Journal of Software Engineering Research and Development, 2026, 14:24, doi: 10.5753/jserd.2026.7689
This work is licensed under a Creative Commons Attribution 4.0 International License.

Revisiting the Energy Footprint of Small Language Models for Unit Test Generation: A Bayesian Comparative Study

Vinicius H. S. Durelli [**Universidade Federal de São Carlos** | ***vinicius.durelli@ufscar.br***]
Rafael S. Durelli [**Universidade Federal de Lavras** | ***rafael.durelli@ufla.br***]
Andre T. Endo [**Universidade Federal de São Carlos** | ***andreendo@ufscar.br***]

Abstract

Context. Manual unit test creation is a cognitively intensive and time-consuming activity, prompting researchers and practitioners to increasingly adopt automated testing tools. Recent advancements in language models have expanded automation possibilities, including unit test generation, yet these models raise substantial sustainability concerns due to their energy consumption compared to conventional, specialized tools.
Goal. Our research investigates whether the energy overhead associated with employing small language models (SLMs) for unit test generation is justified compared to a conventional, lightweight testing tool. We compare and analyze the energy consumption incurred during test suite generation, as well as the fault-finding effectiveness of the resulting test suites, for two SLMs (Phi-3.1 Mini 128k and Qwen2.5-Coder 1.5B) and Pynguin, a purpose-built tool for unit test generation.
Method. We posed three research questions: *(i)* Is the additional energy cost incurred by employing SLMs for unit test generation justified when compared to a lightweight, purpose-built alternative?; *(ii)* To what extent does the energy cost associated with SLM-based unit test generation vary across different SLM architectures?; and *(iii)* To what extent do unit test suites generated by Phi, Qwen, and Pynguin differ in their fault-finding effectiveness across a range of faulty programs? To rigorously address the first two research questions, we employed Bayesian Data Analysis. For the third research question, we conducted a complementary empirical analysis using descriptive statistics.
Results. The results from our Bayesian analysis provide robust evidence indicating that Phi and Qwen consistently consume significantly more energy than Pynguin during test suite generation.
Conclusions. These findings underscore significant sustainability concerns associated with employing even SLMs for routine Software Engineering tasks such as unit test generation. The results challenge the assumption of universal energy efficiency benefits from smaller-scale models and emphasize the necessity for careful energy consumption evaluations in the adoption of automated software testing approaches.

Keywords: *Unit test generation; Energy consumption; Language model; Sustainable Software Engineering*

1 Introduction

Testing is a fundamental part of the software development process. However, software testing is widely acknowledged as a resource-intensive phase in the software development lifecycle, with manual test case generation accounting for a substantial share of the associated costs and effort. In effect, the quality of test cases plays a pivotal role in shaping the effectiveness and efficiency of the software testing process (Myers et al., 2011). Nevertheless, due to its inherently taxing nature, manual test case creation is often prone to human error. To address these challenges and reduce the overall cost, significant advancements have been made in the automated generation of test cases (Anand et al., 2013; Fontes et al., 2023). Essentially, the goal of automated test case generation approaches is to find a small set of test cases (comprising valid and diverse inputs) that verify the correctness of the system under test. Owing to this importance, automated test case generation has remained a prominent research topic for several decades, leading to the development of a myriad of approaches and tools (McMinn, 2004; Godefroid et al., 2005; Ali et al., 2010; Lakhotia et al., 2010).

Recent advances in Large Language Models (LLMs) have significantly extended their applicability beyond the domain of Natural Language Processing (NLP). Given their training on vast corpora, LLMs can reasonably be considered general-purpose response generators. Consequently, these advances have paved the way for the automation of a wide range of Software Engineering activities, including test case generation (Wang et al., 2024). Specifically, given the widespread use of LLMs in various coding-related tasks, it is reasonable to expect that they would be effective in software testing activities, particularly in the generation of unit tests (Schäfer et al., 2024; Abdullin et al., 2025). However, despite the transformative impact of models such as ChatGPT (OpenAI, 2025) and Claude (Anthropic, 2025) across multiple domains, their growing complexity comes with significant computational and data demands. As a result, deploying these models outside high-performance data centers remains largely impractical. The lifecycle of LLMs, encompassing training, fine-tuning, and inference (i.e., serving) phases, requires substantial computational resources and results in significant energy consumption (Luccioni et al., 2023). While model training is known to span several months and demand extensive computational power, recent studies indicate that the energy consumption associated with the serving phase has now surpassed that of training (Ding and Shi, 2024), thereby contributing to the increasing carbon footprint of the technology sector. As global energy consumption continues to rise, concerns regarding the environmental impact of

LLMs have become increasingly pressing. The widespread adoption of LLMs in mainstream applications further exacerbates these concerns.

