

Green AI: Cost of LLM-Based Code Completion

Negar Alizadeh ✉ 

Utrecht University, Utrecht, The Netherlands

Nishant Saurabh ✉ 

Utrecht University, Utrecht, The Netherlands

Fernando Castor ✉ 

University of Twente, Enschede, The Netherlands

Abstract

Background. Code completion is one of the most widely used applications of large language models (LLMs) in software development. In addition to proprietary coding assistants, powerful open-weight LLMs are increasingly adopted for locally deployed code completion systems, partly motivated by privacy concerns. Despite advances in LLM accuracy, the energy cost of inference in code completion tasks remains underexplored, particularly under large-context workloads and across programming languages.

Aims. This study investigates the trade-off between accuracy and energy consumption in LLM-based code completion and analyzes how workload characteristics, context size, and model scale influence inference energy usage.

Method. We evaluate 25 open-weight LLMs on two complementary code completion workloads: repository-level next-line completion (left-to-right prediction) with varying context sizes on the RepoBench dataset, and fill-in-the-middle (FIM) code completion across Python, Java, and Rust on the McEval dataset. We further analyze the influence of input tokens, output tokens, model size, and their interactions on energy consumption using correlation analysis and cluster-robust linear regression models.

Results. Our findings show that the dominant drivers of energy consumption depend strongly on the structure of the completion task. In RepoBench, energy consumption is primarily influenced by input context size and its interaction with model scale, whereas in McEval, output generation and its interaction with active parameter count become the dominant factors. We further observe that output generation is substantially more energy-intensive per token than prompt processing. Across both benchmarks, smaller and heavily quantized models frequently achieve Pareto-optimal trade-offs, often providing accuracy comparable to larger FP16 models while consuming substantially less energy.

Conclusions. The energy behavior of LLM inference in software engineering tasks depends strongly on workload structure, context length, and model scale. Our results suggest that increasing model size or context length does not necessarily lead to proportionally better completion quality, while quantization can substantially improve energy efficiency with limited accuracy degradation. These findings contribute toward more energy-aware deployment strategies for sustainable AI-assisted software development.

2012 ACM Subject Classification Software and its engineering → Software creation and management

Keywords and phrases Green AI, Green LLMs, Code Completion, Coding Assistant, Energy Efficiency, Trade-Offs, Software Development, Model Quantization

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.42

Category Technical Track Paper

Supplementary Material

Software (Replication Package and Appendices): <https://doi.org/10.5281/zenodo.21971227> [2]



© Negar Alizadeh, Nishant Saurabh, and Fernando Castor;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 42; pp. 42:1–42:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Since the emergence of large language models (LLMs) and their rapid advancement in software development tasks, these models have become increasingly popular among software companies and individual programmers, with 84% of developers using or planning to use AI tools [51]. Furthermore, numerous IDEs now incorporate LLM-based code completion tools (e.g., GitHub Copilot, JetBrains AI Assistant, Gemini Code Assist, Claude Code, and Tabnine) and many AI-powered coding environments such as Cursor and Devin are emerging.

LLMs are known to be energy-intensive. Prior work has examined the energy consumption of language models [37, 45] with earlier work focusing mostly on the training phase [37, 38] and more recent papers also investigating the costs of inference [1, 18, 47, 56]. Training is orders of magnitude more expensive, but inference grows with scale and can represent the majority of the energy cost of LLMs for long-term energy use. Prior measurements report that approximately three-fifths of ML energy use is attributed to inference and two-fifths to training [45]. Despite this, most previous work has focused on natural language tasks, with few studies investigating the energy efficiency of LLMs in software development. Existing studies of inference energy indicate that energy usage varies across tasks [3, 38]. This highlights the importance of evaluating LLM performance and energy efficiency not only for natural language tasks but also for other domains such as programming. Even within coding tasks, performance and energy consumption may vary across programming languages due to differences in syntax complexity, ecosystem maturity, and training data availability. For example, Python’s syntax is less verbose than Java’s, with the former being dynamically typed and the latter statically typed. Furthermore, Rust has a unique type system among mainstream programming languages and comparatively less training data.

This study investigates the energy use of LLMs in code completion tasks. Previous studies on LLM inference energy focused primarily on output-intensive tasks, such as code generation and bug fixing [3], and input tokens are often considered relatively inexpensive in comparison [19, 46, 50]. However, in long-context software engineering workloads such as repository-level code completion, processing large input contexts can become a major energy bottleneck. In scenarios where input contexts consist of thousands of tokens while the output contains only a few lines, the relative contribution of input processing becomes significantly larger. This study aims to quantify the relative impact of input and output tokens across different code completion workloads. We focus our investigation on locally-deployable, open weight models. The ability to run these models locally enables us to obtain precise energy measurement while controlling for interference from external factors. The architecture of smaller models often mirrors that of larger ones and results are to some extent transferrable.

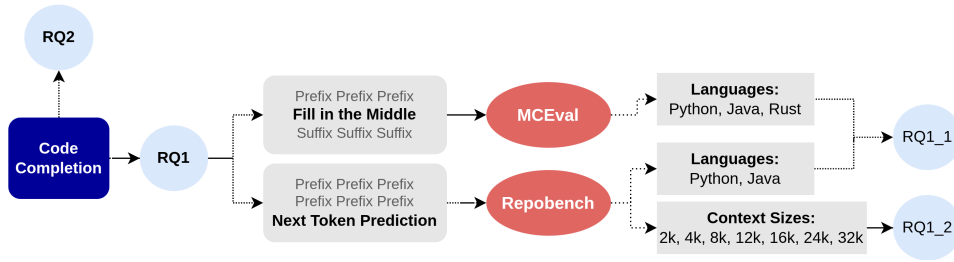
Our study examines two real-world code completion scenarios: (i) **Repository-level next-line completion**, where cross-file context is also included in the prompt; and (ii) **Infill code completion**, where the model fills in missing code using both preceding and succeeding context. We further investigate LLM performance and energy consumption across multiple programming languages, namely Python, Java, and Rust, with the goal of uncovering deeper insights into the energy-performance dynamics of LLMs in realistic coding tasks. We address the following research questions, aiming to contribute to more energy-aware development and deployment strategies for AI-assisted programming tools:

RQ1 What is the trade-off between energy consumption and accuracy in code completion?

RQ1.1 How does the trade-off differ across programming languages?

RQ1.2 How does this trade-off vary across different context sizes?

RQ2 What is the relative influence of input and output token counts on total energy consumption, and how do these drivers interact with model scale?



■ **Figure 1** Schematic representation of the study.

2 Research Method

Figure 1 presents a schematic outline of our study along two directions, corresponding to the two primary categories of code completion [20]: Left-to-Right (L2R) prediction (aka. next token prediction) and Fill-in-the-Middle (FIM) prediction (aka. infilling).

Left-to-Right (L2R) completion refers to the traditional autoregressive setting (one token at a time) in which the model continues the code solely based on the preceding context. This paradigm is commonly used in repository-level completion tasks, where the model may receive large amounts of contextual information from multiple files, functions, and project-wide dependencies. As a result, L2R completion often involves long-context workloads that require processing thousands of input tokens before generating the prediction. In contrast, Fill-in-the-Middle (FIM) completion predicts missing code between a given prefix and suffix. This setting better reflects interactive editing scenarios in modern IDEs, where developers modify or insert code within existing files rather than only appending new content at the end. Our experimental setup is specifically designed to address each category using the appropriate methodology, as described in the remainder of this section.

2.1 Execution Environment

To run LLMs locally, an LLM inference engine is required. Several open-source LLM inference engines have recently emerged to enable local deployment and inference of open-weight LLMs, including PrivateGPT (57.4K GitHub stars), GPT4All (77.4K), llamafile (25.5K), vLLM (87.6K), Ollama (177K) at the time of this study. We selected Ollama (v0.13.0) due to its popularity, as reflected by its GitHub star metric. It is particularly useful for its support of multiple quantization levels, enabling efficient execution on commodity hardware. Furthermore, its seamless model-switching capability and extensive API features make it well-suited for exploration and reproducible experiments. Additionally, its strong adoption among developers, as reported in the 2025 Stack Overflow Developer Survey [51], further reinforces the external validity of our tool selection.

