IJHSR
ijhighschoolresearch.org
RESEARCH ARTICLE

Evaluating the Financial and Environmental Impact of Prompt Engineering and Quantization on LLMs for Code Generation

Ateeksh Seth

Oberoi International School, 4VR8+2W8, Jogeshwari - Vikhroli Link Rd, Jogeshwari East, Mumbai, Maharashtra, 400060, India; ateeksh.seth@gmail.com

ABSTRACT: Prompt engineering is a technique that helps extend a Large Language Model's (LLMs) capability without altering the core model. However, these prompting techniques can often be inefficient as they require multiple LLM inferences and large context windows. LLMs have gotten exceptionally big and are being used billions of times every day. This implies that inefficient prompt engineering techniques can have huge impacts when scaled up, especially financial and environmental impacts. As resources are used inefficiently, due to prompt engineering, they often lead to higher costs in electrical usage, which can make LLMs more expensive to use and run. Additionally, this higher electricity use can raise carbon emissions, increasing the cost to the environment. Most cloud providers don't use renewable sources of energy. This underscores why a deeper analysis of the environmental impacts of LLMs is necessary. Any inefficiencies can often scale up, leading to huge costs, negatively impact the environment, and make LLMs cost-intensive. This study aims to evaluate the trade-offs between accuracy and the environmental and financial costs of various prompting techniques, enabling practitioners to make informed decisions about the cost efficiency and sustainability of their LLM pipelines.

KEYWORDS: Computer Science, Artificial Intelligence, Large Language Model (LLM), Quantization Prompt.

Introduction

Large language models are Natural language processing models designed to predict word sequences based on the transformer architecture.[1] They are trained on vast amounts of text data to understand and generate human-like language, and are capable of performing tasks such as translation, summarization, question answering, and code generation.[2] Though very capable models, they often can't match human-level reasoning and intelligence in logical contexts. Hendrycks *et al.*[3] mention that LLMs, even the big ones, are far behind Humans in complex math tasks, and Patel *et al.*[4] mention that LLMs often struggle with simple grade school math regardless of size.

To extend their performance in complex logical contexts, LLMs use prompting techniques. This is done by engineering and tuning the prompts sent to the model. Prompt engineering is especially powerful because it allows the model to function better than its standard capacity and also improves the model's ability in unseen or untrained contexts, while not requiring any tuning or training.[2]

Recent advances in prompt engineering have brought the models close to human-level reasoning.[5] Regardless, prompt engineering is often inefficient. Most prompting techniques often utilize a large portion of the context or require multiple inferences.[2]

This paper compares different techniques used to improve LLMs, examining whether the additional accuracy of some techniques justifies the cost. As LLMs are getting bigger and more widespread, this question becomes more and more important as inefficiencies in the model scale up. These issues are especially impactful in two segments: Financial and environmental.

Most APIs often charge for the token count. This makes the large context windows for many techniques costly. Additionally, the electrical input and computational power required lead to huge carbon emissions, which create an environmental cost. This makes LLMs unsustainable for the environment. This is why this study measures electrical and token output throughout different prompting techniques to compare and identify how prompting techniques fare on common reasoning datasets.

The data collected will include prompt tokens, completion tokens, and the electrical output, and will evaluate the different prompting techniques on factors such as accuracy and relative environmental emissions. The tests will be run on a Mistral-7B-Instruct-v0.1 quantized at 4bit and 8bit precision levels to gain a more diverse perspective on how size and quantization might have impacts. Additionally, the evaluation will also consider the differences between self-hosting and the API for the model.

Background Theory

The goal of an LLM is to create an intelligent natural language model that is capable of using its token-predicting capabilities to synthesize and generate text automatically. LLMs are first trained on huge corpora of text data, such as the common crawl, Wikipedia, online books, GitHub, Arxiv, and open source.[6] This data is for pretraining used to train the model into creating a token prediction model that can predict patterns and generate a distribution of next possible token combinations.

LLMs learn to understand the logic between different tokens, but this isn't just restricted to human languages. Trained on millions of examples, LLMs can go beyond basic conver-

DOI: 10.36838/IJHSR811.27

sation and generate code. LLMs are becoming exceptionally common for software development due to their ability in coding tasks, especially those that are tedious, with exceptional speed.[6] Therefore, code generation is an important factor to measure the LLMs' performance in the real world, because code generation is one of the most common uses for LLMs.[6]

There are many ways to improve the ability of LLMs, but the simplest and most efficient way is to use prompt engineering.

