Optimizing Test Case Generation and Bug Fixing Efficiency Through Hyperparameter Tuning of Local Large Language Models

RUBEN VAN DER LINDE, University of Twente, The Netherlands

Large Language Models (LLMs) have become incredibly popular, but using them costs a lot of energy. Hyperparameter tuning is a promising way to reduce this energy consumption. This study investigates the effect that hyperparameter tuning has on local large language models while performing test case generation and bug fixing tasks in Java. To accomplish this, 12 models from four different families were evaluated and compared across multiple configurations. The focus was on commonly tuned parameters—temperature, top-p, and max tokens—and their impact on accuracy and energy consumption. Additionally, the influence of including a method description in the prompt for bug fixing was examined. The results show that temperature has the most noticeable effect on performance, while top-p and max tokens have limited influence, as long as the token limit is high enough to avoid incomplete outputs. Lower bit-width models showed comparable performance across both tasks, and their energy efficiency remained very similar. Furthermore, including the method description led to a significant increase in accuracy and even a small reduction in energy usage. These findings provide a better understanding of how local LLMs behave under different hyperparameter settings and prompt designs.

Additional Key Words and Phrases: Large Language Models, Model Bit-Width, Energy Efficiency, Hyperparameter Tuning, Temperature, Token Limit, Top Probability, Test Case Generation, Bug Fixing, Method Description, Java.

1 INTRODUCTION

Large Language Models (LLMs) have become incredibly popular. New, improved models are launched frequently. These models have to be trained, which already costs a lot of energy[14]. But using these models in daily activities, such as a Google search, also adds up. According to [5], if Google were to fully integrate AI into its search services, its energy use could match that of an entire country comparable to Ireland. This energy consumption sparks the discussion on whether it can be reduced.

To address this problem, it is possible to investigate whether changing the configuration of LLMs can reduce energy usage while maintaining desired performance. For example, [2] studied differences in energy consumption across various models and model bit-widths, highlighting that architecture choice significantly impacts efficiency. Other studies, such as [7] and [17], have explored how to balance model performance with energy efficiency and underscore how important this balance is. Similarly, [8] and [19] demonstrated that hyperparameter tuning can substantially influence the energy efficiency of code generation models. [3] has also shown that hyperparameter tuning can affect model performance, though that study did not examine energy usage. [1] investigated how the fine-tuning of LLMs, further training them on their dataset, can enhance both performance and efficiency for test case generation tasks.

There remains a critical gap in the literature: while some existing research has explored performance–efficiency trade-offs in LLMs and shown the impact of hyperparameter tuning and fine-tuning for code generation, there are no studies on the energy optimization of test case generation and bug fixing using hyperparameter tuning. Yet, these tasks are equally critical in software development. A survey [4] that asked software developers how much of their time they spend on test generation and bug fixing showed that this is 15.8% and 25.32%, respectively, highlighting the time that can be won by automating these tasks using LLMs. Also, there are limited studies on the creation of Java code; all of the studies mentioned above focus on Python. It is unknown whether these findings, for Python code generation, can be transferred to other languages like Java. To investigate this, Java was chosen as the programming language for this research.

The goal of this research is to identify the result of the hyperparameter tuning for Java test case generation and bug fixing of local code generation models. To achieve this goal, the following research questions are formulated:

- **RQ1** What is the influence of hyperparameter tuning on the energy consumption and accuracy of locally executable LLMs during test case generation?
- **RQ2** What is the influence of hyperparameter tuning on the energy consumption and accuracy of locally executable LLMs during bug fixing?
- **RQ3** What is the influence of including the method description together with the buggy code inside the prompt on the energy consumption and accuracy of locally executable LLMs during bug fixing?

RQ3 was motivated by early experimental observations during prompt development, where including the method description alongside buggy code noticeably improved the performance of the models. This aligns with findings from [18], which showed that providing additional contextual information, such as method descriptions, can significantly enhance model output quality. Given this, RQ3 aims to evaluate how this affects both accuracy and energy usage in bug fixing tasks.

2 METHODOLOGY

This section will go into detail on the resources and methods that will be used to answer the research questions.

TScIT 43, July 4, 2025, Enschede, The Netherlands

2.1 Model Selection

Four different models will be evaluated, each with three bit-widths: FP16, Q8, and Q4 (see Table 1). These models have been chosen based on their high ranking on the BigCode Leaderboard on Hugging Face [9] and also on their relatively low number of parameters since this is a limiting factor based on accessible hardware. One way to reduce a model's memory footprint and computational cost is by using quantization, the process of converting high-precision floating-point parameters into lower-precision formats (e.g., 16-bit, 8-bit, or 4-bit). This allows models to retain much of their predictive power while using fewer resources. The bit-width of a model refers to the number of bits used to represent each parameter—16 in FP16, 8 in Q8, and 4 in Q4 [6]. Low-bit quantization can significantly reduce energy consumption during inference by lowering memory bandwidth and computational load, enabling more efficient deployment of LLMs on resource-constrained devices [2, 6]. This is already a good step in the direction of reducing the energy usage of a local LLM. Therefore, this research will also examine the difference in performance of these bit-width models.

Table 1. Models

Model Name	Parameters	Bit-widths
DeepSeek Coder V2	15.7B	FP16, Q8, Q4
QwenCoder2.5	7B	FP16, Q8, Q4
CodeLlama	7B	FP16, Q8, Q4
CodeGemma	7B	FP16, Q8, Q4

2.2 Hyperparameters

The following parameters, temperature, top probability, and max tokens, were chosen because they are among the most commonly changed parameters by users. Additionally, all three can influence the model's output size, which is the primary focus of this research.

2.2.1 Temperature. The temperature is a value between 0 and 2 [13, 15]. This value indicates the randomness of the outcome. A higher temperature will make the generated text more random and distinct, while a lower temperature will generate something more predictable/deterministic.

2.2.2 Top Probability. The top probability (top_p) is a value between 0 and 1 that defines a cumulative probability threshold for the selection of tokens. At generation time, the model considers only the most probable tokens whose cumulative probability mass exceeds top_p, then samples from that subset. Lower top_p values make the output more focused and deterministic, while higher values allow for more varied and creative responses [19].

2.2.3 Token Limit. The token limit sets the maximum allowed tokens, or words, that the model is allowed to generate per prompt. The model will stop generating once it reaches this limit, even though it might not have finished its answer. his does not mean the model will always use the full token limit. For example, when the max tokens is set to 400, the model may only output 250 tokens. This can occur if the model encounters a stop token, which acts as a signal for it to conclude its response.

2.3 Testing Environment

During these tests, the GPU energy consumption will be measured to assess how each parameter affects efficiency. The hardware used for this evaluation is a CPU/GPU node with an NVIDIA A10 GPU (see Table 2).

The environment uses a containerized Jupyter setup on Ubuntu 22.04.3 LTS. Java runs via OpenJDK 17.0.10 (Eclipse Temurin), using the 64-Bit Server VM in mixed mode. The default JVM uses the G1 Garbage Collector (G1GC), with InitialHeapSize=2147483648 and MaxHeapSize=32178700288. Parallel and Serial collectors are disabled.

2.4 Running an LLM

All of the selected models are open source, so they can be downloaded using Ollama [11], a widely used tool for managing and running LLMs locally. Ollama is also very useful for running different bit-widths of a model.

Table 2. Hardware setup

Hardware
CPU: 56 Cores / 128 Threads each – Max. 2.0 GHz
Memory: 256 GB
GPU: NVIDIA A10 – VRAM: 24 GB – TDP: 150W

3 EXPERIMENTAL PROCEDURE

3.1 Evaluation

To evaluate the performance of the models on both tasks, the HumanEvalPack benchmark was used. This dataset contains 164 Java coding problems, each with a predefined method and corresponding test case. For the bug fixing task, model outputs were assessed by checking whether the corrected method passed the official test case. For test case generation, the inverse was done: the generated test case was executed against the canonical (correct) solution. Each outcome-pass, fail, or compilation error—was recorded. The accuracy of a configuration was defined as the success rate: the number of passed cases divided by the total number of test cases.

