

Bigger Is Not Always Better: Performance and Sustainability Trade-Offs in LLMs for Python Bug-Fixing Tasks

Syed Fakhar Abbas Naqvi  

University of Tartu, Estonia

Hina Anwar  

University of Tartu, Estonia

Abstract

Background. Large Language Models (LLMs) are increasingly used in software engineering tasks such as code generation, bug detection, and program repair. However, their use introduces inference-time costs that are rarely considered together with functional performance, especially in debugging and repair workflows where models may be invoked repeatedly.

Aim. This study evaluates the correctness-energy trade-offs of LLMs across model scales in Python bug-fixing tasks. We examine how model size relates to code correctness, inference time, and GPU energy consumption.

Method. We conduct a controlled empirical evaluation of six open-source code-oriented LLMs, ranging from 1.5B to 15B parameters, on 40 real-world Python bugs from the BugsInPy benchmark. Each model is executed multiple times per task under a consistent hardware and prompting setup. Repair performance is measured using Pass@k and test-suite outcomes, while computational cost is measured using inference time and GPU energy consumption. We also analyze generated outputs to identify recurring failure patterns.

Results. Increasing model size substantially raised computational cost without consistently improving correctness. The smallest model, Qwen 1.5B, achieved the best overall performance, with a Pass@1 score of 43% and an average runtime of 38.7 seconds. In contrast, StarCoder 15B achieved a Pass@1 score of 23%, required 96.7 seconds, and consumed approximately 7 times more energy per inference while solving fewer tasks. Higher success rates were also observed in web-related projects, suggesting that structurally localized bugs are more effectively handled by current models.

Conclusion. On the evaluated localized Python bug-fixing tasks, smaller LLMs offered a more favorable trade-off between correctness, inference time, and energy consumption. These results suggest that scaling model size does not automatically lead to more efficient LLM-based repair, and that inference cost should be considered alongside functional correctness when comparing models.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Empirical software validation; Computing methodologies → Artificial intelligence

Keywords and phrases Large Language Models, Bug Fixing, Energy Consumption, Empirical Software Engineering, Sustainability

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.34

Category Technical Track Paper

Supplementary Material *Software (Source Code):*

<https://github.com/syedfakhar25/empirical-llm-bugfix-benchmark/tree/main> [15]

archived at [swh:1:dir:e2f5ea38e5e8b633db8ef0baf818d58b194468a9](https://www.swh.io/dir/e2f5ea38e5e8b633db8ef0baf818d58b194468a9)

Acknowledgements This work was carried out as part of the first author's thesis research.



© Syed Fakhar Abbas Naqvi and Hina Anwar;
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. 34; pp. 34:1–34:20



Leibniz International Proceedings in Informatics

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

1 Introduction

Large Language Models (LLMs) are increasingly used in software engineering tasks, including code generation, bug detection, automated debugging, and program repair [12]. Code-oriented LLMs can generate patches, complete missing code, and assist developers in reasoning about faulty program behavior. As these models become more accessible through local deployment and integrated development tools, they are likely to be invoked repeatedly during software development workflows.

This growing use of LLMs raises an important sustainability concern. AI models consume substantial energy during training and continue to consume energy during inference after deployment [2], [8], [9], [22]. For example, the energy consumed during LLaMA 3.1 training has been compared to the emissions from 83 years of electricity use by a single household in the United States [2]. Prior work has also reported that training a large neural network can produce emissions comparable to those of multiple cars over their lifetimes [9]. With the increasing demand for large-scale AI systems, data centers are projected to consume a growing share of global electricity, reaching approximately 20% by 2030 [9].

However, in software engineering workflows, sustainability concerns extend beyond model training. Once deployed, LLMs consume energy during inference, the process by which trained models generate outputs [8], [9], [22]. Inference is particularly relevant in debugging and repair because these tasks are often iterative: developers may request multiple fixes, regenerate patches, or apply models across many defects. As a result, even moderate per-inference costs can accumulate when LLMs are used frequently.

This concern aligns with Green AI, which argues that AI research should consider computational cost alongside model performance [6]. For LLM-based software engineering, this means evaluating not only whether a model produces correct outputs, but also the inference cost required to produce them. Prior studies have examined the environmental impact of large-scale AI systems, the energy efficiency of LLM-generated code, and differences between smaller and larger models in programming tasks [2], [9], [16], [20]. These studies provide useful evidence, but energy measurements are highly dependent on hardware, task configuration, implementation choices, and measurement boundaries [1]. Therefore, results from one type of programming task do not necessarily generalize to other software engineering activities.

Bug fixing is a particularly relevant setting for studying correctness-energy trade-offs. Developers spend substantial time locating, understanding, and repairing defects [25], [21], [23]. Debugging includes several subtasks, such as bug localization, classification, and repair, and different bug types may require different levels of reasoning and contextual understanding [24]. Python bug fixing is also a practical and challenging target. Python is widely used in domains such as data science, machine learning, automation, web development, and backend systems. At the same time, Python bugs can be difficult to repair because of dynamic typing, heterogeneous data structures, and domain-specific library usage [24].

Despite growing interest in Green AI and LLM-based software engineering, limited empirical evidence is available on the inference-time cost of using open-source code LLMs for real-world bug fixing. Much prior work evaluates the efficiency of generated code after it is produced, for example, by measuring the runtime or energy consumption of the generated program. In contrast, this study focuses on the energy consumed by the model during repair generation. This distinction matters because, in LLM-based debugging, sustainability concerns arise from both the code eventually produced and the repeated inference required to obtain it.

In this paper, we investigate the correctness–energy trade-offs of LLM-based Python bug fixing, with particular emphasis on how model size influences both repair effectiveness and inference cost. We evaluate six open-source code-oriented LLMs, ranging from 1.5B to 15B parameters, on 40 real-world bugs from the BugsInPy benchmark under a consistent hardware and prompting setup. Our goal is to determine whether increasing model size improves correctness–cost behavior, or whether smaller models can offer more favorable trade-offs under the evaluated conditions.

The study is scoped to localized bug repair, where the faulty code can be isolated to a single function or class and replaced as a complete block. Repair scenarios that require repository-level reasoning, coordinated edits across multiple files, or a broader architectural understanding are outside the scope of this work. Accordingly, the findings and conclusions should be interpreted in the context of this class of localized repair tasks.

This paper makes the following contributions:

- We provide a controlled empirical comparison of six open-source code-oriented LLMs on real-world Python bug-fixing tasks from BugsInPy.
- We measure repair correctness, inference time, and GPU energy consumption under a consistent experimental setup, making inference cost visible alongside functional correctness.
- We analyze correctness–energy trade-offs across model scales and show that larger models do not necessarily provide more efficient LLM-based repair in the evaluated setting.
- We characterize recurring failure patterns in generated repairs to help explain differences in model behavior.

The rest of the paper is organized as follows. Section 2 discusses related work on Green AI, energy-aware LLM evaluation, and LLM-based software repair. Section 3 presents the research questions and describes the methodology and experimental setup. Section 4 presents the results. Section 5 discusses implications, and Section 6 presents threats to validity. Section 7 concludes the paper.

2 Related Work

Energy-aware software engineering has increasingly examined how software systems, programming practices, and AI-assisted development affect energy consumption. In the context of LLM-assisted programming, recent work has studied both the energy behavior of generated code and the computational cost of using the models themselves. This section summarizes representative studies that motivate our focus on inference-time energy in LLM-based bug fixing.

