Green My LLM: Studying the Key Factors Affecting the Energy Consumption of Code Assistants

Anonymous Author(s)

Abstract

In recent years, *Large Language Models* (LLMs) have significantly improved in generating high-quality code, enabling their integration into developers' *Integrated Development Environments* (IDEs) as code assistants. These assistants, such as GitHub Copilot, deliver real-time code suggestions and can greatly enhance developers' productivity. However, the environmental impact of these tools, in particular their energy consumption, remains a key concern. This paper investigates the energy consumption of LLM-based code assistants by simulating developer interactions with GitHub Copilot and analyzing various configuration factors. We collected a dataset of development traces from 20 developers and conducted extensive software project development simulations to measure energy usage under different scenarios.

Our findings reveal that the energy consumption and latency of code assistants are influenced by various factors, such as the number of concurrent developers, model size, quantization methods, and the use of streaming. Notably, a substantial portion of generation requests made by GitHub Copilot is either canceled or rejected by developers, indicating a potential area for reducing wasted computations. Based on these findings, we share actionable insights into optimizing configurations for different use cases, demonstrating that careful adjustments can lead to significant energy savings.

Keywords

language models, code assistants, energy consumption, Green AI, Empirical Software Engineering;

ACM Reference Format:

Anonymous Author(s). 2026. Green My LLM: Studying the Key Factors Affecting the Energy Consumption of Code Assistants. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In recent years, *Large Language Models* (LLMs) for code have significantly become better at generating code, facilitating their seamless integration into developers' *Integrated Development Environments* (IDEs) as code assistants. Code assistants 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, the energy consumption of software has gained prominence as a significant environmental and societal concern. In the context of Artificial Intelligence (AI), Green AI has been defined by Schwartz *et al.* [29] as research that yields novel results while considering the computational cost, making practitioners ideally reduce resources spent. Specifically to Green AI applied to LLMs, studies have observed the environmental impact of training and using such LLMs. 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 [27]. 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.* [20]. 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 [9, 33].

In this paper, we aim to determine how much energy an average developer consumes when using a code assistant similar to GitHub Copilot and 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 latency 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.

While our primary focus is on the energy consumption of the code assistant, we also consider latency (*i.e.*, the time taken to generate suggestions), as it directly impacts the usability of the tool. If energy-saving configurations introduce excessive latency, the assistant may no longer provide timely support to developers, thereby undermining its practical utility.

RQ2: How many generation requests made by GitHub Copilot are useful? Knowing how many generations are

EASE 2026, Glasgow, Scotland, United Kingdom

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

useful to the developer could allow future work to improve the efficiency of code assistants.

RQ3: How much energy does a developer use when using a code assistant similar to GitHub Copilot under 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 Copilot specifically for three reasons: *(i)* It is the most used code assistant according to the 2024 StackOverflow developer survey,[1] *(ii)* the inference server provided by GitHub Copilot exhibits low latency and correct quality [11], which we may not be able to reproduce with our own inference server, and *(iii)* GitHub Copilot provides many mechanisms making it one of the state-of-the-art code assistants, 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 prompts that take into account multiple files. We are aware of the existence of these mechanisms thanks to the Copilot Explorer project [1], while evidence of similar mechanisms in other code assistants is unclear, and verification is time-consuming. In this paper, 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 latency.
- 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.

Outline. From section 2 to section 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 on the results of our evaluation and provide a critical discussion in section 6. In section 7, we discuss the limitations of our study. Finally, section 8 presents related works, and section 9 concludes the paper.

2 Code Assistant 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 Copilot to 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 energy 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. Our setup is illustrated by Figure 1.

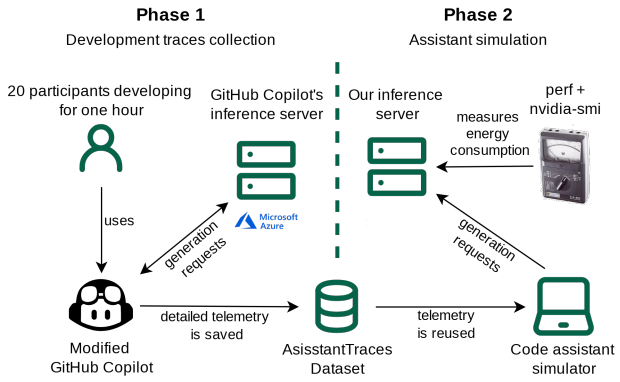

Figure 1: The setup of our experiment. The left part represents the dataset collection with human participants, and the right part represents how we reproduced GitHub Copilot by sending captured telemetry to our own server.