One of the simplest prompting techniques is a few-shot. It involves giving the model a set of examples in the prompt. These examples are designed to provide the LLM with additional context to help with the problem. The number of examples chosen can range from 2 to 100, as many as possible in the context window, and they showcase impressive increases in accuracy. Though they perform worse than state of the art fine tuned models.[1] Regardless, they allow the model to significantly improve accuracy without an abundance of data samples or any retraining.

Chain of Thought (Chain of Thought) is a promoting technique that prompts an LLM to think and process complex processes similar to a human. When humans approach a multi-step or complex problem, they often approach it step by step. Similarly, COT encourages the model to think about the task in multiple steps rather than answering the question directly. It is highlighted to have multiple benefits for reasoning performance. It significantly improves LLM performance on reasoning tasks, especially big models like PaLM 540B: boosting accuracy on GSM8K from 17.9% to 58.1%.[7] For this study, we will compare 0-shot COT.

Sample reasoning

```markdown
To solve the task, we need to follow these steps:
1. Convert the input string to a list of numbers using the `split()` function.
2. Sort the list of numbers using the `sorted()` function.
3. Convert the sorted list of numbers back to a string using the `join()` function.
```

Scratchpad is a form of reasoning technique that allows the model to generate an arbitrary sequence of intermediary tokens before producing the answer. Unlike COT, which is designed to think before an answer, the scratchpad is meant to use its reasoning tokens as a notebook and pen to solve the problem. When Zero Shot COT generates steps for complex problems, it can often make errors in reasoning or skip steps, leading to the LLM making errors while executing. Scratchpad performs steps in the planning, which improves accuracy as the answer is less likely to be impacted by faulty reasoning. Scratchpad improved performance by approximately 50% on the MultiArith benchmark and by almost 40% on the Python code execution benchmark.[8]

Sample reasoning

```
* **Step 1:** Take the input string of numbers, e.g., `"3 1 2"`.
* **Step 2:** Split the string into a list using `split()`.
* Input: `"3 1 2"` → `["3", "1", "2"].`
* **Step 3:** Sort the list using `sort()` (or `sorted()`).
* Sorted list: `["1", "2", "3"].`
* **Step 4:** Join the sorted list back into a string using `' '.join()`.
* Result: `"1 2 3"`
```

Structured Chain of Thought (SCOT) is another prompting technique that builds on COT. COT, designed to improve the reasoning ability of LLMs, doesn't do as well on Code generation tasks. SCOT is a version of COT that specializes in code through precise instructions, such as loops and branches.[9] Instead of writing in natural language like COT, it generates reasoning in pseudocode. These pseudocode instructions are often more precise and specific, which allows for easier conversion, leading to better accuracy than their counterparts. SCOT reveals better performance than COT by 13.79% on Human Eval benchmark.[10]

Sample reasoning

```
// Sequence: Parse input and prepare data
INPUT ← READ_INPUT_STRING()
LIST ← SPLIT(INPUT, delimiter=" ")
// (Optional branch: ensure elements are numeric)
IF ANY_ELEMENT_NOT_NUMERIC(LIST) THEN
ERROR ← "Invalid input: non-numeric detected."
RETURN ERROR
END IF
// Sequence: Sort operation
SORTED_LIST ← SORT(LIST)
// Sequence (could be seen as a loop underpinning join)
OUTPUT ← ""
FOR EACH ELEMENT in SORTED_LIST DO
OUTPUT ← CONCATENATE_WITH_SPACE(OUTPUT, ELEMENT)
END FOR
RETURN TRIM_LEADING_SPACE(OUTPUT)
```

Quantization is a process that can help with both financial and environmental aspects of running an LLM. Quantization is the process of shrinking the precision of numbers used in LLMs by reducing the number of bits used to represent them. This process rounds values, thereby reducing the number of calculations done by the computer during inference. However, the process of rounding can reduce the accuracy and performance of LLM. Regardless, according to Giagnorio *et al.*,[11] A 4-bit quantization to Deepseek Coder 33B led to 70% reduction in memory footprint without significant loss in performance, highlighting its usefulness in running LLMs with limited hardware.

Methods

Datasets:

Human Eval is a dataset by OpenAI. It contains many samples of coding problems meant to test LLM's ability in code generation. It asks the user to write code based on an input and then tests the code using an evaluator to provide a quantitative measurement of the model's ability through its 164

samples. Exactly 100 samples will be used in this study for each prompting technique.

