7B-instruct	CodeGen2.5	7	
CodeLlama-7B-instruct	CodeLlama	7	
CodeLlama-7B	CodeLlama	7	
CodeLlama-7B-python	CodeLlama	7	
CodeLlama-13B-instruct	CodeLlama	13	
CodeLlama-13B-python	CodeLlama	13	
replit-code-v1-3b	replit-code	3	
WizardCoder-pythin	WizardCoder	7	
SantaCoder	Santacoder	1.1	✓
StarCoder	StarCoder	15.5	
InCoder 6B	Incoder	6	✓
InCoder 1B	Incoder	1	✓
CodeParrot	Codeparrot	1.5	✓

Table 4.1: LLMs considered in our study. Models with the RQ1 checkmark were also evaluated on the "old" dataset. Size is in billions of parameters

4.3 Experiment Setup

This section describes our experiment setup to generate and validate the solutions produced by the LLMs. First, we describe the models we used as well as how solutions were generated from each LLM. Then, we outline the three-step process used to filter invalid solutions. Finally, we explain how the run time of the generated solutions was measured.

4.3.1 LLMs Under Study

Our empirical study covers a total of 18 LLMs, specifically designed for coding purposes. We selected the 18 popular code LLMs from Hugging Face,[5] as well as GitHub Copilot, which is an online closed-source code assistant. The LLMs were selected based on the number of downloads and likes they exhibited. We chose GitHub Copilotto offer a comparison between a commercial LLM and open-source LLMs, but did not choose any other GPT models from OpenAI because of their cost. Table 4.1 summarizes all the LLMs considered in this study. This includes variants of the same base LLM—*i.e.*, models with varying sizes (in billions of parameters) or different training data, such as variants of CodeGen, InCoder, and CodeT5. LLMs belonging to the same family are models closely related in terms of training data and method. We first performed our experiment in March 2023 with a subset of the models presented here, with the "old"

[5] https://huggingface.co

dataset. We then performed it a second time in September 2023 with all the models and the "new" dataset.

4.3.2 Code Generation

We generated 10 solutions for each problem from our dataset by varying the temperature of the LLMs used to generate them. Specifically, we considered 6 different temperatures (`0.1`, `0.2`, `0.4`, `0.6`, `0.8`, and `1.0`). As Copilot's temperature cannot be configured, we used its default temperature for generating all the problems.

Generating code with GitHub Copilot. Automatically generating with GitHub Copilotfor a reproducible experiment proved to be a difficult task. Firstly, the code suggestion feature of Copilot activates when typing in a text editor with the installed Copilot plugin. Additionally, GitHub Copilotproduces code based on a context that encompasses the current file and the files previously accessed by the user, impacting the generated solutions. Lastly, we noticed a caching mechanism on the GitHub Copilotserver side, which resulted in very similar or identical solutions if we generated multiple solutions for a given problem in the same session. To address these issues, we used a generation method similar to the one used by Döderlein *et al.* [29] by instrumenting the GitHub CopilotNeovim plugin[6] and restarting the plugin between every generation to avoid the caching effect. This method allowed us to automatically generate solutions in a quick and isolated fashion. On top of that, GitHub Copilotprovides 2 means of generation: *inline* generations (the suggested code is integrated with the editor) and *panel* generations (Copilot generates at most 10 completions and displays them on a panel next to the editor). We chose to exclusively use inline generations, as panel generations yielded worse results in terms of functional correctness than inline generations.

Generating code with open-source models. Regarding the open-source models, we generated solutions by deploying the models on servers provided by the Grid5000 platform [12]. We used the Deepspeed library to make the generation process faster and fit larger models on our GPUs. Concerning the sampling, we used the same methods as Chen *et al.* [20] and used nucleus-sampling [43] with top $p = 0.95$. The maximum number of tokens to be generated was set to 600. This is because the LLMs we used have a limited context size of 1024 tokens (prompt included). Thus, to avoid exceeding the context size, we had to limit the number of tokens to generate. We also verified it did not significantly impact the functional validity of the LLMs.

[6]https://github.com/github/copilot.vim

In total, we generated 2,040 solutions with COPILOT and 12,240 with each of the eight other models, resulting in 210,120 generated solutions overall.

4.3.3 Validation

Each generated solution was tested to ensure its functional correctness following a three-step process. At every step, if a solution was found to be invalid, it was excluded from subsequent stages of the experiment. The process was as follows:

i) Local validation. We filtered out code generations that included easy-to-spot errors, such as syntax errors or runtime errors, by using the small inputs we fetched earlier (see section 4.2 - *Test cases*). While this step was not strictly necessary, it quickly reduced the number of solutions to be validated in the next step;

ii) Leetcode validation. Next, we submitted the solutions to the Leetcode judge system using the Leetcode GRAPHQL API, where they underwent a rigorous test suite managed by Leetcode. We kept the solutions that passed all the test cases or exceeded Leetcode's allocated time limit. The latter was kept to ensure that correct solutions that were too slow remained included in the benchmark;

iii) Exclusion of timeouts and other errors. Finally, we excluded the solutions that reported errors when executed with our benchmarking setup using the large inputs fetched beforehand. We invalidated the code generations that took more than 10 seconds to run or raised an error. Most of the errors raised in this step were recursion errors caused by differences between Leetcode's Python interpreter and ours. Indeed, Leetcode's seemed to have a higher recursion limit than ours, which we set to 10,000 instead of the default 1,000 (we could not manage to set it any higher). Additional errors occurred because we developed our helper classes differently from Leetcode's implementation. Although these classes are provided by Leetcode during the submission process, they are not publicly available. Out of the 4,930 invalidated solutions that were caught in this step, 4,863 (98.6%) were due to timeouts, 20 (0.04%) to recursion errors, and 47 (0.1%) to other errors. Following this validation process, the initial set of 210,120 generated solutions was pruned down to 7,481 (3.6%) valid solutions remaining across the 18 LLMs.

4.3.4 Measuring run time

We measured the performance as the run time of the generated solutions using pytest-benchmark,[7] which runs pytest unit tests multiple times to obtain run times statistics. The measurements were performed using parameters that ensured each solution ran at

[7] https://github.com/ionelmc/pytest-benchmark

least 10 times and for at least 1 second in total. We did not perform warm-up runs of the benchmarks, as we did not notice any significant difference in the measured time during preliminary testing. To facilitate the measurement protocol, the generated solutions were sorted into "runs", based on the specific LLM and temperature that were used during the code generation. Within each run, which was defined by a unique combination of model and temperature, the solutions were measured in sequence during a single program execution. Furthermore, in every run, we added the canonical solutions we previously collected, which would run alongside the generated solutions. This approach ensured that the same canonical solutions were executed in every run, allowing us to maintain measurement stability. Specifically, we calculated the standard deviation of the canonical solution run times across all runs, thus providing a reliable measure of the variability of the measurement protocol. We observed that over 96% (196 out of 204) of the canonical solutions had a standard deviation lower than $1/10^{th}$ of their average run time, which we deemed to be an acceptable level of variation.

The cluster we used to run the benchmark was the chiclet cluster of the Grid5000 testbed.[8] It hosts 2 AMD EPYC 7301, with 16 cores per CPU and 128GB of memory. When using the node, all the cores of both CPUs were reserved, but only one was used at a time to maximize the stability of the measurement protocol.

4.3.5 Replication package

All the artifacts of this study, including our results, code, and datasets, are available in the following public repository: https://zenodo.org/doi/10.5281/zenodo.7898304.

4.4 Data Analysis

In this section, we describe the methods we adopted to analyze our results. These methods fall into one of the two following categories: functional correctness and code performance.

4.4.1 Functional Correctness

The functional correctness of an LLM defines how much the LLM outputs code conforming to the program contract (as specified by the input prompt). To evaluate the functional correctness of our LLMs, we computed their pass@k metrics with $k = 1$ and $k = 10$, using

[8] http://grid5000.fr

the unbiased estimator proposed by Chen *et al.* [20]. The pass@k unbiased estimator which, from k samples produced, considers the test as successful if one of these samples passes all the tests, is computed as follows (with n the total number of samples, c the number of correct samples and $\mathbb{E}$ the expected value):

$$pass@k := \mathop{\mathbb{E}}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4.1}$$

As Chen *et al.* [20] suggest, we calculated the pass@k for each temperature when evaluating an LLM's functional correctness and considered the best one as the pass@k for that LLM.

4.4.2 Code Performance

To measure the code performance, we considered three different metrics. First, we used the memory usage reported by Leetcode. Then, we computed the median of the run times measured by pytest-benchmark for every generated solution. Lastly, to compare the LLMs solutions to human-submitted solutions, we also used the rank reported by Leetcode when validating the solution. This rank is a number between 0 and 100 that indicates the share of submitted solutions that are slower than the current solution (*e.g.*, if a solution has a rank of 90, it is faster than 90% of the submitted solutions on Leetcode).

To assess the LLMs' performances, we conducted pairwise comparisons as follows: For each pair of LLMs, we identified problems where both models generated more than 5 valid solutions. For each identified problem, we conducted a Student t-test on the mean run time of the generations to determine if there was a significant difference. Then, for each pair A-B of LLMs, we computed the ratio of problems where A's code was significantly faster than B's and where B's code was significantly faster than A's.

4.5 Results

In this section, we summarize the key observations from our experiment and answer our research questions. In Table 4.2, you can also find the functional validity results of the different LLMs on both of our Leetcode datasets. Our results are also available in the form of a companion notebook in our replication package. The companion notebook offers more insight into the results and additional graphs.

LLM Model	Pass@1	Pass@10
StarCoder	0.095	0.132
CodeLlama-13B-python	0.093	0.201
GitHub Copilot	0.092	0.196
CodeLlama-7B-instruct	0.082	0.191
CodeLlama-13B-instruct	0.078	0.206
WizardCoder-python-7B	0.075	0.157
CodeGen2.5-7B-mono	0.066	0.147
CodeGen2.5-7B-instruct	0.062	0.142
CodeLlama-7B-python	0.047	0.172
CodeGen-6B-mono	0.045	0.113
CodeGen-2B-mono	0.038	0.103
replit-code-v1-3b	0.025	0.083
InCoder-6B	0.021	0.064
SantaCoder	0.015	0.064
CodeLlama-7B	0.014	0.015
InCoder-1B	0.012	0.039
CodeGen-350M-mono	0.007	0.039
CodeParrot	0.002	0.015

Table 4.2: Functional validity of the LLMs on Leetcode (by decreasing `pass@1`, higher is better)

4.5.1 RQ1: Can Leetcode be used as a dataset and a benchmark platform for evaluating LLMs?

4.5.1.1 Can Leetcode problems be adopted as a dataset for LLM generation?

As on can observe in Figure 4.2, the generated codes exhibit on a significant drop in functional correctness between the two datasets. The difference here is pretty staggering for every tested LLM, reporting on a tenfold decrease in `pass@k`. We believe this issue may stem from data contamination in the old dataset. Data contamination occurs when an LLM is assessed on data that was included in the training dataset, introducing bias into the evaluation process. In our case, a significant number of questions in the old dataset are widely known and have been extensively shared on GitHub. For instance, a search for the prompt of the "3sum" Leetcode problem on GitHub yields approximately 4.000 matches in public repositories. These questions are also old enough to likely be included in the training datasets of the LLMs under study, as the majority of their training datasets have a cut-off date between 2021 and 2022. Due to this data contamination, LLMs tend to recite, reproducing verbatim source code when generating solutions. This phenomenon is more pronounced when the prompt is highly specific and lacks contextual information, as seen in Leetcode prompts that closely match GitHub repositories. The observed shift in functional validity between the two datasets could also arise from a genuine difference

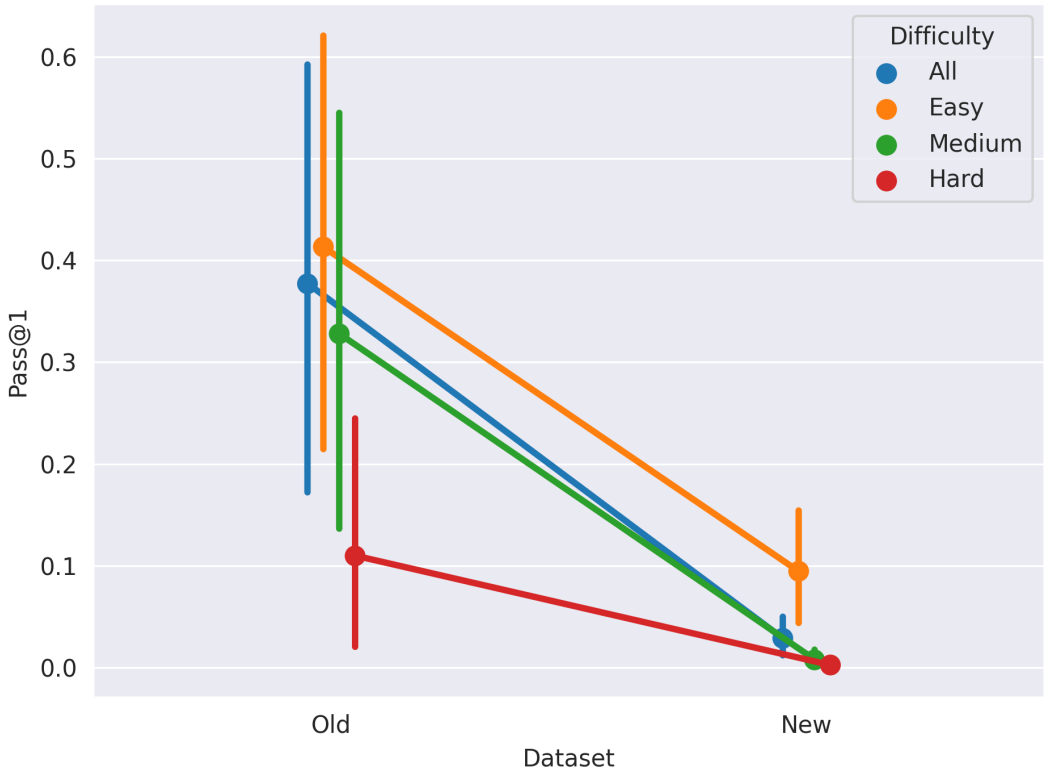

Figure 4.2: Average pass@1 of the evaluated LLMs for every difficulty and dataset, with 95% confidence interval (higher is better)

in the difficulty of the questions within each dataset. However, quantifying this last hypothesis proves to be challenging.

4.5.1.2 Are Leetcode measurements reliable?

Run time. As reported in Figure 4.3, the coefficient of variation of the Leetcode measures (0.089) is slightly higher than the coefficient of variation of the local measures (0.035). This suggests that Leetcode's measuring setup is less suited to ensure accurate benchmarks.

We also study the correlation between our local measurements and Leetcode's, and we notice two issues: (1) the times measured locally and by Leetcode only slightly correlate on average (0.28). For some problems, the measures are highly correlated (> 0.8) while, for others, they are almost not (< 0.2). This is more apparent when we look at the scatter plot showing the measures of some problems in Figure 4.4. In this problem, there are four clusters of generations with a different locally measured time, but the clusters are indiscernible in terms of Leetcode time. This could be due to two main reasons.

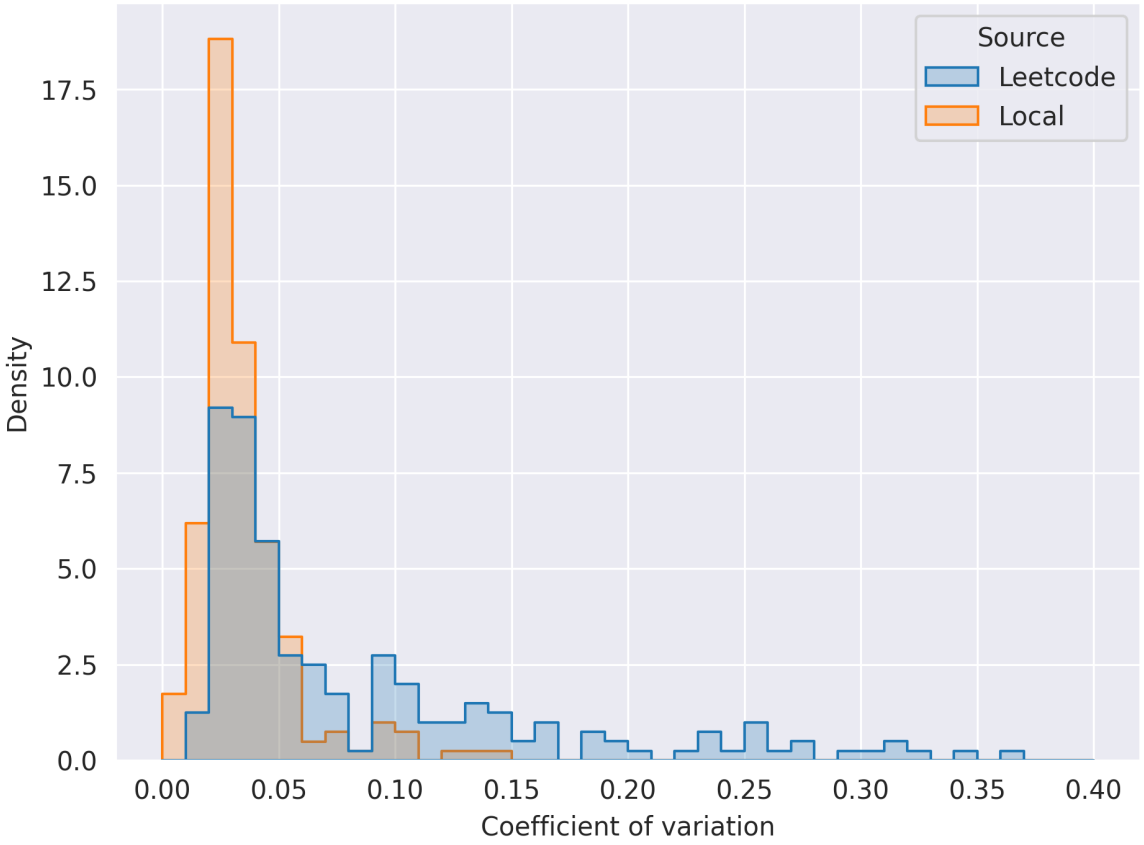

Figure 4.3: Coefficicent of variation of the time measured by Leetcode and locally for every problem using canonical solutions

Firstly, as previously discussed, the variance of the measures from Leetcode is higher and as such, there is much more noise in the measures, decreasing the precision. Secondly, the tests we employed may be more focused on performance testing than Leetcode's. Notably, our test suite comprises only three tests featuring significantly large inputs, potentially accounting for certain disparities in the results.

Although still usable, relying on the time reported by Leetcode introduces some limitations due to its higher variance. Consequently, Leetcode's measures cannot allow us to discern the differences in run time between different code implementations as precisely as locally measured time.

Memory usage. While we did not measure the memory ourselves, we observed the variation of the memory usage measure provided by Leetcode. We notice that the memory usage for the same solutions decreases over time. There is indeed a slight correlation of -0.24 between the day of the year we tested our solution and the memory usage. The

Figure 4.4: Scatter plot of the measures done by Leetcode and locally for every generation for the problem "*Difference between element sum and digit sum of an array*". Orange points are from multiple measures of the same canonical solution and serve as visual references for the measurement error

fact that the memory usage measure evolves renders comparisons between LLMs harder. While we could theoretically offset the memory usage when we detect changes over time, it would require testing the canonical solutions alongside the generated one on Leetcode.

Leetcode rank. The time ranking that Leetcode returns when we test a solution represents the share of submitted and valid solutions that are slower than ours. While this could be a great tool to rank LLMs among human-submitted solutions, we find that the ranking is heavily affected by our submissions and time. As you can see in Figure 4.5, the overall rank of the LLMs we tested decreases over time. To verify this, we tested GitHub Copilottwice, once as the first LLM, and a second time after testing all the LLMs. The first test of Copilot has an overall rank of 77, and the second test ranks down to 54, despite it being tested with the same solutions. This effect of the rank evolving becomes obvious when you consider that the rank is determined using all the previously accepted solutions, including ours. This means that by testing thousands of our solutions on Leetcode, we are actively changing the ranks of future tests.