2.1 Involved participants

We recruited 20 volunteers among Computer Science (CS) students and CS professionals, using mailing lists and word of mouth. Before the experiment, the 20 participants filled out a survey with their age, development experience, and familiarity with GitHub Copilot.

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 CS master's degree and 1 in a CS bachelor's degree), and 7 of them were CS professionals. All the participants were proficient in Java programming. 10 participants did not write any Java code for more than a year, while the other 10 did write Java code in the last year. 1 participant did not know GitHub Copilot before the experiment, 7 participants knew GitHub Copilot but never used it, 4 used it sometimes, and 8 used it regularly. All the participants were familiar with VS Code. This experiment was approved by our institution's Ethical Board.

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 VS Code and GitHub Copilot A skeleton of the project and associated unit tests were provided. The skeleton of the project contained simply the bare minimum Maven project that could function and was provided to avoid participants losing time setting up a working Maven project. It also contained the unit tests and relevant interfaces so that participants could easily verify their implementation. 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 [5].

[1] https://survey.stackoverflow.co/2024/

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

The participants used the GitHub Copilot VS Code 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 assistants (*e.g.*, ChatGPT). This constraint was enforced by the experimenter watching over the participant.

We collected GitHub Copilot's telemetry by modifying the VS Code 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 Copilot decides to send a generation request to the moment of its acceptance from the user. GitHub Copilot's telemetry was collected during the whole experiment, including times when participants were thinking, debugging or testing.

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.

2.3 Dataset description

The AssistantTraces dataset consists of all the telemetry sent by GitHub Copilot during the experiment in JSON format. There are 119,774 telemetry messages in total, including 9,633 generation requests. The dataset is available bundled with our replication package [5], or in standalone format at https://doi.org/10.5281/zenodo.11503612 [4] for ease of use.

3 Experimental 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. This section is also illustrated by Figure 1. Specifically, the inference servers were hosted 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.

3.1 Simulation client

Thanks to the telemetry data from the AssistantTraces dataset, we could reproduce the API requests that GitHub Copilot sent to its inference server. We developed a simulator (the *client*) to mimic developers' interactions with GitHub Copilot by 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, to keep the simulation times short and manageable, we limited the simulation to the first hour of development, even when some developers took more than one hour to complete the task. For the developers who completed the task in less than an hour, we delayed their start times so that the middle point of each simulation aligned, thus creating a more realistic distribution of the server's workload.

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. However, to avoid issues with simultaneous identical requests, we offset each duplicate by a random number of 0 to 30 seconds.

3.2 Inference server

To handle generation requests and generate code suggestions, we set up an inference server that generated code suggestions using an LLMs. In particular, we used the Text Generation Inference (TGI) server,[3]. Our choice was motivated by its ability to operate various LLMs and by its support of sharding and 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.

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 Copilot by activating and deactivating streaming when sending requests. By default, GitHub Copilot uses 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 Copilot has 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. But we also emulated a manual trigger for the generation behavior. This was achieved by only sending generation requests that were completed and displayed to the user based on the assumption that the developer manually triggering GitHub Copilot would 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) [15], StarCoder2-7b, and StarCoder2-15b [17]. 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