Example HumanEvel/01[12]

Question

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

Test

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

Metrics collected:

Tokens refer to the sub-word level encoding that LLMs use to work with information. In this experiment, they are split into prompt tokens, tokens that are generated by the user and imputed into the model, and completion tokens, tokens generated by the model. Separating provides more accurate estimations of the cost, because prompt tokens tend to take less energy than completion tokens, so that prompt-heavy response sparse techniques aren't at a disadvantage.

Electrical input is the energy required to run the model. Beyond hardware costs, which are usually constant for all techniques, electrical input helps measure the true environmental impact and financial cost for self-hosting. Both GPU and total electrical input will be measured. This is because the total electrical input allows trends and provides accurate values for total consumption, but can be noisy and easily impacted by other factors. Whereas, GPU energy is directly linked with LLM inference, and isn't affected significantly by background processes. Code Carbon will be used to measure the electrical input. Code Carbon is a Python library used to estimate the electric consumption and the carbon footprint of executed code.[13,14] It calculates electric power consumption using system hardware.

The financial cost of using a model with an API can be calculated by multiplying the token count by the rate. Usually, prompt tokens cost less than completion tokens, but for the Mistral API, our standard of measure, the cost of input and output tokens is $0.25 per million input tokens,[15] or $0.00000025 per token.

Self-hosting financial cost will be the financial cost of operating the model. This involves setting a standard rate for electricity and cost, and then multiplying it by $0.13 per kilowatt-hour (kWh).[16] Though not identical in all locations, it is comparable between prompting techniques. Iowa was chosen as the location, as the data collection was done on a server in Iowa.

Self-hosting CO_2 emissions are based on multiplying the electrical input by the carbon intensity of electricity. The carbon intensity comes from the same location as the electricity rate, and the carbon intensity for Iowa is 252 g CO_2 / kWh. [17]

Model:

Mistral 7B was chosen because it is an open-source and light model, with 7 billion parameters, while still being comparable to much bigger models in reasoning performance. This allows easy and comprehensive evaluation. Additionally, the data will be collected with 4-bit quantization and 8-bit quantization.

Sampling Strategies/ Inference Settings:

input_ids=inputs,
attention_mask=attention_mask,
max_new_tokens=1000,
do_sample=True,
temperature=0.7,
top_p=0.95

Tools Used:

Nvidia T4x2
Kaggle Notebook

Results and Discussion

Table 1: 4-bit quantization: Prompting techniques performed worse than 0-shot in accuracy.

	Zero Shot	COT	Scratchpad	Few Shot	SCOT
Accuracy at Human Eval(%)	86	86	82	76	81
Prompt tokens	15855	22755	37155	78755	33555
Completion Tokens	33123	38548	26139	30568	32318
Total Tokens	48978	61303	63294	109323	65873
GPU Electrical input (kWh)	0.07771985162	0.09367008827	0.05964057355	0.0744172125 9	0.07820676784
Total Energy Input (kWh)	0.1295784956	0.186484157	0.1041235153	0.1271711545	0.1340371129
Duration (Seconds)	2989.058672	4307.521816	2563.834739	3040.547511	3217.871555
API Financial cost ($)	0.0122445	0.01532575	0.0158235	0.02733075	0.01646825
Self-Hosting Financial Cost	0.01010358071	0.01217711148	0.00775327456 1	0.0096742376 37	0.01016687982
Self Hosting CO2 released (g CO_2)	19.58540261	23.60486224	15.02942453	18.75313757	19.7081055

DOI: 10.36838/IJHSR811.27

Table 2: 8-bit quantization: Improvements in ability for few-shot and reasoning methods, but 0-shot still performs better than prompting techniques.

	Zero Shot	COT	Scratchpad	Few Shot	SCOT
Accuracy at Human Eval(%)	87	81	81	80	81
Prompt tokens	15855	22755	37155	78755	33555
Completion Tokens	28034	41842	28071	30420	30341
Total Tokens	43889	64597	65226	109175	63896
GPU Electrical input (kWh)	0.1291323489	0.2092452868	0.1234488821	0.1539121323	0.1412329944
Total Energy Input (kWh)	0.2411115047	0.3681580369	0.2302056308	0.2737391468	0.255693612
Duration (Seconds)	6454.04618	9158.96719	6152.92477	6906.287735	6596.967327
API Financial cost ($)	0.01097225	0.01614925	0.0163065	0.02729375	0.015974
Self-Hosting Financial Cost	0.01678720535	0.02720188729	0.01604835467	0.0200085772	0.01836028927
Self Hosting CO2 released (g CO_2)	32.54135191	52.72981228	31.10911829	38.78585734	35.59071458