Given that the expansive capacity of LLMs can sometimes hinder accuracy in specific knowledge domains, and considering their substantial computational and energy demands pose significant challenges for sustainable deployment, researchers have been turning to developing models that are more compact and affordable. These alternatives, often termed Small Language Models (SLMs), are streamlined models with significantly fewer parameters than traditional LLMs. Their smaller size results in reduced computational requirements, faster inference times, and easier deployment on resource-constrained devices (Jovanović and Campbell, 2024). Despite their smaller size, SLMs fine-tuned on domain-specific datasets can outperform larger, general-purpose models on targeted tasks.

As mentioned, the use of LLMs and SLMs tailored for code generation has recently gained traction in the context of software testing. These models are increasingly being employed to automate the generation of test cases, often completely replacing traditional tools specifically designed for this purpose. While SLM-based approaches offer the appeal of flexibility and natural language-driven interaction, they introduce non-negligible overhead in terms of resource consumption (not to mention integration complexity). Despite these drawbacks, many researchers and practitioners continue to adopt both large and small language models for test case generation, frequently overlooking the fact that, in many scenarios, conventional tools may be sufficient from an energy consumption standpoint. This trend guided the central research question of our previous study (Durelli et al., 2025): Is the additional energy cost of using even an SLM justified when compared to a more lightweight and purpose-built alternative for test case generation? Specifically, in our previous study, we examined the energy footprint of employing an SLM for automated unit test generation. We focused on a single SLM (i.e., Phi 3.1 Mini 128k) and conducted a Bayesian analysis to estimate and compare its energy consumption and fault-detection effectiveness against a conventional, purpose-built unit test generation tool, Pynguin (Lukasczyk and Fraser, 2022; Lukasczyk et al., 2023).

The increasing adoption of language models for software engineering tasks, including test case generation, motivates a broader inquiry into their sustainability implications. In particular, it remains unclear whether the additional energy cost incurred by such models is an inherent characteristic of SLM-based approaches or whether it depends on architectural and design choices specific to individual models.

To address this question, the present study extends our previous investigation (Durelli et al., 2025) along two important dimensions. First, we broaden the empirical scope by incorporating an additional code-specialized SLM (i.e., Qwen2.5-Coder 1.5B) into the comparison. This inclusion allows us to move beyond a single-model case study and examine whether the previously observed energy overhead is intrinsic to SLM-based unit test generation or instead sensitive to architectural and scaling differences across models. By evaluating two SLMs, we aim to provide a more comprehensive assessment of the sustainability implications of language-model-based test generation. Second, and methodologically more importantly, we adopt a cumulative Bayesian perspective. In our previous investigation, we modeled energy consumption using a Bayesian framework, obtaining posterior distributions for the parameters governing the energy usage of Phi and Pynguin. In the current study, rather than discarding this previously acquired evidence, we use those posterior distributions as informed priors for the updated analysis. This approach leverages one of the main advantages of Bayesian inference: the coherent updating of beliefs in light of new data (Kruschke, 2015; van de Schoot et al., 2021). By treating the results of our previous study as prior knowledge, we are able to formally accumulate evidence across studies and obtain posterior estimates that reflect both past and newly collected observations. This cumulative modeling strategy strengthens the inferential foundation of our conclusions. Instead of performing an isolated re-analysis, we embed the new empirical results within a principled updating process (Kruschke, 2015), thereby enhancing statistical efficiency and interpretability.