To monitor GPU utilization and **to measure power consumption**, as suggested in previous studies [3, 4, 6, 9], we use the Python interface (nvidia-ml-py v13.590.48) for the NVIDIA Management Library (NVML) [44]. For every running model, power usage is sampled every 100ms, and total energy consumption is estimated as the arithmetic mean of the recorded power samples multiplied by the elapsed execution time. While higher sampling frequencies yield a more detailed power profile, we adopt a 10Hz sampling rate, consistent with prior work [3], to avoid the runtime overhead associated with more frequent measurements. The reported power values represent net consumption, computed by subtracting the idle power of the GPU from the measured power. To measure GPU idle power, we executed the

`nvidia-smi` [43] command for 340 seconds, collecting 3400 samples at 100 ms intervals. 3400 collected observations provide a 98% confidence level that the true power consumption lies within $\pm 2\%$ of the observed values. The exact command we have used is provided in the replication package [2]. The mean measured idle power of the GPU was 46.82 W.

All experiments were conducted on a cluster equipped with an NVIDIA A100 80GB PCIe GPU [42] configured in “High Performance” mode, two AMD 7313 CPUs running at 3 GHz with 1 TB of system memory and 32 MB cache, and AlmaLinux 8.10 (64-bit) operating system configured with the “Performance” CPU governor. Each experiment was repeated three times to mitigate transient fluctuations, and the reported values represent the average across the three runs. No other jobs were allowed to run on the GPU during the experiments.

2.2 Dataset

Repobench [35] is designed for repository-level auto code completion. We used the code completion subset, which focuses on next-line prediction using three distinct settings: Cross-File-First simulates predicting code that hasn’t yet appeared in the current file. This is the hardest task that requires LLMs to know about the external files. In Cross-File-Random cases, a random and non-first occurrence of a cross-file line is masked, so the code has already been partially introduced in the file, allowing the LLMs to leverage prior in-file context. Finally, the In-File split contains cases within a single file, where an in-file line unrelated to cross-file modules is masked. Compared to other repository-level benchmarks for code completion [12, 17, 32, 33, 40, 55], Repobench supports a wide range of context sizes, varying from 2000 to 128000 input tokens. This makes it particularly suitable for our study, as it allows us to better investigate the effect of input length on energy consumption. Considering the limitations imposed by the context length of the LLMs under examination, we were only able to carry out experiments on the data with a maximum of 32000 as their context limit. Then, for each context level, 100 samples were extracted for each split (300), resulting in 2100 code completion problems.

McEval [10] is a multilingual code evaluation benchmark, covering 40 languages. Its code completion task can be categorized into single-line completion, multi-line completion, and span completion. In single-line completion, only one line of code is masked, whereas in the other two settings multiple lines are masked. In multi-line completion, the masked lines are dispersed throughout the code, while in span completion they appear consecutively as a single block. To evaluate LLM performance for FIM purposes across different programming languages, we selected Python because it is the most widely used language for machine learning development, Java as a representative of popular compiled languages, and Rust as a less common but still mainstream language, reflecting scenarios where training data is relatively limited. The selected McEval subsets contain 340 Python, 355 Java, and 343 Rust completion tasks. While other languages such as C# or C++, could also be included, we deliberately limited our selection to these three for practical reasons. This choice can be acknowledged as a potential threat to validity.

2.3 Model Selection

To ensure diversity in both model scale and architecture, we collected models ranging from approximately 1B up to 20B parameters, focusing on model sizes practical for local deployment and inference. The list of models under evaluation covers multiple parameter ranges, including models below 4B parameters, several models in the 6B–8B range, and larger models from 13B to 20B parameters. To construct this set, we selected model families based on their

availability in GGUF format and their popularity on the Ollama platform, as indicated by pull counts. Two models (Granite 3.3 8B and Granite 3.3 2B) were not available on Ollama in full-precision and Q8 formats; therefore, we sourced their GGUF files directly from the official IBM Granite repository on Hugging Face [27].

We also included newer models with efficiency-driven approaches, as they are relevant to the energy-efficiency goals of this study. In particular, we included both dense and Mixture-of-Experts (MoE) architectures. Unlike dense models, where all parameters participate during inference, MoE models activate only a subset of their parameters for each token, potentially reducing inference cost while maintaining strong performance [22, 48]. Therefore, throughout this study, we report the number of active parameters for MoE models rather than the total parameter count. To investigate the trade-off between numerical precision, accuracy, and energy consumption, we evaluate models under multiple quantization levels, including FP16, Q8, and Q4, representing 16-bit floating-point, 8-bit, and 4-bit weight precision, respectively. **Quantization** reduces the numerical precision used to represent model weights, lowering memory usage and computational overhead during inference [15, 21, 31].

In line with our focus on code completion, we only include general-purpose and coding-specific models. For experiments using the RepoBench dataset, base versions of the models are required, whereas instruction-tuned variants are needed for the McEval dataset.

Base LLMs are pretrained models optimized for next-token prediction, whereas instruction-tuned are additionally trained to follow natural language instructions. Following the original benchmark papers, RepoBench [35] requires base models because the benchmark is designed to evaluate the completion of the next-line code at the repository-level in an autoregressive setting, where the model must continue the code directly from the provided context without relying on the following behavior of the instruction. In contrast, McEval [10] follows a prompt-based fill-in-the-middle setting, making instruction-tuned variants more appropriate for the evaluation scenario. Moreover, in the RepoBench setting, context length is particularly important because models with larger context windows can better capture long-range dependencies. We also considered the reputation and maturity of model developers, as models from established AI organizations tend to be better maintained and widely adopted. A complete overview of the evaluated models is provided in the online appendix [2].

2.4 Performance metrics

There are numerous metrics to evaluate the correctness of the code generated by LLMs. This section focuses on the key metrics frequently used to assess LLM performance. Given that this research cannot explore all possible metrics, it focuses on the metrics most frequently used to determine the performance of LLMs used for code completion. A recent systematic literature review of LLM-based code completion [26] identified Top- k Accuracy, Levenshtein Edit Similarity (ES), Exact Match (EM), Mean Reciprocal Rank (MRR), and BLEU as the most commonly used evaluation metrics. For the RepoBench dataset, to ensure comparability and reproducibility, we use the same evaluation metrics as defined in the original RepoBench paper [35]: Normalized Edit Similarity and Exact Match. Although these metrics may appear overly strict, since functionally correct code can differ in naming conventions, they are appropriate in a repository-level setting. In such scenarios, the model is provided with extensive contextual information, including cross-file references and project-wide naming patterns. Therefore, we expect the LLM to generate a piece of code that is not only functionally correct but also adheres to the intended naming conventions for variables, functions, and classes. Moreover, even when the generated code is not fully correct, high textual similarity can still be practically useful in software development, as developers may adapt and refine the generated code instead of writing it entirely from scratch.

For the generated code in the McEval dataset, we utilized the Docker container prepared by the dataset authors to compute Pass@1 accuracy based on the provided test cases. This metric corresponds to the Plausible@1 criterion recommended in recent LLM evaluation guidelines [7]. Since McEval consists of relatively small and self-contained examples, it is feasible to execute the generated code and verify correctness through automated testing. All necessary environments are preinstalled in the Docker image, and the correctness is assessed by executing the generated code against the provided tests [10]. This approach is better suited for this use case, where no cross-file repository context is involved, ensuring that the evaluation reflects functional correctness rather than syntactic similarity.

In contrast, applying execution-based evaluation to RepoBench would require resolving repository dependencies and building complete software systems, which would substantially limit the scale and reproducibility of this study. Therefore, for RepoBench, we rely on similarity-based metrics instead. This approach is better suited for the repository-level setting while still allowing us to evaluate how well the generated code aligns with the expected implementation and project-wide conventions.

2.5 Experimental Procedure

To ensure consistency, we used identical hyperparameters for all models within each dataset. The temperature and top-p values were set to 0.2 and 0.95, respectively, as prior studies [11,35] suggest these settings yield better performance in code generation tasks.

For **RepoBench**, the maximum number of output tokens was set to 128, following the configuration of the original study. Only the context size was varied to accommodate different input lengths. To construct the prompts, we used the *construct_prompt* function provided with the dataset. For **McEval**, we first measured the length of the longest canonical solution for each language (Python: 1227 tokens, Java: 1274 tokens, Rust: 1017 tokens). We then set the maximum number of output tokens to 1300 to ensure coverage of the longest solutions across all three languages. The context size remained at its default value of 2000 tokens. As for the prompt, we used the *instruction* column provided in the dataset.