3.2 Prompt Design

The prompt that is sent to the model is a very important part of the experiment. One of the goals of the prompt was for the response not to include any unnecessary words, which would be a waste of tokens, time, and energy. The prompt also had to give enough direction for the generated Java code to compile. For both tasks, the prompt contained the request not to include any natural language in any form. For test case generation, the method for which test cases needed to be generated was given, together with an outline of what the test case generation class should look like. For bug fixing, the buggy method was given together with a description and failure symptoms of the bug. Additionally, for the bug fixing tests with a method description, a method description was provided in

the prompt. Full prompts used in the experiments can be found in Appendix A.

3.3 Hyperparameter Tuning

Different hyperparameter values were tested to observe their effect on model behavior. Therefore, it was first necessary to determine in which steps these values would be changed. It is not recommended to change both the temperature and the top_p at the same time [12, 13], as they both affect the randomness of the model. The max tokens could be changed together with either the temperature or the top_p, but this would require a lot more computations. So, because of the time and resource constraints of this research, the hyperparameters were changed separately. Also, because of these constraints, the number of configurations tested had to be reduced.

For the ranges, the temperature is a value between 0 and 2. Previous study [16] has shown that a temperature between 0.0 and 1.0 did not result in any major changes in results. [16] also says that too high a temperature can lead to hallucinations of a model, which means that it starts to make things up that are factually incorrect. This may benefit tasks that require more creativity. Based on these considerations, a range that focuses on lower temperatures, with a few higher values included for comparison, was selected. Specifically, we use the following values for temperature: [0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 2].

The top p is a value between 0 and 1. To minimize the number of values, but still keep a set with which a pattern can hopefully be seen, the following were chosen: [0, 0.1, 0.2, 0.4, 0.8, 1]. The default values will be set to 0.1 and 0.95 for temperature and top_p, respectively.

Finally, for the max tokens, it is a bit harder to decide which steps to choose since it is possible to set this infinitely high. Another thing that makes this a difficult range to choose is that two completely different outputs are generated, test cases and fixed code, which have different sizes. But to allow for a fair comparison between these distinct outputs, the same max token values were chosen for both tasks. The HumanEvalPack has 164 examples, with an average of 41 tokens, a minimum of 4 tokens, and a maximum of 142 for the canonical solutions(i.e., the output generated with bug fixing). The tests of the HumanEvalPack (i.e., the output generated with test case generation) have an average of 69 tokens, a minimum of 36 tokens, and a maximum of 426. During initial tests, it was discovered that setting the max tokens too low resulted in a lot of compilation errors, since the LLM was not yet done generating the code. Therefore, the max tokens as a default value was set to 450, so no test could potentially fail because it ran out of tokens. The values to test were chosen to still check whether a lower max token value would influence the energy consumption, and if a higher value would result in better accuracy. The following values were used for the max tokens: [250, 350, 500, 750].

3.4 Energy Measurement

Due to the lack of administrator access in the used environment, it is not possible to measure the CPU power. Therefore, only the GPU power will be measured during the experiment. For this, NVIDIA's NVIDIA Management Library (NVML) [10] was used through the pynvml Python wrapper. A measurement rate of 10 Hz was chosen.

During the experiment, the amount of data loaded onto the GPU was measured. A threshold was determined based on the size of the model, the estimated overhead for activations, attention mechanisms, and KV cache, plus an additional safety margin buffer (1000MB). Upon reaching this threshold, the experiment would be terminated, ensuring that energy measurements remained unaffected by other programs running on the GPU.

The idle power consumption of the NVIDIA A10 GPU has been measured by executing the 'nvidia-smi' command at a sampling frequency of 100 Hz over a total duration of 600 seconds. This procedure was repeated ten times, yielding ten independent sets of 6000 power draw samples each. To ensure the integrity of the idle power measurements, the system was monitored to confirm that no other processes were using the GPU during the data collection period. These results show that in the idle state, the A10 uses 16.11 W on average with a standard deviation of 0.6132 W. The idle power draw of 16.11 W was not used in the rest of the research; however, it can serve as a reference for modeling or estimating background energy consumption in similar GPU workloads.

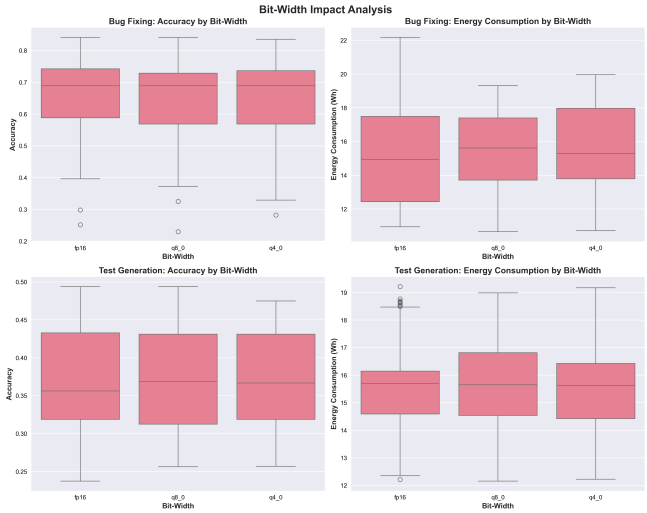

Fig. 1. Comparison of the three bit-widths by combining the results of all models and configurations

4 RESULTS

In this paper, only the bug fixing results without the method description are compared directly to the test case generation results (RQ1 vs RQ2). The variant with method description (RQ3) is evaluated separately due to its significantly different prompt structure and performance.

When evaluating overall model performance, Deepseek coder, despite having 15.7B parameters, often shows lower accuracy rates

than models like Qwen and Codegemma, which have 7B parameters each.

Three different bit-widths of each model were evaluated and are compared in Figure 1. For bug fixing, the FP16 bit-width shows a higher maximum energy usage and a lower median. In contrast, lower bit-width models (Q8 and Q4) demonstrate a slightly higher median energy consumption while maintaining comparable accuracy.

For test case generation, differences in performance across model bit-widths are minimal. Although the Q4 model shows more consistent results in terms of accuracy. And the upper quartile of the FP16 model shows lower energy consumption.

4.1 RQ1 Analysis: Test Case Generation

Figure 4 shows that accuracy decreases as temperature increases. In addition, it can be seen that the models perform best at temperatures of 0.1 and 0.4.

Moving on to the top_p parameter, the impact of varying its values is relatively limited. A slight increase in accuracy is observed around 0.8 and 1.0, but the impact is not substantial.

Regarding the max tokens setting, while some fluctuation is observed, the overall effect of max tokens changes is slight. For the CodeLlama and Deepseek models, the accuracy is at its lowest at a maximum of 250 tokens.

In terms of energy consumption, changes in these configurations do not significantly affect energy consumption.

Finally, Figure 2 shows the trade-off between energy usage and accuracy for test case generation. The large Pareto frontier suggests that many good configurations balance accuracy and energy efficiency. The qwen2.5-coder:7b model (blue points), especially in Q8 bit-width, performs best overall, with accuracy close to 0.5. While deepseek-coder-v2:latest (red points) has the lowest energy consumption among the models tested, it does perform worse than Codegemma and Qwen. Some of its configurations obtain accuracy in the 0.30–0.35 range, but use much less energy with values between 12 and 13 Wh.

4.2 RQ2 Analysis: Bug Fixing

Turning to bug fixing, Figure 5 also shows a progressive decrease in accuracy as temperature increases. The accuracy peaks at 0.1, 0.2, and 0.4, which is similar to the test case generation results.