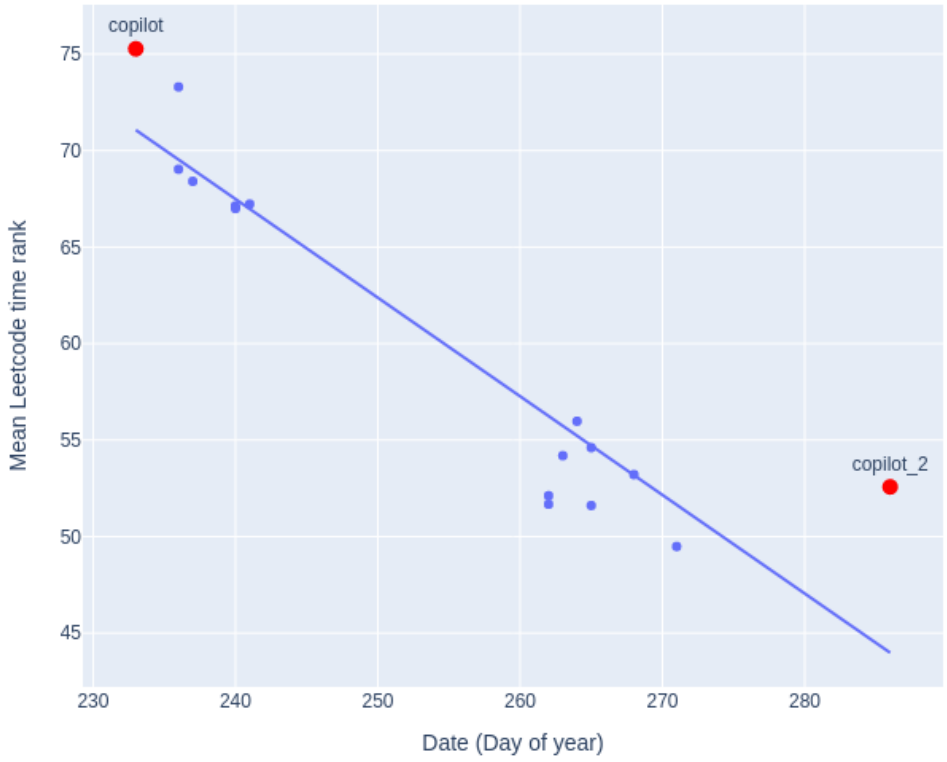

Figure 4.5: Scatter plot of LLM's rank and date they were tested on Leetcode. The two models in red are the same model tested on different dates

RQ1: The evaluation of LLMs using Leetcode questions as a dataset presents some challenges. Although Leetcode's questions could serve as a valuable dataset akin to HumanEval, limitations arise due to the constraint that only the problems published after the LLM's training dataset formation are usable for evaluating the LLM in question. This creates potential difficulties in reproducibility, particularly as new LLMs emerge, especially if they do not exclude Leetcode problems from their training datasets [49]. Additionally, while Leetcode's provided metrics, such as run time, memory usage, and rank may offer practicality in various scenarios, their usability and reliability are questioned when compared to more traditional measurement methods. The presence of these challenges emphasizes the need for careful consideration and scrutiny when adopting Leetcode to evaluate LLMs.

4.5.2 RQ2: Are there notable differences in performances between LLMs?

The pairwise comparison depicted in Figure 4.6 reveals subtle distinctions in the performances of various LLMs. Notably, some models, such as StarCoder and the CodeLlama model with 13B parameters specialized in Python, consistently exhibit slightly superior results compared to others. Despite these observed variations, the mean Cohen's d effect size measures a mere 0.024, a statistically insignificant magnitude. This suggests that the practical impact of these differences on the mean speed of code generation is remarkably small. For instance, when comparing CodeLLama-13-instruct and CodeGen25-7B-mono, CodeLLama outperforms the latter in a statistically significant manner in 3 problems out of 8. However, it is crucial to note that the mean performance difference between these models is a mere 0.02 standard deviation. It thus seems that improving an LLM in terms of functional validity does not significantly impact the performance of the code it generates. This may be due to different factors, such as the fact that most LLMs share the same datasets or that they are trained to produce valid code and not fast code. Improving the performance of an LLM could be done by curating a training dataset of only efficient code and fine-tuning one of the foundational models we used, or by using reinforcement learning to "teach" the model to produce better code. Madaan *et al.* produced an LLM that could improve the performance of code [70]. This LLM could be leveraged in a generation pipeline to directly improve the generated code.

> **RQ2**: Our analysis uncovers statistical differences in the performance of generated code among different LLMs. However, the effect size, as measured by Cohen's d, is so negligible that it raises questions about the practical significance of these differences. Despite some models consistently outperforming others, the overall impact on the mean efficiency of LLM-generated code appears to be minimal.

4.5.3 RQ3: Is there an effect of the functional validity of the LLM and its temperature on the generated code's performance?

Functional validity. When calculating for every problem the correlation between the success rate of the LLM that generated the solution and the run time of the solution, we find that there is only a very slight negative correlation (-0.08) between the success

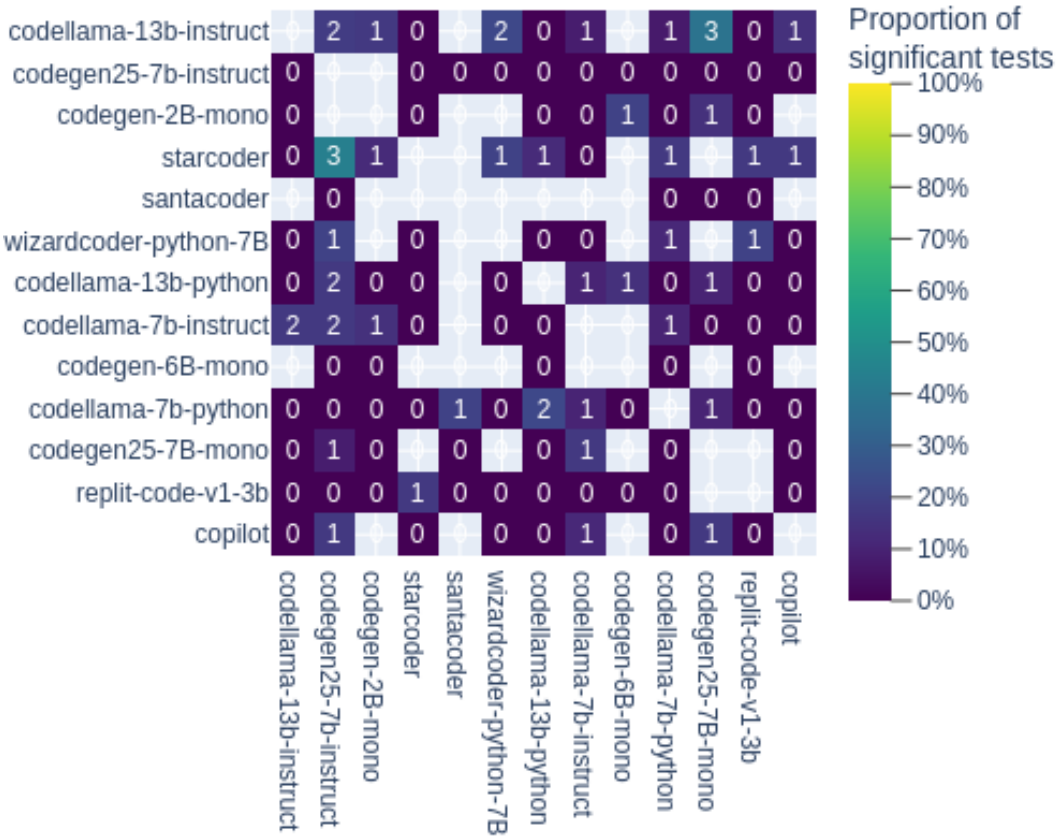

Figure 4.6: Number of problems where an LLM (row) is better than another (column)

rate and the performance. There is close to no correlation (-0.11) observed between the success rate of the model and the variation in the performance of the generated code.

Temperature. There is no correlation (0.05) observed between the temperature of the generations and the performance of the generated code. This means that the temperature does not affect how fast the solutions are. However, we observe that temperature is moderately correlated (0.41) with higher variations in performances. This means that higher temperatures tend to increase the variation in performance across generations. The complete distribution of the correlation for each of the 24 problems can be seen in Figure 4.7. So, while increasing the temperature leads to a lower success rate [29], it can help find a faster solution with an extended exploration of generations. The fact that the temperature increases the variation in performances comforts the idea that higher temperatures lead to more diverse outcomes.

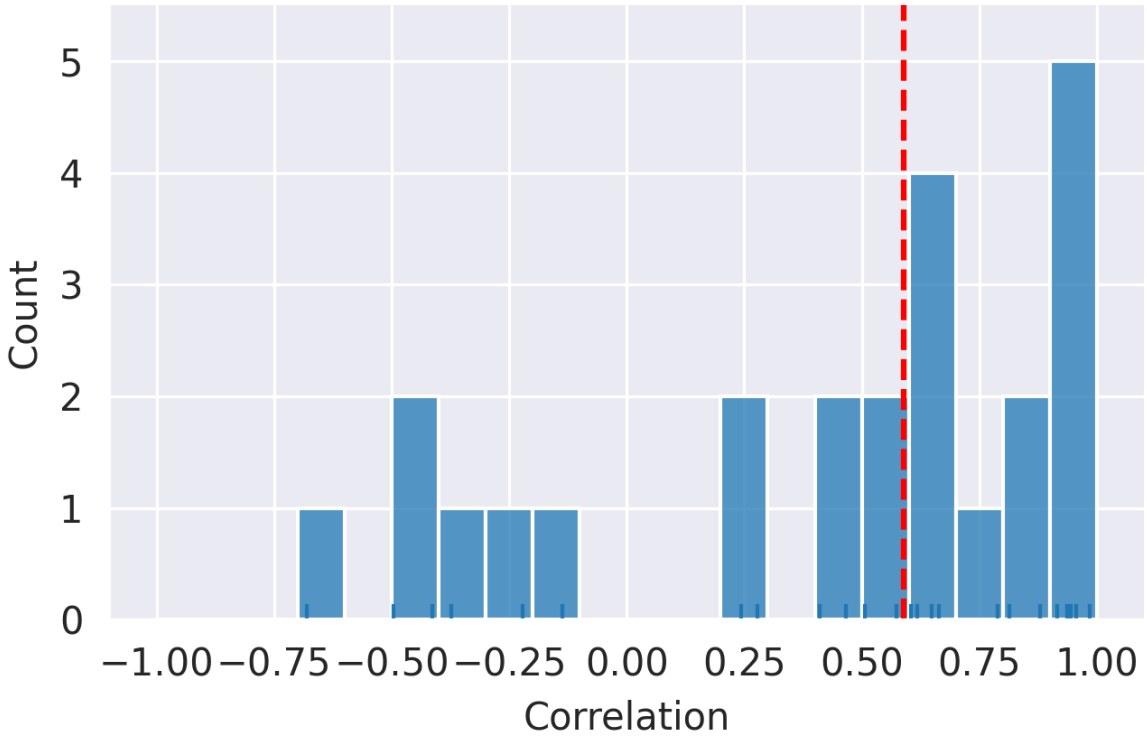

Figure 4.7: Distribution of correlations for every problem between the temperature and the variation of the performance. The red line is the median

RQ3: Our analysis of LLMs indicates that the quality of generated code does not have a substantial impact on its performance. However, we observed that modifying temperature settings within an LLM significantly affects the diversity of code performances produced. This implies that, while code validity may not be a decisive factor in performance, adjusting temperature settings can be a valuable strategy to enhance the variety of outcomes in code generation processes.

4.5.4 RQ4: How fast are LLMs compared to humans ?

As previously stated, the Leetcode time ranking evolves, so we chose to compare the second model we tested on Leetcode with humans (Copilot being the first, it did not have enough generations overall because of its lack of a temperature setting). The results of this comparison are depicted in Figure 4.8. The comparison is done using the Leetcode ranking, with the assumption that most of the previous submissions were made by humans.

We observe in Figure 4.8 that the solutions generated from LLMs are faster than most previous submissions with a mean rank of 73%, and that it even generated some solutions that were faster than 95% of the previous submissions.

> **RQ4**: It seems that the LLMs are faster than most of the human solutions on Leetcode, on average. If the LLM we tested were in an actual competition, his valid solutions would be on average faster than 73% of the other solutions on Leetcode.

4.6 Discussion

In this section, we discuss the results reported in the previous section and their implications.

On the Leetcode measures and usability. The data contamination issue we unveiled poses a significant challenge in the evaluation of LLMs, as it prevents an accurate evaluation of their real performances. Because Leetcode's problems are not filtered from the training datasets, as research on LLMs continues, even the newer problems might contaminate future training datasets, thus rendering reproduction of our study harder [49]. This conclusion also holds for any study using Leetcode as an evaluation dataset, such as [16, 29, 76]. However, we believe that the methodologies employed and the conclusions drawn would likely hold validity with alternative sets of questions from Leetcode or other performance-oriented datasets. This suggests that future studies seeking to replicate our findings would primarily need to change the dataset employed for assessing LLMs. Addressing the data contamination concern could involve leveraging pre-filtered evaluation datasets, such as HumanEval, already separated from the training processes, and repurposing them into performance evaluation datasets.

On functional correctness. The ranking of the functional correctness of the LLMs is consistent with the previous evaluations of the LLMs on HumanEval [20, 4, 77, 9, 57, 69, 90]. However, it seems that InCoder performs worse than presented in its introductory paper [37], which may be due to our experimental protocol—using it only for left-to-right generation instead of infilling like it was built for.

We observe that Leetcode's problems seem harder for the LLMs to solve than HumanEval's problems. Indeed, StarCoder, the model that performed the best with a `pass@1` of 0.09, had a `pass@1` of 0.408 on HumanEval [57]. We believe this is due to multiple factors. First, our Leetcode prompts and expected solutions are longer than in HumanEval (the average length of solution in HumanEval is 180 characters *vs* 425 characters in our dataset), thus increasing the chance of a generation to fail. Indeed,

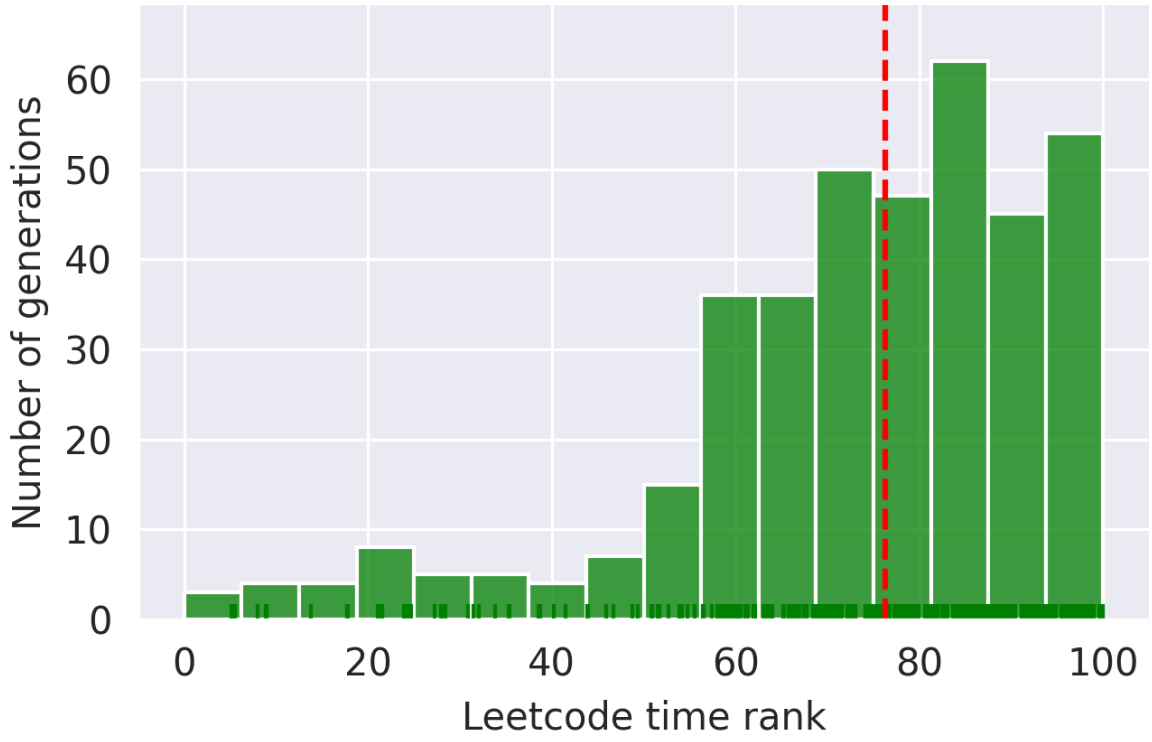

Figure 4.8: Distribution of the ranking for the CodeGen-6B-mono model

having to generate more code can lead to a higher chance of making a mistake, as shown by a correlation of -0.30 between the solution's length and the success rate of the problem. Second, Leetcode problems come from programming competitions and need a lot of thinking to be solved. The causal generation of the LLMs does not allow a "thinking" process to happen, which for harder problems causes a drop in functional validity. This could be solved by making the LLMs mimic human thinking with methods, such as Chain-of-Thought prompting [106].

On the performance of generated code. To the best of our knowledge, our methodology for evaluating LLMs based on the performance of generated code is novel and could serve as a benchmarking approach for future studies involving new LLMs and datasets. While we aimed for a singular performance score for LLMs, similar to pass@k, the varying rates at which LLMs generate valid solutions posed a challenge, leading us to employ pairwise comparisons. An improvement to our method could involve using an optimal solution as a reference point and comparing the speed of each generated solution to the optimal solution's speed. Speed would thus be expressed as a factor relative to the optimal time, addressing the issue of problems having different time scales and potentially laying the groundwork for a more comprehensive "performance score".

The low success rate of LLMs on Leetcode posed a considerable challenge for their comparison. Among the 204 problems, only 24 had *(i)* valid solutions from at least 10 different models (out of the 18 assessed LLMs) and *(ii)* at least 10 valid solutions in total (see to the companion notebook). This low success rate complicated pairwise comparisons, as numerous models could not be compared due to an insufficient number of common problems with a substantial number of generated solutions.

Our discoveries offer valuable insights for developers in their choice of LLMs. For example, when developers are considering an LLM for tasks like code generation, such as with GitHub Copilot, the performance of the generated code may be an important concern. With our findings, developers can be assured that there is no significant variance in the performance of code generated by different LLMs. This means that if they aim to have fast code, there's no necessity to consistently opt for the largest model available; instead, a smaller one suffices. We also hope that our findings will incentivize further research on building new LLMs that produce even more efficient code.

4.7 Limitations & Threats to Validity

In this section, we address several limitations of our study.

Regarding functional correctness, although we did our best to generate code in the most optimal conditions, some changes to the input prompt (*i.e.*, adding examples and constraints), or the configuration of the models may have changed the performance of the LLMs. However, we believe that this should not significantly impact the validity of our experiment, as all the models were configured similarly.

GitHub Copilotbeing a closed-source tool, it might be retrained without the community's knowledge, which could potentially lead to a modification in the tool's performance in upcoming experiments.

The majority of our validation and benchmarking process relied on Leetcode's test suite and online judge system. Thus, it is possible that some big test cases we extracted for the benchmarking favored some types of implementations over others. We mitigated this issue by systematically fetching three test cases and by having a large dataset of problems to generate from. During our experiments, we also tried to generate plausible inputs using random generators, but this method did not yield satisfying results because randomly generating data structures with specific shapes, or properties (*e.g.*, generating valid regular expressions) is difficult to achieve.

One gap in our study is that we do not consider memory usage at all when studying the LLMs. This is because our benchmarking setup did not allow for memory monitoring

and doing otherwise would have cost us a lot of time. Moreover, we believe our results obtained with only the run time to be self-sufficient.

The way we compared a LLM to a human using Leetcode rankings gives a good idea of where the LLM stands in terms of performance. However, because the ranking evolves and we have no information about the population the LLM is ranked against, the results here should be taken with a grain of salt.

It is also important to note that Leetcode as an evaluation dataset suffers from some issues. As we only evaluate the LLMs on algorithmic problems, the performances of the LLMs are hard to generalize across all programming fields. However, this is difficult to improve on for similar reasons to HumanEval: we only have a limited context size and have to make the LLM generate in one go a completion to some code, which must be self-contained (meaning, all the information needed to generate the solution must be in the prompt). Also, in terms of performance, it is difficult to find another kind of self-contained code than algorithmic problems that have, such variations in performance. While studying the LLMs on SQL generation could be a good idea, it would not fit into our studies with generalist programming languages.

Regarding Leetcode as a platform, we are heavily dependent on it for validating the generated solutions and are limited by its daily rate limits of $1,000$ submissions per account. For future studies, it would be great to consider alternatives to Leetcode, such as the project CodeNet [86], which does not have as many restrictions as Leetcode.

4.8 Conclusion

In this chapter, we provided a preliminary answer to the question *"How efficient are the solutions generated by LLMs?"* by presenting a comprehensive analysis of the performance of code generated by various LLMs, which uses a novel methodology that measures and compares the runtime speed of solutions to algorithmic problems. Our findings suggest that the performance of the generated code is largely similar across different models, regardless of their size or training data. Furthermore, increasing the temperature parameter during code generation leads to a greater variance in performance, though not necessarily to better or worse solutions on average. We also critically evaluated the suitability of Leetcode as a dataset and benchmark platform for assessing LLMs. The results indicate that, while Leetcode's problems are suitable for performance evaluation, their measures should be used cautiously due to issues with stability and reliability. Additionally, we observed that the use of newer Leetcode problems is essential to avoid data contamination and ensure the validity of LLM evaluations.

This work opens up several avenues for future research, including the development of performance-oriented training datasets and the fine-tuning of LLMs for performance improvement. As the field of AI-assisted programming continues to evolve, studies such as ours will play a critical role in understanding and enhancing the capabilities of LLMs.