3 Results

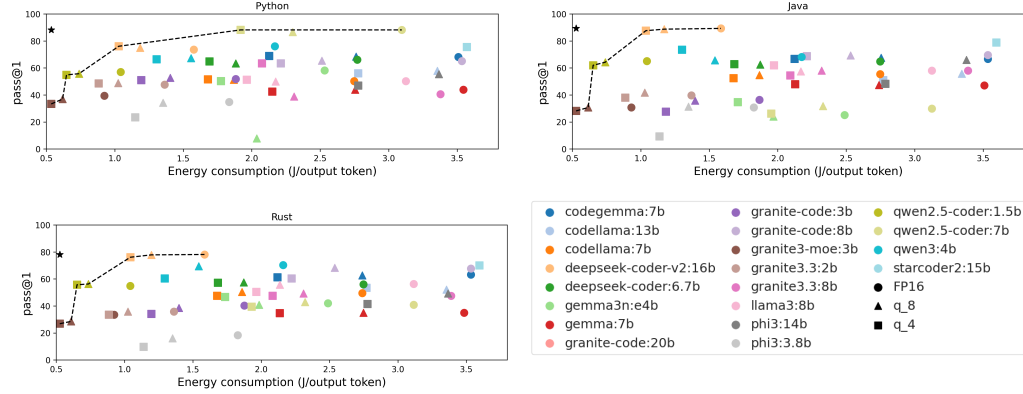
In this section, we present our findings following the order of the research questions.

3.1 RQ1 – What is the trade-off between energy consumption and accuracy in code completion?

To address this RQ, we first collected the energy consumption of all LLMs while executing the complete workload. For McEval, we computed the mean Pass@1 score and total energy consumption across the complete workload of each language subset. Since both the size and the content of the workloads differ across subsets, we report the energy consumed per generated token (J/token) to enable a fair comparison between models. The same consideration applies to RepoBench, where the Python and Java subsets also contain different problems. Similar to McEval, we report aggregated energy consumption values together with the mean Edit Similarity scores. The results are presented in Tables 1 and 2. Due to page limitations, only the FP16 results are included in the main paper, while the results for 8-bit and 4-bit quantizations are provided in the online appendix [2]. To analyze the trade-off between accuracy and energy consumption, we employ the Pareto frontier, a multi-objective optimization method that identifies nondominated configurations across competing objectives.

■ **Table 1** Results of inference for instruction-tuned models in Python, Java and Rust on McEval. Total energy (E) is in Wh, elapsed time (T) is in seconds, J/t indicates the energy per generated token and P@1 indicates pass@1. Note that the workload across languages is not the same.

Model	Python				Java				Rust			
	T	E	J/t	P@1	T	E	J/t	P@1	T	E	J/t	P@1
codegemma:7b	997.86	66.32	3.51	68.18	898.07	59.42	3.53	66.76	898.13	59.41	3.53	63.26
codellama:13b	1288.67	88.94	4.98	56.97	1293.52	89.56	4.98	54.65	1357.33	94.07	4.98	53.35
codellama:7b	839.38	57.79	2.75	50.30	997.59	68.85	2.75	55.49	934.90	64.58	2.74	49.56
deepseek-coder-v2:16b	582.85	30.98	1.58	73.64	603.88	32.54	1.58	89.30	596.67	31.48	1.58	78.13
deepseek-coder:1.3b	847.85	48.53	0.74	5.76	805.12	46.16	0.73	7.61	883.53	50.69	0.73	3.79
deepseek-coder:6.7b	1264.01	87.28	2.77	66.06	1394.03	96.45	2.75	64.79	1254.37	86.60	2.75	55.98
deepseek-llm:7b	773.20	51.26	2.76	35.15	831.19	55.43	2.76	33.24	851.87	56.89	2.75	19.53
gemma3n:e4b-it	993.42	38.42	2.53	58.18	1052.29	41.54	2.49	25.07	1030.64	41.94	2.49	41.98
gemma:2b	398.66	24.54	1.25	25.15	383.24	23.45	1.25	25.92	346.91	20.97	1.25	20.12
gemma:7b	1850.42	125.69	3.55	43.94	1718.55	116.72	3.51	47.04	1744.44	118.70	3.48	34.98
granite-code:20b	2416.59	166.91	7.75	64.24	2006.78	138.60	7.63	65.92	2003.26	138.45	7.63	63.26
granite-code:3b	605.66	36.32	1.88	51.82	455.01	27.07	1.87	36.34	479.93	28.42	1.88	40.23
granite-code:8b	940.17	64.20	3.53	65.15	813.09	55.48	3.53	69.58	800.20	54.54	3.53	67.64
granite3-moe:3b	663.20	25.38	0.92	39.39	446.74	16.95	0.93	30.70	644.38	24.49	0.92	33.53
granite3.3:2b	454.22	27.30	1.36	47.58	469.16	27.99	1.37	39.72	516.69	30.90	1.36	35.86
granite3.3:8b	707.74	47.96	3.38	40.61	820.77	55.99	3.39	58.03	829.16	56.49	3.39	47.52
llama3:8b	612.80	38.92	3.12	50.30	697.16	44.67	3.12	58.03	706.18	45.47	3.12	56.27
mistral:7b	985.96	68.04	2.85	44.54	1070.87	73.95	2.84	38.59	972.74	67.09	2.85	37.61
phi3:14b	1347.70	93.26	5.13	54.85	1375.09	95.39	5.13	65.92	1323.67	91.80	5.13	50.73
phi3:3.8b	694.29	47.46	1.83	34.85	857.20	58.86	1.82	30.70	844.41	57.92	1.83	18.37
qwen2.5-coder:1.5b	373.94	16.64	1.04	56.97	345.69	15.47	1.04	65.07	352.92	15.40	1.04	54.81
qwen2.5-coder:7b	493.23	31.16	3.10	88.18	429.44	26.60	3.12	29.86	460.24	28.83	3.12	40.82
qwen3:4b	520.68	31.44	2.17	76.06	465.15	28.00	2.17	68.17	531.88	32.49	2.16	70.26
starcoder2:15b	1503.64	103.40	6.02	76.36	1516.90	104.56	6.06	81.97	1571.24	108.33	6.10	69.68



■ **Figure 2** Relation between accuracy and energy consumption (Joule/token) for different languages in McEval. The black dashed line indicates the Pareto frontier and ★ denotes the ideal point.

3.1.1 RQ1.1 – How does the energy-accuracy trade-off vary across programming languages?

To further investigate whether the observed trade-offs differ across programming languages, we performed statistical analyses on the achieved accuracy scores. In the McEval dataset (Table 1), applying Friedman’s test revealed a significant effect of programming language on Pass@1 ($\chi^2 = 40.36$, $p < 0.001$). Post-hoc Wilcoxon signed-rank tests with Bonferroni correction ($\alpha = 0.0167$) showed no significant difference between Python and Java ($p = 0.90$). However, both Python and Java significantly outperformed Rust ($p < 0.001$), with median paired differences of 5.75 and 5.01, respectively. We did not perform similar tests for energy and time because the workloads differ across programming languages.

■ **Table 2** Results of inference for base models in Python and Java on RepoBench. Total energy is reported in Wh, elapsed time in seconds, J/t indicates the energy per generated token and ES stands for Edit Similarity. The full list of models and Exact Match results is provided in the online appendix [2]. Note that the workload across context sizes and languages is not the same.