Rani et al. [17] studied whether LLMs can improve the energy efficiency of existing code. They optimized 400 MATLAB scripts using GPT-3, GPT-4, Llama, and Mixtral, and compared the results with optimizations produced by a senior MATLAB developer. Their evaluation considered energy consumption, memory usage, and other performance metrics. The results showed no clear energy benefit from LLM-based optimization and indicated negative effects on memory usage. Their findings suggest that LLM-assisted code optimization does not automatically lead to more sustainable software and that energy effects must be evaluated empirically.

A prerequisite for such evaluations is reliable energy measurement. Sallou et al. [18] addressed this challenge by introducing EnergiBridge, a cross-platform tool for measuring software energy consumption on Linux, Windows, and macOS. This is relevant because many earlier energy measurement tools were platform-specific, making comparisons difficult

across environments. However, even with improved tooling, energy measurements remain sensitive to hardware, operating system behavior, workload characteristics, and experimental configuration.

Several studies have evaluated the energy efficiency of code generated by LLMs. Islam et al. [9] compared code generated by 20 LLMs with human-written code using energy consumption, runtime, memory usage, and correctness. Their results showed that human-written code was approximately 1.17 times more energy-efficient than DeepSeek-v3 and more efficient than other evaluated LLM outputs. They also found that increasing problem complexity was associated with greater inefficiency in LLM-generated code. These findings show that functional correctness alone is insufficient for evaluating AI-generated code.

Ashraf et al. [2] studied energy-aware code generation across model scales. They compared three small open-source models, StableCode-3B, StarCoderBase-3B, and Qwen2.5-Coder-3B-Instruct, against two large proprietary models, GPT-4.0 and DeepSeek-Reasoner, using 150 Python programming tasks from LeetCode. Their evaluation measured correctness, runtime, memory usage, and energy consumption during execution of the generated code. The results showed that large models achieved stronger correctness and runtime performance, while small models were more energy-efficient in more than half of the cases where their outputs were correct. Their study is closely related to ours because it compares models of different sizes under a sustainability lens. However, their measurement target is the execution cost of generated code, whereas our study measures the inference cost of generating repairs.

This distinction is important because energy consumption in AI-assisted software engineering arises at different points in the workflow. Early Green AI work emphasized the need to account for computational costs across the AI life cycle [6, 22]. More recent work shows that inference can substantially increase the overall energy footprint of deployed AI systems, as models may be executed repeatedly at scale [9]. Fernandez et al. [7] similarly emphasize that inference can represent a major share of energy consumption in deployed AI systems. In LLM-based debugging, this repeated-use concern is especially relevant: developers or tools may request multiple patches, regenerate outputs, or apply models across many defects before arriving at a correct fix.

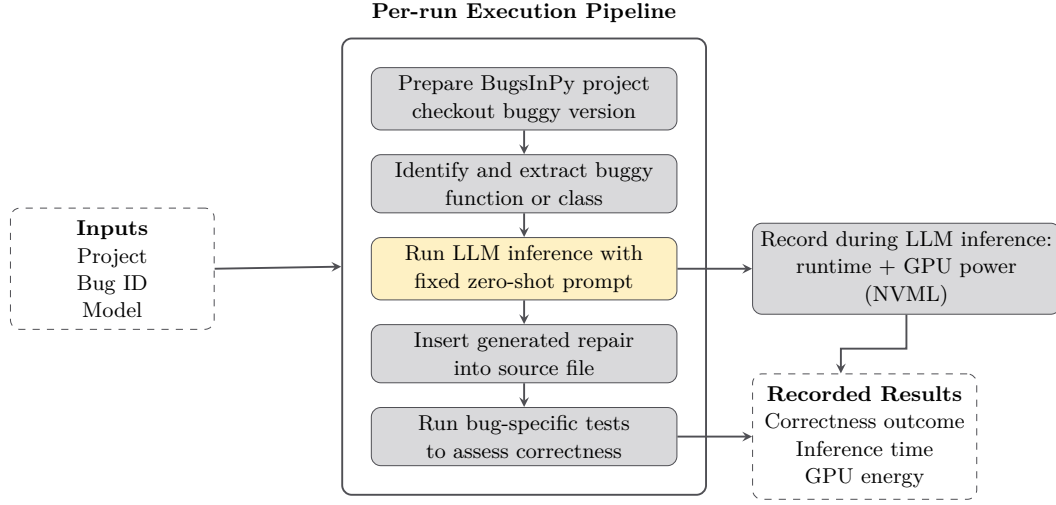
Overall, prior work provides important evidence on software energy measurement, LLM-generated code efficiency, and model-scale comparisons. However, most related studies focus on the energy behavior of generated code after execution or on general programming problems. Less is known about the inference-time energy cost of using open-source code-oriented LLMs for real-world bug-fixing tasks. This study addresses that gap by evaluating six open-source code LLMs of different sizes on Python bugs from BugsInPy, measuring repair correctness, inference time, and GPU energy consumption during patch generation under a consistent experimental setup.

3 Methodology

We conduct a controlled experiment to evaluate open-source code-oriented LLMs of different sizes on Python bug-fixing tasks. The goal is to examine how model size relates to repair correctness, inference time, and GPU energy consumption during repair generation.

3.1 Research Questions and Hypotheses

The study is guided by three research questions:



■ **Figure 1** Per-run experimental workflow. Runtime and GPU power are recorded only during LLM inference; tests are used to determine repair correctness.

RQ1. How do LLMs of varying sizes differ in their ability to fix Python bugs in terms of repair correctness and inference time?

RQ2. What is the inference-time energy cost associated with running these models on Python bug-fixing tasks?

RQ3. Which model provides the most favorable trade-off between correctness, inference time, and energy consumption?

We investigate two hypotheses:

H1. Smaller models can achieve comparable repair correctness to larger models while requiring less inference time and energy consumption.

H2. Larger models require greater energy resources during inference than smaller models, regardless of whether the generated fix is correct.

3.2 Experimental Design

For each experimental run, the pipeline takes a selected model and a selected bug as input. It prepares the buggy project state, extracts the buggy function or class, prompts the model to generate a repair, inserts the generated repair into the source file, and executes the associated tests to determine correctness. Figure 1 summarizes this per-run pipeline.

The study evaluates block-level repair: each generated output replaces the extracted buggy function or class. This supports automated insertion and testing, but does not evaluate repository-wide or multi-location repairs.

Runtime and GPU power are recorded only during LLM inference, from immediately before repair generation until the model output is returned. Repository setup, extraction, prompt construction, patch insertion, and test execution are excluded. Because power is measured using NVML, reported energy values represent GPU-side inference energy, not full-system energy.

■ **Table 1** Overview of selected models.

Identifier	Model Name	Parameters	Size
Qwen1.5	Qwen2.5-Coder-1.5B	1.5B	Small
StableCode3	StableCode-3B	3B	Small
DeepSeek6	DeepSeek-Coder-6.7B-Instruct	6.7B	Small
StarCoder7	StarCoder2-7B	7B	Medium
CodeLlama13	CodeLlama-13B-Instruct	13B	Medium
StarCoder15	StarCoder2-15B	15B	Medium

3.3 Benchmark and Models

Python was selected because of its widespread use in AI, data science, automation, web development, and backend systems [19]. We use BugsInPy [24], a benchmark of real-world Python bugs from open-source GitHub projects. We selected 40 bugs from BugsInPy projects, including Ansible, Black, Cookiecutter, FastAPI, Luigi, Sanic, Scrappy, TQDM, and Youtube-dl. The selected bugs cover several software domains, including web frameworks, automation, developer tools, utilities, and data-processing projects. The selection was constrained by the block-level repair setting used in this study: a bug was included only if the buggy code could be localized to a function or class, the relevant block could be extracted for prompting, and the associated test suite could be executed in our environment. Bugs requiring multi-file repairs, repository-level changes, unavailable dependencies, or non-reproducible test environments were excluded.