Comparable to test case generation, the influence of different top_p values remains marginal overall. The best top_p value lies somewhere between 0.8 and 1.0.

As for the max tokens setting, adjusting the max tokens does not result in very conclusive results. The lowest accuracy across models is observed at 250 max tokens.

When it comes to energy usage, the overall impact of changing these parameters is not substantial. Nevertheless, a small increase can be identified for some models at a temperature of 2.0.

Lastly, Figure 3 shows that the Pareto frontier for bug fixing is quite short. This means it's harder to find accurate and energy-efficient configurations. In most cases, improving one comes at the cost of the other. It also suggests that the models may already be close to their performance limits for this task, given the current setup.

All results of the task case generation and bug fixing experiments can be found in Appendix B.

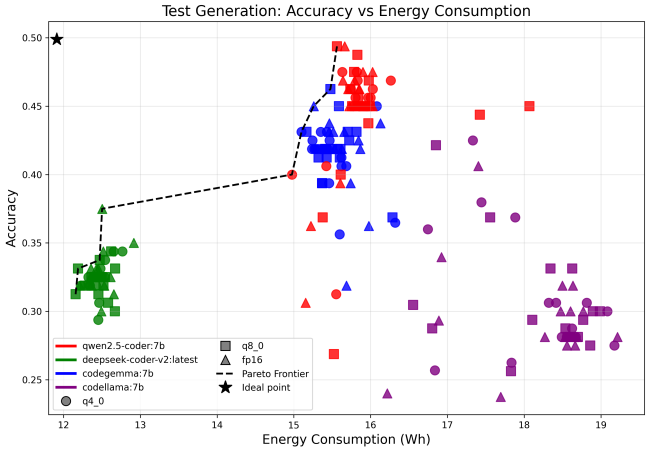

Fig. 2. Test case generation accuracy vs energy consumption for all hyperparameter configurations across all models and bit-widths. Each point represents a specific combination of temperature, top-p, and max tokens values.

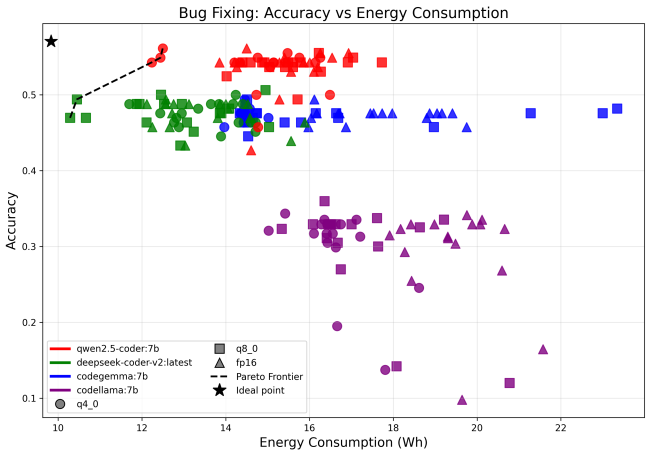

Fig. 3. Bug fixing accuracy vs energy consumption for all hyperparameter configurations across all models and bit-widths. Each point represents a specific combination of temperature, top-p, and max tokens values.

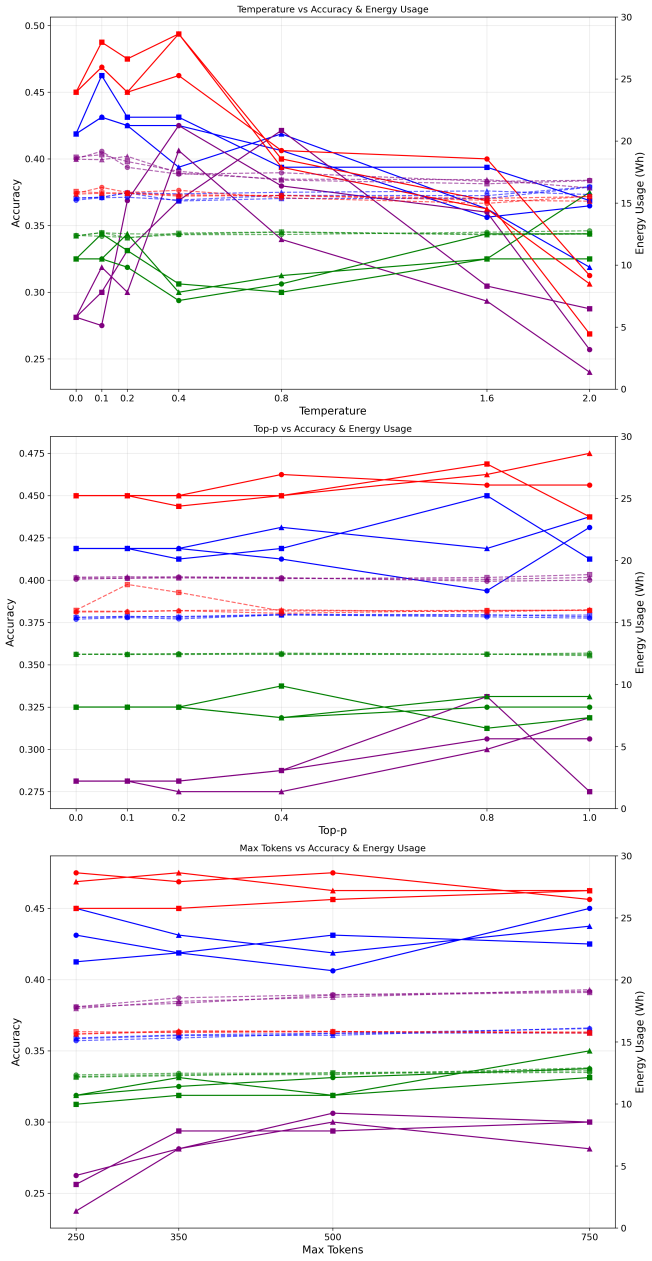

Fig. 4. Test case generation performance across all hyperparameter values: temperature, top_p, and max tokens. Lines show performance across all models and bit-widths for each parameter value.

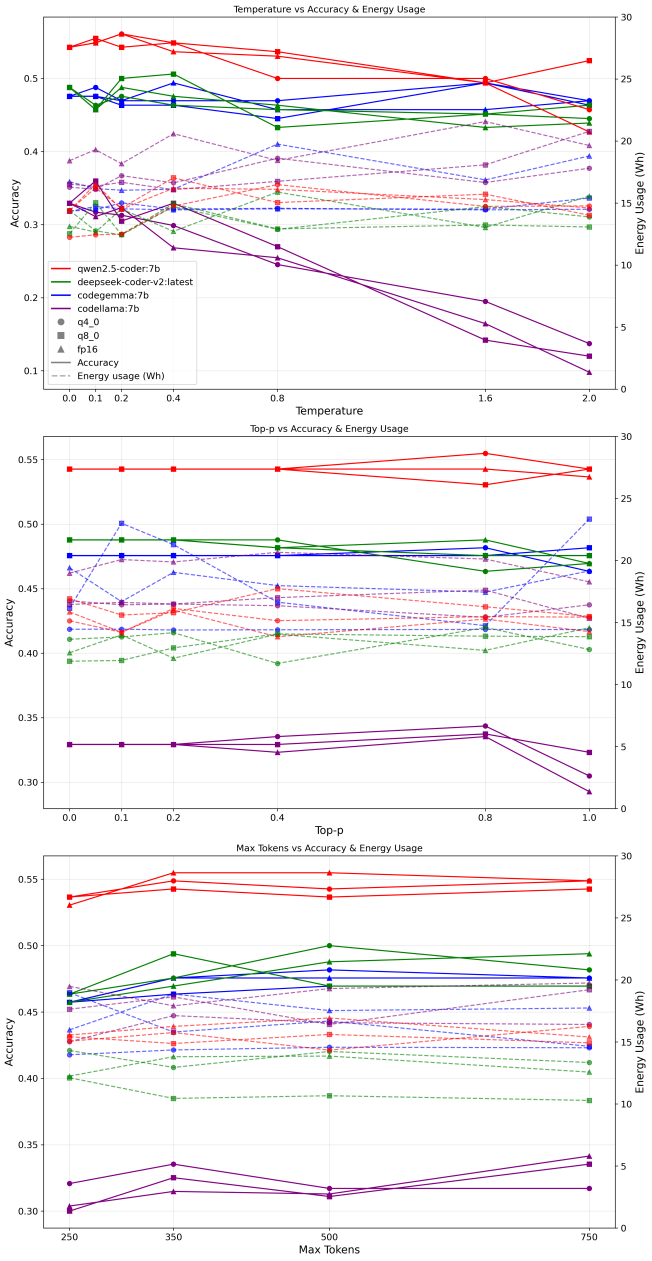

Fig. 5. Bug fixing performance across all hyperparameter values: temperature, top_p, and max tokens. Lines show performance across all models and bit-widths for each parameter value.