ctx	Model name	Python				Java			
		Time	Energy	J/t	ES	Time	Energy	J/t	ES
2k	codegemma:7b	624.62	40.16	3.62	56.08	640.51	41.26	3.62	75.25
	codellama:13b	885.98	60.45	5.10	47.86	894.54	61.08	5.10	67.36
	codellama:7b	506.89	34.16	2.84	48.12	506.04	34.05	2.83	70.58
	deepseek-coder-v2:16b	432.11	19.88	1.54	44.97	403.57	18.72	1.56	64.36
	deepseek-coder:1.3b	153.59	8.68	0.84	42.51	153.84	8.52	0.83	58.55
	deepseek-coder:6.7b	450.25	29.96	2.85	45.87	434.74	28.66	2.83	67.06
	gemma:2b	256.03	14.67	1.30	45.55	254.84	14.65	1.29	69.32
	gemma:7b	600.35	38.60	3.58	51.04	616.57	39.67	3.57	72.95
	granite-code:20b	1015.81	69.11	8.26	27.89	1253.24	85.28	8.00	52.57
	granite-code:3b	250.75	14.28	1.98	20.38	339.74	19.51	1.96	47.76
	granite-code:8b	464.86	30.63	3.57	27.61	528.69	35.02	3.55	48.16
	llama3:8b	508.80	31.74	3.17	47.96	543.33	34.02	3.16	68.59
	qwen2.5-coder:1.5b	261.24	11.55	1.11	40.87	260.97	11.38	1.11	63.04
	qwen2.5-coder:7b	512.73	32.58	3.17	47.94	461.37	28.87	3.15	68.52
	qwen3:4b	340.74	19.94	2.22	44.13	335.78	19.55	2.22	57.79
4k	codegemma:7b	733.47	47.70	3.80	52.95	713.41	46.23	3.79	74.33
	codellama:13b	1074.49	73.46	5.35	47.11	1031.47	70.56	5.35	62.29
	codellama:7b	624.59	42.26	3.01	47.88	597.12	40.31	3.00	65.44
	deepseek-coder-v2:16b	639.20	30.01	1.75	44.88	528.30	24.55	1.66	62.58
	deepseek-coder:1.3b	209.20	11.98	0.94	38.95	198.94	11.30	0.93	53.49
	deepseek-coder:6.7b	596.32	39.85	3.03	45.37	566.51	37.83	3.01	65.70
	gemma:2b	301.12	17.38	1.42	45.39	289.61	16.52	1.39	67.16
	gemma:7b	717.37	46.52	3.76	53.27	703.91	45.55	3.73	72.83
	granite-code:20b	1386.40	94.53	8.77	28.22	1469.76	100.45	8.70	52.81
	granite-code:8b	602.68	40.15	3.69	27.76	626.71	41.72	3.69	50.06
	llama3:8b	614.40	39.07	3.30	47.90	614.45	38.87	3.29	66.84
	qwen2.5-coder:1.5b	308.24	13.43	1.14	40.51	301.51	12.80	1.11	60.81
	qwen2.5-coder:7b	602.72	38.63	3.25	46.05	539.13	34.09	3.23	66.10
	qwen3:4b	409.62	24.32	2.29	42.26	383.46	22.56	2.26	59.27
8k	codegemma:7b	901.72	59.13	4.08	56.14	870.22	56.78	4.08	70.96
	codellama:13b	1404.75	96.13	5.81	45.79	1387.39	94.89	5.88	61.77
	codellama:7b	841.48	56.94	3.32	51.75	826.93	55.89	3.38	66.08
	deepseek-coder-v2:16b	848.72	39.24	1.76	45.67	790.12	36.68	1.83	59.28
	deepseek-coder:1.3b	297.69	17.44	1.12	37.32	296.44	17.37	1.12	48.12
	deepseek-coder:6.7b	826.18	55.51	3.37	47.08	822.59	55.14	3.41	61.99
	gemma:2b	358.90	20.68	1.56	30.56	353.80	20.16	1.56	32.11
	gemma:7b	892.09	58.43	4.05	53.30	868.27	56.53	4.07	69.27
	granite-code:20b	1958.13	133.17	10.00	31.60	2021.28	137.26	10.00	49.71
	llama3:8b	765.83	49.11	3.53	47.34	750.91	48.03	3.50	63.99
	qwen2.5-coder:1.5b	359.49	16.56	1.26	38.44	361.44	16.15	1.25	58.84
	qwen2.5-coder:7b	719.25	46.34	3.40	47.24	677.95	43.23	3.42	60.56
	qwen3:4b	496.04	29.59	2.38	44.53	474.78	28.03	2.39	58.72
12k	codellama:13b	1876.01	128.58	6.54	43.60	1910.71	131.02	6.62	58.72
	codellama:7b	1146.49	77.94	3.86	46.46	1170.43	79.50	3.92	62.24
	deepseek-coder-v2:16b	1113.96	52.34	2.10	42.22	1091.98	51.59	2.15	54.78
	deepseek-coder:1.3b	433.49	25.99	1.40	34.51	448.06	26.80	1.42	42.56
	deepseek-coder:6.7b	1184.34	79.90	3.92	44.26	1215.03	82.01	3.98	56.88
	qwen2.5-coder:1.5b	433.16	20.54	1.42	38.30	439.49	20.70	1.44	54.30
	qwen2.5-coder:7b	857.51	55.53	3.62	45.54	855.15	55.21	3.66	56.04
	qwen3:4b	562.17	34.01	2.50	44.93	579.82	34.46	2.54	54.40
16k	codellama:13b	2357.74	161.85	7.25	41.79	2641.62	181.57	7.35	56.72
	codellama:7b	1511.87	102.78	4.52	44.97	1651.18	112.54	4.46	59.93
	deepseek-coder-v2:16b	1326.67	63.53	2.42	42.76	1459.92	70.92	2.49	50.30
	deepseek-coder:1.3b	600.05	36.04	1.69	32.23	667.24	40.36	1.74	41.98
	deepseek-coder:6.7b	1582.16	106.92	4.46	40.33	1738.78	117.83	4.55	55.11
	qwen2.5-coder:1.5b	488.73	23.27	1.55	37.44	529.41	25.87	1.62	54.84
	qwen2.5-coder:7b	971.66	62.70	3.85	43.74	1038.28	67.33	3.94	54.55
24k	qwen3:4b	637.32	37.94	2.62	45.16	686.45	40.88	2.70	55.23
	deepseek-coder-v2:16b	1921.41	95.33	2.94	40.66	1993.03	98.77	2.90	53.94
	qwen2.5-coder:1.5b	637.91	32.11	1.84	34.66	673.71	34.57	1.88	53.11
	qwen2.5-coder:7b	1275.13	83.22	4.26	44.52	1368.26	89.59	4.27	54.83
32k	qwen3:4b	803.91	48.82	2.88	38.26	848.24	51.55	2.92	52.63
	deepseek-coder-v2:16b	1816.94	91.11	3.41	40.58	2822.69	143.56	3.60	45.90
	qwen2.5-coder:1.5b	679.48	32.80	2.00	35.49	911.62	48.67	2.26	44.92
	qwen2.5-coder:7b	1322.70	85.11	4.59	42.15	1874.55	123.37	4.81	48.11
32k	qwen3:4b	813.82	48.39	3.00	46.41	1083.22	66.65	3.21	46.93

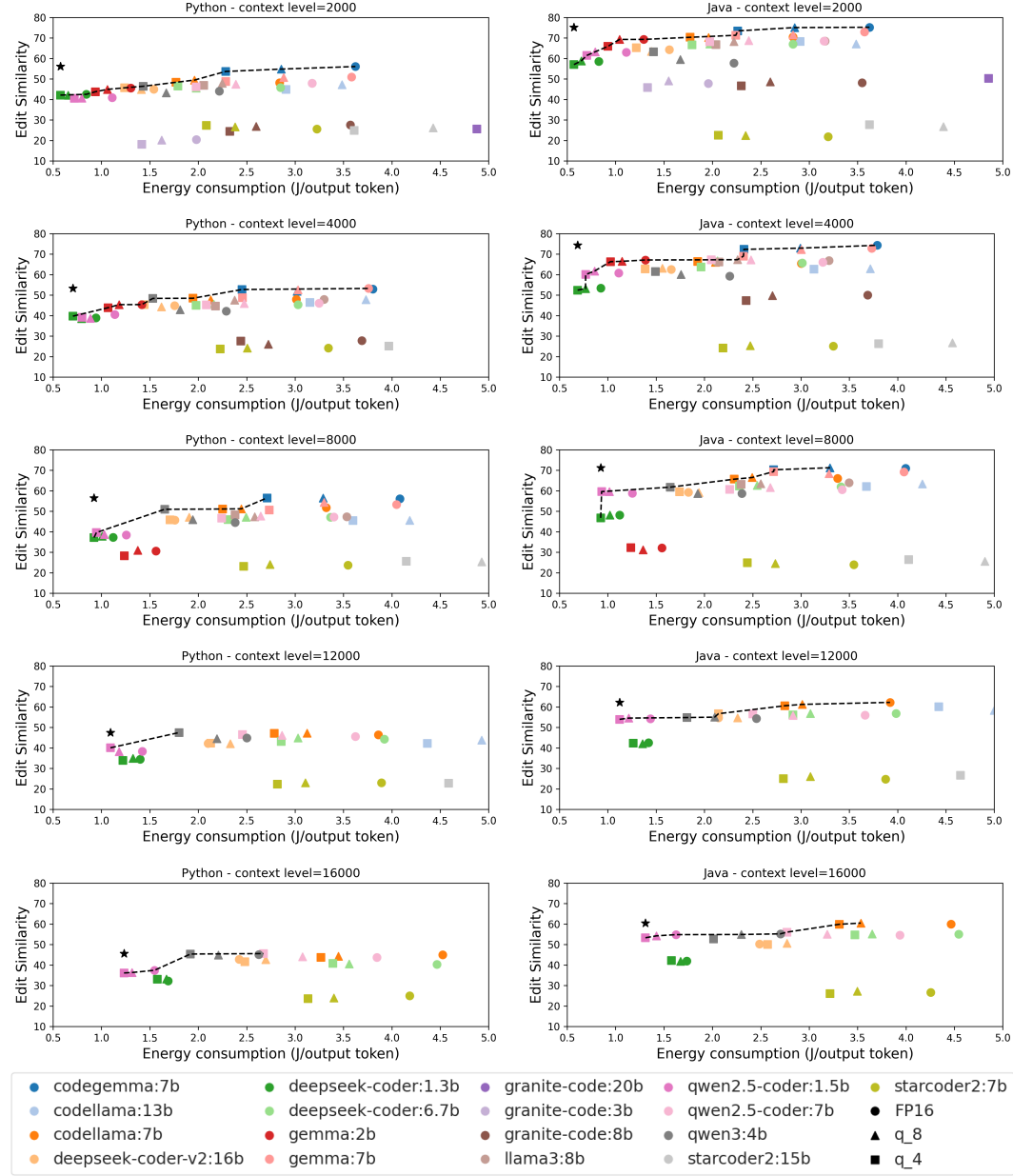


Figure 3 Relation between accuracy and energy per generated token (Joule/token) for different context levels in Repobench dataset. The left column is the results for generating Python code, and the right column depicts the results for generating Java code. The black dashed line indicates the Pareto frontier. The ★ denotes the ideal point. (Data points exceeding the x-axis limit are not displayed.)