We evaluate six open-source code-oriented LLMs ranging from 1.5B to 15B parameters. Models were selected based on open-source availability, Python support, and prior use or performance in code-related benchmarks such as HumanEval [5] and CodeXGLUE [13]. Following prior work [2], we categorize models up to approximately 6–7B parameters as small and models from 7B to 15B parameters as medium-sized. Table 1 summarizes the selected models and their parameter sizes. All selected models are publicly available on the Hugging Face¹ model repository. All models were executed locally on the same hardware. Proprietary API-based models were not included because their inference energy cannot be measured directly under the same hardware and measurement conditions.

3.4 Prompting and Execution

A fixed zero-shot prompt was used for all models and tasks to ensure a consistent evaluation setting. We deliberately used a single standardized prompt rather than model-specific prompting or alternative prompting strategies, such as few-shot prompting. This design reduces prompt engineering as a source of variation and enables the models to be compared under identical prompting conditions [4] [3]. The prompt instructs the model to generate a complete Python function or class and to output only executable code.

¹ <https://huggingface.co/>

```

You are a Python expert.
Fix the following {block_type} named '{block_name}'.
STRICT RULES:
- Output ONLY valid Python code
- DO NOT explain anything
- DO NOT include markdown
- DO NOT include text before or after code
- The output MUST start with: def {block_name} OR class {block_name}
- The code MUST be complete and executable
BUGGY CODE:
{buggy_block}

```

Generated outputs were evaluated without manual intervention. Outputs containing non-code text, Markdown formatting, incomplete blocks, syntax errors, unexpected function or class structures, or missing return statements were counted as unsuccessful repairs (counted as failures in Pass@k). We intentionally did not edit, clean, or reformat generated outputs because adherence to the fixed output constraints is part of the evaluated zero-shot repair setting. For executable repairs, correctness was determined solely by whether the generated repair passed the associated bug-specific test suite.

All models used the same prompt template and were executed without task-specific or model-specific decoding tuning. Generation followed the defaults of the inference library/model configuration used in our scripts. The batch size was 1. Model loading time was excluded from inference-time and energy measurements. Random seeds were not fixed; instead, each bug-model pair was executed 10 times to capture output and measurement variability.

For each bug-model pair, we perform 10 independent runs, resulting in 2,400 intended model executions. Each run takes a project identifier, bug identifier, and model identifier as input; prepares the BugsInPy workspace; extracts the buggy block; runs LLM inference; inserts the generated repair; executes the bug-specific tests using `pytest`; and records correctness, inference time, and GPU energy.

A run is successful if the generated repair passes all bug-specific tests. A bug is considered fixed by a model if at least one of the model's 10 generated repairs passes the tests. Runs failing due to syntax errors, runtime errors, failed tests, invalid output format, or timeout are counted as unsuccessful. The timeout policy was governed by the overall SLURM job time limit.

All experiments are conducted on a dedicated GPU node equipped with an NVIDIA Tesla V100 GPU with 32 GB memory, 4 CPU cores, and 128 GB RAM. Each experiment is executed as an independent job with an identical configuration, and a fixed delay is introduced between repeated runs to reduce thermal effects.

All runs used the same software environment, model execution scripts, and measurement scripts. The models were loaded before each measured run. Model-loading energy and time are excluded from the reported inference cost.

3.5 Measurements and Analysis

Repair correctness is evaluated using execution-based testing. A generated repair is correct if it passes the bug-specific test cases. Since each bug-model pair is evaluated over 10 runs, we use Pass@k to summarize repair success [11, 5]. Pass@1 is computed as:

$$\text{Pass@1} = \frac{\text{Total Successful Attempts}}{\text{Total Attempts}} \quad (1)$$

Pass@1 is computed as the observed proportion of successful repairs across repeated independent runs. For Pass@10, a bug is considered solved for a given model when at least one out of the 10 generated repairs successfully passes the bug-specific test suite.

Inference time is measured using `time.perf_counter`. GPU energy is measured during inference using NVML, with GPU power sampled every 100 ms and integrated over time:

$$E_{\text{measured}} = \sum P(t) \times \Delta t \quad (2)$$

where $P(t)$ is instantaneous GPU power and Δt is the sampling interval. Baseline idle energy is subtracted from measured energy:

$$E_{\text{final}} = E_{\text{measured}} - E_{\text{baseline}} \quad (3)$$

Because NVML reports GPU-side power, the reported values represent GPU inference energy rather than full-system energy. Baseline idle GPU energy was measured under the same job configuration and sampling interval, and subtracted from measured inference energy to estimate the additional GPU energy associated with repair generation.

For each run, we record the model identifier, bug identifier, execution status, number of tests passed, total number of tests, correctness outcome, inference time in seconds, and GPU energy in Joules.

The collected data are aggregated across repeated runs. For RQ1, we analyze Pass@1, the number of bugs fixed, and inference time. For RQ2, we analyze GPU energy consumption. For RQ3, we jointly compare repair correctness, inference time, and GPU energy consumption to identify correctness-energy trade-offs across models.

To compare energy differences between models, we aggregate inference-energy measurements per bug-model pair across the 10 runs and use the per-bug average as the unit of analysis. This gives one paired energy value per bug for each model. We use the Wilcoxon signed-rank test [14] for pairwise comparisons and report Vargha–Delaney A_{12} effect sizes [10] to quantify the magnitude of differences. Because multiple comparisons are performed, p-values are interpreted together with effect sizes and descriptive trends rather than as the sole basis for conclusions.

Finally, we conduct an exploratory inspection of generated repairs that failed to fix the observed failure to identify recurring failure patterns, such as invalid syntax, incomplete repairs, incorrect function or class structure, incorrect repair logic, or outputs that do not address the observed failure. This inspection supports the interpretation of the quantitative results rather than producing a validated taxonomy.

4 Results

This section presents the results organized by the three research questions. We analyze repair correctness, inference time, GPU energy consumption, and the resulting correctness-energy trade-offs across the evaluated models. Only bug-model pairs with 10 completed runs were included in the analysis. Of the 40 selected bugs, 35 had complete and consistent 10-run coverage across all six models and were retained for analysis. Bugs with incomplete runs due to execution timeouts or missing run logs were excluded to ensure a balanced comparison. The final dataset, therefore, contains 35 bugs (2,100 runs). The complete experimental results are available in the replication repository as a CSV file.

4.1 RQ1: Repair Correctness and Inference Time

How do LLMs of varying sizes differ in their ability to fix Python bugs in terms of repair correctness and inference time?

This RQ evaluates H1 by examining whether smaller models can achieve repair correctness comparable to larger models while requiring less inference time. Table 2 summarizes repair correctness and inference time for each evaluated model. Pass@10 is reported as the number of bugs, out of the 35 bugs retained for aggregate analysis, for which a model generated at least one test-passing repair across its 10 runs, while Pass@1 represents the observed proportion of successful repair attempts across all runs. The difference between these measures reflects the repeated-run setting, in which some bugs are solved consistently across multiple runs, while many others are not solved at all.