Within this extended framework and building upon our previous study, we compare two SLMs (i.e., Phi 3.1 Mini 128k and Qwen2.5-Coder 1.5B) with a traditional automated test generation tool (i.e., Pynguin) in terms of *(i)* energy consumption during test suite generation and *(ii)* the fault-detection effectiveness of the resulting test suites. This comparative analysis examines whether SLM-based approaches incur greater energy costs than a purpose-built alternative and whether such costs are accompanied by corresponding gains in test effectiveness. In this follow-up study, we further conduct a direct comparison between the two SLMs to assess whether energy consumption varies across models. By evaluating three distinct test generation approaches on a benchmark of faulty Python programs (Lin et al., 2017), this work contributes to the broader discussion on sustainable Software Engineering.

The remainder of this article is organized as follows: Section 2 details the experimental design, including subject program selection, experimental variables, and the rationale behind employing a Bayesian approach to data analysis. Section 3 presents results from our Bayesian analysis, highlighting differences in energy consumption and fault-detection effectiveness between the two test code generation approaches. Section 4 discusses threats to the validity. Section 5 presents related work. Section 6 summarizes key insights and implications of our research, emphasizing sustainability considerations in automated Software Engineering practices.

2 Study Design

In this follow-up investigation, we compare Pynguin (Lukasczyk and Fraser, 2022) with two SLMs, namely Phi 3.1 Mini 128k[1] and Qwen2.5-Coder 1.5B,[2] in terms of their energy consumption and the overall quality of the generated unit tests. While our previous investigation (Durelli et al., 2025) focused exclusively on Phi,

[1]For brevity, we refer to Phi 3.1 Mini 128k simply as *Phi* throughout the remainder of this article.

[2]Hereafter referred to as *Qwen*.

the inclusion of Qwen allows us to examine whether the energy-related patterns observed earlier are model-specific or instead characteristic of SLM-based unit test generation more broadly.

Specifically, the primary objectives of this investigation are twofold: *(i)* to examine and compare the energy consumption profiles of SLM-based approaches and a conventional unit test generation tool, and *(ii)* to assess the overall fault-detection effectiveness of the resulting test suites. By incorporating multiple SLM architectures into the comparison, we aim to determine whether differences in model scale and architectural design influence the sustainability and practical utility of language-model-based test generation.

Consistent with the findings of our prior study (Durelli et al., 2025), we hypothesize that employing zero-shot prompting with SLMs does not yield test suites that are substantially more effective than those generated by a conventional unit test generation tool. This hypothesis is grounded in a broader, sustainability-oriented interpretation of effectiveness, which considers not only the computational demands measured through energy consumption but also evaluates whether any additional energy expenditure translates into demonstrably more robust test suites capable of detecting a greater number of faults. In this context, effectiveness is interpreted as a trade-off between fault-detection capability and the energy cost required to achieve it.

While SLMs are known for their compactness and, consequently, their capacity for reduced computational overhead during inference (i.e., serving), their application in the domain of test data generation still demands considerable computational expenditure. The iterative processes of prompt engineering, model inference, and subsequent validation of generated test cases, even within a zero-shot paradigm, contribute significantly to the overall energy footprint. This substantial resource usage raises critical questions regarding the environmental sustainability and practical viability of deploying even SLM for routine unit test generation, particularly when juxtaposed against conventional tools.

We contend that the maturity of conventional unit test generation tools has led to highly optimized algorithms and heuristics that achieve comparable test suite effectiveness with a significantly lower resource-usage footprint and, consequently, a more favorable energy consumption profile. These tools often leverage well-established, state-of-the-art test data generation algorithms (Fraser and Arcuri, 2013; Lukasczyk et al., 2023), which, while computationally intensive in their own right, have been refined over the years to operate within more constrained resource environments. Therefore, the incremental benefit, if any, offered by SLM-based approaches for test generation might not outweigh the associated increase in energy consumption, rendering them less desirable from both an economic and ecological perspective. To systematically investigate these trade-offs and provide empirical evidence for the considerations outlined in our prior and current studies, we reformulated our RQs as follows:

> **RQ_1:** Is the additional energy cost incurred by employing SLMs for unit test generation justified when compared to a lightweight, purpose-built alternative?