In the RepoBench dataset (Table 2), we report Edit Similarity (ES) as a proxy for accuracy. A Wilcoxon signed-rank test reveals a statistically significant difference between Python and Java ($W = 103.0$, $p < 0.001$). Java achieved a higher median ES (56.43) compared to Python (43.02), with a median paired difference of -14.90 . One possible explanation for these findings lies in the characteristics of the programming languages. Python, being dynamically typed, and Java, being statically typed, may influence how LLMs perform under

syntax-based evaluation metrics such as ES. In particular, the stricter and more explicit structure of Java code could partially explain its higher ES scores. However, when evaluated using execution-based metrics such as Pass@1 (as in McEval), the performance of Python and Java becomes comparable. In contrast, Rust is consistently outperformed by both Python and Java. This may be due to the relatively smaller amount of Rust code available in the training corpora of contemporary LLMs.

As shown in Figure 2, in McEval, both quantized variants of the small **Qwen2.5-coder:1.5b** model are located close to the ideal point while still achieving competitive Pass@1 scores across all three languages. This indicates that smaller quantized models can provide a favorable balance between energy efficiency and accuracy. Among the larger models, the 4-bit quantized variant of **Deepseek-coder-v2:16b** stands out as one of the most efficient high-performing models. Despite containing 16B total parameters, this MoE model activates only approximately 4B parameters during inference, allowing it to achieve the highest Pass@1 scores in Java and Rust while remaining on the Pareto frontier. In Python, however, **Qwen2.5-coder:7b-q4** achieves the highest accuracy overall. The improvement over **Deepseek-coder-v2:16b-q4** is approximately 16% while requiring nearly twice as much energy per generated token.

Notably, the Pareto frontiers across languages are largely composed of the same model families, suggesting that the energy-accuracy trade-off remains relatively stable across programming languages. One exception is **Qwen2.5-coder:7b**, which performs particularly well in Python but is less competitive in Java and Rust. Most Pareto-optimal configurations correspond to quantized models. Even when FP16 models appear on the frontier, their quantized counterparts often achieve nearly identical accuracy while consuming substantially less energy per generated token. Several larger FP16 models are dominated by smaller quantized alternatives, indicating that increasing parameter count and precision does not necessarily yield better energy-accuracy trade-offs. Overall, quantized models frequently provide the best balance between accuracy and energy efficiency, with small accuracy degradation relative to their energy savings.

3.1.2 RQ1.2 – How does the energy-accuracy trade-off vary across different context sizes?

In RepoBench, as shown in Table 2, increasing the context size generally leads to a higher energy cost per generated token (J/token) in both Python and Java. For example, in Python, **Qwen2.5-coder:1.5b** increases from 1.11 J/token at a context size of 2000 to 2.00 J/token at 32000, with a similar trend observed in Java. This pattern is reflected in Figure 3, where models gradually shift toward higher J/token values as the context size increases. This trend is further summarized across models in the online appendix [2].

Although the workloads across different context sizes are not identical in terms of both content and size, they are designed to represent repository-level completion tasks of comparable complexity. We found out that increasing the context size does not consistently translate into substantial improvements in ES, suggesting diminishing returns from larger context windows. For some models, such as **Granite-code:20b** and **Gemma:7b**, ES slightly improves as larger contexts are provided. In contrast, for models such as **CodeLlama:7b** and **CodeLlama:13b**, performance gradually decreases with increasing context size. Other models, including **Qwen2.5-coder:1.5b** and **Qwen2.5-coder:7b**, exhibit modest improvements up to a certain context length, after which ES begins to decline.

Notably, the composition of the Pareto frontier remains relatively stable across context sizes, with quantized variants frequently appearing among the Pareto-optimal configurations. This confirms that increasing model precision or context size does not necessarily result

in proportionally better quality also in repository-level code completion. In several cases, quantized variants achieve nearly identical ES compared to their FP16 counterparts while consuming substantially less energy per generated token.

Model rankings were not stable in terms of per-token energy consumption across context sizes. In the models with size lower than 2B range, `Deepseek-coder:1.3b` exhibited lower J/token values than `Qwen2.5-coder:1.5b` up to a context size of 12K. However, at 16K context, `Qwen2.5-coder:1.5b` became more energy-efficient. A similar trend was observed in the 4B range between `Deepseek-coder-v2:16b-lite-base-fp16` and `Qwen3:4b-fp16`, both of which have approximately 4B active parameters. While the DeepSeek model remained more efficient at smaller contexts, `Qwen3:4b-fp16` achieved lower J/token values at 32K context. Likewise, among the 7B models, `Qwen2.5-coder:7b-base-fp16` became more energy-efficient than both `Codellama:7b-code-fp16` and `Deepseek-coder:6.7b-base-fp16` starting from 12K context size. These observations suggest that certain model families may exhibit better scalability under long-context workloads.

Summary. The trade-off between energy consumption and accuracy of LLMs is not always strict. In code completion, it is strongly influenced by the characteristics of the completion task, programming language, and context size. Increasing context size consistently raises energy consumption, while the corresponding accuracy improvements are often limited. Therefore, no single model consistently provides the best trade-off across all tasks, programming languages, and context sizes.

Takeaway. Practitioners should select models based on the programming language and expected context length of their target use case rather than a model’s reported accuracy alone. Furthermore, model providers should report inference energy alongside accuracy.

3.2 RQ2 – What is the relative influence of input and output token counts on total energy consumption, and how do these drivers interact with model scale?

■ **Table 3** Spearman correlations between total energy consumption and the studied predictors, including input tokens, output tokens, active parameter count, and their interaction terms. Bonferroni correction was applied ($\alpha = 0.05/6 = 0.0083$) and all correlations remain significant.

Predictor	RepoBench	McEval
Input Tokens	0.57	0.24
Output Tokens	0.40	0.36
Active Parameter Count	0.60	0.75
Input Tokens $\times$ Active Parameter Count	0.94	0.79
Output Tokens $\times$ Active Parameter Count	0.67	0.92
Input Tokens $\times$ Output Tokens	0.58	0.40

To quantify the drivers of energy consumption, we first analyzed the pairwise relationships between energy usage, token counts (input and output), and active parameter count using correlation analysis. The results are presented in Table 3. We selected input and output token counts because both prompt processing and autoregressive generation contribute directly to inference cost. Active parameter count is a compact proxy for model scale and computational complexity, as larger models consistently consumed more energy than smaller ones under comparable workloads. For McEval, correlation was high for active parameter count and weak for the other two variables. This reflects the characteristics of this dataset: input and

output lengths are short, thus model scale dominates energy use. For RepoBench, output length is short, and input length is very long, so their impact is comparable to the model scale. In this case, energy use is moderately correlated with the three variables.