Table 2 Repair correctness and inference time across models.

Model	Size	Pass@10	Pass@1	Avg Time (s)
Qwen1.5	Small	15	0.43	38.7
StableCode3	Small	7	0.20	45.7
DeepSeek6	Small	5	0.14	54.3
StarCoder7	Medium	8	0.23	48.2
CodeLlama13	Medium	14	0.40	68.5
StarCoder15	Medium	8	0.23	96.7

As shown in Table 2 and Figure 2, Qwen1.5 achieves the strongest repair performance among the evaluated models, fixing 15 unique bugs and reaching a Pass@1 of 0.43 while maintaining the lowest average inference time (38.7s). CodeLlama13 performs competitively in terms of correctness, fixing 14 bugs with a Pass@1 of 0.40, but requires substantially longer average execution time (68.5s). The remaining models fix fewer tasks and exhibit lower overall Pass@1 values.

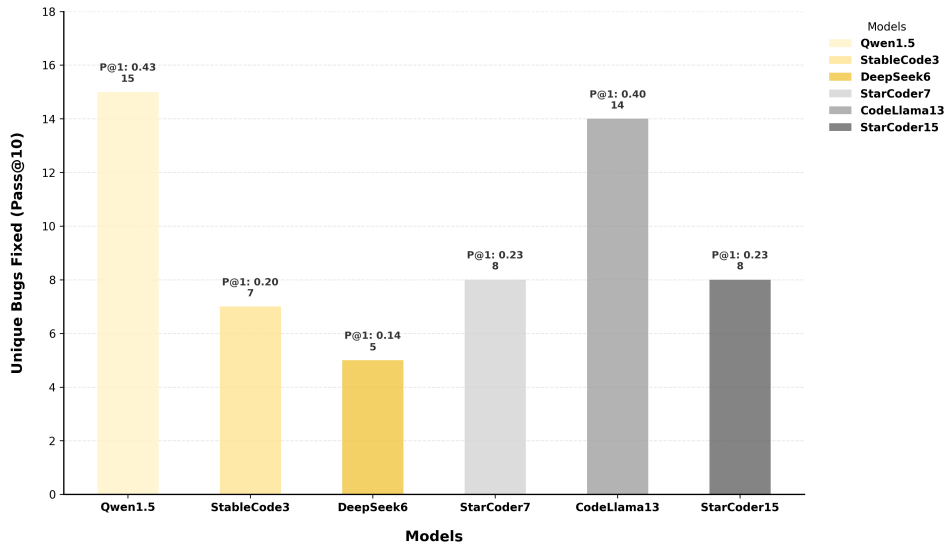
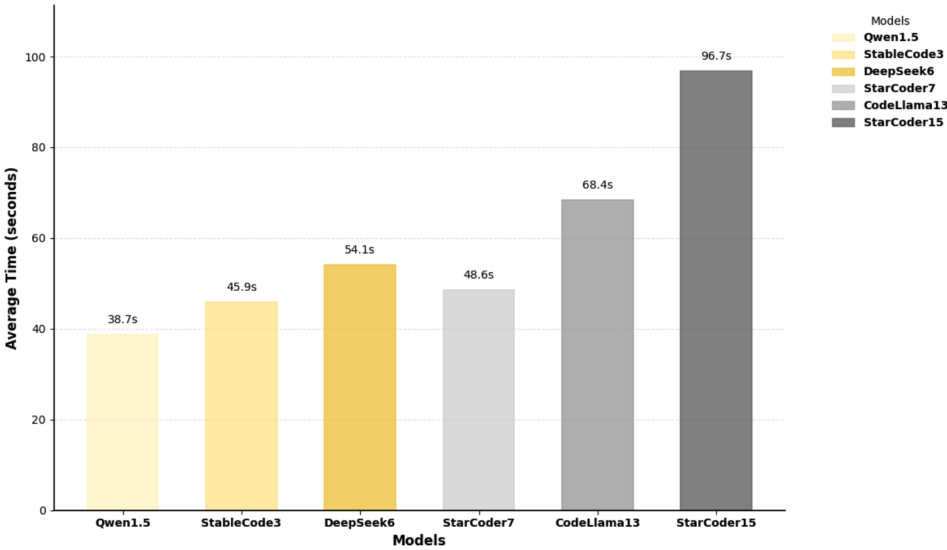


Figure 2 Repair correctness across models.



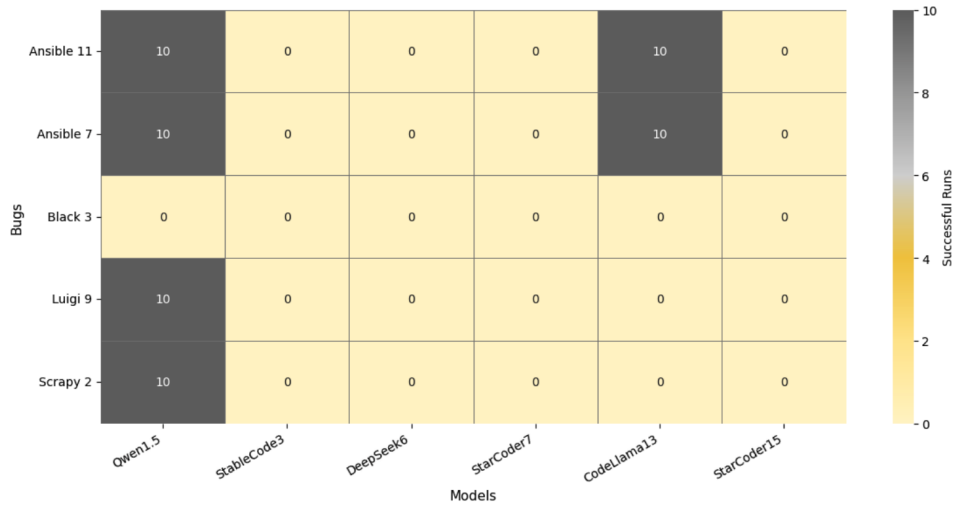
■ **Figure 3** Average inference time across models.

These results indicate that increasing model size does not consistently lead to higher repair correctness or lower inference time in this setting. For example, StarCoder15, the largest evaluated model, fixes only 8 bugs, achieves a Pass@1 of 0.23, and requires the highest average inference time (96.7s). As illustrated in Figure 3, overall inference runtime increases with model scale, although the trend is not strictly monotonic across model families.

Summary: Qwen1.5 achieved the highest Pass@10 and Pass@1 while also requiring the lowest average inference time. This supports H1 in the evaluated setting, although CodeLlama13 was competitive.

4.1.1 Exploratory Failure Analysis

Although Qwen1.5 achieves the highest Pass@1, its score remains below 0.50, indicating that even the best-performing model does not reliably generate a test-passing repair in a single attempt. We conducted a targeted manual inspection of repairs generated for 20 bugs across all six models, covering 1,200 outputs (20 bugs $\times$ 6 models $\times$ 10 runs). The analysis was exploratory and aimed at identifying recurring failure patterns rather than constructing a validated failure taxonomy. Figure 4 presents five illustrative cases selected from this inspection to show recurring success and failure behavior across repeated runs.



■ **Figure 4** Number of successful runs for selected bugs across models (out of 10 independent attempts).