While evaluating the runtime performance of LLM-generated code is essential to understand their capabilities, it only captures one part of the picture. As these tools become more widely adopted, it also becomes important to look beyond what the generated code does, and examine what it costs to produce it–especially in terms of energy. Developers using LLM-based assistants like GitHub Copilotrely on remote inference servers to generate code suggestions, but the energy implications of these interactions remain largely unexplored. In the next chapter, we shift focus from the output of LLMs to their use, and explore the real-world energy consumption of code assistants during software development.

Chapter 5

Green My LLM: Studying the key factors affecting the energy consumption of code assistants

Chapter summary. In this second contribution, I looked at what's usually invisible to the developer: how much energy is actually used when working with a code assistant like GitHub Copilot. To do that, we ran a user study with participants who used GitHub Copilotto develop a Java project, and built a dataset of real development traces, which we then replayed on our own inference server to measure the energy consumption of a typical code assistant under different conditions. We found that a surprising amount of energy was wasted–mainly due to unwanted or canceled suggestions. Some simple changes, like switching to manual triggering of the code assistant or tweaking server configurations, could cut down this waste significantly. We also showed that things like the number of developers sharing a server, the model used, or whether quantization is used, all have a big impact on energy consumption. Overall, this study reveals just how much hardware and configuration matter, and why both users and tool providers should think more critically about the energy side of these tools.

5.1 Introduction

In recent years, LLMs for code have significantly become better at generating code, facilitating their seamless integration into developers' IDEs as code assistants, which offer auto-completion suggestions that developers can either accept or reject. The generation

process is typically initiated automatically after a brief pause in typing, but can also be manually triggered via a command or keyboard shortcut.

Numerous code assistants, like GitHub Copilot, Tabnine, and CodeWhisperer, including open-source options—like Tabby and Cody—offer IDE extensions and manage inference servers for code suggestions.

However, as explained in chapter 2, the energy consumption of software–and more specifically, that of LLMs–has gained prominence as a significant environmental and societal concern. For instance, Samsi *et al.* benchmarked the energy consumption of LLM inference and were able to estimate the energy of a single response from an LLM [91]. Other works focused more on the impact of training the model, such as the carbon footprint of the Bloom model estimated by Luccioni *et al.* [66]. However, the evaluation of LLMs' energy consumption remains challenging when assessing LLMs dedicated to specific purposes. In the context of LLMs for code, existing studies have only focused on the impact of the generated code [99, 23].

In this chapter, we want to provide an answer to the question *"How much energy is used when using code assistants like GitHub Copilot?"*. Notably, we aim to determine how much energy an average developer consumes when using a code assistant similar to GitHub Copilotand how to reduce it. To the best of our knowledge, our work is one of the first to study the energy consumption of LLMs in code assistants. In particular, we wish to deliver actionable insights into the energy consumption of the code assistant from the perspective of both the service provider (*e.g.*, GitHub Copilot) and the end-user interacting with the code assistant. All the more in the context of code assistants, like GitHub Copilot, the end-user knows little about the internal workings and impacts of the service: as the LLM inference is executed remotely in the cloud, it is difficult for the developer to perceive the computing impact of using a code assistant. Thus, we aim to answer the following research questions:

RQ1: *What is the impact of certain factors on the energy consumption and performance of code assistants?* Specifically, we study the impacts of the number of concurrent developers using the assistant, the streaming and manual triggering of the requests, the model and its quantization, the maximum number of concurrent requests, and the number of GPUs.

RQ2: *How many generation requests made by GitHub Copilotare useful?* Knowing how many generations are useful to the developer could allow future works to improve the efficiency of code assistants.

RQ3: *How much energy does a developer consume when using a code assistant similar to GitHub Copilotunder different scenarios and objectives?* We aim to get a broad look at the potential impacts of a code assistant by leveraging the knowledge about the various factors we gathered when answering RQ1.

We chose to study GitHub Copilotspecifically for three reasons: *(i)* it is the most used code assistant according to the 2023 StackOverflow developer survey,[1] *(ii)* the inference server provided by GitHub Copilotexhibits low latency and correct quality [29], which we may not be able to reproduce with our own inference server, and *(iii)* GitHub Copilotprovides many mechanisms making it one of the state-of-the-art code assistant, such as preventing some generations requests that may not be useful, caching the results of the previous generations requests, canceling the previous request when a new one is sent, and building of prompts that take into accounts multiple files. We are aware of the existence of these mechanisms thanks to the Copilot Explorer project [2], while evidence of similar mechanisms in other code assistants is unclear, and verification is time-consuming.

In this chapter, we share the following contributions:

- We provide the first dataset of development traces from developers using GitHub Copilot, which allows the simulation of developers using a code assistant on a real inference server.
- We study the impact of some configuration options of the inference server and the code assistant on its energy consumption and performance.
- We analyze the ratio of useful generation requests from GitHub Copilot.
- We estimate how much energy a developer would consume when developing under different configurations of the inference server.

Main findings & implications. Our results reveal that a considerable portion of energy is wasted due to suggestions being cancelled or not wanted by the users. Thus, manually triggering the generation of the code assistant can significantly reduce its energy consumption, by minimizing unnecessary requests. Additionally, the number of concurrent developers on a single inference server greatly affects the energy consumption of a single developer, as having more concurrent developers better uses the resources of the server. Providers should aim to maximize the number of developers per server or share inference servers in order to maximize efficiency. Lastly, the configuration of the

[1] https://survey.stackoverflow.co/2023/#section-most-popular-technologies-ai-developer-tools

inference server of the code assistant plays a significant role in its energy consumption. For instance, smaller and models tends to use less electricity, reducing the number of GPUs decreases the energy consumption at the cost of a higher latency, and some quantization techniques can positively both the energy consumption and latency of the code assistant.

Outline. From section 5.2 to section 5.4, we describe how we collected our dataset, performed our experiments, and explain the methodology we followed to analyze the results obtained. We report in section 5.5 on the results of our evaluation and provide a critical discussion in section 5.6. In section 5.7, we discuss the limitations of our study. Finally, section 5.8 concludes the chapter.

5.2 Dataset

To investigate our research questions, it was imperative to measure the energy consumption of a code assistant in a realistic usage setting. To that end, we designed an experiment with participants using GitHub Copilotto gather a dataset of development traces, enabling us to simulate developers using a code assistant on an inference server under our control. This approach was necessitated by our inability to access GitHub Copilot's inference server and our objective to measure its power consumption. Moreover, conducting this experiment in two distinct phases—(i) having participants use a code assistant to develop a small application followed by (ii) exploring the energy consumption through traces replay—allowed us to gain more freedom when it came to controlling the configuration of the code assistant and facilitated the reproducibility of our experiment. Hereafter, we refer to the traces dataset we collected as AssistantTraces.

5.2.1 Involved participants

We recruited 20 volunteers among Computer Science (CS) students and CS professionals, using mailing lists and word of mouth. All of the participants were proficient in Java programming. 13 participants were between 18 and 25 years old, 6 were between 26 and 35 years old, and 1 was between 36 and 45 years old. 13 of them were students in CS (12 in a master's degree and 1 in a bachelor's degree), and 7 of them were CS professionals. 1 participant did not know GitHub Copilotbefore the experiment, 7 participants knew GitHub Copilotbut never used it, 4 already used it a little, and 8 used it regularly. All of the participants were familiar with VSCode. The participants were compensated

for their time with 50€ each. This experiment was approved by our institution's Ethical Board.

Before the experiment, the 20 participants filled out a survey with their age, development experience, and familiarity with GitHub Copilot. Following the experiment, 19 out of the 20 participants agreed to complete a subsequent survey concerning their experience and feelings during the experiment. This last survey was used to report ideas for future research.

5.2.2 Assigned task

The task given to the participants consisted of developing a CLI Connect 4 game[2] in Java in one hour using VScode and GitHub Copilot, it is derived from a project given in an university OOP introductory course. A skeleton of the project and associated unit tests were provided. The participants then had to design and write the game loop and logic and handle the board display to the players using the CLI. The instructions for the participants, the skeleton project, and the projects created by the participants are available in our replication package.

The participants used the GitHub Copilot VScode extension only through inline and panel completions; that is, they could not use other features of GitHub Copilot, such as the chat. They were also given access to the Internet while being forbidden from using other AI assistant (*e.g.*, ChatGPT).

We collected GitHub Copilot's telemetry by modifying the VScode extension and redirecting the telemetry to the computer of the participant. GitHub Copilot's telemetry includes a plethora of different messages that enable us to retrace the history of a generation, from the moment GitHub Copilotdecides to send a generation request to the moment of its acceptance from the user.

The participants all used the same laptop: A Dell Latitude 7410, with 32 GiB of memory, an Intel Core i7-10610U and a CML GT2 Mesa Intel graphics card. It was running on Debian (bookworm). The monitor was a Dell U2720Q.

5.2.3 Dataset description

The AssistantTraces dataset consists of all the telemetry sent by GitHub Copilotduring the experiment in JSON format. There are 119,774 telemetry messages in total, including 9,633 generation requests. The dataset is available at https://doi.org/10.5281/zenodo.11503612.

[2] https://en.wikipedia.org/wiki/Connect_Four

5.3 Experiment setup

To estimate the energy consumption of GitHub Copilot, we leveraged the collected traces to simulate the developers' behavior and the front end of the code assistant on an inference server. Specifically, the inference servers were run on a cluster, whose nodes consist of AMD EPYC 7513 (Zen 3), with 512 GiB of memory and 4 Nvidia A100-SXM4-40GB (40 GiB). The server's distribution and OS were Debian 6 on Linux 5.10.0-28-amd64.

All the artifacts of this study, including our results, code, and datasets, are available in the following public repository: https://doi.org/10.5281/zenodo.13167546.

5.3.1 Simulation client

Thanks to the telemetry data from the AssistantTraces dataset, we could reproduce the API requests that GitHub Copilotsent to its inference server. We developed a simulator (the *client*) to mimic developers' interactions with GitHub Copilotby replaying generation requests to the inference server. The main benefit of our approach is that we can make multiple simulations with different scenarios by varying *e.g.*, the number of developers or the behavior of the code assistant.

When performing simulations with concurrent developers, we limited the simulation to the first hour of development to account for some developers who needed more than one hour to complete the task and to keep the simulation times short and manageable. For other developers who completed the task in less than an hour, we delayed their start times so that the middle of each simulation aligned, thus creating a more realistic distribution of the server's workload. For example, a 1-hour telemetry session starts at minute 0 and reaches its midpoint after 30 minutes, while a 50-minute session begins at minute 5 and also reaches its midpoint at minute 30.

When simulating less than 20 developers, we repeated the simulation multiple times with different developers until all were included (*e.g.*, we repeated a simulation of 2 developers 10 times with varying pairs of developers every time, so that all 20 developers would be simulated). For simulations with more than 20 developers, we randomly picked replicates among the 20 developers. To avoid issues with simultaneous identical requests, we offset each duplicate by a random number of 0 to 30 seconds.

5.3.2 Inference server

To handle generation requests and generate code suggestions, we set up an inference server. Code suggestions are generated using an LLMs. In particular, we used the Text

Generation Inference (TGI) server,[3] a server for text generation inference that is easily configurable to operate with different LLMs and that supports sharding between multiple GPUs and multiple parallel requests (using continuous batching). The server was set up with its default parameters, except for the number of shards, quantization method, and the number of concurrent requests which we describe in the next section.

5.3.3 Studied factors

Using the aforementioned setup, we performed several simulations with varying elements in the setup's configuration. We chose to study specific factors because we hypothesized they would affect the energy consumption of the code assistant. The factors we studied are as follows:

Number of concurrent developers: We varied the number of developers concurrently querying the code assistant ranging from 1 to 500 with discrete values: 1, 2, 5, 10, 20, 30, 50, 75, 100, 150, 200, 300, 400, 500.

Streaming the requests: We emulated different request-sending behaviors from GitHub Copilotby activating and deactivating streaming when sending requests. By default, GitHub Copilotuses streaming, which allows it to cancel a previous request that is still generating when a new one is sent. Deactivating streaming signifies that every request that is sent by GitHub Copilothas to complete the triggered generation, even if it is no longer useful for the user.

Manually triggering the code assistant: Typically, GitHub Copilot's generation mechanism is triggered automatically. We also considered emulating a manual trigger for the generation behavior by only sending generation requests that were completed and displayed to the user based on the assumption that the developer manually triggering GitHub Copilotwould wait for a response. While this method is not perfect, it serves as an approximation of the user behavior.

Code assistant model: We tested three different LLMs from the StarCoder family, namely: StarCoder (15.5B parameters) [57], StarCoder2-7b and StarCoder2-15b [65]. We chose these models as they are popular open-source models for code generation. The range of models allows us to see the impact of the size of the model, as well as its architecture. Indeed, StarCoder is a decoder-only transformer with *Multi-Query-Attention* and positional embeddings, whereas StarCoder2 is a decoder-only transformer with *Grouped-Query-Attention* and *Rotary-Positional-Encodings*.

[3] https://github.com/huggingface/text-generation-inference

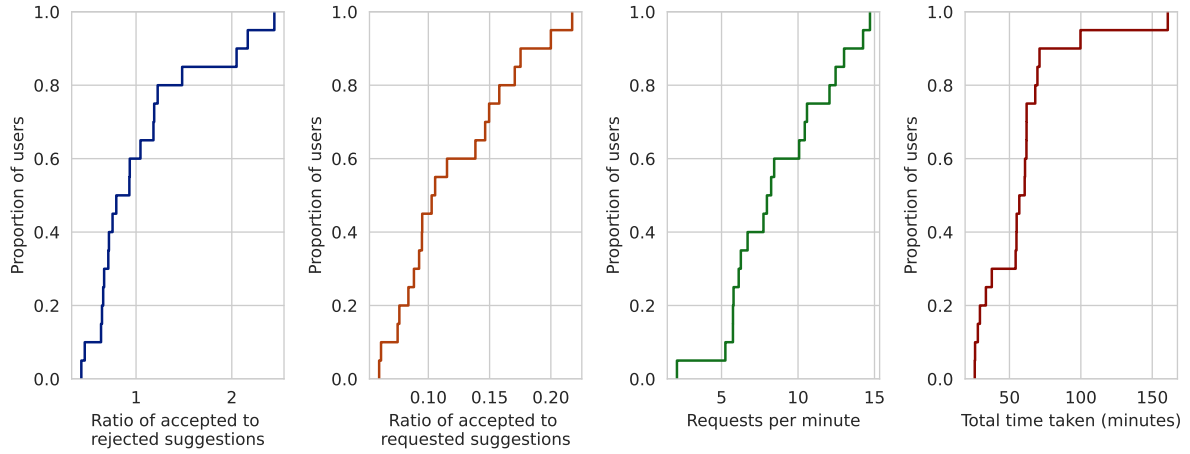

Figure 5.1: Various statistics on the participants' usage of the code assistant and their time taken to finish the experiment. The first figure represents the ratio of the number of suggestions accepted by the user to the number of suggestions by the user. The second figure represents the ratio of the number of accepted suggestions to the total number of suggestions requested by the code assistant. The third one focuses on the frequency of requests made by the participant, and the last figure represents the time taken by the participant to finish the experiment.

Quantization method: Quantization is a method used to reduce the computational and memory costs of running inferences. We studied multiple quantization methods for running the models on the inference server: EETQ [75], BitsAndBytes-NF4, BitsAndBytes-FP4 [26] and no quantization at all.

Maximum number of concurrent requests: We varied the number of concurrent requests on the TGI server which effectively modified the number of requests that could be queued simultaneously. When the server was overloaded, it returned an error to the caller.

Number of GPUs: All of our servers had 4 GPUs available which enabled us to run the simulations using a subset of the available GPUs, such as two or just one GPU. When using less than 4 GPUs, we considered the unused GPUs nonexistent and did not account for their consumption.

5.3.4 Configuration space exploration

We define a configuration as a set of factors with a specific value. One example of configuration is [20 concurrent developers; streamed requests; requests are automatically triggered; the model used is StarCoder2-7b; no quantization method; 4 GPUs].

The combination of these multiple factors and their varying modalities results in $4,896$ unique possible configurations. Considering each configurations takes between 1

and 20 hours to simulate and measure, we decided not to explore the whole configuration space. To keep the exploration manageable yet informative, some factors were fixed while exploring the impact of others (*e.g.*, fixing the model and number of GPUs while varying the number of developers or the quantization method). In total, we performed 829 simulations with 314 unique configurations. As explained in subsection 5.3.1, configurations with less than 20 concurrent developers needed to be simulated multiple times (2 for 10 developers, 4 for 5 developers, 10 for 2 developers, 20 for 1 developer), which partly explains why there are more simulations than configurations. Moreover, we simulated some other configurations up to five times to measure the stability of our setup.

5.3.5 Energy consumption measurements

We measured the energy consumption of the inference server's CPU and GPU using `perf` and `nvidia-smi` utilities, respectively. We also collected the time taken for generations to complete (latency), the number of rejected requests (due to server saturation), and the number of completed generation requests. When measuring a configuration multiple times, we found that the standard deviation of the power consumption was only 1.4% of the mean power consumption. We considered this level of standard deviation acceptable and concluded that our measuring setup was stable. To estimate the carbon emissions related to the energy consumption measured during our experiments, we considered France's 2023 carbon intensity, equivalent to 56 g of CO_2 per kWh [34].

5.4 Data analysis

In this section, we describe our methodology for analyzing the results we obtained from section 5.3.

5.4.1 Simulations analysis

To get an overview of the data, we computed the mean power consumption (in Watts) and total energy used (in Wh) for every simulation, as well as the mean latency, the number of rejected requests, and the number of completed generations. We also derived the power consumption per developer, which allocates an equal share of the server's energy consumption to every developer using the code assistant. When analyzing a simulation with multiple concurrent developers, only the period in the simulation where all developers were active in parallel was considered.

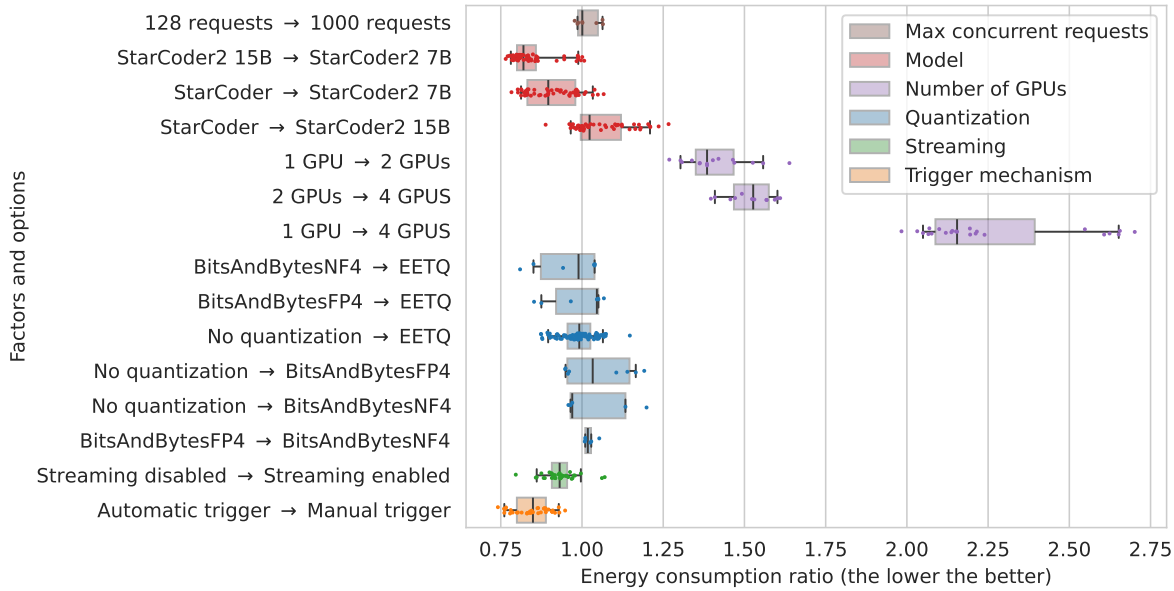

Figure 5.2: Energy impact ratio from switching from one option to another. A ratio of 1 means no change, a ratio of 2 means the energy consumption doubled, and so on. Points correspond to the ratio in energy when comparing neighboring configurations.

Server saturation. When the server receives more requests than it can handle, the number of rejected requests or the latency may increase, depending on the client and server configuration. We consider that a server is saturated when the number of rejected requests exceeds 10% or when the mean latency of the requests exceeds 20 seconds. Note that this saturation threshold was set arbitrarily; searching for an ideal saturation threshold is out of the scope of this study.

Measuring the impact of a factor. To assess the impact of a factor on energy consumption or latency, we compared pairs of configurations that differed only by the factor being studied. This method ensures that any observed differences are solely due to the factor in question.

5.4.2 Participant analysis

For each participant, we performed analyses using the gathered telemetry and survey data. Specifically, we calculated the number of generations requested, displayed, and accepted, and those remaining in the code after a certain period. To determine which generation requests remain in the participant's code, we leveraged GitHub Copilot's telemetry, which indicates whether a generation is still present in the code. This is assessed using edit distance at the character and word levels, with the generation considered removed when the word edit distance falls below 50% [2].