In addition, we included interaction terms between token counts and active parameter count to capture the intuition that the computational cost of processing or generating tokens depends not only on the number of tokens, but also on the size of the model performing the computation. Moreover, the interaction between input and output tokens was included to account for the fact that autoregressive generation depends on the size of the provided context. This effect is also reflected in the RepoBench Pareto plots (Figure 3), where models consistently require higher energy per generated token as context size increases, despite the output length being capped at 128 tokens. This suggests that generating the same number of output tokens may incur substantially different costs under short and long context workloads. As shown in Table 3, the first two interaction variables exhibit strong correlation with energy consumption across both benchmarks, while the interaction between input and output tokens is moderately correlated. These interaction terms are particularly relevant in our setting, since RepoBench emphasizes long-context processing whereas McEval contains generation-heavy workloads. To further analyze the contribution of these factors, we fitted multiple linear regression models using standardized variables ($\mu = 0, \sigma = 1$). As input token count, output token count, and active parameter count are measured on different scales, we standardized these predictors to enable direct comparison of their regression coefficients. The regression assumptions were reasonably satisfied, and the corresponding diagnostic plots are provided in the online appendix [2]. The general regression model is defined in Equation 1.

$$E = \beta_0 + \beta_1 I + \beta_2 O + \beta_3 P + \beta_4 I \cdot O + \beta_5 I \cdot P + \beta_6 O \cdot P + \beta_7 W + \beta_8 L + \varepsilon \quad (1)$$

where E denotes total energy consumption (Wh), I and O represent the standardized input and output token counts, P is the standardized active parameter count, W denotes bit width (quantization), L represents the programming language, and ε is the error term. The interaction terms capture that the energy cost of generation depends jointly on context size and model scale. To facilitate interpretation of the regression coefficients, the standard deviations (SD) of the predictors are reported in Table 4. The dependent variable is the total energy consumption in Wh. As predictors, we included standardized input tokens, output tokens, and active parameter count, as well as interaction terms between these variables. Bit width and programming language were treated as categorical variables. To account for multiple observations from the same model families, we utilized Cluster-Robust Standard Errors, grouping by model families. This ensures our p -values are not artificially inflated by intra-model correlation. For RepoBench, the statistics and regressions are computed using the common models evaluated across all context sizes up to 16K to ensure consistent model coverage, as only a few models support 24K and 32K contexts. For McEval, all evaluated models are included. The regression results are presented in Table 5.

■ **Table 4** Standard deviations of the standardized predictors used in the regression models. Token statistics represent the aggregated number of tokens across all prompts within each benchmark.

Benchmark	Input Tokens	Output Tokens	Parameter Count
RepoBench	1,316,930.93	2,309.96	4,563.42
McEval	21,161.11	39,171.61	4,781.41

■ **Table 5** Regression coefficients (β), standard errors (SE), statistical significance (p -value), and model fit metrics for RepoBench and McEval. Cluster-robust standard errors are used (clustered by family). Q4 is used as the reference category for bit width, and Java is the reference language in both RepoBench and McEval.

RepoBench					
Predictor	coef (β)	SE	p-value	R^2	Adj. R^2
Intercept	49.285	1.889	<0.001	0.961	0.960
Bit width (Q8)	2.389	1.059	0.024		
Bit width (FP16)	3.008	1.014	0.003		
Language (Python)	1.433	0.535	0.007		
Input Tokens _{std}	22.976	1.431	<0.001		
Output Tokens _{std}	4.902	1.916	0.011		
Active Parameter Count _{std}	26.604	1.497	<0.001		
Input Tokens _{std} $\times$ Output Tokens _{std}	3.978	1.583	0.012		
Input Tokens _{std} $\times$ Active Parameter Count _{std}	9.788	1.328	<0.001		
Output Tokens _{std} $\times$ Active Parameter Count _{std}	1.448	0.987	0.142		
McEval					
Predictor	coef (β)	SE	p-value	R^2	Adj. R^2
Intercept	40.748	3.718	<0.001	0.940	0.937
Bit width (Q8)	5.841	1.232	<0.001		
Bit width (FP16)	21.947	3.027	<0.001		
Language (Python)	-0.030	6.149	0.996		
Language (Rust)	0.058	1.434	0.968		
Input Tokens _{std}	0.427	2.887	0.882		
Output Tokens _{std}	23.496	2.189	<0.001		
Active Parameter Count _{std}	27.259	1.171	<0.001		
Input Tokens _{std} $\times$ Output Tokens _{std}	-0.352	0.209	0.093		
Input Tokens _{std} $\times$ Active Parameter Count _{std}	-0.562	0.394	0.154		
Output Tokens _{std} $\times$ Active Parameter Count _{std}	15.674	1.351	<0.001		

For RepoBench, the regression model achieved a high R^2 (0.961), indicating that the selected predictors explain most observed variation in energy consumption. Input token count and active parameter count exhibit the strongest positive associations with energy usage, with coefficients of 22.98 and 26.60, respectively. In contrast, output token count has a smaller but statistically significant effect. This is expected, as repository-level context can grow substantially while generation length remains capped at 128 tokens to predict the next line.

Because the predictors were standardized, we can directly compare the regression coefficients to measure the real-world impact of scaling up data sizes. The coefficients represent the expected change in total energy consumption (in Wh) for every one-SD increase in that variable. For example, in RepoBench, a one-SD increase in input tokens corresponds to approximately 1.3 million (1,316,930) additional aggregated input tokens across the 300 prompt instances (~ 4333 tokens per prompt), which is associated with an increase of approximately 23 Wh in total energy consumption. On the other hand, a one-SD increase in output tokens (2310 tokens), imposes 4.9 Wh more energy in total energy usage.

To find out how many output tokens would generate that same 22.98 Wh (~ 23) increase, we can calculate their energy equivalence. While a 23 Wh increase in energy consumption is driven by an additional 1.3 million input tokens, that exact same 23 Wh energy increase would be triggered by generating just 10,857 aggregated output tokens (~ 36 token per prompt). Under the RepoBench workload configuration, generating a single token consumes as much energy as processing approximately 121 input tokens (1,316,930/10,857).

The interaction between input tokens and active parameter count is also statistically significant, indicating that the energy impact of increasing context size becomes larger for bigger models. Similarly, the interaction between input and output tokens is significant, suggesting that generation becomes more expensive when conditioned on larger contexts.

However, the interaction between output tokens and parameter count is not statistically significant, likely because the variation in generated output length is relatively limited in RepoBench compared to the variation in input size.

Bit width also has a statistically significant effect on energy consumption. Compared to the Q4 baseline, Q8 and FP16 variants require additional energy, confirming that quantization (lower bit width) can improve energy efficiency even after controlling for token counts and parameter count. Moreover, the programming language variable is statistically significant, suggesting that language-specific characteristics such as tokenization patterns and repository structure may influence energy consumption in long-context workloads.

For McEval, the regression model also achieved a high goodness of fit ($R^2 = 0.940$). However, the contribution of the predictors differs substantially from RepoBench. In this benchmark, input size is capped at approximately 2K tokens, while the generation length can reach up to 1300 tokens. As a result, output token count becomes one of the dominant predictors of energy consumption, with a coefficient of 23.50, whereas input token count is no longer statistically significant. This indicates that, in generation-heavy workloads, the decoding phase contributes more strongly to total energy usage than prompt processing.

The interaction between output tokens and parameter count is highly significant in McEval, indicating that generating longer outputs becomes increasingly expensive for larger models. In contrast, the interactions involving input tokens are not statistically significant, which aligns with the relatively limited variation in prompt length in this benchmark. Quantization effects remain statistically significant, with FP16 models showing substantially higher energy consumption than Q4 variants. However, unlike RepoBench, the programming language variables are not statistically significant after controlling for token counts and parameter count, suggesting that language-specific effects become less influential when the workload is dominated by generation rather than long-context processing. The regression behavior closely matches the intended workload characteristics of the two benchmarks.

These findings align with the earlier correlation analysis, where interaction terms exhibited the strongest associations with energy consumption in their respective code completion workload settings. Overall, the two benchmarks provide complementary perspectives on the energy behavior of LLM inference. RepoBench primarily captures the effect of long-context processing, where input size and its interaction with parameter count dominate energy consumption. In contrast, McEval emphasizes generation-heavy workloads, where output length and its interaction with parameter count become the primary drivers. Together, these results indicate that the dominant contributors to energy consumption depend strongly on the structure of the workload. Context-heavy software engineering tasks are mainly influenced by prompt processing costs, whereas generation-heavy tasks are primarily driven by autoregressive decoding behavior.

Summary. Our regression analysis shows that the dominant drivers of energy consumption strongly depend on the characteristics of the completion task. In RepoBench, energy usage is primarily influenced by input context size and its interaction with active parameter count, highlighting the cost of long-context processing. In contrast, McEval is mainly driven by output generation, its interaction with active parameter count, and quantization level, indicating that autoregressive decoding dominates in generation-heavy workloads. The results also reveal that output generation is substantially more energy-intensive on a per-token basis than prompt processing. Furthermore, the significance of the interaction terms suggests that the energy cost of processing and generating tokens is better explained by considering token counts and the scale of the model together.