As shown in Figure 4, repair outcomes exhibit a pronounced 0-or-10 pattern: a model either succeeds in all 10 runs or fails in all 10 runs. For Ansible 7 and Ansible 11, Qwen1.5 and CodeLlama13 produce test-passing repairs in all 10 runs, whereas the remaining models fail in every run. Qwen1.5 similarly achieves 10/10 successful runs for Luigi 9 and Scrappy 2, while no evaluated model successfully repairs Black 3.

Ansible 11 provides an illustrative example of how output structure can affect repair success. The bug requires updating the `map_obj_to_commands` function to construct CLI command strings for networking banners based on state conditions (`absent` vs. `present`). Specifically, when the state is `present`, the function must determine whether the target banner text differs from the existing text and format the string using specific delimiter tags (`@\n...\n@`).

As illustrated by the following generated repair, Qwen1.5 produces a concise, self-contained function that satisfies the required structure and passes the associated tests. Qwen1.5 and CodeLlama13 both achieve 10/10 successful runs for this bug.

```
def map_obj_to_commands(updates, module):
    commands = list()
    want, have = updates
    state = module.params['state']

    if state == 'absent' and 'text' in have.keys() and have['text']:
        commands.append('no_banner %s' % module.params['banner'])

    elif state == 'present':
        if want['text'] and (want['text'] != have.get('text')):
            banner_cmd = 'banner %s' % module.params['banner']
            banner_cmd += '\n@'
            banner_cmd += want['text'].strip()
            banner_cmd += '\n@'
            commands.append(banner_cmd)

    return commands
```

Manual inspection revealed that models like StarCoder7 and DeepSeek6 generated the correct `banner_cmd` logic but failed to execute because they wrapped the code in triple-backtick Markdown blocks (`"python ... "`) or redeclared the parent module import after the return statement. Since our execution pipeline treats syntax/insertion errors as immediate failures, these minor formatting deviations caused complete test suite failures.

Taxonomy of Recurring Failures. Across all 20 manually inspected bugs, failures fall into three main categories:

1. **Structural & Format Violations (~50% of failures in medium models):** Inclusion of Markdown tags, explanatory text, duplicated function signatures, or code outside the target block.
2. **Incomplete Repairs (~30% of failures):** Correct handling of primary branches but omission of required initializations, return statements, or edge-case null checks.
3. **Incorrect Logic (~20% of failures, predominant in Qwen1.5):** Flawed API usage, invalid state checks, or incorrect enumeration (enum) indexing (e.g., in *Black 3*).

Unexpected Findings. The manual inspection showed that structural and formatting violations were more common than complex algorithmic misunderstandings (such as incorrect recursion or invalid mathematical operations). In several cases, generated outputs partially captured the intended repair logic but failed because they violated the required block structure or output format. This suggests that within the evaluated zero-shot block-repair setting, output compliance can be as important as the quality of semantic repair.

4.1.2 Project and Category-Level Performance

We also examine repair success across projects and broader project categories to understand whether model behavior varies by software domain. Table 3 reports the number of bugs fixed by each model for each project included in the selected benchmark subset.

■ **Table 3** Project-wise number of bugs successfully fixed by each model.

Project	Qwen1.5	StableCode3	DeepSeek6	StarCoder7	CodeLlama13	StarCoder15
Ansible	2	0	0	0	2	0
Black	0	0	0	0	0	0
CookieCutter	0	0	0	0	0	0
FastAPI	4	4	3	4	4	4
Luigi	1	0	0	0	1	0
Sanic	1	1	1	1	1	1
Scrapy	3	0	0	0	3	1
TQDM	4	2	1	3	3	2
Youtube-dl	0	0	0	0	0	0

The results show that *FastAPI*, *Sanic*, and *TQDM* are solved more consistently across models, while *Ansible*, *Luigi*, and *Scrapy* are more selective. Qwen1.5 performs strongly across several projects, particularly *FastAPI*, *Scrapy*, and *TQDM*. CodeLlama13 also performs competitively, while StableCode3 and DeepSeek6 show more limited success across projects. In contrast, none of the evaluated models successfully solved bugs from the *Black*, *CookieCutter*, and *Youtube-dl* projects.

Figure 5 aggregates the project-level results into broader categories. The heatmap shows the percentage of bugs fixed by each model within each category.

The evaluated *Web* category is solved consistently across all models. This may indicate that the evaluated web-related bugs are more localized or structurally regular, although confirming this would require a more detailed bug-characteristic analysis. Performance is

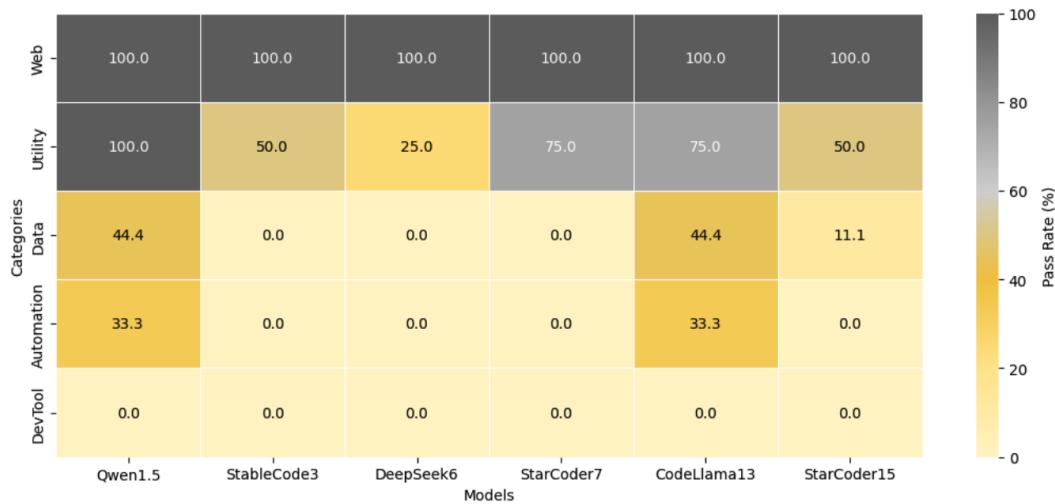


Figure 5 Model performance across project categories.

more moderate in the *Utility* category and lower in the *Data*, *Automation*, and *DevTool* categories. No model solves the evaluated *DevTool* bugs, suggesting that this category was particularly challenging in the selected benchmark subset. These observations indicate that repair success depends not only on model size but also on the characteristics of the underlying bugs.

4.2 RQ2: Inference-Time GPU Energy Cost

What is the inference-time energy cost associated with running these models on Python bug-fixing tasks?

This RQ evaluates H2 by comparing GPU energy consumption during inference between larger and smaller models. Inference-time GPU energy consumption is one factor in the sustainability of deployed LLM-based software engineering systems, especially when models are invoked repeatedly. Figure 6 shows the average GPU energy consumption per inference after subtracting baseline idle energy.