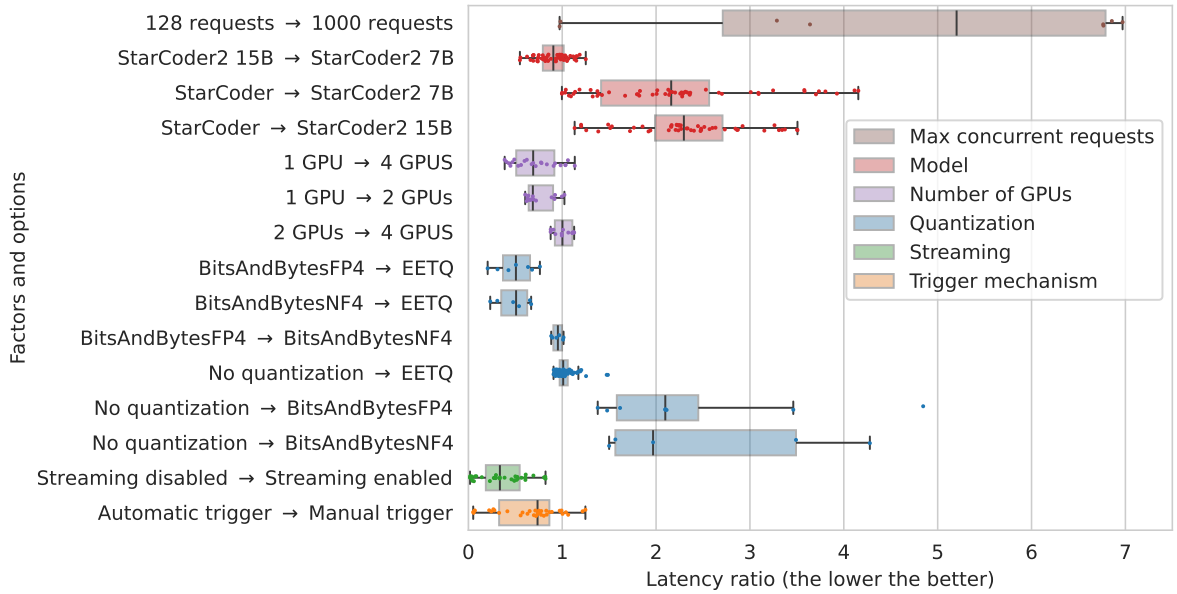

Figure 5.3: Latency impact ratio from switching from one option to another. A ratio of 1 means no change, a ratio of 2 means the latency doubled, and so on. Points correspond to the ratio in latency when comparing neighboring configurations.

In Figure 5.1, we share various statistics on the participants code assistant usage and time taken to realize the experiment. We observe variations in the way the participants use GitHub Copilot, notably in the ratio of accepted suggestions over the number of rejected suggestions or total sent generations requests. We also notice that some participants trigger generations more frequently than others. Lastly, there is also a great variation when it comes to the time the participant took to finish the experiment, indeed, 5 participants requested to have more than one hour to finish, and 6 completed the task within 40 minutes. Out of the 20 participants, 9 did not complete the task they were given and decided to end the experiment at the one hour mark. When calibrating the duration and complexity of the task given to the participants, we aimed at having the most development traces as possible. Thus, we chose to reduce the chances of the participants finishing early, thereby increasing the chances of the participant finishing late or giving up after one hour. We find that participants not completing the whole task is not an issue for our experiment, as the usage of the code assistant roughly stays the same during the whole task.

5.5 Results

In this section, we summarize the key observations from our experiment and answer our research questions. We make our complete results available in the companion notebook in the replication package.

Metric	Independent variable	Groups	Test used	Stat	P-value	Significant
Ratio of accepted suggestion over rejected suggestions	Computer experience	professional, student	t-test	-2.496	0.022	✓
	GitHub Copilot familiarity	regularly, sometimes, never used	ANOVA	1.743	0.205	✗
	Coded Java in the last year	did code in java, didn't code in java	t-test	-0.570	0.576	✗
	Finished	finished, didn't finish	t-test	0.127	0.900	✗
Requests per minute	Computer experience	professional, student	t-test	0.808	0.429	✗
	GitHub Copilot familiarity	regularly, sometimes, never used	ANOVA	1.034	0.377	✗
	Coded Java in the last year	did code in java, didn't code in java	t-test	0.243	0.811	✗
	Finished	finished, didn't finish	t-test	0.799	0.434	✗

Table 5.1: Inferential statistics of the effect of three independent variables on two metrics. The significance was asserted using a Benjamini-Hochberg correction with a False Discovery Rate of 0.20

5.5.1 Relationship between participant's characteristics and their usage of GitHub Copilot

Before answering any of the research questions, we wanted to have a look at the relationship between some of the participant's characteristics, such as their experience, familiarity with GitHub Copilot, if they coded in java during the last year, or whether or not they finished the task, and their usage of GitHub Copilot, that is, the rate at which the participants made requests, and the ratio of code suggestions they accepted when presented to them. We performed this investigation in order to find if any noticeable effect could bias our subsequent evaluations, so the metrics were chosen because they become important later on. In total, we performed 8 post-hoc tests, which are described in Table 5.1. In order to reduce the Type I error rate (false positives), we also performed a Benjamini-Hochberg procedure to determine the significance of the tests using a false-discovery rate (FDR) of 0.20.

From the data in Table 5.1, we cannot conclude that there is any effect between most of the characteristics studied and the studied metrics, except between the computer experience and the ratio of accepted suggestions over rejected suggestions. When analyzing the data, we see that professional developers have a ratio of 0.67 whereas students have a ratio of 1.26. This indicates that the professional developers in our sample are more conservative towards the suggestions made by GitHub Copilotcompared

to the students. The students tend to accept the suggestions made by Copilot twice as much as the professional developers.

5.5.2 RQ1: What is the impact of studied factors on the energy consumption and performance of code assistants ?

In this section, we report on our findings on the different studied factors and their impacts. In Figure 5.2 and Figure 5.3, we depict the impact of switching from one option to another on energy consumption and request latency, respectively. The impacts were calculated using the method described in subsection 5.4.1. Each hue represents one factor, while each row represents the switch from one option to the next. The X-axis reflects the ratio in measured latency or energy consumption between the first and second options.

Number of concurrent developers. When increasing the number of concurrent developers, we observe an increase in the average power consumption of the machine and the latency of the server up to a certain point. Thanks to the continuous batching techniques used by TGI, adding more developers only marginally increases the latency and consumption of the server, thus reducing the energy consumption per developer. This is illustrated in Figure 5.4, where we can observe that the energy consumption per developer decreases whenever a developer is added. On the other hand, with too many developers, the latency becomes excessively high, making the code assistant unusable. With the configuration shown in Figure 5.4, the average latency reaches 16 seconds with 75 concurrent developers, and increases exponentially from that point with each developer added (50 seconds at 100 developers, 210 seconds at 150 developers, *etc.*). It finally reaches a plateau past 150 concurrent developers as the server reaches the maximum number of concurrent requests limit and starts rejecting new requests.

Streaming the requests. Enabling streaming to send requests—i.e., canceling previous generation requests when a new one arrives—reduces server latency by 62%, on average, and reduces power consumption by 7%. The lesser reduction in power consumption compared to the reduction in latency is due to the inference server still spending most of its time generating responses. As a result of the lower latency, streaming allows more concurrent developers to use the assistant.

Manually triggering the code assistant. Manually triggering the code assistant by only requesting the generations that were proposed to the developer, reduces energy consumption by 15% and reduces latency by 35%. The highest reduction in energy consumption (25%) is observed with the configuration [StarCoder2; no quantization; no streaming automatic trigger; maximum 1000 concurrent requests; 2 GPUs; 5 developers],

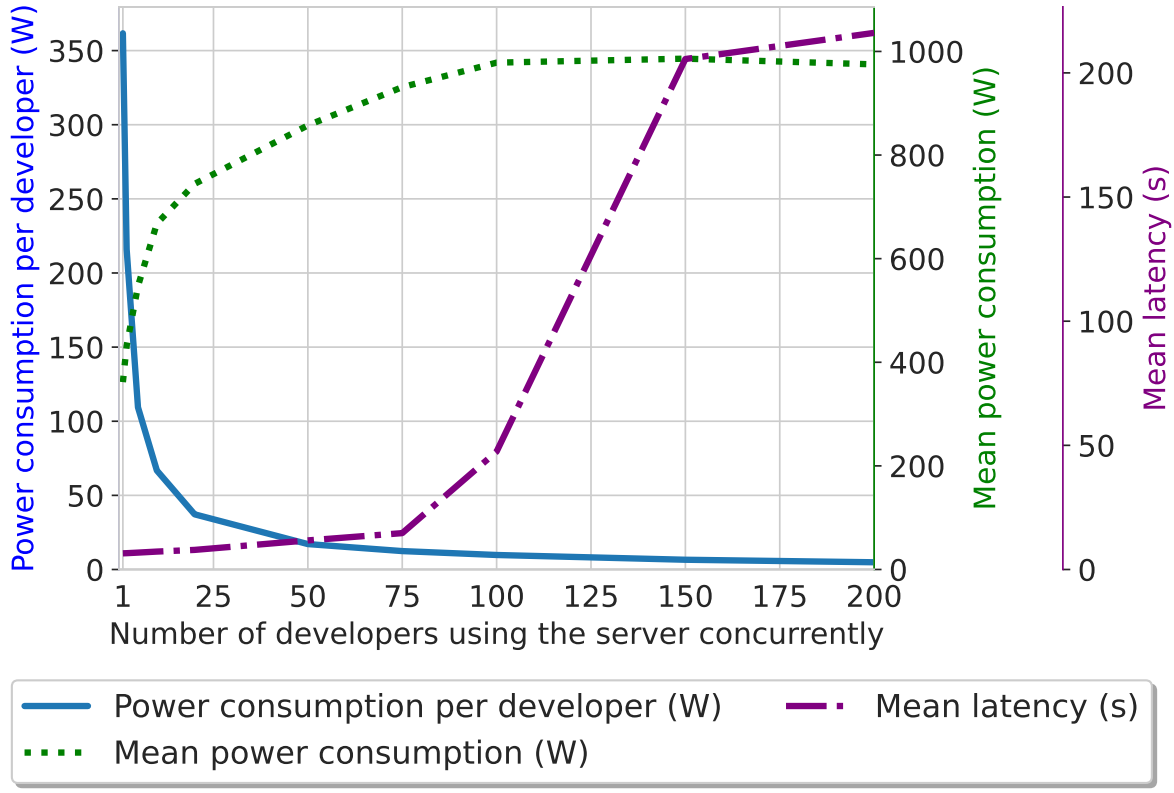

Figure 5.4: Evolution of the average power consumption and the latency depending on the number of developers, for the following configuration: StarCoder2-7B; no quantization; no streaming; manual trigger; maximum 1000 concurrent requests; 4 GPUs. The latency and power consumption are superposed in order to easily visualize how they interact when the number of developer increases.

which also reduces the latency by 25%. On the other hand, the lowest reduction of the energy consumption (5%) is observed with the configuration [StarCoder2-7b; no quantization; no streaming; automatic trigger; maximum 1000 concurrent requests; 4 GPUs; 75 developers] which, however, reduces the latency by 93%. In this specific configuration, enabling an automatic trigger made the server saturate (226 seconds of latency and 59% of rejected requests), while using a manual trigger allowed it to handle the load from the 75 developers (14.6 seconds of latency and 0% of rejected requests).

Quantization. Quantization generally increases latency. Notably, employing the method EETQ over no quantization at all increases the latency by 3.1%, and using BitsAndBytes-NF4 or BitsAndBytes-FP4 increases the latency by 138.6% and 156%, respectively. In some cases, the use of EETQ also reduces energy consumption: on average, it reduces energy consumption by 1.1%, but it can reduce it by up to 12.6% in the case of the configuration [StarCoder; streaming; automatic trigger; maximum 1000 concurrent requests; 4 GPUs; 5 developers]. BitsAndBytes, however, slightly increases the power consumption by 5% on average.

Model. Using LLMs with fewer parameters reduces energy consumption and latency. For instance, switching from StarCoder2-15B to StarCoder2-7B reduces energy consumption by 15.6% and latency by 10.0%. However, the model architecture also plays an important role in latency. For example, while StarCoder (15.5B) and StarCoder2-15B exhibit a similar number of parameters and power consumption, using the latter over the former increases the latency by 149% on average. We believe this might be due to differences in the architecture of the two models. For instance, StarCoder uses *Multi-Query-Attention*, whereas StarCoder2 adopts *Grouped-Query-Attention*.

Maximum number of concurrent requests. Increasing the maximum number of concurrent requests allowed by the TGI server does not affect the energy consumption of the server, but greatly increases latency when there are too many concurrent developers—*i.e.*, up to 597% increase with the configuration [StarCoder2-7B; EETQ; no streaming; automatic trigger; 4 GPUs; 50 developers]. Providers should prefer having a lower maximum to reject the requests early rather than making the users wait for several minutes.

Number of GPUs. As intuitively expected, reducing the number of GPUs allocated to the inference server reduces the energy consumption of the server and increases latency. For example, using 4 GPUs instead of 2 increases consumption by 51.8% and can decrease the latency in some cases, hence enabling some more concurrent developers to use the code assistant as more memory is available. Downsizing the number of GPUs can be a viable option for saving energy if the number of concurrent developers is low enough.

Usage of the code assistant. Participants' usage of GitHub Copilotvaried, with the average amount of requests per minute ranging from 1.9 to 14.7, overall averaging 9 requests per minute. We observe that the more a developer uses GitHub Copilot(*i.e.*,, the more generation requests are made), the more power consumption increases (correlation of 0.93). We infer that in its current state, the usage of a code assistant is directly correlated with the amount of code a participant writes—*i.e.*, the more time a participant spends writing code in the IDE, the more a code assistant triggers generations.

> **RQ1**: Various factors significantly impact the energy consumption and performance of code assistants. Increasing the number of concurrent developers improves energy efficiency, but risks server saturation. Streaming requests and manually triggering generations both significantly reduce energy consumption and latency. Quantization methods and the choice of LLM also affect performance, with smaller models showing better efficiency. Adjusting the number of GPUs is crucial for optimizing energy use. Our findings, therefore, confirm the importance of carefully selecting and optimizing these factors.

5.5.3 RQ2: How many generation requests made by GitHub Copilotare actually useful?

Figure 5.5 illustrates the final state of generation requests sent by GitHub Copilotduring our experiment. Out of the 9,634 generation requests made by GitHub Copilotover the 20 participants, only 2,944 finished generating and were displayed to the user (30%). Of these, 1,066 were accepted (11% of all requests), and 816 (8.5%) were kept in the code at the end of the experiment according to GitHub Copilot.

Our analysis reveals that the majority of generation requests made by GitHub Copilotwere canceled, often because they were started while the user was still typing, leading to new requests that replaced the previous ones. Empty completions typically occur at the end of a sentence, or before a closing parentheses or brackets. The completions that were displayed but not accepted are most likely due to either the users not wanting or needing the suggestion in the first place, or to the suggestions not matching what the user expected.

Previously, we discussed the energy implications of these inefficiencies when presenting the energy saving of manually triggering GitHub Copilot. By eliminating unnecessary requests, *i.e.*, canceled and empty requests, we could save 15% of energy on average.

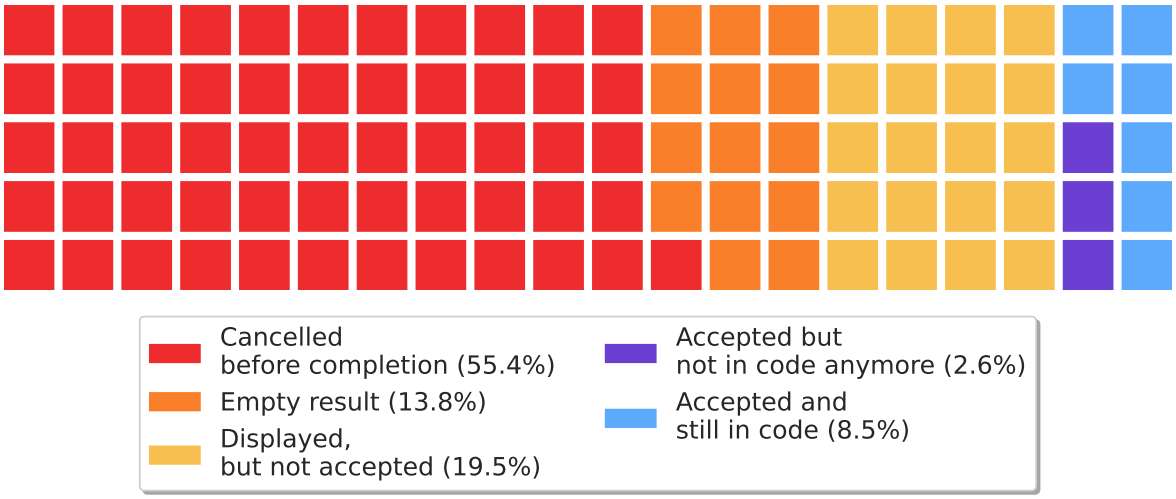

Figure 5.5: Percentage of requests sent by GitHub Copilotdepending on their completion state. Red-tinted categories represent requests that did not benefit the user. Blue-tinted categories represent requests that benefited the user.

This relatively modest energy saving, despite removing almost 70% of the generations is due to the lower-than-average computing time used by canceled and empty generations, compared to completed generations. Assessing the energy impact of suggestions displayed, but not accepted, by users is left for future research.

> **RQ2**: An analysis of GitHub Copilot's generation requests reveals a significant share of them are not useful to the end user, indicating that many generation requests are either unnecessary or not helpful to users. This inefficiency highlights the potential for substantial energy savings by optimizing when and how generation requests are made, possibly through more intelligent triggering mechanisms or better user interaction designs.

5.5.4 RQ3: Under different scenarios and objectives, how much does a developer using a code assistant such as GitHub Copilotcost in energy?

For the sake of simplicity, this section only focuses on a subset of configurations and their impacts, given the large number of different configurations available (314). However, the entire set of configurations can be found in our companion notebook. When presenting

these results, we assume that the server load is constant—*i.e.*, all developers are working simultaneously—and that the server is turned on only when the developers are working.

To select the configurations, we defined three different development scenarios, each with two objectives. The goal of these scenarios is to highlight how different use cases require different configurations and have varying impacts. The scenarios are as follows:

- **Small team:** 5 developers concurrently requesting a single dedicated server machine.
- **Medium team:** 20 developers concurrently requesting a single dedicated server machine.
- **Distributed service (*e.g.*, GitHub Copilot):** A large number of developers (sharing) requesting many server machines. The aim is to maximize the number of developers per machine while not saturating the servers. We report on the impact of a single machine in this scenario.

The two objectives a development team may aim for we considered in our study are as follows:

- **Performance:** By design, the generation requests are triggered automatically, and the previous ones are canceled (using streaming). The latency between requests is thus minimized. Generations are triggered automatically so that the developer always receives suggestions as quickly as possible. This is also the default triggering and streaming mode of GitHub Copilot.
- **Frugality:** The energy consumption per developer is minimized. The generation requests are triggered manually, and the previous ones are not canceled (not using streaming), as we expect the developers to wait for their manually triggered generation requests to finish.

Table 5.2 shows the best configurations for each scenario and objective. For each of the 6 configurations, its latency and various energy consumption metrics are presented. To better put in perspective the energy usages of the configurations, we also show them in Figure 5.6. As we can see from both the table and the figure, the energy consumption of a single developer varies significantly across the different configurations. The setup machine's high idle power consumption (~270 W for four GPUs and one CPU) results in a substantial per-developer energy impact when fewer developers utilize many GPUs. Conversely, increasing the number of concurrent developers reduces individual energy

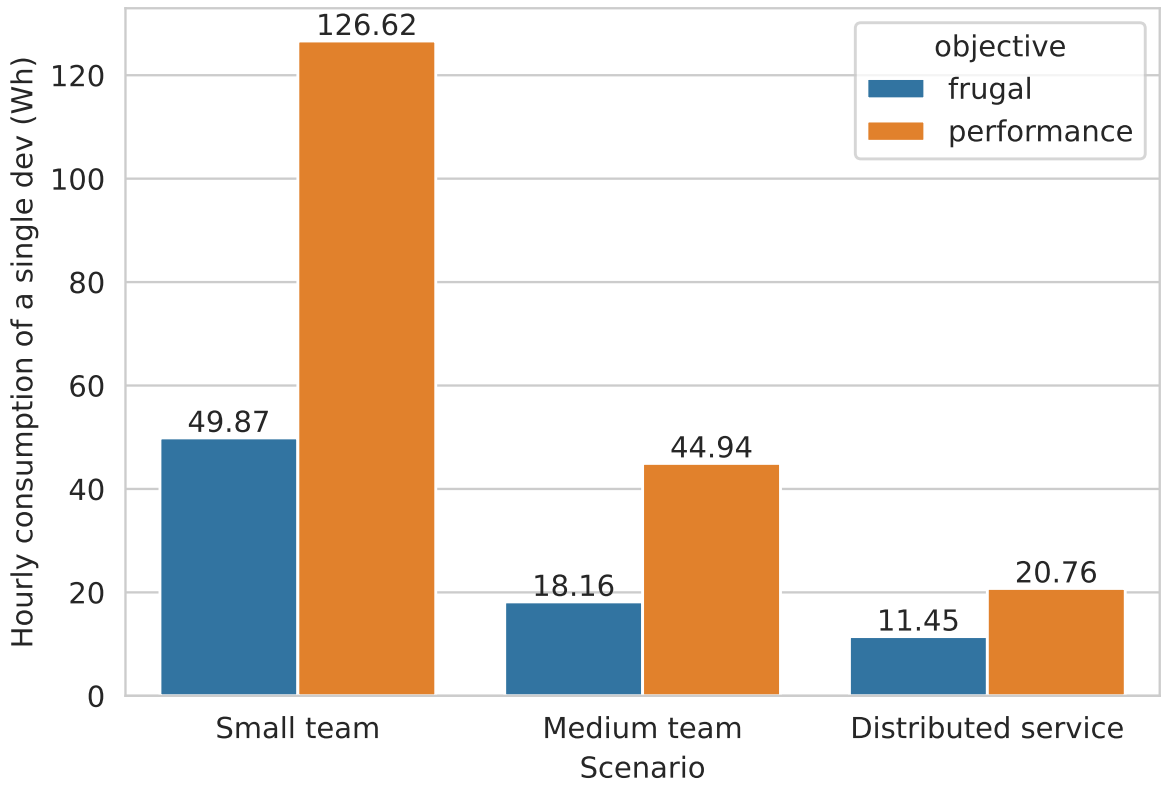

Figure 5.6: Hourly energy consumption (Wh) of a single developer based on the objective and the number of developers using the assistant.

consumption by leveraging the continuous batching from the TGI server, aligning with the findings discussed in subsection 5.5.2.