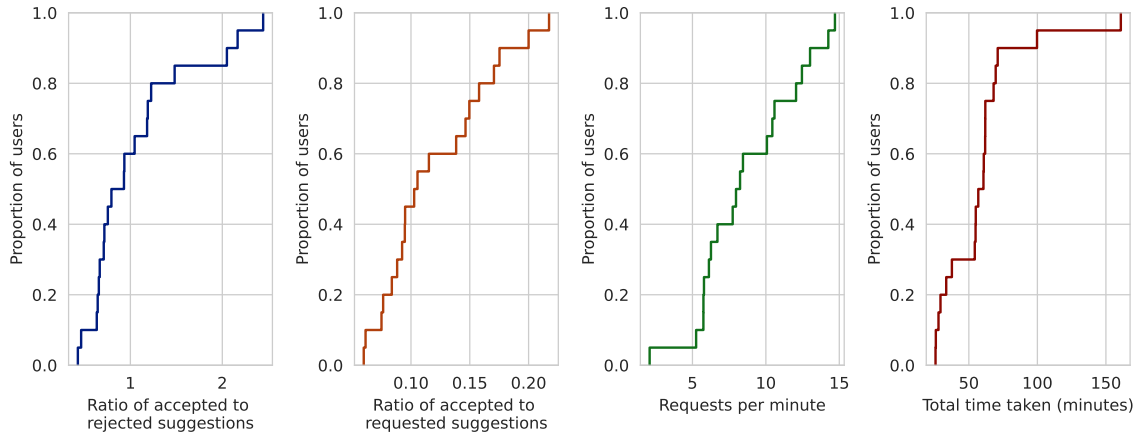

Figure 2: Various statistics on the participants' usage of the code assistant and their time taken to finish the experiment. The figure respectively represent: (i) the ratio of the number of suggestions accepted by the user to the number of suggestions by the user, (ii) the ratio of the number of accepted suggestions to the total number of suggestions requested by the code assistant, (iii) the frequency of requests made by the participant and (iv) 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 post-training quantization methods: EETQ [23], BitsAndBytes-NF4, BitsAndBytes-FP4 [10] as well as a baseline configuration without quantization. We chose EETQ and BitsAndBytes because they are the only natively supported methods by the TGI framework which are also compatible with our LLMs.

Maximum number of concurrent requests: We varied the number of concurrent requests on the TGI server, which effectively modified the number of requests to be queued simultaneously.

Number of GPUs: 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.

3.4 Configuration space exploration

A configuration is a set of factors with a specific value (*e.g.*, [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 that take between 1 and 20 hours each to simulate. Thus, because of timing and budget constraints, 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. The simulation results can be found in our replication package [5].

3.5 Energy & latency measurements

We define the **energy consumption** of a request as the total amount of energy (in Wh) used by the CPU and GPU of the inference server to generate a suggestion. Correspondingly, the **power consumption** refers to the instantaneous or average rate of energy use (in W) during inference. Finally, the **latency** of a request is the time elapsed (in seconds) between the moment the generation request is sent to the inference server and the time its response is received.

We measured the energy consumption of the inference server's CPU and GPU using `perf` and `nvidia-smi` utilities, respectively. `perf` is a Linux utility that uses RAPL, an interface that provides built-in CPU energy counters. `nvidia-smi`, on the other hand, provides energy measurements made directly on the GPU. RAPL and `nvidia-smi` have been used by most studies measuring the energy consumption of AI [7, 19, 20, 27, 30, 33].

We also collected the time taken for generations to complete—the latency—as well as the number of rejected requests, and the number of completed generation requests. Lastly, we chose not to consider the functional validity or quality of the outputs produced by the various models. We discuss this issue at the end of section 7

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 [12].

When assessing the noise of our measurements, we found the coefficient of variation of the total energy consumption of repeated simulations to be, on average, 0.016 (SD = 0.029). We considered this level of standard deviation acceptable and concluded that our measuring setup was stable.

4 Data analysis

In this section, we describe our methodology for analyzing the results we obtained from section 3.

4.1 Simulations analysis

To get an overview of the data, we computed the **mean power consumption** (in Watts) and **total energy consumption** (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

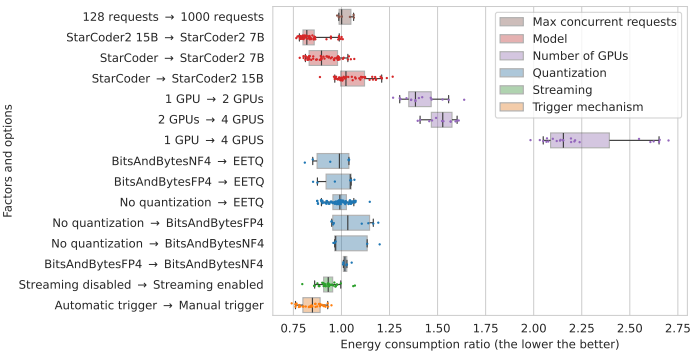

Figure 3: Energy impact ratio when switching one option. A ratio of 1 means no change, a ratio of 2 means the energy consumption doubled. Points correspond to the ratio in energy when comparing neighboring configurations.