Takeaway. Practitioners should select models carefully, as larger models do not necessarily yield better accuracy but can considerably increase energy consumption. Furthermore, limiting unnecessary input and output tokens can reduce inference energy.

4 Related Work

Recent research on LLMs for software engineering (SE) focuses on improving code completion accuracy, while the inference energy footprints remain underexplored. A systematic review [53] shows that 76% of Green AI literature was published recently and nearly half of these primary studies focus only on the training phase. Most remaining inference benchmarks are algorithm-agnostic or use image datasets. This leaves open questions about how inference energy scales across programming languages, context windows, and model precisions.

Several studies have investigated LLM-based code completion from an accuracy perspective, including efficient training strategies for fill-in-the-middle (FIM) models, syntax-aware infilling benchmarks, practical IDE-based evaluation settings, alignment techniques for improving code generation quality, and empirical analyses of LLM-generated software quality [8, 20, 29, 41, 49]. These studies provide valuable insights into code completion performance and software quality, but they do not primarily analyze the energy cost of inference.

Repository-level code completion has also received increasing attention, since real-world code completion often depends on information across multiple files. Recent studies emphasize long-context reasoning and cross-file dependency modeling through retrieval augmentation and contextual integration strategies [23, 24, 36]. Zhang et al. [55] proposed RepoCoder, which combines retrieval and generation for repository-level code completion, while Ding et al. [16] studied code completion by jointly modeling in-file and cross-file context. Recent work extends repository-level completion to specific domains, such as RTL code completion [54], and API-centric repository-level code completion [34]. These studies emphasize repository context and retrieval, but they mainly evaluate completion quality rather than the energy implications of processing larger contexts.

Beyond accuracy, related work studies sustainability in code generation. Some focus on the energy efficiency of the generated code itself, examining whether LLM-generated programs are more or less energy-efficient than human-written solutions [5, 14, 25, 28, 52]. This work is complementary to ours: it studies the runtime energy behavior of the produced software, whereas our study focuses on the energy usage of the LLM during the code completion.

Energy consumption of LLMs has also been studied from Green AI and inference-efficiency perspectives. Fernandez et al. [19] analyze the energy implications of LLM inference optimizations across diverse workloads, software stacks, hardware accelerators, and serving configurations. GreenMyLLM [13] studies the energy consumption of code assistants by simulating developer interactions and analyzing factors such as model size, quantization, streaming, and rejected suggestions. Jareno et al. [30] further investigate energy-efficient LLM inference from an SE perspective, proposing compiler-level optimization of the *llama.cpp* inference engine and showing that hardware-aware code transformations can reduce energy consumption without modifying the underlying model. More recently, SweetSpot [46] proposed an analytical model for predicting LLM inference energy efficiency as a function of input and output sequence lengths, highlighting non-linear efficiency regimes and energy “sweet spots” under controlled synthetic workloads. These works demonstrate the growing importance of inference energy, but they do not focus specifically on comparing realistic code completion workloads across context sizes, programming languages, and completion styles.

The studies closest to ours jointly consider LLM inference energy and SE tasks. Solovyeva and Castor [50] perform a phase-level analysis of LLM inference energy in software development benchmarks, distinguishing between input processing and output generation and showing that excessive generation can substantially increase energy consumption. Mehditabar et al. [39] propose BRACE, a benchmark-oriented framework for evaluating coding LLMs in

terms of both functional accuracy and energy efficiency. These studies are highly relevant, but they address different levels of analysis. The former focuses on phase-level inference behavior, while the latter provides a unified energy-accuracy benchmarking perspective.

Our work complements these studies by examining workload-level energy behavior in realistic code completion scenarios. We study both repository-level left-to-right completion with varying context sizes and multilingual fill-in-the-middle completion across Python, Java, and Rust. Rather than only ranking models by accuracy or energy, we analyze how input length, output length, active parameter count, quantization, programming language, and context size shape energy-accuracy trade-offs. We show that long-context repository completion and generation-heavy FIM completion stress LLM inference differently, and that energy-efficient model selection depends on the characteristics of the completion task.

5 Threats to Validity

A potential threat to **internal validity** relates to the occasional incomplete responses returned by the Ollama API. In a small number of cases (typically one or two prompts per model), the model failed to return a complete generation and skipped the query. This issue became more noticeable in RepoBench under large context sizes (e.g., beyond 16K tokens), where missing a single long-running prompt could noticeably reduce the measured elapsed time and energy consumption, potentially making some FP16 models appear more energy-efficient than lower-bit quantized variants. To mitigate this issue, we manually inspected the results and excluded incomplete executions when computing the energy cost per generated token. Another threat to internal validity arises from the limited number of repetitions per experimental configuration. Each setup was executed three times to capture transient runtime variability while keeping the overall computational cost manageable. Increasing the number of repetitions could further improve statistical confidence, but would significantly increase the required execution time and resource usage. Nevertheless, since each run processes hundreds of prompts over several minutes and GPU power is sampled at 10Hz, the measurements exhibit relatively stable utilization patterns, reducing the impact of transient fluctuations. Another threat concerns the approximation used for energy estimation. Total energy consumption was computed using the arithmetic mean of the sampled GPU power values multiplied by the elapsed execution time, rather than performing numerical integration over the full power trace. To assess the impact of this approximation, we conducted an additional validation experiment on the Java subset of RepoBench. The observed discrepancy between the two approaches remained small, with an average difference of 0.46% and a maximum deviation of 2.3%. Finally, energy was measured using a software-based GPU power profiler rather than external hardware instruments, which may affect measurement precision. However, the same procedure was applied consistently across all configurations.

A potential threat to **external validity** of our study concerns the selection of programming languages. Although Python, Java, and Rust cover dynamically typed, statically typed, and relatively lower-resource mainstream languages, the results may not generalize to all programming ecosystems or domain-specific languages. Another potential threat to external validity is that the measurements on the A100 platform are limited to GPU power obtained through NVML, since privileged access required for CPU-side energy measurements was unavailable on the cluster. Consequently, the reported energy values do not include the contribution of CPU, DRAM, or PCIe communication overheads. Although this may lead to an underestimation of total system-level energy consumption, all experiments were conducted under identical hardware configurations and comparable data transfer conditions, which helps preserve the validity of fair comparisons across models and workloads.

A potential threat to **conclusion validity** concerns regression assumptions. Diagnostic Q-Q plots exhibit deviations from normality in the residual tails. Alternative specifications using a log-transformed outcome and Gamma regression did not consistently improve the diagnostics. Therefore, we acknowledge these deviations as a limitation of the analysis.

6 Conclusion

This study investigated the trade-off between accuracy and energy consumption in LLM-based code completion across different programming languages, context sizes, and completion workloads. Using RepoBench and McEval, we evaluated a total of 25 open-weight LLMs under long-context repository-level completion and generation-heavy fill-in-the-middle completion scenarios. Our results show that the dominant drivers of energy consumption depend strongly on the characteristics of the completion task. In long-context workloads such as RepoBench, energy consumption is primarily influenced by input context size and its interaction with model scale. In contrast, generation-heavy workloads such as McEval are mainly driven by output generation and its interaction with active parameter count. These findings indicate that different software engineering tasks stress different components of LLM inference.

We further observed that output generation is substantially more energy-intensive per token than prompt processing. In addition, increasing context size consistently raised the energy cost per generated token, while the corresponding improvements in completion quality were often limited and highly model-dependent. Across both benchmarks, smaller and heavily quantized models frequently achieved Pareto-optimal trade-offs, often providing accuracy comparable to larger full precision models while consuming substantially less energy. Moreover, model rankings in terms of energy efficiency were not stable across context sizes, suggesting that certain model families scale more efficiently under long-context workloads. Overall, our findings demonstrate that larger models and longer contexts do not necessarily provide proportionally better completion quality, emphasizing the importance of energy-aware model selection and deployment strategies for AI-assisted software engineering tools.

Future work could complement our regression analysis with controlled ablation studies that vary one factor at a time while keeping others fixed, enabling a more precise understanding of how context length, output length, quantization, and model scale influence energy consumption. In addition, extending this analysis to emerging agentic workflows involving iterative multi-turn interactions, self-correction, and tool usage would help better understand the sustainability challenges of large-scale AI-assisted software engineering systems.

Data Availability Statement

The replication package for this study, along with the appendices, is publicly available [2].

References

- 1 Marta Adamska, Daria Smirnova, Hamid Nasiri, Zhengxin Yu, and Peter Garraghan. Green prompting. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.10666.
- 2 Negar Alizadeh. Replication Package, August 2026. doi:10.5281/zenodo.21971227.
- 3 Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. Language models in software development tasks: An experimental analysis of energy and accuracy. In *Proceedings of the 22nd MSR*, pages 725–736. IEEE, 2025. doi:10.1109/MSR66628.2025.00109.