If GitHub Copilotwere using a setup similar to ours with maximized server load, the average power consumption per developer could range from 10 W to 20 W depending on their objective. However, with dedicated servers for small teams, under-utilization could result in a high power consumption per developer (~120 W). This can be mitigated by using fewer GPUs and smaller models, or considering server sharing.

To put it in perspective, we measured the energy of the laptop and the monitor of the last two participants using a power-meter. The average power consumption of the laptop and monitor when used by our participants was 19.7 W and 18 W, respectively. Thus, in GitHub Copilot's best-case scenario using our setup, the power consumption per developer would be about 58% the consumption of a laptop's consumption, and at worst it would be roughly equal to that of a laptop.

We observe that all performance configurations uses StarCoder due to its low latency. However, among the three studied models, StarCoder is the worst at generating code:

StarCoder has a 33.6 pass@1 on HumanEval while StarCoder2-7B and 15B have 35.4 and 46.4 pass@1 [57, 65], respectively.

Scenario	**Small team** (*dedicated server*)		**Medium team** (*dedicated server*)		**Distributed service**	
Objective	frugal	performance	frugal	performance	frugal	performance
Number of concurrent developers	5	5	20	20	75	50
Model	StarCoder2-7B	StarCoder	StarCoder2-7B	StarCoder	StarCoder2-7B	StarCoder
Quantization method	-	-	-	-	EETQ	-
Number of GPUs	1	4	1	4	4	4
Streaming	✗	✓	✗	✓	✗	✓
Manual trigger emulation	✓	✗	✓	✗	✓	✗
Average latency (s)	6.4	1.2	7.7	1.7	16.5	2.3
Average server power (W)	249.3	633.1	363.2	898.8	858.9	1038.2
Energy per 1000 generation requests (Wh)	12.0	30.7	0.9	2.2	0.1	0.4
Energy per hour per developer (Wh)	49.9	126.6	18.2	44.9	11.4	20.8
CO2 emissions per hour per developer (g)	2.8	7.1	0.9	1.0	0.6	1.2

Table 5.2: Optimal configurations for each scenario and objective, and their energy and performance impacts.

> **RQ3**: The energy cost for a developer using a code assistant like GitHub Copilotvaries significantly based on its configuration. Small teams with dedicated servers can see high per-developer energy consumption (~120 W), while large-scale services can achieve much lower consumption (10–20 W) by maximizing server load. Overall, our results emphasize the importance of choosing appropriate configurations based on team size and objectives to optimize both energy consumption and performance when using code assistants.

5.6 Discussion

In this section, we discuss the results reported in the previous section and their implications.

On energy usage. As reported in our results, the configuration choice for a code assistant significantly impacts per-developer power consumption. However, a considerable portion of the energy spent on generations is wasted due to cancellations and disinterest from users. Substantial energy savings could be achieved by requiring users to manually trigger GitHub Copilotand encouraging them to rethink their interactions with the tool, or by enhancing the request-triggering mechanism. For instance, Mozannar *et*

al. [74] proposed a novel method for triggering generation requests by leveraging human feedback collected from GitHub Copilot's telemetry. Furthermore, Barke *et al.* [13] showed that developers exhibited two kinds of behaviors when interacting with GitHub Copilot—*acceleration mode* and *exploration mode*—which could also be leveraged to adapt the triggering mechanism of GitHub Copilot.

One of our objectives with this contribution is to raise awareness among practitioners, tool providers and researchers about the environmental impact of using code assistants. Developers using code assistants can significantly reduce their energy usage by manually triggering the generations and using the assistant in a more conscious manner to avoid unnecessary generations. On the other hand, the providers of the assistants should maximize the number of users per inference server and carefully select the quantization method so as to decrease the resource usage. The choice of the model is also paramount, as it dictates the amount of memory left for batching requests, and the computational cost of a single request.

We encourage researchers to pursue our work by finding more ways to enable users and providers to reduce the energy consumption of their code assistants. One avenue of work is to evaluate the environmental footprint of the assistant using life-cycle analysis, in order to incorporate the hardware and training costs. Lastly, while code assistants and more specifically GitHub Copilotcan improve developer productivity [6], the rebound effects [28] stemming from faster development needs to be assessed. We hope that our dataset, AssistantTraces, will serve as a valuable resource for researchers and tool providers. We encourage researchers to build upon AssistantTraces, enhance its collection methodology, and refine it through their own user studies.

On the number of developers. Our results indicate that using a inference server for a small number of developers can be inefficient in terms of hardware and energy. Optimizing the number of developers per machine improves utilization, but a balance is needed between developer load and code assistant performance. Having few developers complicates hardware and model selection, as smaller GPUs may struggle with large models, while smaller models may generate lower-quality code. In this case, practitioners should consider sharing hardware and its costs with others, so that they may all benefit from higher-end hardware and a lower individual impact. We also encourage researchers to develop and improve sharing models for LLM inference, such as Petals [15].

On the difference in latency between StarCoder models. We observed a sharp increase in latency between the StarCoder and StarCoder2-15B models. While we believe this is due to the architecture changes introduced by the StarCoder2 model [65], more research is needed to get a better insight into the reasons for such an increase. As

latency plays a key role in the maximum number of concurrent developers an inference server can handle, finding ways to reduce said latency could help further improve the efficiency of code assistants.

On quantization. In our results, we found that quantization effectively increased the latency and had close to no effect on the energy consumption. We find this result surprising considering quantization reduces the LLM's size. It is possible that these specific quantization method increase the computational cost of inference. We advise researchers to investigate the reasons for this effect, and explore the energy savings of other quantization methods.

On the generalization of our results. Our study uses one specific setup and varies its configuration. While the energy consumption we observed in our different scenarios give a good idea of the scale of the energy required to run a code assistant, the observed energy consumption is specific to the context of our experiment, and should not be generalized to other code assistants, especially GitHub Copilot. However, even though other code assistants might use a different hardware, model or serving framework, they can still benefit from our results. Another setup, completely different from ours, with users making a different amount of requests per minute (which is highly correlated with the energy consumption of the assistant) would still save energy by maximizing the number of concurrent developers per machine, by using smaller models, by having the users manually trigger the generations and by cancelling them early if they're not needed anymore, *etc.*

On the environmental footprint of the code assistants We only calculated the CO2 emissions that were due to the energy consumption of the inference server. Future studies could also include the other sources of impact such as the hardware the server is hosted on, and the datacenter its in. This could also be an opportunity to present other metrics such as resource depletion or water use, which can be as important as CO2 emissions when talking about ICT. [95]

5.7 Limitations & Threats to Validity

In this section, we address several limitations of our study:

Participants and task selection. Selecting participants by word of mouth, and mailing lists from university students and professionals allowed us to easily recruit enough qualified participants for our study. Moreover, the task assigned to the participants was designed to be short and simple for an experimental setting, yet long and complex enough to simulate a realistic development session. One alternative was to perform our

dataset collection on developers working for their companies or school projects. While this alternative could have provided us with more realistic development traces, it would have been logistically harder to perform. Another alternative was to assign multiple short tasks, following Vaithlingam *et al.* [98], which could simplify generalization, while compromising the realism of the full development process, including including design, debugging, and testing.

Having a non-representative sample of the population of developers and a not complex-enough task may affect some of our results such as the number of accepted/rejected generation requests, as well as the number of requests per minute made by the participants. As the rate of generation requests is highly correlated (0.93) with the energy consumption of code assistant, the raw energy consumption numbers would also be affected. If our sample was indeed biased, we believe it would not affect the results from RQ1 and RQ3, as our experiment focuses mainly on the impact of changing configuration options of the code assistant. Another setup, completely different from ours, with users making a different amount of requests per minute (which is highly correlated with the energy consumption of the assistant) will still save energy by maximizing the number of concurrent developers per machine, by using smaller models, by having the users manually trigger the generations and cancel them early if they're not needed anymore, *etc.*

Configuration space and exploration. To keep the number of simulations low, we had to limit both the configuration space's size and its exploration. Indeed, we restricted our evaluation to three models from the StarCoder family and ran our simulations without varying the hardware. Additionally, we only considered a subset of TGI parameters. As a result, some configurations yielding different or optimal results might not be explored. We mitigated this effect by curating the configurations to explore so that they cover a large spectrum.

Emulation of a manual trigger of a code assistant. Our method of emulating the manual triggering of GitHub Copilotby only sending generation requests displayed to the user is only an approximation that allowed us to save time and resources by simplifying our dataset collection process. In reality, user behavior may differ, and this emulation might not fully capture the nuances of manual interactions. A dedicated study involving real users manually triggering generations is needed to obtain more accurate insights.

Simulation of the developers. Simulating the developers instead of performing live measures allowed us to have more freedom and explore more configurations. In return, our simulations do not allow for any kind of change in behavior from the simulated developer—*i.e.*, the same request will be sent at the same times no matter the latency or

quality of the responses. An alternative was to measure the energy as the participants were developing and modify GitHub Copilotto use our inference server. While this would yield more accurate reactions from the developers, it would greatly limit the amount of configurations we could explore. A compromise was to develop autonomous agents simulating the developers, which we deemed to difficult to realize.

Quality of the generated code. When performing the simulations, we decided not to assess or take into consideration the quality or correctness of the code generated by the LLMs. It was possible to estimate the quality of the code by comparing it to the code written to the developer using a *BiLingual Evaluation Understudy* (BLEU) score [78]. However, as shown by Chen *et al.* [20], any score relating to the proximity between two texts is unreliable at best when evaluating LLMs for code. Another alternative was to evaluate the generations by hand, which would have been too much time intensive. A third alternative was to consider the reported pass@k from the model's creators, which could have been easy to implement, but would have made our search of optimal configuration harder in subsection 5.5.4. The rationale for our choice can be explained by the lack of methods to evaluate the quality of the generated code in the context of our experiment and the simplicity of not considering the code quality when searching for the best models under specific scenarios and objectives. Nevertheless, the trade-off between energy consumption, performance, and code quality is a consideration that should be addressed in future research.

5.8 Conclusion

In this chapter, we explored the question *"How much energy is used when using code assistants like GitHub Copilot?"* by studying the energy consumption of code assistants powered by LLMs, with a user study. Our findings indicate that various configuration factors, such as the number of concurrent developers, model size, and use of streaming, impact both the energy consumption and performance of these tools. Notably, a substantial amount of energy is spent on generations that are ultimately unused, highlighting an area for potential improvement.

Our results indicate that significant energy savings can be achieved by service providers and end-users if they follow the following steps: disable the automatic triggering of code assistant suggestions on the client side, utilize batching techniques to maximize inference server usage and support more concurrent developers, and reduce model resource consumption through effective quantization and the use of more efficient models.

Overall, while LLM-based code assistants offer productivity benefits, it is crucial to consider their environmental impact. By adopting more efficient configurations and usage practices, we can make these tools more sustainable without compromising their utility.

While our investigation into code assistants revealed ways to reduce their energy consumption during typical usage, this raises a broader question: can these models also be used to actively improve the energy efficiency of the software they help create? In other words, beyond measuring the cost of using LLMs, can we also measure their potential to reduce future energy use, especially when they're tasked with optimizing existing code? This is the focus of the next chapter, where we shift our attention from the energy cost of inference to the energy savings that might be achieved through LLM-powered code optimization, and ask whether these tools can truly deliver a net environmental benefit.

Chapter 6

When faster isn't greener: the hidden costs of LLM-based code optimization

Summary of contribution. In this final contribution, I examined an often-overlooked aspect of LLM-based code optimization: its actual energy cost. While it's tempting to assume that faster code inherently leads to lower energy consumption, the process of generating these optimizations using large models is itself highly resource-intensive. To investigate whether such optimizations are truly worthwhile, we benchmarked eight optimization strategies on 118 tasks from EvalPerf and introduced the concept of a *Break-Even Point* (BEP)–the number of program executions needed for the energy saved by the optimized code to compensate for the energy consumed during its generation.

Our findings reveal a more complex picture than the common assumption suggests. Although many optimization strategies did yield significant energy savings, those gains often came at a steep initial cost. In several cases, the optimized code had to be run tens or even hundreds of thousands of times before breaking even. However, smaller models–like Qwen2.5-Coder-7B–were often able to generate efficient optimizations with much lower overhead, making them more suitable for scripts or code that isn't executed frequently. These results highlight that while LLM-based optimization holds promise, its real-world utility is highly context-dependent. Not every piece of code needs to (or should) be optimized using an LLM, especially if the energy required to produce the optimization outweighs the eventual benefit. In that sense, this study calls for more restraint and more critical assessment when deciding when and how to use LLMs for code optimization.

6.1 Introduction

As presented in chapter 2, among the numerous interesting applications of LLMs in software engineering, one particularly promising application is *code optimization*: automatically rewriting programs' code to improve performance. Recent advances in code-oriented LLMs have made it increasingly feasible to generate optimized versions of existing code, potentially reducing execution time, energy consumption, and resource usage without requiring deep expertise in performance engineering [112, 44, 81]. Most of these methods rely on iterative refinement [71] which enables them to adapt and improve based on the outcomes of each optimization step.

However, the growing popularity of these models coincides with increasing concerns about the environmental sustainability of computing, as also presented in chapter 2.

In this context, it is tempting to view LLM-based optimization as a straightforward win for sustainability, faster code often resulting in greener code. Unfortunately, the reality is more nuanced. While GPUs have become increasingly efficient, their overall energy consumption and environmental impact keep exponentially increasing [73]. LLMs, which depend heavily on GPU-based computation, are known to be very resource-intensive [91, 68, 14], requiring powerful inference servers for deployment and operation [22]. Thus, any potential savings achieved by optimizing code must be weighed against the substantial energy cost of using LLMs for the optimization itself. This concern raises a critical question for the community: *What is the real energy cost of using LLM-powered code optimizers? Is the performance and energy gain of optimized code sufficient to justify the energy cost of optimization?*

In this chapter, we address this question by assessing the energy cost of LLM-based code optimizations and determining when it is energetically beneficial to optimize a given program. Additionally, our evaluation setup allows us to also compare the effectiveness of different LLM-based optimization methods in terms of both performance and correctness. To the best of our knowledge, this is the first study to examine not only the benefits of LLM-based code optimization but also its environmental trade-offs.

Specifically, we investigate the following research questions:

RQ1: *What is the best method to optimize code?*

RQ2: *What is the energy necessary to produce one optimized result?*

RQ3: *In terms of energy, at what point is it profitable to optimize a given code?*

To answer these questions, we evaluated 8 code optimization methods using the EvalPerf [64] benchmark. For each experiment, we measured both the energy consumption

caused by the optimization process and the energy difference between unoptimized and optimized code. To quantify the energy trade-offs, we introduce a new metric that measures the number of optimized code executions required to compensate for the energy cost of the optimization.

Main findings & implications. Our results show that, while most LLM-based optimization strategies are capable of reducing the energy consumption of code, the energy savings they deliver do not always offset their own energy costs. In some cases, the optimized code must be executed tens of thousands of times before the energy spent on optimization is compensated by the energy saved. For example, a large model like DeepSeek-R1-14B combined with a powerful optimization method can reduce energy usage by up to 49% on average, but only becomes energetically beneficial after approximately 200,000 executions. In contrast, smaller models like Qwen2.5-Coder-7B can achieve worthwhile savings with far fewer executions, making them more suitable for lightly used code samples or one-off scripts. Notably, in some high-consumption scenarios, we observed energy reductions from 318 J to just 2.6 J, which offset the optimization cost in under 100 executions. These findings demonstrate that while LLM-based optimization can yield substantial energy savings, its actual profitability is highly context-dependent and must be carefully evaluated based on how frequently the optimized code is expected to run.

The remainder of this chapter is structured as follows : In section 6.2, we explain the methodology of our experiment. We report our results in section 6.3 and discuss them in section 6.4. Finally, we discuss our limits in section 6.5 and conclude in section 6.6.

6.2 Methodology

In this section, we explain how we set up and led our experiment. Namely, we list the various optimization methods we chose and how we implemented them, as well as the LLMs chosen to be used alongside the optimization methods. We also present the metrics considered to analyze our results.

6.2.1 Dataset

We evaluated the optimization methods on the 118 tasks from the EvalPerf dataset [64], which is based on HumanEval+ and MBPP+. This dataset is well-suited for our evaluation purposes as it contains both performance-exercising programming tasks and tests, as well as reference solutions with varied performances. EvalPerf's tasks were

selected because they are computationally non-trivial, which enable a higher runtime variation in the possible solutions of the tasks. This higher variation, coupled with the performance exercising tests make it easier to differentiate good and bad optimizations.

6.2.2 Methods and their implementation

We selected 8 methods to optimize code, which can be classified in two distinct categories: i) single-code optimizers and ii) batching optimizers. Single-code optimizers take one sample as input and output one code sample, while batching optimizers may take multiple inputs and output multiple samples. The key consideration about batching optimizers is that they produce batches of code samples, in which only the sample with the best performance, in terms of number of CPU instructions, is selected as a result for analysis.

The single-code optimization methods considered in this study are:

- **Simple prompting.** This method consists of simply asking the LLM to optimize the program.
- ***Chain-of-Thought* prompting (CoT).** This method [106] is similar to simple prompting, but asks the LLM to think step by step before generating its answer.
- ***Chain-of-Draft* prompting (CoD).** This method [110], while being inspired by CoT prompting, aims at reducing the number of tokens produced by the LLM by prompting it with the instruction "*Keep a minimum draft for each thinking step, with 5 words at most*".
- **Self-refinement with Natural Language Feedback.** This method and the associated prompts come from the works of Madaan *et al.* [71] and Waghjale *et al.* [103], respectively. For every code sample, the LLM is prompted to give feedback on the solution, then prompted a second time to optimize/fix the code sample.
- **Self-refinement with Execution Feedback.** This method and the associated prompts come from Waghjale *et al.* [103]. Instead of LLM-generated natural language feedback, the LLM is enhanced with execution feedback, which includes either runtime information or test execution results, depending on whether the sample passes the unit tests.

The batching optimization methods considered in this study are:

- **LLM4EFFI.** This method [112] revolves around the idea of first producing a correct and efficient algorithm, and then producing the corresponding code. In this

method, iteration 0, which is normally used to generate the baseline, is tasked with generating efficient code, and the subsequent iterations, which are normally used to optimize the samples, are only focused on improving the correctness of non-correct samples. Using this method, 10 algorithms and their associated code samples are generated to produce a single result, which is the fastest of the code samples. As we use our own test cases, we did not implement the test case generation feature of LLM4EFFI. LLM4EFFI generates batches containing 10 samples.

- ***Evolution of Heuristics* (EoH).** This method from Liu *et al.* [62] uses an evolutionary algorithm to make LLMs produce better code. While this method was used on optimization problems, such as the traveling salesman problem, we thought it would be interesting to apply it to code optimization. Instead of evolving the code samples directly, this method uses the LLM to evolve ideas (heuristics) through various mutations. Each idea is also associated with a code sample generated with the idea. We set the population size parameter to 10 (meaning, after every generation of 50 heuristics, only the 10 best heuristics are selected), and the parents parameter (for exploratory mutations) to 3. EoH generates batches of 10 or 50 samples, depending on the amount of correct samples in the input. If at least one correct samples is found, then the batch size is 50, as 10 samples are sampled for each of the five mutation, otherwise, the batch size is 10, as 10 samples are selected and a fix is attempted on them.