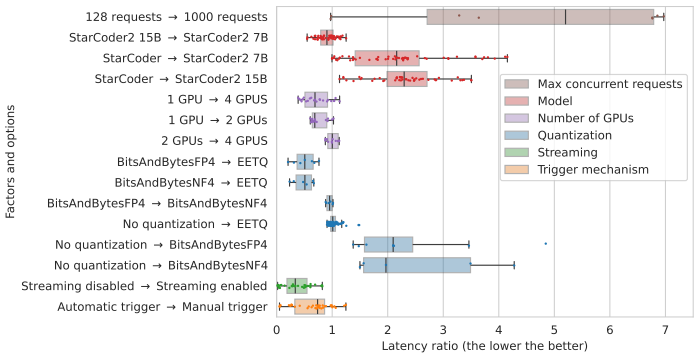

Figure 4: Latency impact ratio when switching one option. A ratio of 1 means no change, a ratio of 2 means the latency doubled. Points correspond to the ratio in latency when comparing neighboring configurations.

allocates an equal share of the server's power 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 was considered (see subsection 3.1).

Server saturation. When the server receives more requests than it can handle, it may reject them or put them in a waiting queue, depending on its 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. These saturation threshold was set arbitrarily as searching for ideal thresholds is out of the scope of this paper.

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 ensures that observed differences can be attributed to this factor.

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, using edit distance, indicates whether a generation is still present in the code [1].

In Figure 2, 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 as many development traces as possible. Thus, we chose to reduce the chances of the participants finishing early, thereby increasing the chances of the participants 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 Results

This section summarises our experiment's key observations and answers to RQs. Complete results are included in the replication package.

5.1 How do a developer's characteristics affect their use 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 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 determine 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 1. To reduce the Type I error rate (false positives), we also employed 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 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 Copilot compared to the students. The students tend to accept the suggestions made by Copilot twice as much as the professional developers. We make the original data we used available in our replication package [5].

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 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

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

In this section, we report on our findings on the different factors and their impacts. In Figure 3 and Figure 4, we depict the impact of switching from one option to another on energy consumption and request latency, respectively. The impacts were calculated using the pairwise comparison method described in subsection 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, 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, the average latency reaches 16 seconds with 75 concurrent developers, and increases exponentially from that point with each developer added (*e.g.*, 50 seconds at 100 developers, 210 seconds at 150 developers). 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 from the same user 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;

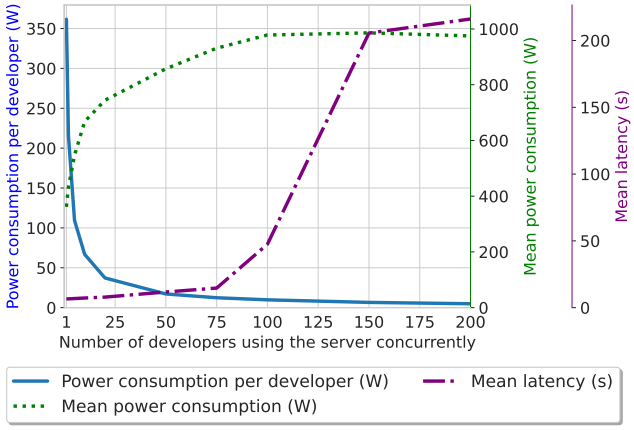

Figure 5: Evolution of the average power consumption and the latency depending on the number of developers, for the following configuration: StarCoder2-7B; no quantization; no streaming; manual trigger; maximum 1000 concurrent requests; 4 GPUs. The latency and power consumption are superposed in order to easily visualize how they interact when the number of developers increases.

no quantization; no streaming automatic trigger; maximum 1000 concurrent requests; 2 GPUs; 5 developers], 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 Copilot varied, 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 strongly impact the energy consumption and latency of code assistants. Increasing the number of concurrent developers improves energy efficiency, but risks server saturation. Streaming requests and manually triggering generations both strongly reduce energy consumption and latency. Quantization methods and the choice of LLM also affect latency, 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.