The Impact of Quantization on Prompting Efficacy:

The results from Table 1 and Table 2 reveal that for 4-bit quantization, the zero-shot tends to perform as well as zero-shot COT, and better than scratchpad and other techniques; furthermore, higher prompt tokens were negatively linked to accuracy. This means that adding more details to your prompt context can lead to a loss of accuracy. However, most papers, such as Brown *et al.*,[1] Zhou *et al.*,[18], and Wei *et al.*[7], highlight that adding more context, examples, and reasoning steps tends to improve the model. The main reason for the reduction in accuracy is quantization. Quantization can reduce the precision of the bits, which doesn't affect accuracy too much, but reasoning is often sensitive and can be affected by quantization. Research reveals that quantization, especially at 4 bits, reduces the performance of the model.[19] Furthermore, quantization can have a huge impact on the logical reasoning ability of LLMs.[20] These highlight that quantization, especially with a small model like Mistral 7b, can make reasoning techniques ineffective.

Additionally, the 4-bit model had greater issues with hallucination. Instead of writing the code, the model would be stuck in a loop, repeating similar lines till it reached the token limit. However, when quantization was reduced, the model was much more consistent in its replies and didn't hallucinate in such a way. Additionally, both 4-bit and 8-bit underscored similar performance. This implies that even though they have similar accuracy, reducing quantization might be good for better structure and formatting tasks.

Table 3: Few shot Vs 0 Shot with 8-bit SCOT.

	SCOT	Few Shot SCOT
Accuracy at Human Eval(%)	81	86
Prompt tokens	33555	114155
Completion Tokens	30341	33766
Total Tokens	63896	147921
GPU Electrical input (kWh)	0.1412329944	0.1706562693
Total Energy Input (kWh)	0.255693612	0.3056164071
Duration (Seconds)	6596.967327	7778.500091

Moreover, in 4-bit quantization, adding a few shots highlighted a reduction in performance, but in 8-bit quantization, a few-shot approach performed better than 0-shot versions. For example, in the exploratory sample above in Table 3, where SCOT was run with a few shots, it performed 5% better. While the same trials in 4 bits revealed that the model actually performed worse. This implies that heavy quantization could reduce the few-shot performance of LLMs.[21]

Overall, quantization was found to be very useful in reducing the GPU energy usage by 46% and the time by 51%. This highlights that reducing the bit precision by half can reduce the processing required by approximately half, without a significant decrease in quality. This underscores the ability of quantization to make LLMs more accessible and easier to run.

Methods of improving efficiency:

Furthermore, while analyzing the outputs across all prompting techniques, one common waste of electricity and token count was post-reasoning. Post-reasoning is when a model reasons and thinks after writing the code. This can help explain code to the user in an interactive chatbot environment, but in the context of code generation, it is usually ineffective. When forced to think before writing the code, the reasoning capability of models increases, but thinking after writing the code can't improve the code; therefore, tokens are often wasted. To reduce post-reasoning, the model can be prompted to avoid reasoning after code, or can be terminated after detecting patterns, such as three back ticks (``"), that signify that the code is complete.

Another way to improve the performance would also be to merge the techniques. Though the paper works with and discusses discrete prompting techniques, many of these techniques can be merged to create more efficient or capable techniques. For example, Scratchpad could be combined with a few-shot approach to improve model performance with complex tasks.

Comparative Efficiency of Prompting Techniques:

Overall, the best performance is showcased by the 0-shot approach, even though it's the most basic technique. This is because quantization reduces the ability of the model to handle large context windows, which benefits 0-shot learning because it doesn't fill context with examples and thinking.[21]

COT can compete with 0 shot accuracy, but due to reasoning tokens, it takes much longer. This leads to 120% the energy usage when compared to 0 shots. Scratchpad, on the other hand, falls by 4% on accuracy, but is the most efficient prompting technique. Using 20% less electricity when compared to 0 shot and only 35% less than COT. This means it is recommended for efficient workloads that require reasoning.

Few-Shot is usually an efficient technique, but examples for code generation can take a huge portion of the context window, and quantization isn't able to handle huge context windows as well. In 8-bit quantization, we see a huge jump for few-shot, while other techniques maintain their accuracy. This means that for higher bit precision, few should be an effective technique.