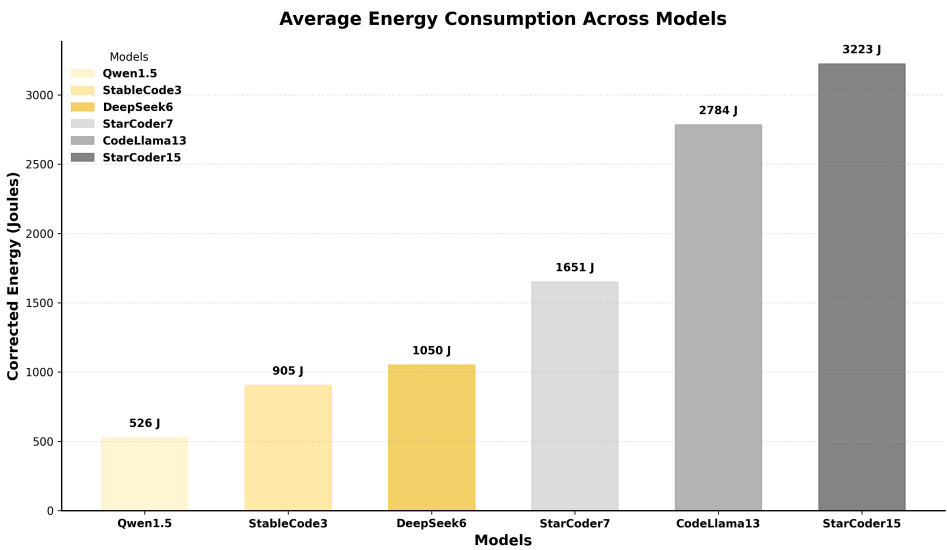
Average GPU energy consumption increases with model size in the evaluated set. Qwen1.5 consumes approximately 526 J per inference, followed by StableCode3 with 905 J and DeepSeek6 with 1,050 J. Among the medium-sized models, StarCoder7 consumes approximately 1,651 J, CodeLlama13 requires around 2,784 J, and StarCoder15 consumes over 3,200 J per inference. These results show that medium-sized models incur substantially higher GPU inference energy costs than smaller models.

Figure 7 shows the distribution of GPU energy consumption across tasks for each model. Smaller models consume less GPU energy overall, but their energy values vary more across tasks. Manual inspection suggests that this variation may be related to differences in generated output length and structure, although output length was not analyzed as a separate quantitative variable. In contrast, CodeLlama13 and StarCoder15 show consistently higher but relatively less variable energy consumption, indicating more stable but more expensive inference behavior.

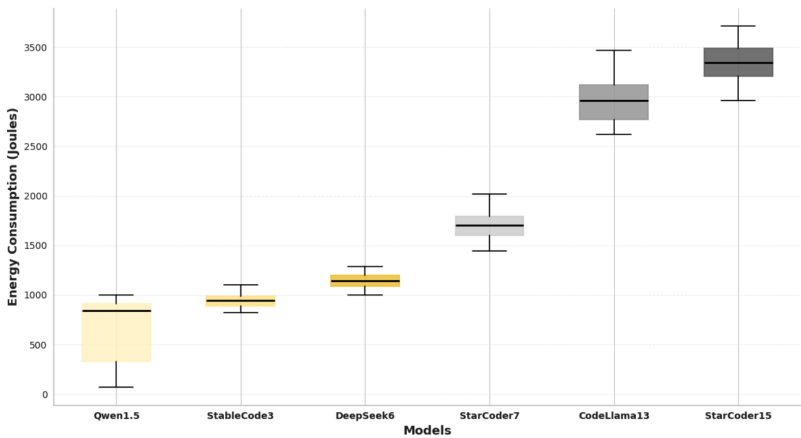
We further analyze GPU energy differences using Vargha–Delaney A_{12} effect sizes. Figure 8 presents pairwise comparisons, where each cell represents the probability that the model on the vertical axis consumes more GPU energy than the model on the horizontal axis.

Values close to 1.0 indicate that the row model usually consumes more GPU energy than the column model, while values close to 0.5 indicate similar distributions. The heatmap reports lower-triangular pairwise comparisons to avoid duplicate symmetric entries.

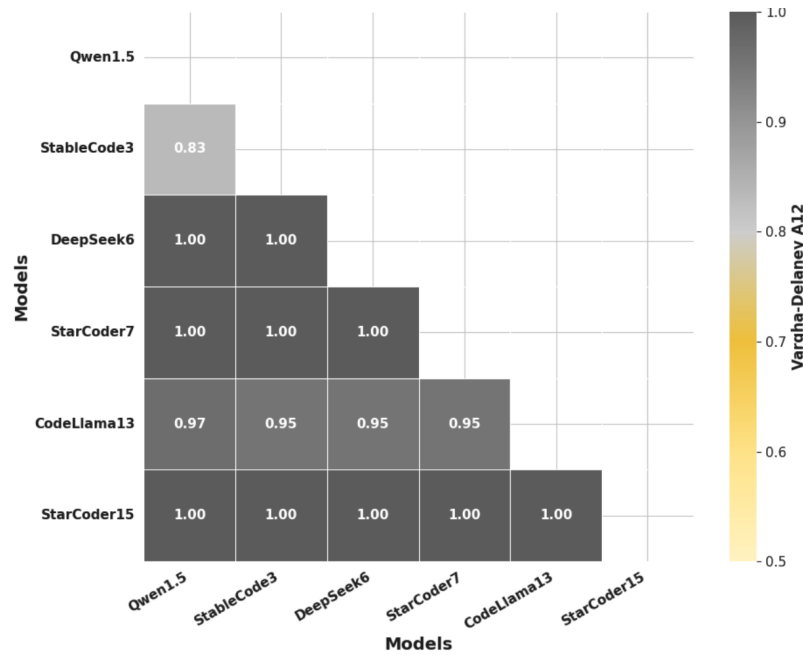
The A_{12} results show clear differences between small and medium-sized models. Comparisons involving StarCoder7, CodeLlama13, and StarCoder15 produce values close to 1.0 against smaller models, indicating a high probability that these models consume more GPU energy across tasks. Differences among smaller models are less pronounced, although Qwen1.5 remains the lowest-energy model overall.



■ **Figure 6** Average GPU energy consumption per inference across models.



■ **Figure 7** Distribution of GPU energy consumption across models.



■ **Figure 8** Pairwise comparison of GPU energy consumption using Vargha–Delaney A_{12} effect size.

■ **Table 4** Pairwise Wilcoxon signed-rank test results comparing Qwen1.5 with other models.

Comparison	p-value	Significant
Qwen1.5 vs StableCode3	< 0.001	Yes
Qwen1.5 vs DeepSeek6	< 0.001	Yes
Qwen1.5 vs StarCoder7	< 0.001	Yes
Qwen1.5 vs CodeLlama13	< 0.001	Yes
Qwen1.5 vs StarCoder15	< 0.001	Yes

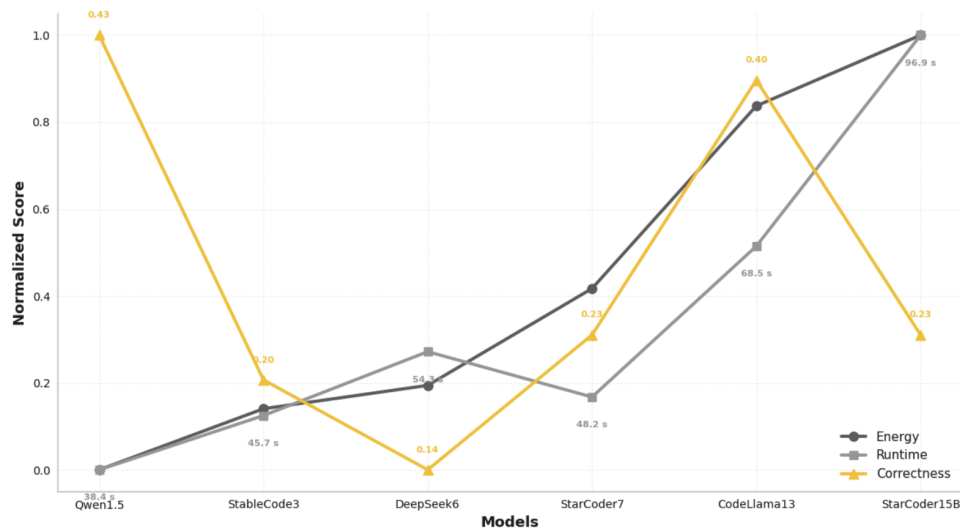
Because Qwen1.5 has the lowest average GPU energy consumption, we use it as the reference model for pairwise Wilcoxon signed-rank tests. Using per-bug average GPU energy values, the tests show statistically significant differences between Qwen1.5 and all other evaluated models.