5.3 RQ2: How many generation requests made by GitHub Copilot are actually useful?

Figure 6 illustrates the final state of generation requests sent by GitHub Copilot during our experiment. Out of the 9,634 generation requests made by GitHub Copilot with the 20 participants,

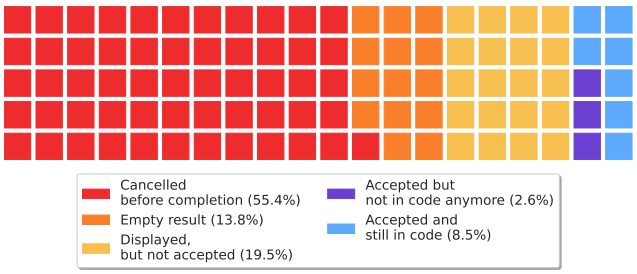

Figure 6: Percentage of requests sent by GitHub Copilot depending on their completion state. Red-tinted categories represent requests that did not benefit the user. Blue-tinted categories represent requests that benefited the user.

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 results reveal that the majority of generation requests made by GitHub Copilot were canceled. A manual analysis of the GitHub Copilot logs revealed that the cancellations mostly happened because the requests were sent while the user was typing. When the user typed a new character, the ongoing request would be cancelled by GitHub Copilot as it was no longer necessary. Then, if applicable, GitHub Copilot would send a new generation request. Empty completions typically occur at the end of a sentence, or before a closing parentheses or brackets. They are due to the GitHub Copilot extension requesting a generation, and the inference server finding nothing relevant to generate.

We hypothesize that 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. However, further research is needed to investigate this hypothesis.

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

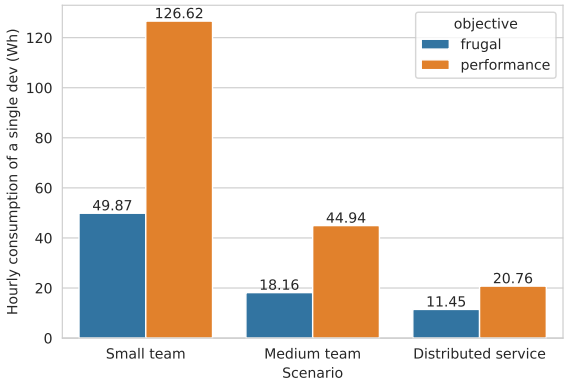

Figure 7: Hourly energy consumption (Wh) per developer based on the objective and the number of developers.

5.4 RQ3: Under different scenarios & objectives, how much energy is used by a developer using a code assistant like GitHub Copilot?

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 replication package. 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 this subset of 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):** Many developers (sharing) requesting many server machines. As the number of servers is considered infinitely scalable by the service provider, it aims to maximize the number of developers per machine while not saturating the servers. We report on the impact of a single almost saturated machine in this scenario.

We considered two objectives that a development team may aim for:

- **Performance:** To minimize latency, generation requests are triggered automatically, and streaming is enabled. This is also the default triggering and streaming mode of GitHub Copilot [1].
- **Frugality:** To minimize per-developer energy consumption, generation requests are triggered manually, and the previous ones are not canceled (streaming is disabled), as we expect the developers to wait for their manually triggered generation requests to finish. Here, frugality is meant as in *"Avoiding waste, using the least resources as possible"*.

Table 2 shows the best configurations for each scenario and objective. To better put in perspective the energy usages of the configurations, we also show them in Figure 7. 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 server's high idle power consumption (~270 W from its four GPUs and single 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 consumption by leveraging the continuous batching from the TGI server, aligning with the findings discussed in Section 5.2.

If GitHub Copilot were deployed using a setup similar to ours, under conditions of maximized server utilization, the average power consumption per developer would likely 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 contextualize these figures, we measured the power consumption of the laptops and monitors used by the last two participants in our user study. Both devices were connected to the same power strip, which in turn was connected to a PowerSpy2 watt meter operating with its default settings. The meter recorded the average power consumption every second and transmitted the data via Bluetooth to a secondary computer not involved in the participant's task. Due to technical issues, these measurements were only available for the final two participants.