4.3 RQ3 Analysis: Method description inclusion

As shown in Table 3, the inclusion of the method description significantly improved performance across all metrics. The results show that the accuracy increases dramatically by 47.71% relative improvement (from 44.51% to 65.74% absolute accuracy) across all configurations. The improvement varied by model, with Qwen2.5-Coder 7B showing the largest gain (+53.1%), followed by DeepSeek-Coder v2 (+50.1%), while CodeGemma 7B and CodeLlama 7B both achieved approximately +42% improvements. To evaluate how well a model balances performance and energy use, we define an efficiency metric that captures the trade-off between task accuracy and energy consumption. Specifically, the metric combines the accuracy with the logarithmically normalized inverse of energy usage, thereby rewarding configurations that solve more problems while consuming less power. A model is considered optimal when it maximizes this efficiency metric. Under this definition, the Qwen2.5-Coder model achieves optimal efficiency when using FP16 quantization with temperature 0.1, top_p 0.95, and max tokens of 500, 750, 350, or 250, ranked in descending order of performance. DeepSeek-Coder v2 performs best with either FP16 or quantized models (Q4, Q8) at temperature settings between 0.1-0.4 and top_p 0.95. CodeGemma shows the highest efficiency with Q4 quantization, temperature 0.1, and lower top_p values (0.2-0.4). CodeLlama's optimal configuration uses FP16 quantization with temperature 0.0-0.1 and top_p 0.95, though with notably lower accuracy than other models. The complete ranking of the top five configurations for each model family, including detailed parameter settings and performance metrics, is presented in Appendix C.

Interestingly, the energy consumption for the tests with method descriptions included was 2.37% lower on average (15.27 Wh vs 15.64 Wh). Unlike for bug fixing without a method description, the results with a method description, shown in Figure 6, have a way larger Pareto frontier, indicating that more optimal configurations are possible when the method description is included.

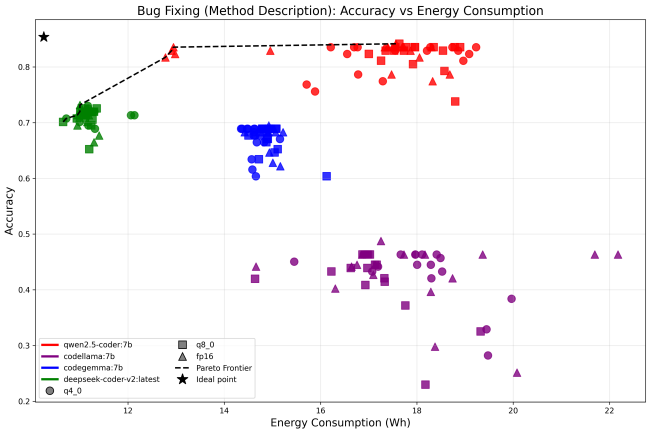

Fig. 6. Bug fixing with method description accuracy vs energy consumption for all hyperparameter configurations across all models and bit-widths. Each point represents a specific combination of temperature, top-p, and max tokens values.

Metric	Without method descriptions	With method descriptions	+/- %
Overall Performance			
Accuracy (%)	44.51	65.74	+47.71%
Energy Consumption (Wh)	15.64	15.27	-2.37%
Inference Time (s)	491.48	463.71	-5.65%
Compilation Failure Rate (%)	5.34	3.84	-28.09%
Efficiency	0.119675	0.178440	+49.10%
Model-wise Accuracy (%)			
CodeGemma 7B	47.3	67.3	+42.3%
CodeLlama 7B	29.7	42.4	+42.8%
DeepSeek-Coder v2	47.4	71.2	+50.1%
Qwen2.5-Coder 7B	53.6	82.0	+53.1%
Model-wise Energy Consumption (Wh)			
CodeGemma 7B	16.27	14.84	-8.8%
CodeLlama 7B	17.75	17.74	-0.1%
DeepSeek-Coder v2	13.29	11.15	-16.1%
Qwen2.5-Coder 7B	15.25	17.34	+13.7%

Table 3. Comparative Analysis: Bug Fixing Performance With vs Without Method Descriptions. The values in the table are averages of all configurations.

5 DISCUSSION

5.1 Hyperparameter Effects

Across both tasks, temperature emerged as the most impactful hyperparameter, with higher values consistently lowering accuracy. In contrast, top-p and max tokens had relatively minor effects. This suggests that randomness in generation plays a larger role in performance than output length or sampling cutoff, at least in the context of Java code generation. These findings are in line with earlier work that also identified temperature as the dominant factor influencing accuracy and energy behavior in code generation models [4, 8]. Additionally, previous studies found that lower temperatures, especially around 0.1, tend to produce better results, which influenced the default settings in this study. While the high accuracy at 0.1 aligns with these expectations, the relatively strong performance at 0.4 was less anticipated, suggesting that a small amount of randomness may help in certain scenarios. For the CodeLlama models, the accuracy slightly decreases at a max of 250 tokens. The lowest accuracy for the max tokens configuration was observed at 250. This is attributable to a rise in compilation failures. The model lacked sufficient tokens to output the full method, and as a result, the generated code failed to compile. Detailed pass/fail statistics for each configuration, also for bug fixing, are available in Appendix D. Besides the compilation errors, the max tokens value has minimal impact on energy use or accuracy. As indicated by the figures, energy usage remains relatively constant across configurations. This suggests that the choice of model seems to make a much larger difference in the energy consumption than the hyperparameters themselves.

When comparing the two tasks, test case generation and bug fixing, distinct differences in trade-offs emerge. Test case generation

showed a broader Pareto frontier, indicating that many configurations achieved a favorable balance between energy consumption and accuracy. In contrast, bug fixing showed fewer such optimal trade-offs, making it more difficult to improve one metric without negatively affecting the other. Additionally, energy consumption during test case generation was more consistent across models, whereas bug fixing revealed clearer patterns in accuracy across different configurations.

5.2 Effect of Java as Target Language

As all referenced studies focused on the generation of Python code, the effect that the use of Java has may explain why some energy or accuracy trends observed in this study differ from previous work. Java's more verbose syntax and stricter type requirements might also lead to longer generated outputs, possibly influencing both compilation success and energy consumption.

5.3 Model Performance and Size

DeepSeek Coder, despite being the largest model in the experiment (15.7B parameters), consistently showed lower accuracy than some of the smaller 7B models. Interestingly, it also consumed less energy on average, making it one of the most energy-efficient models overall. This raises questions about how we define efficiency: while DeepSeek uses less energy, a significant portion of its computation results in unsuccessful outputs, suggesting inefficiency in terms of task performance. Nonetheless, this observation supports previous findings [2], indicating that larger models do not always outperform smaller ones in terms of accuracy. However, the result diverges from prior work on energy consumption. [2] reported that increasing parameter count typically increases energy usage—a trend not seen here. DeepSeek appears to be optimized for inference efficiency, possibly at the cost of output quality, making it a good option when low energy usage is important. Overall, this suggests that model size is not a reliable predictor of either performance or energy consumption.