- **Simple10 (custom method).** As LLM4EFFI and EoH both generate batches of optimizations and take the best one out of the batch, they are set apart from the other methods which generate single code optimizations ("batches" of one optimization). To study whether the differences between LLM4EFFI and EoH and the single-code optimizers are due to their higher batch size, or due to other aspects of the methods, we implemented Simple10, a variant of the simple method which generates batches of 10 optimizations using the prompt from the simple method, and considers the best sample of the batch as the result.

6.2.3 LLMs under study

During our study, we tested the optimization methods using 5 different LLMs. The models we used were Starcoder2-15B (instruct), Qwen2.5-Coder-{3,7,14}B (instruct) and DeepSeek-R1-Distill-Qwen-14B. We selected those models specifically because we wanted to see the difference between models of different families and sizes, and between instruction-tuned models and reasoning models.

6.2.4 Experimental setup

Our experiment aims to optimize code samples using LLMs in multiple iterations while assessing the energy consumption of the optimization process as well as the energy consumption of the programs before and after optimization. Thus, we split our experiment into two distinct phases: a) the optimization phase and b) the program energy consumption evaluation phase. Separating the experiment into these two phases helped us manage our server budget, as evaluating the energy consumption of the programs is not required by the optimization process and does not need GPUs. Our experiment was conducted on the 118 tasks from the EvalPerf dataset with 8 optimization methods, using 5 models. For a given configuration (*i.e.*, method + model combination) and for the single-code optimization methods, 40 independent samples were generated by task. Simple10 generated 40 independent batches by task, EoH generated 3 independent batches by task and LLM4EFFI generated 5 independent batches by task.

Optimization phase. The optimization phase consists of one base iteration (iteration 0) and four optimization iterations (iterations 1–4). Each iteration is split into three steps: i) code generation ii) correctness evaluation and iii) performance evaluation. In the base iteration (iteration 0), during the code generation step and depending on the optimization method, we either generate a solution to a problem, or use the optimization method to generate a solution if the method allows it (LLM4EFFI and EoH are the only two methods concerned). The code samples that were generated in iteration 0 with just the problem's base prompt are then used as baseline for comparison with the optimized programs. In subsequent iterations, in the code generation step, we generate optimizations of samples from the previous iteration using the optimization methods' techniques.

In the correctness evaluation step and performance evaluation step, we evaluate each code sample that was produced during this iteration using the correctness and performance tests from EvalPerf. Only samples that passed all the correctness tests are evaluated on performance. During each step of the optimization process, we measured the energy consumed by the machine. While the CPU was always taken into account, the GPU was only recorded during the code generation phase, as it is the only one using the GPU. Additionally, at the start of each iteration, we performed a calibration for 30 seconds where we recorded the energy consumption of the machine at rest.

Program energy consumption evaluation phase. During this phase, we measured the marginal CPU energy consumption of every valid sample that was generated in the optimization phase. The samples were executed with the performance-exercising tests from EvalPerf. Before evaluating each configuration, a calibration phase of 30 seconds

was realized, in order to get the idle consumption of the machine. This idle consumption as well as the energy measured during the evaluation were used to calculate the marginal energy consumption of the samples [50]. When measuring the energy consumption of the samples, our idle measures had a coefficient of variation of 0.014, which indicates a great stability of the measures across the runs.

To increase the accuracy of our results, we repeated each sample's execution until at least one second was elapsed. As we had to evaluate 1,147,198 samples, this minimum duration seemed like a good trade-off between measure accuracy and feasibility of the experiment. When analyzing the data, to aggregate the energy consumption at the task level and then at the configuration level, we either averaged the energy consumption of all generated samples for single-code optimizers or averaged the energy consumption of the best samples from each batch for batching optimizers. Many parts of our experiment were built upon the EvalPerf and DPE experiment framework from Liu *et al.* [64]. Namely, we adapted their generation and evaluation framework to fit our needs.

6.2.5 Hardware and software setup

The optimization-related phases of our experiment ran on devices with 1 AMD EPYC 7513 (Zen 3), as well as 4 Nvidia A100-SMX4-40GB. The energy profiling of the individual samples was executed on devices with 2 AMD EPYC 7301 (Zen). We used VLLM [54] to run our LLMs. It was executed using Docker, with tensor parallelism set to 4 (the match the number of GPUs). The generations made by the LLMs were made using temperature of 0.2, which is the temperature that was chosen by most of the previous studies on LLM-based optimization. To maximize GPU usage and get more realistic energy data, generation requests were sent concurrently, and then batched internally by VLLM.

In total, this experiment consumed 651.24 KWh which amounts to 36.47 kg of CO2eq using France's energy mix's carbon intensity, which is where the servers hosting the experiment were located. The replication package of our experiment, with all of the data from our experiment as well as a companion notebook with additional figures and tables is available at https://doi.org/10.5281/zenodo.15526179.

6.2.6 Evaluation metrics

To evaluate the quality of the optimization methods and models, we used multiple metrics to quantify the performance of the samples generated, their correctness as well as their energy cost.

DPS_{norm}. The *Differential Performance Score* (DPS) is a metric developed by Liu *et al.* [64]. DPS uses clustering to more accurately assess the runtime performance of a given program on a lot of different tasks, compared to traditional metrics such as speedup. This metric is based on CPU performance counters, which is more reliable than CPU time. Given a set of reference solutions to a problem and a sample solution, the DPS is the cumulative ratio of the reference that is immediately slower than the sample solution. For example, a DPS of 80% implies that overall the LLM generates code whose efficiency can match or improve 80% of all LLM-generated solutions. We use a variant of DPS, named DPS_{norm}, a normalized version of DPS that ignores the volume of solutions at each level of efficiency. This metric is more suitable for our study as it is better at highlighting differences in efficiency levels among a large variety of tasks. We also define the $\text{DPS}_{\text{norm}}\Delta$ as the difference in the DPS_{norm} of the baseline and the DPS_{norm} obtained in the final optimization iteration.

Pass@k. We evaluate the correctness of the code produced by a given method using the Pass@1 developed by Chen *et al.* [20]. This metric represents the proportion of generated samples that are functionally correct, that is they pass all the unit tests of the problem. For optimization methods that produce batches of samples (*e.g.*, EoH, LLM4EFFI, Simple10), we compute Pass@1 over all individual samples in the batch, regardless of whether they were ultimately selected as the final result. However, since this does not necessarily reflect the correctness of the selected output, we additionally introduce a metric called *Pass@result*. This metric evaluates the correctness of the chosen result per optimization attempt: for single-code optimizers, Pass@result is equivalent to Pass@1, as only one sample is generated per attempt; for batch-based optimizers, Pass@result measures the proportion of batches whose selected result passes all tests. In this sense, Pass@result acts as a dynamic variant of Pass@k, where k corresponds to the batch size used by the method.

Efficient@k. We adapt the Efficient@k metric from Peng *et al.* [82]. While pass@k is the probability of generating a correct solution, Efficient@k is the probability that at least one of the top-k correct code samples for a problem is more efficient than our baseline. For example, an Efficient@1 of 0.7 means that 70% of the generations that are correct are also better than the baseline. Our version of the metric is slightly different from the original one, because incorrect samples are excluded from its calculation whereas, in the original, incorrect samples would count as "not more efficient" samples and lower the score. We chose to alter the metric in order to really focus on the probability of a

successful optimization to actually be faster than the baseline.

Energy reduction. While the Efficient@k gives us a good idea of the probability of a successful optimization, the Energy reduction is designed to give insights on how much more efficient the optimized code is compared to the baseline. The Energy reduction is simply defined as the ratio of the energy consumed by the optimized sample over the energy consumed by the baseline. For instance, an Energy reduction score of 0.60 means that the optimized sample's energy consumption is 40% lower than that of the baseline.

Break-Even Point (BEP). Let e_{orig} be the energy consumed per execution of the original (non-optimized) program, e_{opt} be the energy consumed per execution of the optimized program, and E_{optim} be the one-time energy cost of the optimization process. The BEP is defined as:

$$\text{BEP} = \frac{E_{\text{optim}}}{e_{\text{orig}} - e_{\text{opt}}}$$

This metric captures the number of executions required for the cumulative energy savings of the optimized program to outweigh the energy cost of performing the optimization. In the case where the "optimized" version of the program consumes more energy than its original version, then the BEP is considered as undefined and is not taken into account in further calculations.

To obtain more representative averages for BEP and Energy reduction, we removed the lowest and highest 5% of values across tasks before computing the mean. This approach is standard in robust statistical analysis [107] and was chosen due to the heavy-tailed nature of these metrics, which are especially sensitive to rare but extreme outliers. We confirmed that this trimming did not alter the overall trends observed across configurations.

6.3 Results

In this section, we share our results addressing the three research questions targeting (i) the optimization capacity of the tested methods (subsection 6.3.1), (ii) the energy consumption of the optimization process (subsection 6.3.2), and (iii) the cost-effectiveness of code optimization (subsection 6.3.3).

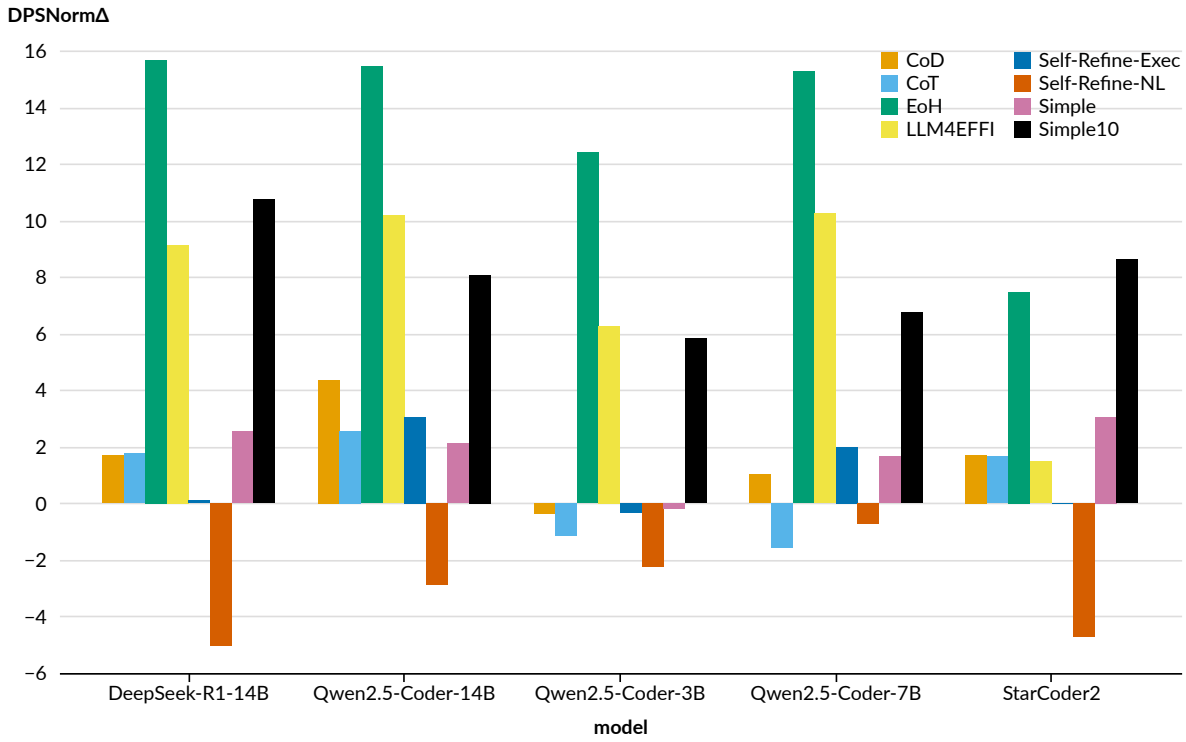

Figure 6.1: $\text{DPS}_{\text{norm}}\Delta$ of each method and model combination, at the last iteration. Negative results show decreases in DPS_{norm} compared to the baseline. A higher $\text{DPS}_{\text{norm}}\Delta$ is better.

6.3.1 RQ1: What is the best method to optimize code?

The optimization methods we evaluated exhibited varying performance in terms of both their capacity to optimize code and their ability to maintain sample correctness.

Optimization capacity. To visualize the optimization capacity of each method, Figure 6.1 reports on the performance improvements in DPS_{norm} across all method-model combinations. The batching-based methods (EoH, LLM4EFFI and Simple10) exhibit the best optimization capacity and show the highest average gains over the baseline—*i.e.*, 13.24, 7.46 and 7.99 DPS_{norm} points by the final iteration, respectively. In contrast, other methods show more modest optimization capacities, with overall DPS_{norm} gains ranging from −3.12 (Self-Refine-NL) to 1.83 (Simple). In particular, some methods tend to decrease the overall performance during the optimization process. Namely, Self-Refine-NL significantly reduces the average DPS_{norm} of the programs, particularly when combined with the DeepSeek-R1 model, resulting in a drop of 5.03 DPS_{norm} points. We hypothesize

that this reduction stems from the model's extensive reasoning steps, which may lead to inefficient optimizations.

As shown in Figure 6.1 and Table 6.2, Simple10 often matches or outperforms EoH and LLM4EFFI in terms of optimization capacity. Although one might assume that LLM4EFFI and EoH outperform traditional optimizers due to their "ideas before code" approach, this assumption is challenged by the effectiveness of a simple batch strategy, like Simple10, which achieves comparable or better results on several models. While LLM4EFFI has a better $\mathsf{DPS}_{\mathsf{norm}}$ at iteration 0, it is eventually matched by Simple10 at iteration 2 (Figure 6.2). However, as reported in Table 6.2, Simple10 achieves a notably lower Pass@result score (60.41) compared to EoH (91.41) and LLM4EFFI (90.07), indicating a higher likelihood of producing incorrect final outputs. This suggests that, while naive batching can enhance optimization capacity, it may do so at the expense of correctness. The stronger correctness of EoH and LLM4EFFI highlights that their success is likely not due to batching alone, but also to additional factors such as explicit correctness refinement (in LLM4EFFI) or evolutionary selection mechanisms (in EoH).

Figure 6.2 further illustrates how both the optimization capacity and correctness evolve over iterations. For most methods, $\mathsf{DPS}_{\mathsf{norm}}$ peaks at iteration 2, with further rounds either stabilising or decreasing performance. One exception is EoH, which continues improving until plateauing at iteration 4. LLM4EFFI, in contrast, peaks early. Indeed, optimization occurs at iteration 0, any subsequent $\mathsf{DPS}_{\mathsf{norm}}$ gains only resulting from converting invalid code into valid code.

Correctness of the optimizations. As shown in Figure 6.2, all methods but EoH and LLM4EFFI decrease in correctness over iterations. This trend is expected for methods focusing primarily on optimization, such as Simple, CoT and CoD, but is more surprising for methods like Self-Refine-Exec and Self-Refine-NL, which are designed to preserve correctness over iterations. Self-Refine-Exec, nonetheless, is the best single-code optimizer thanks to its use of test failures in prompts, enabling effective error correction. These results are consistent with those of Waghjale *et al.* on similar optimizers [103].

In contrast, LLM4EFFI and EoH improve the average correctness of the code over time, albeit relying on different mechanisms. After the initial generation, LLM4EFFI only focuses on improving code correctness. This avoids re-optimizing valid code, thereby reducing the risk of invalidation. EoH, leveraging evolutionary algorithms to implement its selection phase, retains only valid samples in each generation and attempts to repair non-passing solutions if no valid ones are found in a batch, as it is not able to produce optimizations without valid code samples.

	Cumulative energy cost of one optimization (Wh)				
optimizer	Iter-0	Iter-1	Iter-2	Iter-3	Iter-4
CoD	0.79	0.86	0.94	1.02	1.09
CoT	0.79	0.94	1.12	1.34	1.53
EoH	0.74	6.15	10.18	14.11	18.19
LLM4EFFI	3.75	5.06	5.73	6.32	6.85
Self-Refine-Exec	0.79	0.96	1.15	1.37	1.62
Self-Refine-NL	0.79	1.04	1.45	1.92	2.39
Simple	0.79	0.94	1.06	1.18	1.30
Simple10	0.79	2.47	3.81	5.10	6.31

Table 6.1: Total energy cost to get one optimized result at the last iteration depending on the optimization method over all models. For all methods but EoH and LLM4EFFI, Iter-0 is the baseline generation phase.

RQ1: Our results show that optimization capacity is strongly influenced by both the method and the LLM used. Most methods failed to reliably outperform the baseline, but batching-based optimization methods demonstrated superior performance. Interestingly, Simple10 often matched or exceeded the optimization capacity of more complex methods, highlighting that performance gains can result from effective sampling rather than complexity alone. These findings emphasize that effective optimization does not solely depend on complex reasoning or additional refinement steps, but also on strategic sample selection and iterative control. Selecting an appropriate optimization strategy thus requires balancing performance gains with the risk of introducing errors, especially in iterative workflows.

6.3.2 RQ2: What is the energy necessary to produce one optimized result?

To answer this question, we first present the distribution of energy consumption across the phases of the optimization process, and we then examine the energy consumption required to produce one optimized result in four iterations of optimization.

Energy usage by phase. The generation phase accounts for between 90.3% (Qwen2.5-Coder-7B) and 99.2% (DeepSeek-R1-14B) of the total energy consumed during the optimization process. On average, about two-thirds of the remaining energy is consumed

Method	Iteration	Efficient@1	Energy reduction	Energy / result (Wh)	BEP	Pass@1	Pass@result	DPS_{norm}	$DPS_{norm}\Delta$
CoD	2	0.819	0.803	0.94	63,086	68.1	68.1	78.1	1.09
CoT	2	0.712	0.912	1.12	95,870	62.7	62.7	78.0	0.96
EoH	2	0.869	0.664	10.18	65,100	69.3	90.2	89.6	12.6
LLM4EFFI	2	0.822	0.759	5.73	84,949	73.4	89.6	84.5	7.44
Self-Refine-Exec	2	0.758	0.822	1.15	109,913	73.1	73.1	78.1	1.08
Self-Refine-NL	2	0.725	0.921	1.45	173,683	51.7	51.7	74.8	-2.21
Simple	2	0.729	0.894	1.06	91,827	66.8	66.8	78.8	1.75
Simple10	2	0.717	0.968	3.81	155,938	66.5	62.0	85.1	8.05
CoD	4	0.795	0.821	1.09	63,586	63.4	63.4	78.7	1.68
CoT	4	0.8	0.808	1.53	102,824	53.9	53.9	77.7	0.65
EoH	4	0.759	0.786	18.19	138,997	70.7	91.4	90.3	13.24
LLM4EFFI	4	0.803	0.789	6.85	104,299	74.8	90.1	84.5	7.46
Self-Refine-Exec	4	0.737	0.877	1.62	153,831	70.3	70.3	78.0	0.96
Self-Refine-NL	4	0.783	0.849	2.39	381,912	44.4	44.4	73.9	-3.12
Simple	4	0.734	0.914	1.30	116,947	61.0	61.0	78.8	1.83
Simple10	4	0.792	0.874	6.31	310,219	59.4	60.4	85.0	7.99

Table 6.2: Energy profitability, correctness and performance of the methods at iteration 2 and 4, over all models. The optimization cost per result is the cumulative cost over all iterations.

by the correctness evaluation (from 0.48% to 6.41%), and one-third by the performance evaluation (from 0.34% to 3.29%). This result is not surprising for the generation phase, as it is the only phase involving an LLM and the GPUs. Interestingly, the energy usage is more influenced by the duration of this phase than by GPU power draw. Indeed, during generation, GPU power remained stable, but latencies varied significantly between models. For instance, generating baseline samples took 195 seconds with Qwen2.5-Coder-3B, while DeepSeek-R1-14B required 9,980 seconds for the same task.

Correctness evaluation consumes twice as much energy as performance evaluation primarily because many generated samples hit the timeout limit during this phase, either due to high time complexity or infinite loops. Since performance is only evaluated on samples that pass the correctness phase, the most computationally expensive ones are already excluded.

Energy usage of the models. Results in Table 6.3 show that larger models tend to consume more energy than smaller ones. For instance, Qwen2.5-Coder-14B and Starcoder2 (14B) require respectively 2.48 and 2.37 Wh per optimization, while the 3B and 7B variants of Qwen2.5-Coder only need 1.87 and 1.69 Wh. Interestingly, DeepSeek-R1-14B is the costliest model in terms of energy, with an average optimization cost of 16.15 Wh. While the architecture of the model itself can impact its consumption, the energy overhead of the optimization process in the DeepSeek-R1 model is largely due to its reasoning phase.

Energy usage of the optimization methods. Interestingly, as reported in Figure 6.3, the ranking of optimization methods by the energy required to produce one optimization

varies across models. For most models, EoH is the most energy-intensive, whereas for StarCoder2, LLM4EFFI consumes the most. This discrepancy arises because LLM4EFFI may repeat its task formalization step until a satisfactory prompt is found, leading to additional inference rounds depending on the model's response quality. We observed that the most energy efficient configuration was Qwen2.5-Coder-14B and CoD with an energy per optimization at iteration 4 of 0.10 Wh. On the other hand, the most energy-intensive configuration was DeepSeek-R1-14B and EoH with 52.6 Wh per optimization at iteration 4, which is almost 500 times more expensive. This underscores how much the choice of the configuration can impact the energy consumption of the optimization. In Table 6.1, we present the evolution over time of the energy required to produce one optimization depending on the optimization method.