The average power consumption of the laptop and monitor was 19.7 W (SD = 8.7) and 18 W (SD = 2.6), respectively. The mean and standard deviation of power consumption were calculated by pooling all 1-second measurements across both participants for each device, treating each reading as an independent sample.

Based on our inference setup, where GitHub Copilot's per-developer power consumption is estimated between 11.4 W and 20.8 W (see Table 2), its use would have led to an estimated increase ranging from 30% to 55% in the total power consumption of developers during our study. This result should be taken with caution, as it is very specific to our inference setup and the hardware used by the participants during our study. Moreover, the fact that we measured the consumption of only two participants makes the measurements difficult to generalize.

We observe that all performance configurations use 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 [15, 17], respectively.

> **RQ3**: The energy cost for a developer using a code assistant like GitHub Copilot varies significantly based on its configuration. Small teams with dedicated servers can see high per-developer power 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 latency when using code assistants.

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 2: Optimal configurations for each scenario and objective, and their energy and latency impacts.

6 Discussion & implications

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

On reducing the energy usage. As reported in our results, the configuration choice for a code assistant strongly impacts each developer's power consumption, yet a considerable portion of the energy spent on generations is wasted due to cancellations and user disinterest. Substantial savings could be achieved by requiring users to manually trigger generations, encouraging more deliberate interactions, and improving the request triggering mechanism. For instance, Mozannar et al. [22] proposed a telemetry-driven triggering method, and Barke et al. [6] identified two interaction modes (acceleration and exploration) that could inform adaptive triggering. More broadly, developers should strive to reduce unnecessary generations, while providers can decrease per-user impact by maximizing the number of users per inference server, selecting quantization methods carefully, and choosing models that preserve memory headroom for batching and keep per-request compute manageable.

On trade-offs and system constraints. Our results show that running an inference server for only a few developers is inefficient in both hardware use and energy consumption, while scaling to more developers improves utilization but requires balancing load against acceptable latency. Latency is therefore a key bottleneck for scaling. Moreover, we observed a sharp increase in latency between StarCoder and StarCoder2-15B, which we suspect is related to architecture changes introduced in StarCoder2 [17]. Regarding quantization, we found that it increased latency and had a limited effect on energy consumption in our setup. This suggests that memory savings may be offset by added inference overhead, and motivates further investigation of the causes and of alternative quantization approaches.

On the usefulness of GitHub Copilot's suggestions. In subsection 5.3, we showed that only about a third of the suggestions were accepted, and that roughly 75% of accepted suggestions remained in the code by the end of the study. Future work could leverage AssistantTraces to more accurately quantify how much of the final project was written by the assistant, study individual differences between participants, and replicate these observations on longer tasks and larger projects, ideally complemented with qualitative analyses.

On generalization & broader environmental footprint. The absolute energy consumption observed in our scenarios is specific to our workload, participants, and infrastructure, and should not be directly generalized to other code assistants, including GitHub Copilot. Nevertheless, the main levers we identify are expected to transfer across implementations because request volume and server utilization largely determine operational energy. These levers include maximizing concurrent developers per machine, selecting smaller models when acceptable, using manual triggering, and canceling early suggestions that are not needed anymore. Moreover, we only calculated CO2 emissions attributable to the inference server's energy use. Future studies should incorporate additional impact sources, such as hardware and datacenter infrastructure, potentially through life-cycle analysis, and consider complementary indicators such as resource depletion or water use, which can be as important as CO2 emissions when discussing ICT [31]. Finally, while code assistants can improve productivity [2], we believe that rebound effects [21] stemming from faster development should also be assessed.

7 Threats to validity

External validity Our participants were recruited through word of mouth and mailing lists, which may yield a sample that is not representative of the broader developer population. In addition, the experimental task was intentionally short and self-contained, and did not cover several aspects of realistic software development (for example, project creation, requirements engineering, test writing, and multi-developer activities such as reviews and coordination). These choices may affect absolute interaction metrics (for example, requests per minute, acceptance rates) and, consequently, absolute energy consumption, since energy is largely driven by request volume.