Previous findings by [2, 8] suggest that low-bit-width models can achieve similar accuracy with reduced energy consumption. However, in this study, while Q8 and Q4 models did show comparable accuracy to FP16 in the bug fixing task, the expected reduction in energy consumption was not observed. Instead, energy usage remained very similar across bit-widths for both bug fixing and test case generation. This deviation from earlier work indicates that the energy-saving potential of quantization may depend heavily on the task type, model architecture, or experimental conditions.

5.4 Prompt Composition: Method Description Impact

Including the method description inside the prompt significantly affects the accuracy of bug fixing, as was also concluded by previous research [18], and slightly decreases the energy consumption of the models, creating a win-win scenario of better accuracy with reduced energy usage. However, it raises the question of whether the model fixed the buggy code with the help of the method description or simply generated a new, correct method based on the method description alone. Since the models can produce correct methods from the method description itself, this distinction is difficult to verify and will need further research.

Beyond the overall benefit of including method descriptions, the top-performing configurations (Appendix C) show that optimal settings vary by model. For example, while Qwen2.5-Coder performed best with FP16, others like DeepSeek and CodeGemma achieved high efficiency with quantized versions. This highlights that model-specific tuning remains important, even when prompt enhancements yield improvements.

5.5 Model Limitations

Even though the LLMs were asked only to return the code and no other explanation, the models did not always follow this. As a result, unnecessary tokens were sometimes outputted. This could have affected the results, since outputting these tokens is wasted energy.

Because a fair comparison between the models was desired, the prompts were not specifically fine-tuned for each model. This may have resulted in suboptimal performance for some models.

5.6 Experimental Environment Limitations

The GPU used for the tests needed to remain idle, but the shared environment allowed others to access it even while it was in use. As a result, the tests had to be restarted multiple times and were not all executed in a single run, which could have affected the reliability of the results.

Moreover, due to time constraints and limited idle access to the GPU, the number of tested configurations had to be reduced, and some irregular jumps in parameter values were necessary. Also, normally, each configuration would be run at least three times to account for measurement variability. If that had been possible, it would have made the results more reliable by reducing the impact of outliers.

Additionally, it was not possible to monitor CPU usage during the experiments. Although the LLMs were executed on the GPU, background activity on the CPU, caused by other users, could have affected the energy measurements, introducing noise in the results. If CPU usage had been trackable, a similar safeguard could have been applied, terminating the experiment when external activity was detected, just as was done for the GPU.

5.7 Measurement Limitations

For test case generation, to ensure that the tests compiled without errors, the decision was made to include the method and inside it three comments stating "Test case that evaluates to boolean (true if correct)." These comments were added to give the model some direction and to prevent it from generating too many test cases, which could cause it to run out of tokens before completing the method, resulting in a compile error. However, by guiding the models to only generate three assertions, the max tokens parameter might not have been fully tested. Since the model was not free to generate many more tokens when the configuration of the max tokens was changed, the results in Figure 4 confirm this. This lower freedom

also led to more compact energy consumption across the different configurations, which can be seen when comparing Figures 2 and 3.

Additionally, the models were asked in the prompt to return the full Java method. It would have been better to only ask for the assertions, which would have saved tokens and reduced energy usage.

6 CONCLUSION

This research highlights several key findings regarding hyperparameter tuning in local LLMs for Java-based test case generation and bug fixing. While the study primarily focused on hyperparameter tuning, it was also observed that incorporating the method description during bug fixing led to the most substantial performance improvement, significantly increasing accuracy while slightly reducing energy consumption. This suggests that prompt content can have a greater effect than hyperparameter values alone.

Furthermore, temperature was identified as the most influential hyperparameter across both tasks; lower temperatures (around 0.1) consistently yielded higher accuracy. In contrast, top-p and max tokens had only minor effects. Nevertheless, ensuring that the max tokens value is high enough remains crucial to avoid premature output termination and compilation errors.

Interestingly, while larger models like DeepSeek Coder did not deliver the best accuracy, they consumed the least energy, indicating that size does not directly correlate with performance. Similarly, lower-bit-width models (Q8 and Q4) demonstrated comparable performance to FP16 models. However, the expected reduction in energy consumption was not observed, especially in test case generation, indicating that the benefits of quantization may vary depending on the task.

In general, test case generation allowed for more flexible trade-offs between energy and accuracy, while bug fixing proved more sensitive to configuration changes and showed clearer performance patterns across models.

In short, bug fixing benefits most from including the method description, while for test case generation, tuning the temperature gives the best results. These findings can help make the use of local LLMs both more effective and more energy efficient.

6.1 Future Work

A key direction for future research is understanding why energy usage remained relatively constant across different bit-widths in this setup, despite prior studies showing clearer energy gains from quantization. Investigating this discrepancy could help determine whether the task type, model architecture, or measurement method played a role.

Another promising area is the exploration of hyperparameter settings, particularly the temperature value around 0.4, which showed unexpectedly strong performance. Similarly, the optimal top_p value likely lies somewhere between 0.8 and 1.0, but could not be determined precisely. While this study lacked the resources to investigate this region in detail, further experiments could help pinpoint more precise optimal values.

It would also be worthwhile to assess whether the observed effects of hyperparameter tuning and energy consumption hold for larger language models with more parameters. Scaling up could reveal new trends or confirm the patterns seen in smaller models.

Additionally, given that the models were used to generate Java code, a less commonly studied target language, the impact of Java's syntax and structure on both performance and energy usage remains unclear. Targeted studies focusing on Java code generation could clarify how language characteristics influence results.

As discussed earlier, it remains uncertain whether the models relied more on the method description or the buggy code when generating their output. Future work could investigate what parts of the input prompt LLMs use to make decisions. Building on this, researchers could also study how different prompt structures or phrasings affect model performance and energy use.

Finally, since including the method description significantly improved accuracy in the bug fixing task, it would be interesting to test whether similar gains can be achieved in test case generation by incorporating this element.

REFERENCES

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, Wannita Takerngsaksiri, Chetan Arora, and Aldeida Aleti. 2025. Enhancing Large Language Models for Text-to-Testcase Generation. arXiv:2402.11910 [cs.SE] https://arxiv.org/abs/2402.11910

[2] Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. 2024. Language Models in Software Development Tasks: An Experimental Analysis of Energy and Accuracy. *arXiv preprint arXiv:2412.00329* (2024). https://arxiv.org/abs/2412.00329v2

[3] Chetan Arora, Ahnaf Ibn Sayeed, Sherlock Licorish, Fanyu Wang, and Christoph Treude. 2024. Optimizing Large Language Model Hyperparameters for Code Generation. arXiv:2408.10577 [cs.SE] https://arxiv.org/abs/2408.10577

[4] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11

[5] Alex de Vries. 2023. The Growing Energy Footprint of Artificial Intelligence. *Joule* 7, 10 (2023), 2191–2194. https://doi.org/10.1016/j.joule.2023.09.004

[6] Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu. 2024. A Survey of Low-bit Large Language Models: Basics, Systems, and Algorithms. arXiv:2409.16694 [cs.AI] https://arxiv.org/abs/2409.16694

[7] Soka Hisaharo, Yuki Nishimura, and Aoi Takahashi. 2024. Optimizing LLM Inference Clusters for Enhanced Performance and Energy Efficiency. TechRxiv. https://doi.org/10.36227/techrxiv.172348951.12175366/v1 Cite as: Soka Hisaharo, Yuki Nishimura, Aoi Takahashi. TechRxiv. August 12, 2024..

[8] Quirijn Hoenink. 2025. Optimizing Code Generation Models Efficiency Through Hyperparameter Tuning. http://essay.utwente.nl/105077/

[9] Hugging Face. 2025. BigCode Models Leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard. Accessed: 2025-05-01.