However, few-shot is much worse in an API context because APIs value prompt tokens almost as much as completion tokens, which leads to higher API costs even though energy usage for prompt tokens is much lower. Overall, few-shot works better for less quantized models, and usually isn't efficient when examples are large, or the API is used.

SCOT is a version of COT, but in these experiments, it performs 4% worse. Though it highlights 20% less energy usage when compared to COT. This implies that pseudocode-like reasoning is much more efficient, but can come at the price of accuracy.

External threats to validity:

Models tend to perform significantly better as the parameter size increases.[5] This means that for bigger models with greater reasoning abilities, small improvements that may seem negligible may start to become more apparent, especially due to the non-linear nature of reasoning and LLM parameter size. This highlights another point of study into how bigger parameter size differences affect the gap in the accuracy between two similar models. Additionally, different LLMs process information differently, which makes generalization across all models difficult. For example, different LLMs require different quantities of energy for the same task.[22] Hence, using the same prompting techniques across different LLMs can give different results.

Furthermore, in our tests, we used 4-bit and 8-bit quantization. However, different levels of quantization could show different results. Quantization's impact on accuracy isn't linear,[23] and this paper does not test across a lot of quantization levels, such as 2-bit and 16-bit. Thereby, quantization could add more complexity and potentially hinder some prompting techniques more than others.

Moreover, some different LLMs have different value prompt tokens and completion tokens. This means that prompt-heavy techniques may have an advantage while using the API, but not be helpful while using another. This can be seen with COT and few-shot in Table 1, where even though few-shot has a much higher API cost, it is cheaper to run, because the API gives prompt tokens and Completion tokens equal value, in spite of prompt tokens taking less energy.

In addition, sampling strategies like top_P or temperature affect the token probability distribution used to predict the next word. For programming and code generation, a low temperature and top_P is recommended,[24] but different values could alter the result.

Internal threats to validity:

Though the experiment took precautions to prevent inaccuracies in data collection, specific issues could act as a threat to the validity of the study. To start, we used a popular code generation dataset, but not every sample from the dataset was used. For example, the first 100 samples for Human Eval. Additionally, Code Carbon measures the energy usage of the hardware components, but it can encounter noise from background tasks, though CPU and RAM usage were removed to reduce noise.

Future Work:

This study opens several avenues for future research. While this study focuses on the Mistral 7B instruct model with 4-bit and 8-bit quantization, future research could explore how the results change when some controlled factors are changed.

For example, evaluating the performance across other models could help substantiate the generalizability of the findings, and examining models with bigger parameter sizes could also highlight how reasoning ability and quantization affect the model.

Moreover, we did not explore methods that involve fine-tuning. Investigating techniques that leverage fine-tuning could reveal approaches that improve performance while reducing environmental and computational costs,[25] providing a broader view of sustainable model optimization.

Furthermore, methods that require the use of multi-inference pipelines, such as Least to Most prompting,[18] Automatic COT,9 and Plan & Solve,[26] are not included in the study. A similar study could be done for these types of techniques.

Conclusion

Overall, this study examined the effect of quantization and prompting techniques on the performance and efficiency of an LLM for code generation. The results indicate that while quantization provides a substantial reduction in energy usage and computational cost, it tends to reduce the model's reasoning ability, making planned reasoning and few-shot generalization through prompting techniques ineffective. It also reveals that quantization can lead to hallucination and also reduces the structural and formatting ability of the LLM. In addition, the study further compares that for heavily quantized models, 0-shot tends to work best for performance, whereas scratchpad tends to work best for cost and environmental efficiency, but also highlights that quantized models struggle with few-shot learning.

Acknowledgments

I would like to thank my mentors, Dr. Siddharth Krishnan and Samuel Lefcourt, for helping me understand the research process and acquire the prerequisite knowledge.