To keep the number of simulations tractable, we restricted the evaluated configuration space to a subset of models (StarCoder family), fixed hardware, and a limited set of inference-server parameters. While we curated configurations to span diverse settings, other untested configurations could yield different optima or alter comparative results.

However, we expect our main comparative findings about configuration choices (for example, smaller models, higher concurrency, manual triggering, and early cancellation) to remain qualitatively robust because they reflect system-level trade-offs that apply across workloads.

Internal validity Although we replay identical traces across configurations, system-level factors may still confound comparisons. Serving behavior can be sensitive to run order and machine state (for example, caching, thermal, and power-management effects), and to non-deterministic scheduling and batching in the inference stack. Moreover, our open-loop simulations prevent developers from adapting their behavior to latency or output quality, which can change the causal pathway that would operate in a live deployment.

Construct validity Two design choices may imperfectly capture real usage. First, we emulated manual triggering by only issuing generation requests that were displayed to the user, which approximates but does not fully reproduce how users decide when to trigger or refrain from triggering suggestions. Second, we simulated developers rather than measuring energy in a closed loop with live users connected to our inference server, which prevents behavioral adaptation to latency or output quality (for example, waiting, retrying, or canceling earlier). Finally, we did not measure the correctness of the generated code, so our configuration comparisons focus on energy and latency only, and do not capture the full trade-off space that determines practical utility.

8 Related Works

Previous research has investigated various aspects of LLMs for code-related tasks, including the security of their suggestions [25, 26, 28], the prevalence of bugs in the generated code [14], how developers interact with them [6, 26, 32] or just the quality and correctness of the code they generate [11, 16, 24, 36]. There have also been efforts to measure the efficiency of LLMs through the creation of benchmarks for comparing them, such as HumanEval [8], MBPP [3], CoderEval [37], APPS [13], CodeXGLUE [18] or ReCode [34]. Xu *et al.* [35] also conducted a comparative evaluation of multiple LLMs for code. Additionally, in the context of LLMs, Vartziotis *et al.* studied the *Green Capacity* of LLMs for code [33]. They quantified the sustainability awareness of ChatGPT, GitHub Copilot and CodeWhisperer, based on the energy consumption and performance of the solutions they generated for Leetcode problems. Coignion *et al.* studied the performance of the code generated by multiple LLMs on Leetcode, and found that the LLMs studied had similar code performance [9]. Luccioni *et al.* measured the inference cost in energy and carbon of multiple LLMs, and found that multi-purpose LLMs are orders of magnitude more expensive than specialized LLMs [19, 20].

Samsi *et al.* benchmarked the inference performance and energy costs of different sizes of LLaMa models on two GPU types [27]. The study provided the words-per-second performance of the LLaMa models, as well as the energy per second, per decoded token, and per response. Multiple factors were studied, such as the dataset used to perform the generations, the batch size, the type of GPU, and the length of the generation. Chakravarty *et al.* analyzed the effect of quantization on the energy efficiency, accuracy, memory usage, and speed of speech recognition models [7].

Our study stands out by addressing the energy consumption of LLM-based code assistants during real-world developer interactions. Unlike previous research that focused on LLM training impacts or isolated inference impacts, we analyze the energy usage of code assistants in a practical setting. Our dataset and methodology allow us to explore various factors influencing energy consumption, such as the number of concurrent developers, offering insights for service providers and end-users to optimize resource usage. By addressing these gaps and highlighting practical implications, our research contributes to Green AI, making it one of the first investigations into the energy consumption of code assistants.

9 Conclusion

This study explored the energy consumption of code assistants powered by LLMs, focusing on GitHub Copilot. 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 latency of the assistants.

Our results reveal that a considerable portion of energy is wasted on suggestions being cancelled or deemed unwanted. Thus, manually triggering the code assistant, rather than relying on automatic suggestions, greatly reduces energy consumption by avoiding unnecessary inference requests. We also found that the number of concurrent developers sharing an inference server has a major impact on per-developer energy consumption. Higher concurrency improves server utilization and reduces energy overhead per user. Consequently, service providers should aim to maximize developer density on inference servers or enable server sharing to improve overall efficiency. The configuration of the inference server plays a significant role in its energy consumption. Using smaller models generally reduces electricity use, while decreasing the number of GPUs can lower energy consumption—though often at the cost of increased latency. In addition, applying quantization techniques can improve both energy efficiency and latency in some cases.