[10] NVIDIA Corporation. 2025. NVIDIA Management Library (NVML). https://developer.nvidia.com/management-library-nvml Accessed: 2025-05-19.

[11] Ollama. 2025. Ollama Library. https://ollama.com/library. Accessed: 2025-05-01.

[12] OpenAI. 2025. OpenAI API Reference - Temperature. https://platform.openai.com/docs/api-reference/responses/create#responses-create-temperature Accessed: 2025-05-08.

[13] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2024. An Empirical Study of the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2024). Version 2, October 2024.

[14] Rajvardhan Patil and Venkat Gudivada. 2024. A Review of Current Trends, Techniques, and Challenges in Large Language Models (LLMs). *Applied Sciences* 14, 5 (2024). https://doi.org/10.3390/app14052074

[15] Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. 2024. Is Temperature the Creativity Parameter of Large Language Models? arXiv:2405.00492 [cs.CL] https://arxiv.org/abs/2405.00492

[16] Matthew Renze. 2024. The Effect of Sampling Temperature on Problem Solving in Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 7346–7356. https://doi.org/10.18653/v1/2024.findings-emnlp.432

[17] Jovan Stojkovic, Esha Choukse, Chaojie Zhang, Inigo Goiri, and Josep Torrellas. 2024. Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference. arXiv:2403.20306 [cs.AI] https://arxiv.org/abs/2403.20306

[18] Xiaolong Tian. 2024. Evaluating the Repair Ability of LLM Under Different Prompt Settings. In *2024 IEEE International Conference on Software Services Engineering (SSE)*. 313–322. https://doi.org/10.1109/SSE62657.2024.00053

[19] Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. 2023. Cost-Effective Hyperparameter Optimization for Large Language Model Generation Inference. In *Proceedings of the Second International Conference on Automated Machine Learning (Proceedings of Machine Learning Research, Vol. 224)*, Aleksandra Faust, Roman Garnett, Colin White, Frank Hutter, and Jacob R. Gardner (Eds.). PMLR, 21/1–17. https://proceedings.mlr.press/v224/wang23b.html

APPENDICES

During the preparation of this work, I used Cursor to help me set up the experiment and create the graphs with the results of the experiment. I also used ChatGPT to help refine the flow of the text, particularly by suggesting appropriate linking words. After using these tools/services, I thoroughly reviewed and edited the content as needed, taking full responsibility for the final outcome.

A APPENDIX: PROMPT TEMPLATES

Example prompts from the first HumanEvalPack test used as input to the models.

1. Test Case Generation Prompt

```
You are an AI that must output **only** valid Java code.

Generate a `Main.java` file that tests this method:
import java.util.*;

public class Solution {
    public boolean hasCloseElements(List<Double> numbers, double threshold) {
        for (int i = 0; i < numbers.size(); i++) {
            for (int j = i + 1; j < numbers.size(); j++) {
                double distance = Math.abs(numbers.get(i) - numbers.get(j));
                if (distance < threshold) return true;
            }
        }
        return false;
    }
}

Follow this exact format and DO NOT include ANY explanations, comments, or text:

public class Main {
    public static void main(String[] args) {
        Solution s = new Solution();
        List<Boolean> correct = Arrays.asList(
            // Test case that evaluates to boolean (true if correct)
            // Test case that evaluates to boolean (true if correct)
            // Test case that evaluates to boolean (true if correct)
        );
        if (correct.contains(false)) {
            throw new AssertionError();
        }
    }
}
```

Listing 1. Test Case Generation Prompt

2. Bug Fixing Prompt

```
You are an AI that outputs **only** corrected Java methods exactly as specified.
Fix the bug in the following Java method and return **only the fixed method** exactly as it should appear
inside a class — no main method, no import statements, no explanations, and no extra wrapping.

Buggy Java method:
public boolean hasCloseElements(List<Double> numbers, double threshold) {
    for (int i = 0; i < numbers.size(); i++) {
        for (int j = i + 1; j < numbers.size(); j++) {
            double distance = numbers.get(i) - numbers.get(j);
            if (distance < threshold) return true;
```

```
        }
    }
    return false;
}

Bug description: missing logic
Failure symptoms: incorrect output
```

Listing 2. Bug Fixing Prompt

3. Bug Fixing with method description Prompt

```
You are an AI that outputs **only** corrected Java methods exactly as specified.
Fix the bug in the following Java method and return **only the fixed method** exactly as it should appear
inside a class — no main method, no import statements, no explanations, and no extra wrapping.

Buggy Java method:
public boolean hasCloseElements(List<Double> numbers, double threshold) {
    for (int i = 0; i < numbers.size(); i++) {
        for (int j = i + 1; j < numbers.size(); j++) {
            double distance = numbers.get(i) - numbers.get(j);
            if (distance < threshold) return true;
        }
    }
    return false;
}

Bug description: missing logic
Failure symptoms: incorrect output

And here is the docstring of the method:
Check if in given list of numbers, are any two numbers closer to each other than given threshold.
>>> hasCloseElements(Arrays.asList(1.0, 2.0, 3.0), 0.5)
false
>>> hasCloseElements(Arrays.asList(1.0, 2.8, 3.0, 4.0, 5.0, 2.0), 0.3)
true
```

Listing 3. Bug Fixing with method description Prompt

B APPENDIX: ALL RESULTS

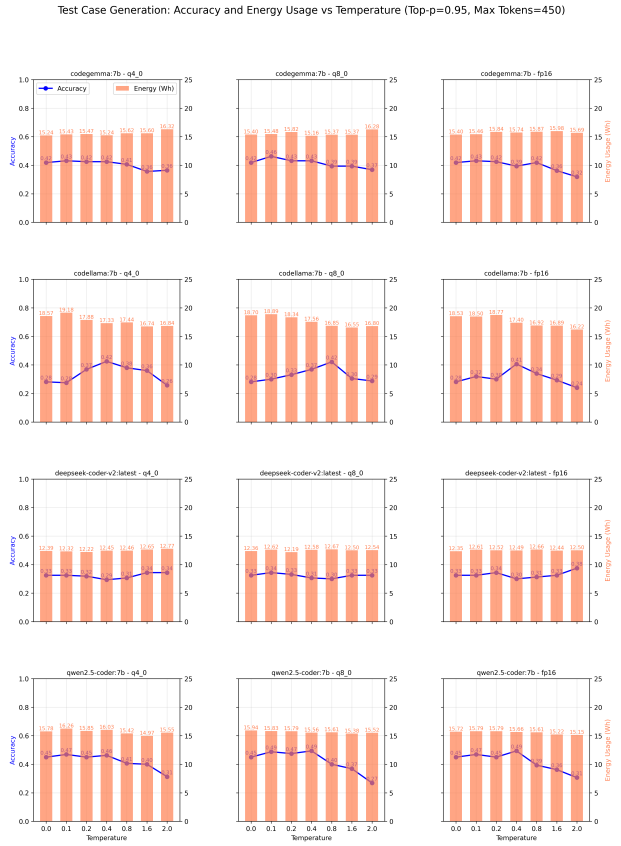

Fig. 7. Test case generation: Temperature results for all models and bit-widths across temperature values [0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 2.0]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