References

1. Vaswani A; Shazeer N; Parmar N; Uszkoreit J. Attention is all you need. Adv. Neural Inf. Process. Syst. 2017, 30.
2. Brown, T; Mann B; Ryder N; Subbiah, M. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 2020, 33, 1877–1901.
3. Hendrycks, D.; Burns, C.; Kadavath, S.; Arora, A Measuring mathematical problem solving with the MATH dataset. arXiv 2021, arXiv:2103.03874.
4. Patel, A.; Bhattamishra, S.; Goyal, N.; Patel, A. Are NLP models really able to solve simple math word problems? arXiv 2021, arXiv:2103.07191.
5. Huang, J.; Chang, K.; Huang, J.; Chang, K. Towards reasoning in large language models: A survey. arXiv 2022, arXiv:2212.10403.
6. Huynh, N.; Lin, B.; Huynh, N.; Lin, B. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. arXiv 2025, arXiv:2503.01245.
7. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M. Chain-of-thought prompting elicits reasoning in large language models. Adv. Neural Inf. Process. Syst. 2022, 35, 24824–24837.
8. Nye, M.; Andreassen, A.; Gur-Ari, G.; Michalewski, H. Show your work: Scratchpads for intermediate computation with language models. arXiv 2021, arXiv:2104.03464. 9. Shubham, V, Dubey, H. Survey of Prompt Engineering Techniques and Challenges, 2024.
10. Li, J.; Li, G.; Li, Y.; Jin, Z. Structured chain-of-thought prompting for code generation. ACM Trans. Softw. Eng. Methodol. 2025, 34 (2), 1–23.

DOI: 10.36838/IJHSR811.27

11. Giagnorio, A.; Mastropaolo, A.; Afrin, S.; Di Penta, M. Quantizing large language models for code generation: A differentiated replication. arXiv 2025, arXiv:2503.07103.
12. OpenAI. openai\humaneval. Hugging Face, 2021. Available online: [https://huggingface.co/datasets/openai/openai\humaneval](https://huggingface.co/datasets/openai/openaihumaneval).
13. Rubei R.; Moussaid A.; di Sipio C.; di Ruscio D. Prompt engineering and its implications on the energy consumption of large language models. arXiv 2025, arXiv:2501.05899.
14. Impact, M. C. Codecarbon: A tool to estimate the carbon emissions of machine learning models, 2024; accessed 2024-03-05. Available online: https://mlco2.github.io/codecarbon/.
15. Mistral AI. API Pricing. Mistral AI. Available at: https://mistral.ai/pricing#api-pricing. Accessed September 1, 2025.
16. EnergySage. How Much Does Electricity Cost in 2025? Available at: https://www.energysage.com/local-data/electricity-cost/. Accessed September 1, 2025.
17. LowCarbonPower.org. Iowa Electricity Generation Mix 2024/2025
18. Zhou, D.; Schärli, N.; Hou, L.; Wei, J. Least-to-most prompting enables complex reasoning in large language models. arXiv 2022, arXiv:2205.10625.
19. Liu, R.; Sun, Y.; Zhang, M.; Bai, H. Quantization hurts reasoning? An empirical study on quantized reasoning models. arXiv 2025, arXiv:2504.04823.
20. Li, Z.; Su, Y.; Yang, R.; Xie, C. Quantization meets reasoning: Exploring LLM low-bit quantization degradation for mathematical reasoning. arXiv 2025, arXiv:2501.03035.
21. Liu, Y.; Meng, Y.; Wu, F.; Peng, S.; Yao, H.; Guan, C.; Tang, C.; Ma, X.; Wang, Z.; Zhu, W. Evaluating the Generalization Ability of Quantized LLMs: Benchmark, Analysis, and Toolbox. arXiv 2024, 2406.12928 (preprint, posted June 15, 2024).
22. Jegham, N.; Abdelatti, M.; Elmoubarki, L.; Hendawi, A. How hungry is AI? Benchmarking the energy, water, and carbon footprint of LLM inference. arXiv 2025, arXiv:2505.09598.
23. Gong, Z.; Liu, J.; Wang, J.; Cai, X. What Makes Quantization for Large Language Model Hard? An Empirical Study from the Lens of Perturbation. Proc. AAAI Conf. Artif. Intell. 2024, 38 (16).
24. Arora, C.; Sayeed, A. I.; Licorish, S.; Wang, F. Optimizing large language model hyperparameters for code generation. arXiv 2024, arXiv:2408.10577.
25. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M. Distilling step-by-step! Outperforming larger language models with less training data and smaller model sizes. arXiv 2022, arXiv:2305.02301. (please confirm identifier)
26. Wang, L.; Xu, W.; Lan, Y.; Hu, Z. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. arXiv 2023, arXiv:2305.04091.

Author

Ateeksh Seth is a passionate student researcher with a deep interest in AI from Mumbai, India. He strong passion for computer science and sustainable technology, and aspires to develop and explore AI-driven solutions that address real-world problems in energy efficiency and software development.