Summary: GPU energy consumption increases substantially with model size in the evaluated set. Qwen1.5 consumes the least GPU energy, and pairwise statistical tests show significant differences between Qwen1.5 and all other models. This supports H2 in the evaluated setting.

4.3 RQ3: Correctness–Energy Trade-offs

Which model provides the most favorable trade-off between correctness, inference time, and energy consumption for Python bug-fixing tasks?

RQ3 compares repair correctness, inference time, and GPU energy consumption jointly. Figure 9 presents normalized values for the three dimensions across models. Energy and runtime are cost dimensions, so lower normalized values are preferable for those lines. Repair correctness is a benefit dimension, so higher normalized values are preferable for the correctness line.



■ **Figure 9** Normalized trade-off comparison of GPU energy consumption, inference time, and repair correctness across models. All metrics are Min-Max scaled to $[0, 1]$, Lower values are preferable for energy and runtime, whereas higher values are preferable for repair correctness. Data labels display raw, unnormalized values for direct reference.

Qwen1.5 provides the strongest overall trade-off between correctness and energy. It is the only model that is best on all four reported dimensions: Pass@10, Pass@1, average inference time, and average GPU energy consumption. CodeLlama13 achieves competitive repair correctness, but at substantially higher inference time and GPU energy cost. StarCoder15 consumes the most energy and takes the longest inference time, yet achieves lower repair correctness than both Qwen1.5 and CodeLlama13.

These results show that higher model size and higher GPU energy consumption do not necessarily translate into better repair performance. In the evaluated setting, repair correctness does not increase consistently with model size, whereas inference time and GPU energy consumption generally do. Therefore, correctness-only comparisons would miss an important part of the model-selection trade-off for LLM-based repair.

Summary: Qwen1.5 provides the most favorable trade-off between correctness and energy among the evaluated models, while also achieving the lowest average inference time. Larger models generally incur higher runtime and GPU energy costs without consistent improvements in correctness in this benchmark.

5 Discussion

The results show that model size alone is not a reliable indicator of repair effectiveness in the evaluated Python bug-fixing setting. Qwen1.5, the smallest model in the study, achieved the strongest repair correctness while also requiring the lowest inference time and GPU energy consumption. In contrast, larger models, such as StarCoder15, incurred substantially higher runtime and energy costs without corresponding gains in correctness. This suggests that scaling model size does not automatically improve correctness; there are energy trade-offs in LLM-based repair.

One explanation is the block-level and format-constrained nature of the repair task. Each model was asked to generate a complete replacement for a function or class and to output only executable Python code. The exploratory failure analysis suggests that several failures were caused not only by incorrect repair logic, but also by output-structure problems such as extra text, duplicated code, incomplete blocks, or code outside the expected structure. In such a setting, concise, structurally compliant outputs may be more useful than longer or more complex outputs.

The project and category-level results further indicate that repair success depends on the characteristics of the underlying bugs. The evaluated web-related bugs were solved more consistently across models, whereas DevTool, Data, and Automation tasks were more challenging. This suggests that future studies should examine bug characteristics more explicitly, including locality, dependency structure, and whether the repair requires a broader repository context.

For researchers, these findings reinforce the need to report inference time and GPU energy alongside repair correctness. For practitioners, the results suggest that smaller open-source code LLMs should not be dismissed solely because of their size, especially in local or repeated debugging workflows. However, these implications are limited to the evaluated models, prompt, benchmark, and hardware setup, and should not be interpreted as a general claim that smaller models are always preferable.

6 Threats to Validity

Construct validity. Repair correctness is measured using execution-based testing: a generated repair is considered correct if it passes the bug-specific test suite. This is common in automated program repair, but passing the available tests does not guarantee semantic equivalence to the developer patch or correctness for all inputs. In addition, this study measures GPU energy during model inference using NVML. The reported values therefore represent GPU-side repair-generation energy, not full-system or end-to-end pipeline energy; CPU, memory, disk, cooling, repository setup, patch insertion, and test execution are excluded. The repair task is also limited to function- or class-level replacement, so bugs requiring multi-location edits, new imports outside the extracted block, configuration changes, or broader repository-level reasoning are not captured.

Internal validity. Energy and runtime measurements can be affected by GPU state, thermal behavior, background processes, caching, and scheduling effects. To reduce these threats, all experiments were executed on the same dedicated GPU node with an identical configuration, a fixed delay between repeated runs, and baseline idle energy subtraction. Full-system energy could not be measured because CPU and host-level energy profiling were restricted in the HPC environment. Model-output variability is another threat. We did not apply task-specific or model-specific decoding tuning and relied on the default generation behavior of the inference library/model configuration. Random seeds were not fixed; instead, each bug-model pair was executed 10 times, and results were aggregated across attempts. Invalid or non-compliant outputs, such as Markdown, explanations, incomplete code, syntax errors, or outputs with an unexpected structure, were counted as unsuccessful repairs and were not manually edited.

External validity. The study initially selected 40 Python bugs from BugsInPy, of which 35 were retained for the balanced aggregate comparison across all six models due to complete and consistent run coverage. The six evaluated open-source code-oriented LLMs range

from 1.5B to 15B parameters and were executed on a single hardware configuration: an NVIDIA Tesla V100 GPU with 32 GB of memory, 4 CPU cores, and 128 GB of RAM. Results may not generalize to proprietary models, larger or newer models, quantized variants, tool-augmented repair systems, other programming languages, other benchmarks, or different hardware. Results may also differ under alternative prompt designs, decoding configurations, or model-specific tuning. Therefore, the findings should be interpreted as evidence for the evaluated models, tasks, prompts, and hardware setup. In particular, our conclusions should not be generalized to automated program repair scenarios requiring repository-level reasoning, cross-file modifications, or broader contextual understanding.

Conclusion validity. Repeated runs, statistical tests, and effect sizes are used to support comparisons across models. However, the benchmark size is limited, and multiple pairwise comparisons may increase the risk of false positives; therefore, statistical results are interpreted together with descriptive trends and effect sizes. The qualitative failure analysis is exploratory and is used to support the interpretation of quantitative results rather than to provide a validated taxonomy of LLM repair failures.