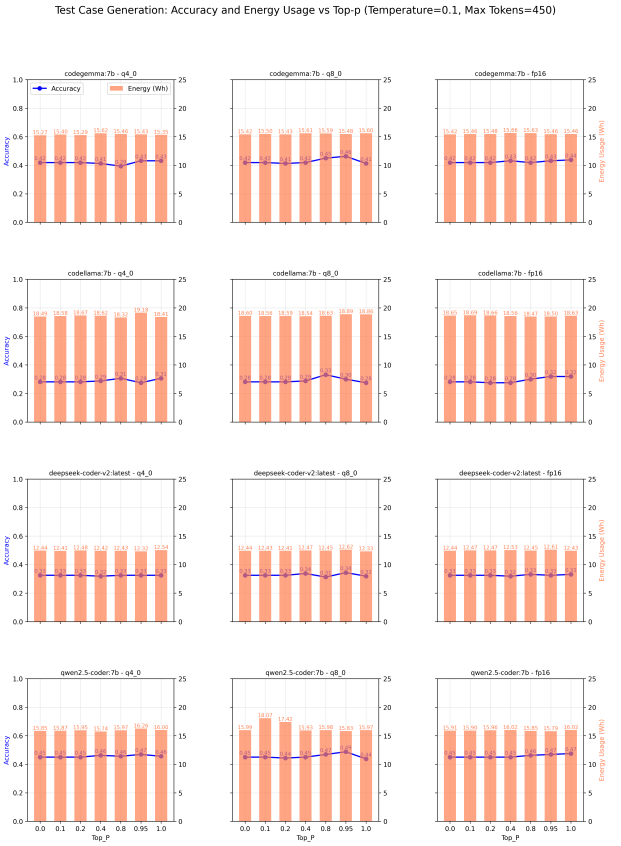

Fig. 8. Test case generation: Top_p results for all models and bit-widths across top-p values [0.0, 0.1, 0.2, 0.4, 0.8, 1.0]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

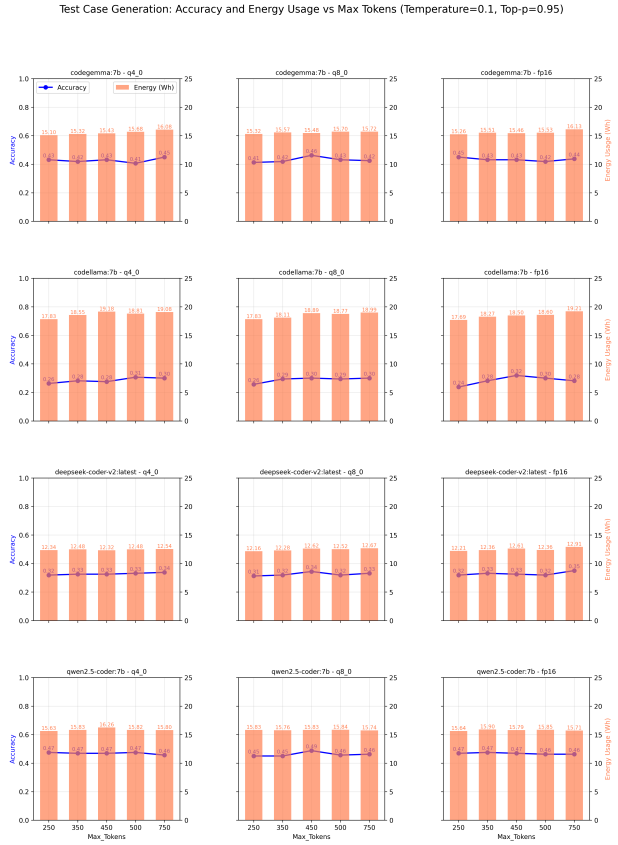

Fig. 9. Test case generation: Max tokens results for all models and bit-widths across max tokens values [250, 350, 500, 750]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

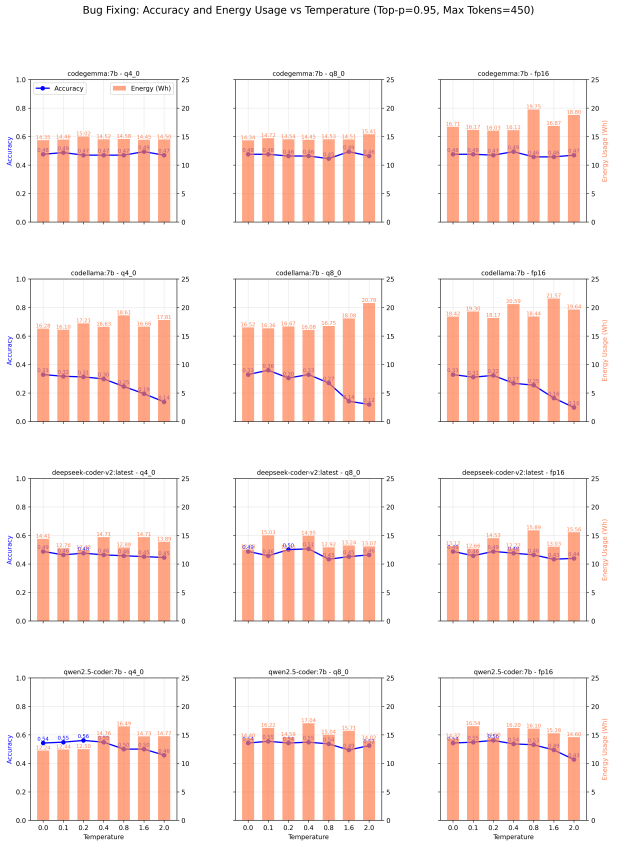

Fig. 10. Bug fixing: Temperature results for all models and bit-widths across temperature values [0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 2.0]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

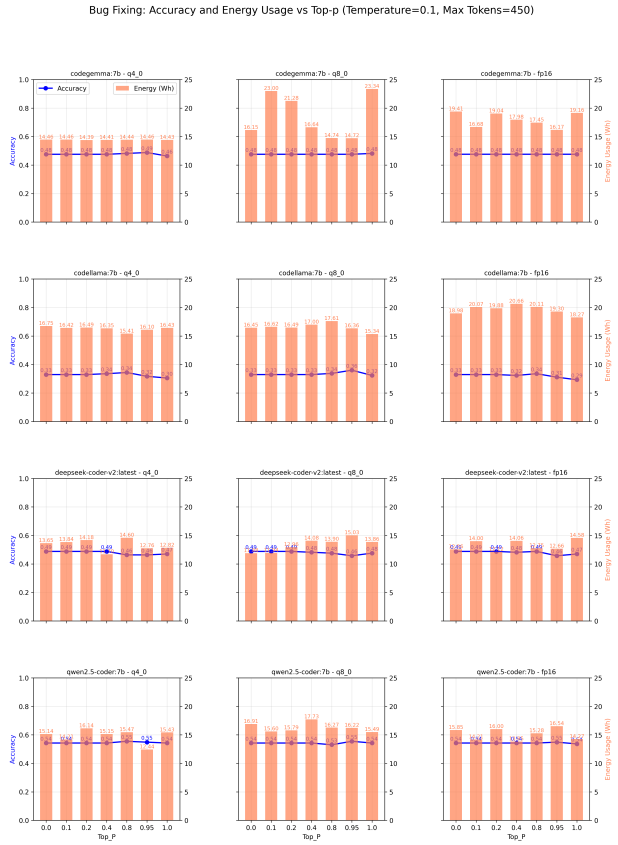

Fig. 11. Bug fixing: Top_p results for all models and bit-widths across top-p values [0.0, 0.1, 0.2, 0.4, 0.8, 1.0]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