- 4 Negar Alizadeh and Fernando Castor. Green AI: a preliminary empirical study on energy consumption in DL models across different runtime infrastructures. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering CAIN*. ACM, 2024. doi:10.1145/3644815.3644967.
- 5 Radu Apsan, Vincenzo Stoico, Michel Albonico, Rudra Dhar, Karthik Vaidhyanathan, and Ivano Malavolta. Generating energy-efficient code via large-language models—where are we now? *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.10099.
- 6 Mauricio Fadel Argerich and Marta Patiño-Martínez. Measuring and improving the energy efficiency of large language models inference. *IEEE Access*, 12:80194–80207, 2024. doi:10.1109/ACCESS.2024.3409745.
- 7 Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. *Empirical Software Engineering*, 2026.
- 8 Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint*, 2022. doi:10.48550/arXiv.2207.14255.
- 9 Fernando Castor. Estimating the energy footprint of software systems: a primer. *CoRR*, abs/2407.11611, 2024. doi:10.48550/arXiv.2407.11611.
- 10 Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. Mceval: Massively multilingual code evaluation. *arXiv preprint*, 2024. arXiv:2406.07436.
- 11 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. Evaluating large language models trained on code. *CoRR*, 2021. arXiv:2107.03374.
- 12 Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint*, 2024. arXiv:2412.00535.
- 13 Tristan Coignon, Clément Quinton, and Romain Rouvoy. Green my llm: Studying the key factors affecting the energy consumption of code assistants. *arXiv preprint*, 2024. doi:10.48550/arXiv.2411.11892.
- 14 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. In *International Conference on the Quality of Information and Communications Technology*, pages 161–176. Springer, 2024. doi:10.1007/978-3-031-70245-7_12.
- 15 Santiago del Rey, Paulo Sérgio Medeiros dos Santos, Guilherme Horta Travassos, Xavier Franch, and Silverio Martínez-Fernández. Aggregating empirical evidence from data strategies studies: A case on model quantization. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 12–22, 2025. doi:10.1109/ESEM64174.2025.00049.
- 16 Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, 2024.
- 17 Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Proceedings of the 37th NeurIPS*, 2023.

- 18 Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. In *Proceedings of the 63rd Annual Meeting of the ACL*, pages 32556–32569, Vienna, Austria, 2025. doi:10.18653/v1/2025.acl-long.1563.
- 19 Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the ACL*, pages 32556–32569, 2025. doi:10.18653/V1/2025.ACL-LONG.1563.
- 20 Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. In *Proceedings of the 41st ICML*. JMLR.org, 2024.
- 21 Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu. A survey of low-bit large language models: Basics, systems, and algorithms. *CoRR*, abs/2409.16694, 2024. doi:10.48550/arXiv.2409.16694.
- 22 Maarten Grootendorst. A visual guide to mixture of experts (moe). <https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mixture-of-experts>, 2024.
- 23 Zhanming Guan, Junlin Liu, Jierui Liu, Chao Peng, Dexin Liu, Ningyuan Sun, Bo Jiang, Wenchao Li, Jie Liu, and Hang Zhu. Contextmodule: Improving code completion via repository-level contextual information. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.08063.
- 24 Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. In line with context: Repository-level code generation via context inlining. *arXiv preprint*, 2026. doi:10.48550/arXiv.2601.00376.
- 25 Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.
- 26 Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92, 2025. doi:10.1016/J.CSI.2024.103917.
- 27 IBM. Granite models. <https://huggingface.co/ibm-granite>, 2024. Accessed: 2025-09-21.
- 28 Md Arman Islam, Devi Varaprasad Jonnala, Ritika Rekhi, Pratik Pokharel, Siddharth Cilamkoti, Asif Imran, Tevfik Kosar, and Bekir Turkkan. Evaluating the energy-efficiency of the code generated by llms. *arXiv preprint*, 2025. doi:10.48550/arXiv.2505.20324.
- 29 Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639138.
- 30 Javier Jareño, José Miguel Aragón-Jurado, Juan Carlos De La Torre, Patricia Ruiz, and Bernabé Dorronsoro. Energy-efficient large language models. *Future Generation Computer Systems*, 182:108483, 2026. doi:10.1016/J.FUTURE.2026.108483.
- 31 Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the ACL*, pages 12186–12215, 2024. doi:10.18653/V1/2024.FINDINGS-ACL.726.
- 32 Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the ACL*, pages 1496–1524, 2025.
- 33 Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37:57619–57641, 2024.
- 34 Zhihao Li, Chuanyi Li, Changan Niu, Ying Yan, Jidong Ge, and Bin Luo. Api-repo: Api-centric repository-level code completion. In *Proceedings of the 16th International Conference on Internetware*, pages 449–460, USA, 2025. doi:10.1145/3755881.3755913.

- 35 Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint*, 2023. doi:10.48550/arXiv.2306.03091.
- 36 Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.07003.
- 37 Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *CoRR*, abs/2211.02001, 2022. doi:10.48550/arXiv.2211.02001.
- 38 Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of AI deployment? In *Proceedings of FAccT 2024*, pages 85–99, 2024. doi:10.1145/3630106.3658542.
- 39 Mohammadjavad Mehditabar, Saurabhsingh Rajput, Antonio Mastropaolo, and Tushar Sharma. Smart but costly? benchmarking llms on functional accuracy and energy efficiency. *arXiv preprint*, 2025. arXiv:2511.07698.
- 40 Microsoft. Lcc_python: Large code corpus for python. https://huggingface.co/datasets/microsoft/LCC_python, 2023.
- 41 Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assunção. Is llm-generated code more maintainable & reliable than human-written code? In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162, 2025. doi:10.1109/ESEM64174.2025.00036.
- 42 NVIDIA Corporation. Nvidia a100 80gb pcie gpu product brief. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf, 2022. PB-10577-001 v03.
- 43 NVIDIA Corporation. NVIDIA System Management Interface (nvidia-smi). <https://developer.nvidia.com/system-management-interface>, 2024. Accessed: 2024-11-06.
- 44 NVIDIA Corporation. Nvidia management library (nvml). <https://developer.nvidia.com/management-library-nvml>, n.d. Accessed: 2026-02-19.
- 45 David A. Patterson, Joseph Gonzalez, Urs Hölzle, Quoc V. Le, Chen Liang, Lluís-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022. doi:10.1109/MC.2022.3148714.
- 46 Hiari Pizzini Cavagna, Andrea Proia, Giacomo Madella, Giovanni Battista Esposito, Francesco Antici, Daniele Cesarini, Zeynep Kiziltan, and Andrea Bartolini. Sweetspot: An analytical model for predicting energy efficiency of llm inference. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering*, pages 83–95, 2026. doi:10.1145/3777884.3797011.
- 47 Soham Poddar, Paramita Koley, Janardan Misra, Niloy Ganguly, and Saptarshi Ghosh. Towards sustainable NLP: Insights from benchmarking inference energy in large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proc. 2025 Conference of the Nations of the Americas Chapter of the ACL*, pages 12688–12704, 2025. doi:10.18653/V1/2025.NAACL-LONG.632.
- 48 Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. *arXiv preprint*, 2022. arXiv:2201.05596.
- 49 Houxing Ren, Zimu Lu, Weikang Shi, Haotian Hou, Yunqiao Yang, Ke Wang, Aojun Zhou, Junting Pan, Mingjie Zhan, and Hongsheng Li. Alignment with fill-in-the-middle for enhancing code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8315–8331, 2025.
- 50 Lola Solovyeva and Fernando Castor. Towards green ai: Decoding the energy of llm inference in software development. *arXiv preprint*, 2026. doi:10.48550/arXiv.2602.05712.

- 51 Stack Overflow. 2025 developer survey – AI agent orchestration tools. URL: <https://survey.stackoverflow.co/2025/ai/#3-ai-agent-orchestration-tools>, 2025.
- 52 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 llm-based green code generation. *arXiv preprint*, 2024. doi:10.48550/arXiv.2403.03344.
- 53 Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of green ai. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(4):e1507, 2023. doi:10.1002/WIDM.1507.
- 54 Peiyang Wu, Nan Guo, Junliang Lv, Xiao Xiao, and Xiaochun Ye. Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.08862.
- 55 Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023. doi:10.18653/V1/2023.EMNLP-MAIN.151.
- 56 Johannes Zschache and Tilman Hartwig. Comparing energy consumption and accuracy in text classification inference. *arXiv preprint*, 2025. doi:10.48550/arXiv.2508.14170.