7 Conclusion

This study presents an empirical evaluation of open-source LLMs for localized Python bug-fixing tasks, where each repair replaces a single extracted function or class. Using the BugsInPy benchmark, models ranging from 1.5B to 15B parameters were evaluated under controlled conditions. The results show that increasing model size does not consistently improve performance for the localized repair scenarios investigated in this study. In particular, the smallest model (Qwen 1.5B) achieved the highest overall performance while consuming significantly less energy. In contrast, larger models incurred substantially higher computational costs without delivering proportional gains in correctness, indicating diminishing returns from scaling. The qualitative analysis further suggests that output characteristics, such as conciseness and structural correctness, can affect repair success in the evaluated block-level setting. In the inspected cases, some medium-sized models produced longer or more structurally complex outputs, which were more susceptible to formatting or structural failures. These findings highlight an important trade-off between performance and sustainability in AI-assisted software engineering. Rather than defaulting to larger models, practitioners should select models based on task requirements, reliability constraints, and resource efficiency.

Future Work

This study can be extended in several directions. First, to include a broader range of models, including larger-scale and proprietary systems, to better understand how model size and architectural differences influence both performance and energy consumption. In addition, exploring alternative prompting strategies, such as few-shot and instruction-based prompting, may provide further insight into how input design affects both correctness and computational efficiency. Second, the evaluation can be expanded to cover additional programming languages and more diverse software systems, enabling analysis of whether the observed trends generalize beyond Python-based bug-fixing tasks. Incorporating a larger set of bugs, including those currently excluded due to dependency or execution constraints, would further improve the representativeness of the results. Finally, future studies can extend energy measurement beyond GPU inference to capture full-system consumption, including CPU, memory, and the complete execution pipeline.

Data Availability Statement

The data and materials supporting this study, including experimental results, generated outputs, and analysis artifacts, are available in the replication repository: <https://github.com/esem2026-ai-study/empirical-llm-bugfix-study.git>.

References

- 1 Mauricio Fadel Argerich, Jonathan Fürst, and Marta Patiño-Martínez. Watt counts: Energy-aware benchmark for sustainable LLM inference on heterogeneous GPU architectures. *CoRR*, abs/2604.09048, 2026. doi:10.48550/arXiv.2604.09048.
- 2 Humza Ashraf, Syed Muhammad Danish, Aris Leivadeas, Yazan Otoum, and Zeeshan Sattar. Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable AI programming. *CoRR*, abs/2508.08332, 2025. doi:10.48550/arXiv.2508.08332.
- 3 Manal Binkhonain and Reem Alfayez. Are prompts all you need? evaluating prompt-based large language models (llm) s for software requirements classification. *Requirements Engineering*, 30(4):423–443, 2025. doi:10.1007/s00766-025-00451-8.
- 4 Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering for large language models. *Patterns*, 6(6), 2025. doi:10.1016/j.patter.2025.101260.
- 5 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. arXiv:2107.03374.
- 6 Luis Cruz, João Paulo Fernandes, Maja H. Kirkeby, Silverio Martínez-Fernández, June Sallou, Hina Anwar, Enrique Barba Roque, Justus Bogner, Joel Castaño, Fernando Castor, Aadil Chasmawala, Simão Cunha, Daniel Feitosa, Alexandra González, Andreas Jedlitschka, Patricia Lago, Henry Muccini, Ana Oprescu, Pooja Rani, João Saraiva, Federica Sarro, Raghavendra Selvan, Karthik Vaidhyanathan, Roberto Verdecchia, and Ivan P. Yamshchikov. Greening ai-enabled systems with software engineering: A research agenda for environmentally sustainable AI practices. *CoRR*, abs/2506.01774(3):14–23, 2025. doi:10.48550/arXiv.2506.01774.
- 7 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 Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32556–32569, 2025. doi:10.18653/v1/2025.acl-long.1563.
- 8 Frederik Floether, Jan Mikolon, and Maria Longobardi. Accelerating the drive towards energy-efficient generative ai with quantum computing algorithms. *Quantum Science and Technology*, 2025.
- 9 Md. Arman Islam, Devi Varaprasad Jonnala, Ritika Rekhi, Pratik Pokharel, Siddharth Cilamkoti, Asif Imran, Tevfik Kosar, and Bekir O. Turkkan. Evaluating the energy-efficiency of the code generated by llms. *CoRR*, abs/2505.20324, 2025. doi:10.48550/arXiv.2505.20324.
- 10 Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. The impact of prompt programming on function-level code generation. *IEEE Trans. Software Eng.*, 51(8):2381–2395, 2025. doi:10.1109/TSE.2025.3587794.
- 11 Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural*

- Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, volume 32, pages 11883–11894, 2019. URL: <https://proceedings.neurips.cc/paper/2019/hash/7298332f04ac004a0ca44cc69ecf6f6b-Abstract.html>.
- 12 Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. An exploratory study on fine-tuning large language models for secure code generation. *CoRR*, abs/2408.09078, 2024. doi:10.48550/arXiv.2408.09078.
 - 13 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shu jie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. Accessed: November 10, 2025. doi:10.48550/arXiv.2102.04664.
 - 14 Crish Nagarkar, Leonid Bogachev, and Serge Sharoff. Can LLM reasoning be trusted? A comparative study: Using human benchmarking on statistical tasks. *CoRR*, abs/2601.14479, 2026. doi:10.48550/arXiv.2601.14479.
 - 15 Syed Fakhar Abbas Naqvi and Hina Anwar. Benchmarking Performance and Sustainability Trade-Offs in LLMs for Python Bug-Fixing Tasks. Software, sw-hId: swh:1:dir:e2f5ea38e5e8b633db8ef0baf818d58b194468a9 (visited on 2026-09-21). URL: <https://github.com/syedfakhar25/empirical-llm-bugfix-benchmark/tree/main>, doi:10.4230/artifacts.28073.
 - 16 Lorena Poenaru-Olaru, June Sallou, Luis Cruz, Jan S Rellermeyer, and Arie van Deursen. Sustainable machine learning retraining: Optimizing energy efficiency without compromising accuracy. In *2025 11th International Conference on ICT for Sustainability (ICT4S)*, pages 100–111. IEEE, 2025. doi:10.1109/ICT4S68164.2025.00019.
 - 17 Pooja Rani, Jan-Andrea Bard, June Sallou, Alexander Boll, Timo Kehrer, and Alberto Bacchelli. Can we make code green? understanding trade-offs in llms vs. human code optimizations. *CoRR*, abs/2503.20126, 2025. doi:10.48550/arXiv.2503.20126.
 - 18 June Sallou, Luis Cruz, and Thomas Durieux. Energibridge: empowering software sustainability through cross-platform energy measurement. *CoRR*, abs/2312.13897, 2023. doi:10.48550/arXiv.2312.13897.
 - 19 Stack Overflow. 2025 stack overflow developer survey. <https://survey.stackoverflow.co/2025/>, 2025. Accessed: 2025-11-10.
 - 20 Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020. doi:10.1609/aaai.v34i09.7123.
 - 21 U. Boran Torun, Mehmet Taha Demircan, Mahmut Furkan Gön, and Eray Tüzün. Past, present, and future of bug tracking in the generative AI era. *CoRR*, abs/2510.08005, 2025. doi:10.48550/arXiv.2510.08005.
 - 22 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.
 - 23 Mike Vizard. Survey: Fixing Bugs Stealing Time from Development – DevOps.com. <https://devops.com/survey-fixing-bugs-stealing-time-from-development/>. [Accessed 10-11-2025].
 - 24 Ratnadira Widayarsi, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1556–1560. ACM, 2020. doi:10.1145/3368089.3417943.
 - 25 Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017. doi:10.1007/s10664-017-9514-4.