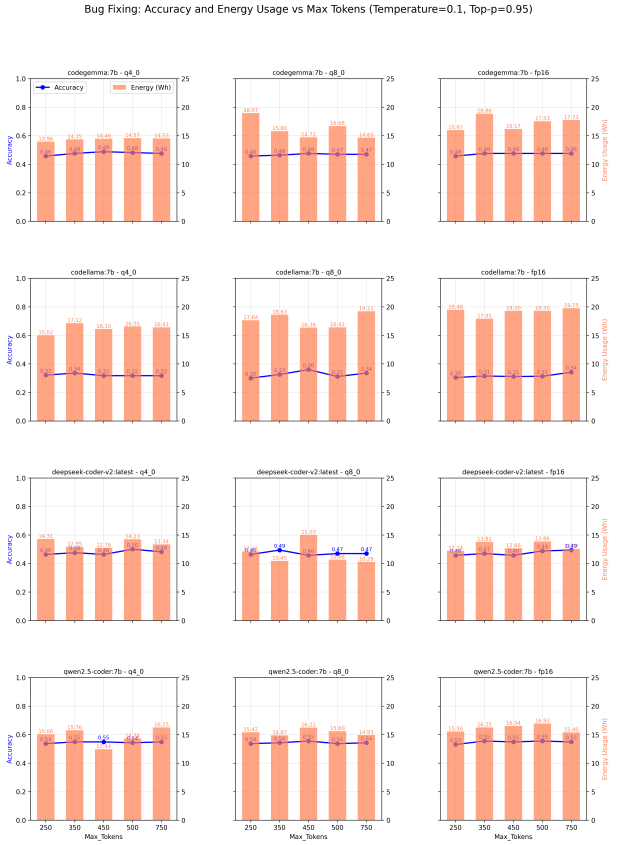

Fig. 12. Bug fixing: Max tokens results for all models and bit-widths across max tokens values [250, 350, 500, 750]. Each subplot shows accuracy and energy consumption for a specific model-bit-width combination.

C APPENDIX: TOP 5 CONFIGURATIONS PER MODEL FAMILY FOR BUG FIXING WITH METHOD DESCRIPTION

Model	Rank	Bit-Width	Temp.	Top-p	Max Tok.	Accuracy (%)	Energy (Wh)	Time (s)	Efficiency
qwen2.5-coder:7b	1	fp16	0.1	0.95	500	83.5	12.944	387.2	0.2346
qwen2.5-coder:7b	2	fp16	0.1	0.95	750	82.9	12.908	386.6	0.2331
qwen2.5-coder:7b	3	fp16	0.1	0.95	350	82.3	12.980	389.5	0.2310
qwen2.5-coder:7b	4	fp16	0.1	0.95	250	81.7	12.777	382.4	0.2303
qwen2.5-coder:7b	5	fp16	0.1	1.0	450	82.9	14.955	452.3	0.2238
deepseek-coder-v2:latest	1	fp16	0.2	0.95	450	73.2	11.004	343.1	0.2153
deepseek-coder-v2:latest	2	q4_0	0.4	0.95	450	73.0	11.166	346.8	0.2139
deepseek-coder-v2:latest	3	q8_0	0.1	0.95	750	72.6	11.040	341.8	0.2133
deepseek-coder-v2:latest	4	fp16	0.1	0.95	750	72.6	11.140	350.2	0.2128
deepseek-coder-v2:latest	5	fp16	0.4	0.95	450	72.6	11.167	350.0	0.2126
codellama:7b	1	fp16	0.1	0.95	450	48.8	17.253	522.5	0.1268
codellama:7b	2	fp16	0.0	0.95	450	46.3	16.861	513.2	0.1212
codellama:7b	3	q8_0	0.0	0.95	450	46.3	16.872	517.3	0.1211
codellama:7b	4	q4_0	0.0	0.95	450	46.3	16.893	518.0	0.1211
codellama:7b	5	q8_0	0.1	0.1	450	46.3	16.915	515.2	0.1211
codegemma:7b	1	q4_0	0.1	0.2	450	68.9	14.348	413.4	0.1881
codegemma:7b	2	q4_0	0.1	0.4	450	68.9	14.369	412.7	0.1880
codegemma:7b	3	fp16	0.8	0.95	450	69.5	14.926	434.4	0.1877
codegemma:7b	4	q4_0	0.1	0.95	350	68.9	14.473	419.8	0.1876
codegemma:7b	5	q4_0	0.1	0.0	450	68.9	14.489	418.5	0.1876

Table 4. Top 5 configurations per model family for bug fixing with the method description included in the prompt

D APPENDIX: PASS, FAIL, COMPILE FAIL GRAPHS

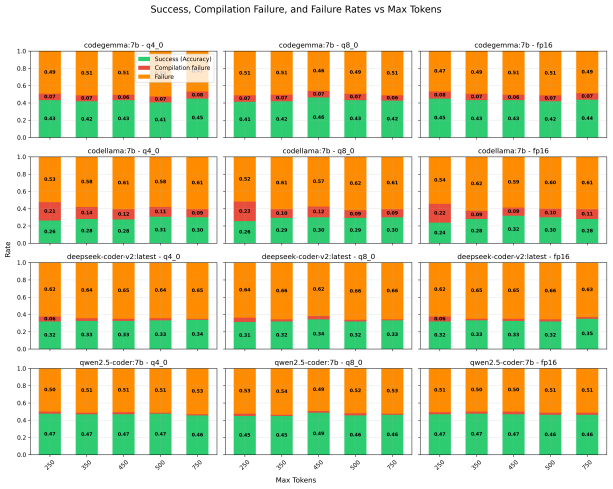

Fig. 13. Test case generation: Max tokens variation showing pass/fail/-compile fail rates across all models and bit-widths for max tokens values [250, 350, 500, 750].

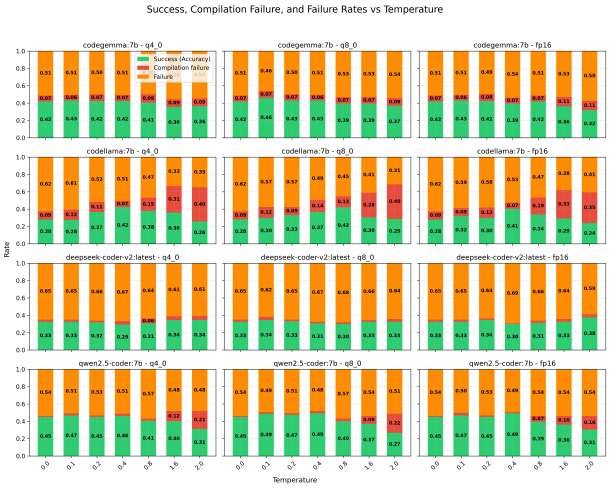

Fig. 14. Test case generation: Temperature variation showing pass/-fail/compile fail rates across all models and bit-widths for temperature values [0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 2.0].

Fig. 15. Test case generation: Top-p variation showing pass/fail/compile fail rates across all models and bit-widths for top-p values [0.0, 0.1, 0.2, 0.4, 0.8, 1.0].

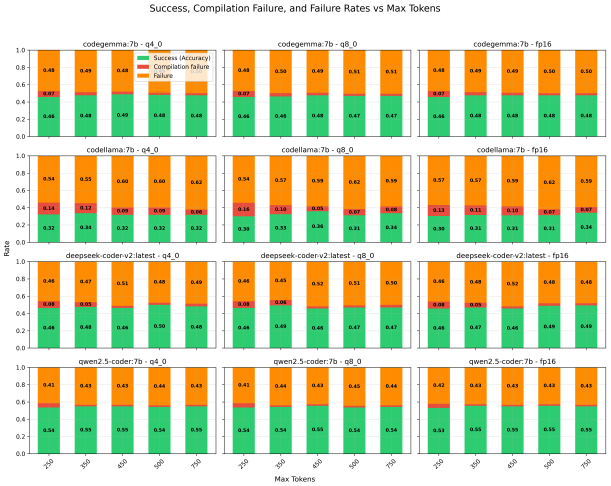

Fig. 16. Bug fixing: Max tokens variation showing pass/fail/compile fail rates across all models and bit-widths for max tokens values [250, 350, 500, 750].

Fig. 17. Bug fixing: Temperature variation showing pass/fail/compile fail rates across all models and bit-widths for temperature values [0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 2.0].

Fig. 18. Bug fixing: Top-p variation showing pass/fail/compile fail rates across all models and bit-widths for top-p values [0.0, 0.1, 0.2, 0.4, 0.8, 1.0].