References

[1] [n. d.]. Copilot-Explorer. https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals.html.

[2] 2024. Measuring GitHub Copilot's Impact on Productivity – Communications of the ACM.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs] doi:10.48550/arXiv.2108.07732

[4] Anonymous authors. 2024. AssistantTraces. doi:10.5281/zenodo.11503611

[5] Anonymous authors. 2024. Green My LLM: Studying the Key Factors Affecting the Energy Consumption of Code Assistants - Replication Package. Zenodo. doi:10.5281/zenodo.13167546

[6] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 78:85–78:111. doi:10.1145/3586030

[7] Aditya Chakravarty. 2024. Deep Learning Models in Speech Recognition: Measuring GPU Energy Consumption, Impact of Noise and Model Quantization for Edge Deployment. arXiv:2405.01004 [cs, eess]

[8] 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. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs]

[9] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A Performance Study of LLM-Generated Code on Leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. ACM, Salerno Italy, 79–89. doi:10.1145/3661167.3661221

[10] Tim Dettmers. 2024. TimDettmers/Bitsandbytes.

[11] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2022. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? (2022). doi:10.48550/ARXIV.2210.14699

[12] Ember. 2024. Carbon Intensity of Electricity Generation – Ember and Energy Institute.

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

[14] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 563–575. doi:10.1109/MSR59073.2023.00082

[15] 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. 2023. StarCoder: May the Source Be with You! (2023). arXiv:2305.06161 [cs.CL]

[16] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. 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).

[17] 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. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs] doi:10.48550/arXiv.2402.19173

[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* 1 (Dec. 2021).

[19] Alexandra Sasha Luccioni and Alex Hernandez-Garcia. 2023. Counting Carbon: A Survey of Factors Influencing the Emissions of Machine Learning. arXiv:2302.08476 [cs]

[20] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. 2023. Power Hungry Processing: Watts Driving the Cost of AI Deployment? arXiv:2311.16863 [cs]

[21] Alexandra Sasha Luccioni, Emma Strubell, and Kate Crawford. 2025. From Efficiency Gains to Rebound Effects: The Problem of Jevons' Paradox in AI's Polarized Environmental Debate. arXiv:2501.16548 [cs] doi:10.48550/arXiv.2501.16548

[22] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 9 (March 2024), 10137–10144. doi:10.1609/aaai.v38i9.28878

[23] NetEase-FuXi. 2024. NetEase-FuXi/EETQ.

[24] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, Pittsburgh Pennsylvania, 1–5. doi:10.1145/3524842.3528470

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 754–768. doi:10.1109/SP46214.2022.9833571

[26] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[27] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC58863.2023.10363447

[28] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. (2022). doi:10.48550/ARXIV.2208.09727

[29] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI (Paper). *Commun. ACM* 63, 12 (Nov. 2020), 54–63. doi:10.1145/3381831

[30] Raghavendra Selvan, Bob Pepin, Christian Igel, Gabrielle Samuel, and Erik B. Dam. 2024. Equity through Access: A Case for Small-scale Deep Learning. arXiv:2403.12562 [cs, stat] doi:10.48550/arXiv.2403.12562

[31] Thibault Simon, Pierre Rust, Romain Rouvoy, and Joël Penhoat. 2023. Uncovering the Environmental Impact of Software Life Cycle. In *2023 International Conference on ICT for Sustainability (ICT4S)*. 176–187. doi:10.1109/ICT4S58814.2023.00026

[32] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. 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*. ACM, New Orleans LA USA, 1–7. doi:10.1145/3491101.3519665

[33] Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. 2024. 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)*. Association for Computing Machinery, New York, NY, USA, 30–37. doi:10.1145/3643795.3648394

[34] 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. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 13818–13843. doi:10.18653/v1/2023.acl-long.773

[35] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3520312.3534862

[36] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, Singapore Singapore, 62–71. doi:10.1145/3558489.3559072

[37] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–12. doi:10.1145/3597503.3623316