Model	Efficient@1	Energy reduction	Optimization cost / result (Wh)	BEP	Pass@1	Pass@result	DPS_{norm}	$DPS_{norm}\Delta$
DeepSeek-R1-14B	0.907	0.625	16.15	459,057	63.5	61.4	82.8	4.57
Qwen2.5-Coder-14B	0.716	0.877	2.48	126,042	71.8	77.4	82.4	5.35
Qwen2.5-Coder-3B	0.691	0.941	1.87	110,706	52.0	61.1	80.7	2.53
Qwen2.5-Coder-7B	0.802	0.831	1.69	66,889	66.4	73.7	82.3	4.32
StarCoder2	0.762	0.924	2.37	95,192	57.4	60.6	76.2	2.4

Table 6.3: Energy profitability, correctness and performance of the models tested at the last iteration over all methods. The optimization cost per result is the cumulative cost over all iterations.

Thanks to the fewer generations they make, single-code methods require much less energy to optimize code than batching methods. As expected, Chain-of-Draft prompts the model to generate fewer tokens, thus reducing the energy use (1.09 Wh). This leads to a threefold reduction in energy per optimization compared the traditional Chain-of-Thought (1.53 Wh). Among single-code methods, Self-Refine-NL is the most energy-intensive (2.39 Wh), mainly because it requires an additional feedback generation round from the LLM which increases the total number of tokens generated.

While LLM4EFFI and Simple10 have similar batch sizes, LLM4EFFI's energy per result increases more slowly between iterations (+0.53 Wh vs +1.21 Wh of additional energy between iteration 3 and 4) as shown in Table 6.1. This is because LLM4EFFI stops optimizing samples once they are correct, whereas Simple10 tries to optimize a sample regardless of correctness or performance.

RQ2: The energy required to produce a single optimized result after four optimization rounds varies significantly depending on the model and optimization method, ranging from 0.10 Wh to 52.6 Wh. Larger models and those that rely on complex reasoning tend to consume more energy. Additionally, optimization methods that involve more inference steps or require longer outputs from the model further increase energy consumption. These findings underscore the need to balance optimization effectiveness with energy efficiency, particularly when selecting models and strategies for large-scale or repeated code optimization tasks.

6.3.3 RQ3: In terms of energy, at what point is it profitable to optimize a given code?

Having assessed both the optimization capacity and energy consumption of all evaluated models and methods, we now discuss their energy profitability, synthesized in Table 6.2 and Table 6.3. Surprisingly, all models achieved relatively consistent Efficient@1 scores, ranging from 0.734 (Simple) to 0.803 (LLM4EFFI). This indicates that all models are generally able to reduce the energy consumption of at least 73.4% of the samples below the consumption of the baseline, provided their solution is correct. In terms of absolute energy savings, EoH and LLM4EFFI were the most effective, with reductions of approximately 21%, in contrast with other methods such as Simple which only reduced energy consumption by 8.6%. Notably, although Simple10 was the second most effective method in terms of optimization capacity, it ranked only sixth in terms of energy savings.

The *break-even point* (BEP)—*i.e.*, the number of executions required for an optimization to offset its energy cost—varied significantly across methods, ranging from 63,586 (CoD) to 381,912 (Self-Refine-NL) executions. Even the most energy-efficient method in terms of BEP still requires, on average, tens of thousands of executions to justify optimization from an energy perspective. Among models, DeepSeek-R1-14B demonstrated the highest Efficient@1 and Energy reduction scores. However, it is also the model for which it takes the most executions to reach the break-even point, suggesting that its substantial energy savings only become profitable in the long run. Interestingly, while Qwen2.5-Coder-14B showed the best optimization capacity in previous analyses, its smaller variant, Qwen2.5-Coder-7B, outperformed it in terms of energy efficiency. The 7B model also had a lower BEP, making it the most cost-effective choice within its family.

The configuration combining DeepSeek-R1-14B with EoH led to the highest reduction in energy consumption, with an Efficient@1 of 0.924, an Energy reduction of 0?493, and a BEP of 220,324. While this configuration has a high BEP, it can be advantageous for

highly energy-intensive code. For instance, one baseline solution consumed an average energy consumption of 318 J, which was reduced to 2.6 J thanks to this configuration, resulting in a BEP of only 74 executions. On the other hand, the configuration using Qwen2.5-Coder-7B with CoD resulted in one of the lowest BEP scores, with an Efficient@1 of 0.805, an Energy reduction of 0.869, and a BEP of 24,016 executions.

Using one "heavyweight" or "lightweight" configuration ultimately depends on the anticipated usage patterns of the optimized code. Heavyweight setups, while energy-intensive during optimization, are suitable for optimizing code that is frequently executed or particularly energy-demanding. Lightweight configurations, on the other hand, offer lower energy costs per optimization and are preferable when optimizing a large number of infrequently executed code samples. In scenarios where the code is unlikely to be executed enough to reach the BEP, not optimizing at all may be the most energy-efficient choice.

Table 6.2 shows that running optimizations for two iterations instead of four can be more profitable in terms of energy. Indeed, for most methods, optimizing after the second iteration yields diminishing returns. For instance, EoH Efficient@1 and energy reduction actually worsen at iteration 4, and while Simple10 does get better in terms of Efficient@1 and energy reduction at iteration 4, it almost doubles its BEP. This shows that in order not to waste a great amount of energy and computing power, careful control of the number of optimization rounds is required.

Interestingly, the optimization capacity of a configuration, as measured by $\mathsf{DPS}_{\mathsf{norm}}\Delta$, and its energy reduction have a Pearson correlation coefficient of only -0.29, indicating that they are weakly correlated. Whereas at the sample level, there is a strong correlation (0.81) between the number of CPU instructions and the energy consumed by the sample. These results outline the need for the development of more energy-aware optimization methods that specifically include energy-related metrics.

RQ3: While most optimization methods and models consistently reduce energy consumption, their energy profitability strongly depends on usage context. High-performance configurations, such as DeepSeek-R1-14B with EoH, offer substantial energy reductions but require a high number of executions to break even, making them suitable for energy-intensive or frequently executed code. In contrast, lightweight setups, like Qwen2.5-Coder-7B with CoD, yield competitive energy gains at a much lower cost, offering a better trade-off for less intensive scenarios. Additionally, running multiple rounds of optimization leads to diminishing returns in terms of energy consumption. These results highlight the importance of selecting optimization strategies not only based on raw performance but also on their contextual profitability, emphasizing that energy-aware optimization should be tailored to the usage patterns of the target code. A careful planning of the number of optimization rounds is also required in order not to waste computing resources.

6.4 Discussion

6.4.1 Practical implications

Our findings offer concrete guidance for software engineers seeking to integrate LLM-based optimizers into real-world development workflows. First, the decision to invoke optimization should depend on the expected execution frequency and energy profile of the target code. Code that powers latency-sensitive hot paths or runs in large batch jobs thousands of times per day can amortize even "heavyweight" optimization costs. Conversely, for one-off scripts or lightly used utilities, using a "lightweight" setup or even skipping optimization may be the more energy-efficient choice.

Second, model and method selection should follow simple heuristics: choose smaller, faster models with token-efficient prompting when the anticipated run count is low, and reserve larger, reasoning-heavy configurations for code that will be executed at scale. Additionally, the number of optimization rounds should be carefully considered, depending on the method used.

Finally, integrating energy-aware optimization into existing CI/CD pipelines (like Google did with their ECO tool [61]) requires reliable instrumentation. Teams must measure both the energy consumed by the optimizer (*e.g.*, via GPU power meters or software hooks) and the downstream savings achieved on representative workloads. Integrating energy regression detection directly into the CI/CD pipeline can help developers focus their use of LLM-based optimizers [25]. Only with this telemetry can teams enforce

energy budgets, detect regressions, and adapt optimization policies according to changing usage patterns.

6.4.2 Decoupling performance and energy

A core insight from our study is that improvements in performance metrics do not automatically translate into energy savings. We observe a weak negative Pearson correlation (-0.29) between normalized performance gains ($\mathsf{DPS}_{\mathsf{norm}}$) and measured energy reduction at the configuration level. Consequently, benchmarks that report only runtime metrics gains risk overstating environmental benefits. Our results underscore the need for benchmarks that report execution time, energy consumption, and optimization overhead. Such multi-metric evaluations will help practitioners and researchers avoid "greenwashing" speedups and ensure that reported performance gains correspond to genuine reductions in energy footprint. Further research is needed to evaluate the correlation between energy reduction and other performance metrics, such as Beyond [32], eff@k [87] or even CPU instructions-based speedups [82].

6.4.3 The pitfalls of chasing efficiency alone

While our study demonstrates that LLM-based optimization methods can yield significant runtime and energy improvements, pursuing efficiency in isolation risks unintended environmental and societal consequences. First, efficiency gains often trigger rebound effects, whereby lower unit costs or faster runtimes induce a greater overall usage. For example, a development team that reduces optimization latency by 50% may be tempted to apply the optimizer not only in production pipelines but also in exploratory and CI workloads, effectively increasing total energy consumption rather than reducing it [67]. In extreme cases, rapid, near-free optimizations might even encourage "optimizing for optimization's sake," multiplying inference calls and model training with negligible per-run cost but huge aggregate footprint.

Second, energy consumption is only one dimension of AI's environmental impact. LLM inference depends on extensive GPU clusters, whose manufacture and operation deplete abiotic resources, consume vast quantities of water for cooling, and generate e-waste when hardware is replaced [109, 14, 18]. These embodied and indirect impacts are not captured by our energy-only metrics but can outweigh operational savings over the hardware life-cycle. For instance, a single optimizer rollout on a state-of-the-art model might save 100 J of energy per invocation, but the GPU fabrication and data-center

cooling required to support that model can entail thousands of liters of water and tens of kilograms of rare-earth mining per server.

Finally, by treating efficiency as entirely beneficial, we risk normalizing ever-greater dependence on opaque, proprietary LLM services. Organizations pursuing "green code" may inadvertently reinforce centralized cloud infrastructures, further entrenching water-intensive data-center construction and prolonging hardware refresh cycles. A more holistic approach, one that couples efficiency targets with usage caps, transparency in resource accounting, and incentives for lightweight or on-device solutions, is therefore essential to truly reducing environmental impact rather than simply shifting the burden [67].

6.5 Limitations & Threats to Validity

While our study provides novel insights into the energy and performance trade-offs of LLM-based code optimization, several limitations constrain the generalizability of our findings. First, we rely exclusively on the EvalPerf benchmark, which focuses on CPU-bound algorithmic tasks. This dataset consists of small programs, often containing one or two functions that solve an algorithmic problem. As such, it does not capture I/O-bound programs, large data-processing pipelines, or real-world microservices, all of which exhibit distinct performance and energy profiles. As a result, our break-even estimates and energy savings percentages may not generalize to domains where memory bandwidth, network latency, or storage I/O dominate. Consequently, there is an urgent need for new LLM-evaluation datasets that encompasses broader software engineering tasks.

Second, our measurements were conducted on NVIDIA A100 GPUs in a controlled environment. Consumer-grade GPUs, CPU inference, and diverse datacenter cooling and power infrastructures can exhibit substantially different power draw curves and thermal behaviors. Therefore, the absolute energy figures we report (*e.g.*, joules per optimization) should be interpreted as indicative rather than definitive, and practitioners should recalibrate our findings to their target hardware.

Third, to increase the precision of the results, we evaluate the code samples with a minimum measurement duration of 1 second. While we noticed a strong correlation between the energy usage and the number of CPU instructions of the samples–indicating a stable enough measuring setup, this measurement duration might still be too short for an accurate comparison of the programs' energy measurements, which, in turn, might incur biases and errors in our results.

Finally, we evaluated only five LLMs that represent current state-of-the-art code-specialized and general-purpose architectures. Given the rapid pace of model development, quantized, distilled, or next-generation transformers may offer different energy–performance trade-offs than those we observed. Thus, our conclusions about model ranking and break-even behavior may shift as new architectures emerge. Extending our framework to include a broader array of models will be necessary to confirm the robustness of these trends.

6.6 Conclusion

In this chapter, we provided an answer to the questions *"What is the real energy cost of using LLM-powered code optimizers? Is the performance and energy gain of optimized code sufficient to justify the energy cost of optimization?"* by systematically evaluating a broad spectrum of LLM-based optimization methods and models, quantifying their performance improvements, correctness preservation, and energy implications. Our results demonstrate that batching strategies, such as EoH and LLM4EFFI, deliver the strongest performance improvement while maintaining correctness over multiple iterations at the cost of a higher energy consumption. Energy profiling revealed that generating the optimization dominates the optimization pipeline, and that larger models incur disproportionately high energy costs. Most importantly, we showed that the energy "break-even" point for these methods can range from under 100 to over 10^5 executions, underscoring the necessity of usage-aware, energy-centric decision making when adopting LLM-driven optimizers.

Looking forward, our study highlights both the potential and the pitfalls of integrating LLMs into energy-sensitive software workflows. While "heavyweight" configurations can yield dramatic energy reductions in hot-path code, "lightweight" setups or even selective forgoing of optimization may be more prudent for less frequently run tasks. Moreover, the broader environmental footprint of LLM inference, encompassing resource extraction, water use, and data-center operations, demands a nuanced perspective beyond operational energy metrics alone. We hope this work serves as a foundation for future research on adaptive, energy-aware optimization frameworks and green AI methods that jointly balance performance and sustainability.

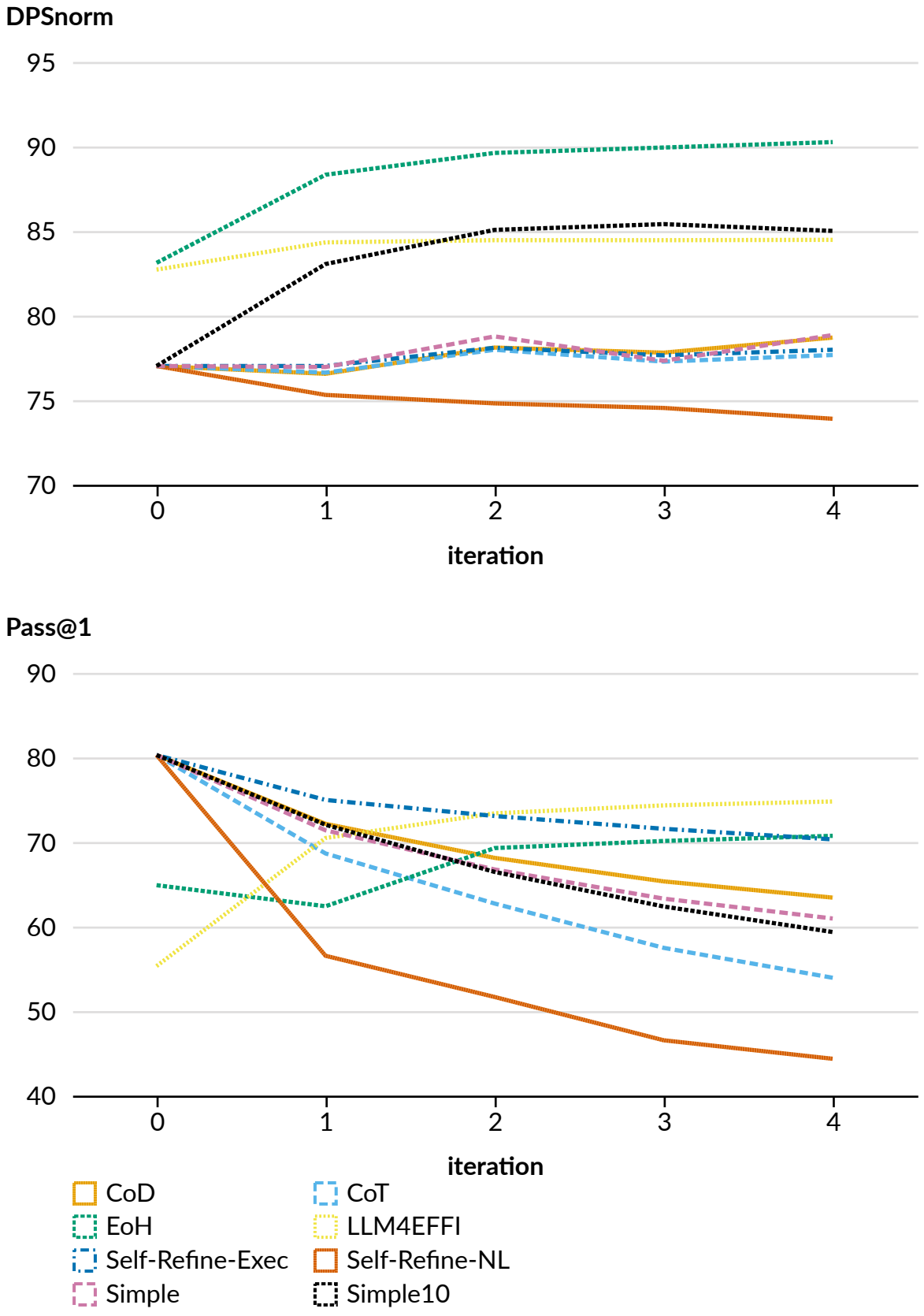

Figure 6.2: Pass@1 and DPS_{norm} across iterations depending on the method. The point at iteration 0 is the same for all methods except LLM4EFFI and EoH, as it is the baseline. A higher Pass@1 and DPS_{norm} is better.

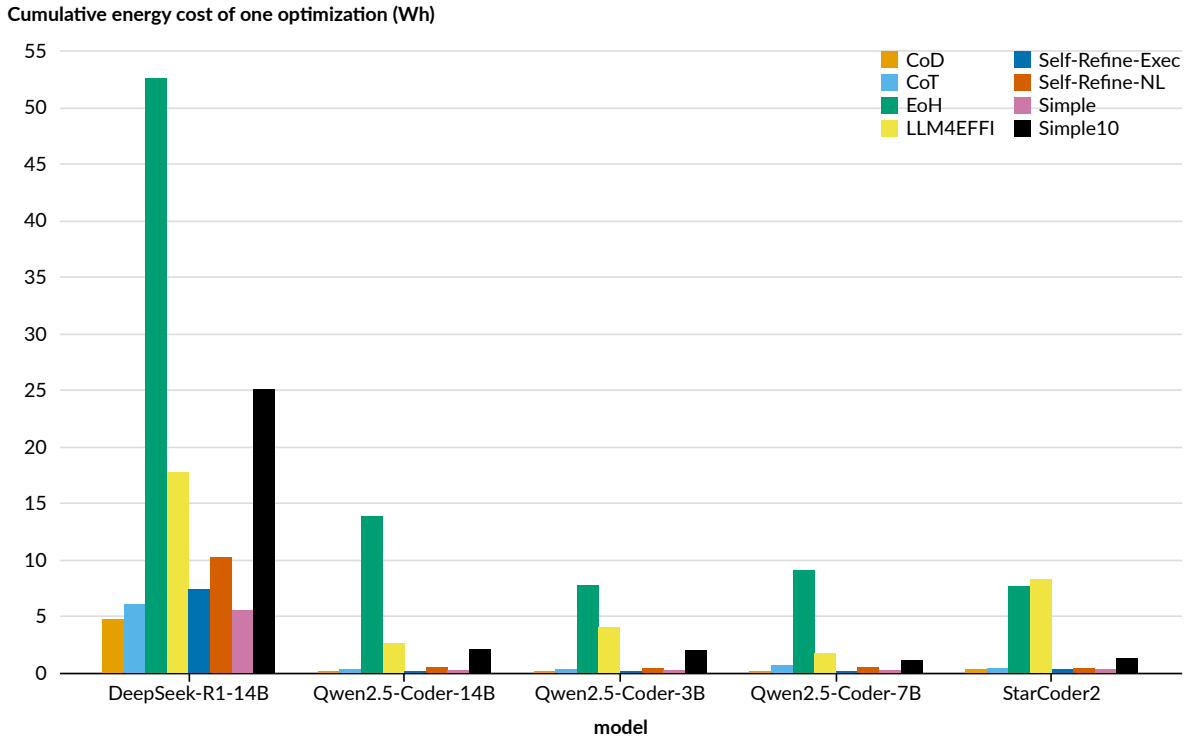

Figure 6.3: Total energy consumption required to get one optimized result at the last iteration depending on the model and optimization method. Lower is better.

Chapter 7

Conclusion

In this chapter, I summarize this thesis by synthesizing my contributions and by providing perspectives related to the work presented in this thesis.

7.1 Summary of the contributions

In this thesis, I set out to explore the energy impact of using Large Language Models (LLMs) in software engineering, specifically for code generation and optimization. This exploration was guided by the overarching question **"What is the negative and positive energy impact of code LLMs?"** which gave rise to three empirical contributions pertaining to parts of this question, each offering a distinct perspective on the sustainability of LLMs in software development.

The **first contribution** focused on the *performance of LLM-generated code* using Leetcode as an evaluation framework. Through a large-scale benchmarking of 18 LLMs across 204 programming tasks, this study introduced a novel methodology to measure the runtime efficiency of generated solutions. The results revealed that while LLMs can often produce code that is competitive, sometimes even outperforming human-written solutions, model configuration (*e.g.*, temperature) plays a major role in determining efficiency and correctness. This work also exposed critical methodological limitations in using public benchmarks like Leetcode due to data contamination, highlighting the need for more robust evaluation protocols.

The **second contribution** quantified the *energy consumption of LLM-powered code assistants* through a user study involving GitHub Copilot. By replaying development traces on inference servers, the study demonstrated that a significant share of energy is wasted on unused or cancelled suggestions. It also identified several levers for reducing energy use, such as shifting from automatic to manual suggestion modes, optimizing

server configurations, and employing model quantization techniques. This work provided actionable recommendations for both tool designers and developers, and stressed the importance of raising user awareness about the hidden energy costs of AI-powered tools.

The **third contribution** evaluated the net *energy impact of LLM-based code optimization*. It introduced the concept of the *Break-Even Point* (BEP), the number of executions required for an optimized program to offset the energy cost of the optimization itself. While LLMs were shown to significantly reduce execution-time energy usage in many scenarios, the BEP analysis revealed that these gains are highly context-dependent. In low-frequency or short-lived applications, the cost of optimization may never be recovered. Moreover, the study highlighted the fragility of current optimization pipelines, with correctness rates and performance improvements varying widely across models and methods.

Taken together, these contributions offer a comprehensive and empirically grounded view of the environmental trade-offs inherent to using LLMs for software development. They show that while LLMs hold promise as tools for sustainable software engineering, this potential is not automatically realized. Rather, it depends on a careful balance between technical efficiency and mindful usage.

The results I presented in this thesis challenge the simplistic narrative that automation and AI necessarily lead to environmental benefits. While LLMs can reduce time-to-solution or improve runtime efficiency, they also consume considerable energy and material resources in the process. The widespread deployment of such tools risks triggering material, economic, and behavioral rebound effects that may ultimately worsen the ICT sector's footprint.

7.2 Perspectives

While the results presented in this thesis provide insights into the energy impacts of code assistants and code optimizers, they also open several promising directions for future research. We outline both short-term and long-term perspectives that could build upon and expand the contributions of this work.

7.2.1 Short-Term Perspectives

Improving the evaluation of code assistants. While this thesis introduced a framework to assess the energy cost of using code assistants like GitHub Copilot, further refinements are warranted. One short-term avenue involves conducting user

studies with **true manual triggering of the code assistant** (as opposed to automatic suggestions or an emulation of the manual triggering), to evaluate how much energy could realistically be saved and how this affects developer experience and productivity, this research avenue could leverage the work of Mozannar *et al.* and Barke *et al.* [74, 13], which laid some groundwork on studying the behavior of humans when interacting with code assistants. Additionally, the existing study could be replicated while focusing on **code quality and correctness of the accepted suggestions**, in order to link energy cost with actual value delivered to the developer. Code quality and correctness of the suggestions could also be integrated in the study as a quality metric as is latency and energy cost, which could give more insights into the various possible configurations and their trade-offs.

Wider configuration and hardware studies. The energy footprint of code assistants and LLM-based tools depends heavily on their deployment configuration. Similar to the research by Fernandez *et al.* and Samsi *et al.* [36, 91], future work could expand the empirical evaluations by considering a **wider range of inference server configurations**, including GPU architectures, memory capacities, batching strategies, and concurrency levels. Similarly, the impact of quantization techniques deserves a more systematic investigation, especially regarding their trade-offs between accuracy, latency, and energy consumption.

Broader evaluation of LLM-based code optimization. The study on LLM-based code optimization presented in chapter 6 focused on a specific benchmark and hardware setup. A natural follow-up would be to **repeat this evaluation on alternative benchmarks**, as presented in subsection 3.3.1, or alternative hardware platforms (notably other kinds of CPUs and GPUs), to understand how the break-even dynamics change. Additionally, **expanding the set of LLMs and optimization algorithms** could provide a more comprehensive picture of the performance and environmental impact landscape.

7.2.2 Long-Term Perspectives

Standardizing sustainable code benchmarks and metrics. The recent proliferation of benchmarks for evaluating LLM-generated code efficiency (*e.g.*, COFFE, ENAMEL and EvalPerf) provided valuable tools for research. However, the lack of shared standards across these benchmarks makes it difficult to compare results or synthesize insights. A fruitful long-term avenue would be to define a **unified, extensible benchmarking**

suite for sustainable code generation.[1] This could include metrics for correctness, runtime, memory use, and energy efficiency, as well as standardized test case generators and contamination-mitigation procedures. Ideally, such a benchmark would support multiple languages and development tasks, allowing for modular evaluation. While making a unified benchmark would certainly be quite a feat, it could be helped by reusing what already works well in the methodologies of existing benchmarks, or using existing benchmarks' tasks as subsets of a bigger benchmark. Establishing standards **backed by the community**–perhaps through organizations like ML Energy or Boavizta–could accelerate progress by aligning incentives, improving reproducibility, and making research findings more interoperable.

Designing sober and sustainable code assistants. Code assistants have the potential to substantially improve developer productivity and code quality. However, as this thesis has shown, their widespread use can also lead to significant energy overheads, especially when they are triggered too frequently or deployed inefficiently. A long-term vision is to develop "sober" code assistants: tools that are designed from the ground up with restraint and sustainability in mind. This might include **adaptive suggestions** that learn a project's execution history and only suggest high-impact edits (*e.g.*, based on static analysis or historical usage data), or **lightweight inference architectures** that minimize computational demand. Developing such assistants would likely require performance-oriented datasets that capture development workflows, metrics for balancing usability with energy cost, and possibly rethinking what "helpfulness" should mean in the context of sustainable software engineering. The development of sustainable code assistants could also require novel methods to **estimate the energy consumption of (part of) a program in real time**. Additionally, the **medium to long term effects of those code assistants** could be studied through A/B studies measuring not just energy saved, but developer satisfaction, cognitive load, and code quality over weeks of real use.

Modeling and mitigating rebound effects. Efficiency gains do not always translate to net resource savings [67]. As hypothesized earlier in the thesis, improvements in LLM efficiency or optimized code could lead to increased usage (more frequent queries, higher throughput systems, or more complex applications being deployed) which ultimately cancels out the environmental benefits. These are known as rebound effects. A compelling research direction is to **model and quantify these effects** more rigorously in the

[1] Relevant XKCD: https://xkcd.com/927/

context of software engineering. **What happens when LLM-based tools make development too frictionless?** Do developers iterate more, accept more marginal suggestions, or automate tasks they wouldn't have otherwise? Answering these questions may require integrating **behavioral modeling** with **monitoring**. For example, answers could be found with empirical studies that instrument CI/CD pipelines and cloud deployments to track how LLM usage correlates with feature and service production, and infrastructure scale which could lead to tooling that helps mitigate the rebound effects (*e.g.*, "energy quotas"). Ultimately, a mature understanding of rebound effects could guide the responsible design of tools and infrastructures.

End-to-end life cycle assessment of code LLMs. Much of the existing work on LLM sustainability focuses on energy consumption, particularly during inference. Yet energy is only one piece of the picture. A more holistic and long-term perspective would be to develop **life cycle assessments of code-focused LLMs and LLM-powered tools**, evaluating not only their electricity usage but also the environmental impacts of training, hardware manufacturing, deployment infrastructure, water usage, mineral depletion, and electronic waste. This direction builds on existing efforts such as those by Berthelot *et al.* [14] for image models and those by Simon *et al.* [95] on software, and would benefit from **interdisciplinary collaboration** between software engineers, environmental scientists, hardware manufacturers, and datacenter operators. Such work would require extensive **data collection** across multiple layers (*e.g.*, hardware, datacenter and network.) for use cases specific to code LLMs (*e.g.*, code assistants, CI/CD tooling) and would need to integrate multiple environmental metrics, that is, not just CO2, but also water usage, mineral depletion and electronic waste increase. A robust LCA framework could inform not only academic studies, but also policy decisions, encouraging more transparent and sustainable AI systems.

These perspectives show that making software development with LLMs more sustainable isn't just a technical problem: it's also about how we design, use, and think about these tools. There's a lot we can still do, from improving the way we measure their impact to rethinking how assistants are built and used. There is still a long way to go until we can make sure that these tools can truly help, not just in terms of productivity, but also in reducing their environmental footprint. I hope this thesis can contribute, even in a small way, to that conversation.

Bibliography

[1] 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/.

[2] Copilot-explorer. https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals.html.

[3] Nvidia GPU. http://www.designlife-cycle.com/nvidia-gpu.

[4] Codeparrot/codeparrot · Hugging Face. https://huggingface.co/codeparrot/codeparrot, October 2022.

[5] The periodic table of mobile phones. *https://www.resources.nsw.gov.au/sites/default/files/2022-11/periodic-table-of-mobile-phones-a3complete.pdf*, 2022.

[6] Measuring GitHub Copilot's Impact on Productivity – Communications of the ACM, February 2024.

[7] Hayri Acar, Gülfem I Alptekin, Jean-Patrick Gelas, and Parisa Ghodous. The impact of source code in software on power consumption. *International Journal of Electronic Business Management*, 14:42–52, 2016.

[8] Datu Buyung Agusdinata, Wenjuan Liu, Hallie Eakin, and Hugo Romero. Socio-environmental impacts of lithium mineral extraction: Towards a research agenda. *Environmental Research Letters*, 13(12):123001, November 2018.

[9] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. SantaCoder: Don't reach for the stars!, January 2023.

[10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, August 2021.

[11] Cornelis P Baldé, Ruediger Kuehr, Tales Yamamoto, Rosie McDonald, Elena D'Angelo, Shahana Althaf, Garam Bel, Otmar Deubzer, Elena Fernandez-Cubillo, Vanessa Forti, et al. Global e-waste monitor 2024, 2024.

[12] Daniel Balouek et al. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. 2013.

[13] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):78:85–78:111, April 2023.

[14] Adrien Berthelot, Eddy Caron, Mathilde Jay, and Laurent Lefèvre. Estimating the environmental impact of Generative-AI services using an LCA-based methodology. *Procedia CIRP*, 122:707–712, 2024.

[15] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative Inference and Fine-tuning of Large Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 558–568, Toronto, Canada, 2023. Association for Computational Linguistics.

[16] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early experiments with GPT-4, April 2023.

[17] Sophie Cerf, Adrien Luxey-Bitri, Clément Quinton, Romain Rouvoy, Thibault Simon, and Catherine Truffert. Untangling the Critical Minerals Knot: When ICT hits the Energy Transitions.

[18] Sophie Cerf, Adrien Luxey-Bitri, Clément Quinton, Romain Rouvoy, Thibault Simon, and Catherine Truffert. Untangling the critical minerals knot: When ICT hits the energy transitions. December 2023.

[19] Angelica Chen, David Dohan, and David So. EvoPrompting: Language Models for Code-Level Neural Architecture Search. *Advances in Neural Information Processing Systems*, 36:7787–7817, December 2023.

[20] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021.

[21] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to Generate Source Code Optimizations in C/C++, October 2023.

[22] Tristan Coignion, Clément Quinton, and Romain Rouvoy. Green My LLM: Studying the key factors affecting the energy consumption of code assistants, November 2024.

[23] Tristan Coignion, Clément Quinton, and Romain Rouvoy. A Performance Study of LLM-Generated Code on Leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 79–89, Salerno Italy, June 2024. ACM.

[24] Vlad-Andrei Cursaru, Laura Duits, Joel Milligan, Damla Ural, Berta Rodriguez Sanchez, Vincenzo Stoico, and Ivano Malavolta. A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama, May 2024.

[25] Benjamin Danglot, Jean-Rémy Falleri, and Romain Rouvoy. Can we spot energy regressions using developers tests? *Empirical Software Engineering*, 29(5):121, July 2024.

[26] Tim Dettmers. TimDettmers/bitsandbytes, May 2024.

[27] Ben Dickson. Why data contamination is a big issue for LLMs - TechTalks, July 2023.

[28] John Dimitropoulos. Energy productivity improvements and the rebound effect: An overview of the state of knowledge. *Energy Policy*, 35(12):6354–6363, December 2007.

[29] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?, June 2023.

[30] Petra Döll, Tim Trautmann, Dieter Gerten, Hannes Müller Schmied, Sebastian Ostberg, Fahad Saaed, and Carl-Friedrich Schleussner. Risks for the global freshwater system at 1.5 °C and 2 °C global warming. *Environmental Research Letters*, 13(4):044038, April 2018.

[31] Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models, May 2024.

[32] Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A Code Efficiency Benchmark for Code Large Language Models, June 2024.

[33] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A Comparative Analysis of Large Language Models for Code Documentation Generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, pages 65–73, New York, NY, USA, July 2024. Association for Computing Machinery.

[34] Ember. Carbon intensity of electricity generation – ember and energy institute, 2024.

[35] U. N. Environment. Global Resources Outlook 2024 | UNEP - UN Environment Programme. https://www.unep.org/resources/Global-Resource-Outlook-2024, Thu, 02/22/2024 - 18:33.

[36] Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy Considerations of Large Language Model Inference and Efficiency Optimizations, April 2025.

[37] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis, April 2022.

[38] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. Search-Based LLMs for Code Optimization, August 2024.

[39] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. DeepDev-PERF: A deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 948–958, New York, NY, USA, November 2022. Association for Computing Machinery.

[40] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. RAPGen An Approach for Fixing Code Inefficiencies in Zero-Shot, July 2024.

[41] Cindy Gordon. ChatGPT Is The Fastest Growing App In The History Of Web Applications. https://www.forbes.com/sites/cindygordon/2023/02/02/chatgpt-is-the-fastest-growing-ap-in-the-history-of-web-applications/.

[42] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 1, December 2021.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The Curious Case of Neural Text Degeneration, February 2020.

[44] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization, May 2025.

[45] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. EffiBench: Benchmarking the Efficiency of Automatically Generated Code, July 2024.

[46] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. SwiftCoder: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning, March 2025.

[47] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder Technical Report, November 2024.

[48] Jerry K. Jacka. The Anthropology of Mining: The Social and Environmental Impacts of Resource Extraction in the Mineral Age. *Annual Review of Anthropology*, 47(Volume 47, 2018):61–77, October 2018.

[49] Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. Stop Uploading Test Data in Plain Text: Practical Strategies for Mitigating Data Contamination by Evaluation Benchmarks, October 2023.

[50] Akshaya Jagannadharao, Nicole Beckage, Sovan Biswas, Hilary Egan, Jamil Gafur, Thijs Metsch, Dawn Nafus, Giuseppe Raffa, and Charles Tripp. A Beginner's Guide to Power and Energy Measurement and Estimation for Computing and Machine Learning, December 2024.

[51] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code, June 2024.

[52] Nidhal Jegham, Marwen Abdelatti, Lassad Elmoubarki, and Abdeltawab Hendawi. How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference, May 2025.

[53] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. Large Language Models and Simple, Stupid Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575, May 2023.

[54] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023.

[55] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach (Artifact)*, 8(OOPSLA1):111:474–111:499, April 2024.

[56] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models, March 2025.

[57] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason

Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: May the source be with you! 2023.

[58] Shuang Li, Yuntao Cheng, Jinfu Chen, Jifeng Xuan, Sen He, and Weiyi Shang. Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, Tsukuba, Japan, October 2024. IEEE.

[59] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-Level Code Generation with AlphaCode, February 2022.

[60] Weixin Liang, Yaohui Zhang, Zhengxuan Wu, Haley Lepp, Wenlong Ji, Xuandong Zhao, Hancheng Cao, Sheng Liu, Siyu He, Zhi Huang, Diyi Yang, Christopher Potts, Christopher D. Manning, and James Y. Zou. Mapping the Increasing Use of LLMs in Scientific Papers, April 2024.

[61] Hannah Lin, Martin Maas, Maximilian Roquemore, Arman Hasanzadeh, Fred Lewis, Yusuf Simonson, Tzu-Wei Yang, Amir Yazdanbakhsh, Deniz Altinbüken, Florin Papa, Maggie Nolan Edmonds, Aditya Patil, Don Schwarz, Satish Chandra, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers, March 2025.

[62] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model, June 2024.

[63] 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.

[64] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating Language Models for Efficient Code Generation, August 2024.

[65] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii

Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation, February 2024.

[66] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power Hungry Processing: Watts Driving the Cost of AI Deployment?, November 2023.

[67] Alexandra Sasha Luccioni, Emma Strubell, and Kate Crawford. From Efficiency Gains to Rebound Effects: The Problem of Jevons' Paradox in AI's Polarized Environmental Debate, January 2025.

[68] Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power Hungry Processing: Watts Driving the Cost of AI Deployment? In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, pages 85–99, Rio de Janeiro Brazil, June 2024. ACM.

[69] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering Code Large Language Models with Evol-Instruct, June 2023.

[70] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. 2023.

[71] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback, March 2023.

[72] Inbal Magar and Roy Schwartz. Data Contamination: From Memorization to Exploitation, March 2022.

[73] Clément Morand, Anne-Laure Ligozat, and Aurélie Névéol. How Green Can AI Be? A Study of Trends in Machine Learning Environmental Impacts, December 2024.

[74] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(9):10137–10144, March 2024.

[75] NetEase-FuXi. NetEase-FuXi/EETQ, May 2024.

[76] Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, Pittsburgh Pennsylvania, May 2022. ACM.

[77] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis, September 2022.

[78] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, USA, July 2002. Association for Computational Linguistics.

[79] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, San Francisco, CA, USA, May 2022. IEEE.

[80] Huiyun Peng, Arjun Gupte, Nicholas John Eliopoulos, Chien Chou Ho, Rishi Mantri, Leo Deng, Wenxin Jiang, Yung-Hsiang Lu, Konstantin Läufer, George K. Thiruvathukal, and James C. Davis. Large Language Models for Energy-Efficient Code: Emerging Results and Future Directions, October 2024.

[81] Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback, November 2024.

[82] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. COFFE: A Code Efficiency Benchmark for Code Generation, February 2025.

[83] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 2785–2799, New York, NY, USA, November 2023. Association for Computing Machinery.

[84] R. A. Pielke, C. Landsea, M. Mayfield, J. Layer, and R. Pasch. Hurricanes and Global Warming. *Bulletin of the American Meteorological Society*, 86(11):1571–1576, November 2005.

[85] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Computing Surveys*, 51(5):1–36, September 2019.

[86] Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T. Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 1, December 2021.

[87] Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark, June 2024.

[88] Pooja Rani, Jan-Andrea Bard, June Sallou, Alexander Boll, Timo Kehrer, and Alberto Bacchelli. Can We Make Code Green? Understanding Trade-Offs in LLMs vs. Human Code Optimizations, March 2025.

[89] Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models, March 2024.

[90] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, August 2023.

[91] 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, September 2023.

[92] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Security Implications of Large Language Model Code Assistants: A User Study. 2022.

[93] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, October 2023.

[94] Alexander G. Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*, October 2023.

[95] Thibault Simon, Pierre Rust, Romain Rouvoy, and Joël Penhoat. Uncovering the Environmental Impact of Software Life Cycle. In *2023 International Conference on ICT for Sustainability (ICT4S)*, pages 176–187, June 2023.

[96] Hamed Taherkhani, Melika Sepindband, Hung Viet Pham, Song Wang, and Hadi Hemmati. EPiC: Cost-effective Search-based Prompt Engineering of LLMs for Code Generation, August 2024.

[97] Jonas F. Tuttle, Dayuan Chen, Amina Nasrin, Noe Soto, and Ziliang Zong. Can LLMs Generate Green Code - A Comprehensive Study Through LeetCode. In *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*, pages 39–44, November 2024.

[98] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–7, New Orleans LA USA, April 2022. ACM.

[99] Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. Learn to Code Sustainably: An Empirical Study on Green Code Generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, pages 30–37, New York, NY, USA, September 2024. Association for Computing Machinery.

[100] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q. Vera Liao, and Jennifer Wortman Vaughan. Generation Probabilities Are Not Enough: Exploring the Effectiveness of Uncertainty Highlighting in AI-Powered Code Completions. 2023.

[101] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023.

[102] Roberto Verdecchia, Giuseppe Procaccianti, Ivano Malavolta, Patricia Lago, and Joost Koedijk. Estimating Energy Impact of Software Releases and Deployment Strategies: The KPMG Case Study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 257–266, November 2017.

[103] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness?, July 2024.

[104] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, April 2024.

[105] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada, 2023. Association for Computational Linguistics.

[106] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, January 2023.

[107] Rand R Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Academic press, 2011.

[108] World Bank Group. *Measuring the Emissions and Energy Footprint of the ICT Sector: Implications for Climate Action*. Other Environmental Study. The World Bank, Washington, D.C, 2024.

[109] Dustin Wright, Christian Igel, Gabrielle Samuel, and Raghavendra Selvan. Efficiency is Not Enough: A Critical Perspective of Environmentally Sustainable AI, March 2025.

[110] Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. Chain of Draft: Thinking Faster by Writing Less, March 2025.

[111] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. Automated Test Case Repair Using Language Models, January 2024.

[112] Tong Ye, Weigang Huang, Xuhong Zhang, Tengfei Ma, Peiyu Liu, Jianwei Yin, and Wenhai Wang. LLM4EFFI: Leveraging Large Language Models to Enhance Code Efficiency and Correctness, February 2025.

[113] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, Lisbon Portugal, February 2024. ACM.

[114] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation, August 2024.