RQ_1 focuses on comparing the energy consumption of SLM-based approaches (i.e., Phi and Qwen) with that of a conventional unit test generation tool (i.e., Pynguin) during the generation of unit test suites for a set of Python programs. This question allows us to assess whether the energy overhead previously observed for Phi persists when considering multiple SLMs and whether such overhead is structural to language-model-based test generation.

RQ_2 shifts the focus from a comparison between approaches to a comparison within the SLM paradigm itself. While RQ_1 examines the difference in energy consumption between SLM-based and conventional unit test generation approaches, RQ_2 investigates whether the energy cost associated with SLM-based unit test generation is sensitive to differences across models. Specifically, it aims to determine whether the previously observed energy overhead is an intrinsic characteristic of SLM-based approaches or whether it varies meaningfully across distinct SLM architectures.

> **RQ_2:** To what extent does the energy cost associated with SLM-based unit test generation vary across different SLM architectures?

Complementing this analysis, we formulated a RQ to examine whether the computational demands associated with each approach correlate with demonstrably more robust test suites, specifically in terms of their effectiveness at fault detection.

> **RQ_3:** To what extent do unit test suites generated by Phi, Qwen, and Pynguin differ in their fault-finding effectiveness across a range of faulty Python programs?

To examine the fault-detection capabilities of the generated test suites, we employed a program repair benchmark comprising both a correct and a faulty version of each program. Specifically, in our experiment, the correct version is used by both tools to generate their respective test suites. These suites are subsequently executed against the faulty version to determine whether the fault can be detected. The benchmark selected for this evaluation is detailed in Subsection 2.1.

2.1 Subjects Selection

For our experiment, we selected programs from the QuixBugs benchmark (Lin et al., 2017), a widely used benchmark in the program repair literature that comprises 40 faulty programs, each of which includes a corresponding correct version, available in both Python and Java. In our experiment, we focus exclusively on the Python implementations. Specifically, we focus exclusively on the Python implementations of QuixBugs for two reasons. First, Python is the target language of Pynguin, the conventional unit test generation tool used as the baseline in this study. Second, keeping the programming language fixed allows us to compare the evaluated approaches under a common execution environment, thereby avoiding confounding factors that could arise from differ-

ences between language ecosystems, testing frameworks, or runtime infrastructures. Since our goal is to compare SLM-based test generation with a purpose-built Python test generation tool, the Python version of QuixBugs provides an appropriate and controlled experimental setting.

A distinctive feature of QuixBugs is that its programs were originally written by participants of the Quixey Challenge (Lin et al., 2017; Ye et al., 2021). As a result, the faults were introduced unintentionally and reflect realistic programming mistakes rather than artificially constructed ones.

We did not use all 40 programs in our experiment: only the programs listed in Table 1 were included in the evaluation. Table 1 gives an overview of the 17 programs from the QuixBugs benchmark used in our study, including their size in lines of code (LoC) and the type of fault present in each faulty version (Ye et al., 2021). Further details on the programs can be found in the replication package of this study.[3] Several programs were excluded due to practical limitations encountered during the setup phase. Specifically, we faced difficulties configuring reliable timeouts and integrating certain programs into our experimental framework. As a result, we opted to exclude those cases to ensure consistency and reproducibility across all runs. The final set of programs used reflects those that could be successfully and consistently executed under the experimental conditions. Although these decisions improve the internal consistency of our study, they also reduce the size of our sample and thus impose limitations on generalizability (as discussed in Section 4).

Table 1. Overview of the QuixBugs benchmark programs, including program name, size in lines of code (LoC), and the type of fault present in the faulty version of each program.

Program Name	LoC	Fault Type
`find_first_in_sorted`	22	Incorrect comparison operator
`find_in_sorted`	19	Missing increment
`flatten`	18	Missing function call
`get_factors`	17	Incorrect constructor call
`hanoi`	53	Incorrect variable reference
`is_valid_parenthesization`	15	Incorrect code replacement
`kheapsort`	29	Missing function call
`kth`	25	Incorrect variable reference
`lcs_length`	48	Incorrect array slice
`lis`	27	Missing logic
`max_sublist_sum`	13	Missing function call
`minimum_spanning_tree`	67	Missing logic
`possible_change`	23	Missing boolean expression
`sieve`	35	Incorrect method call
`sqrt`	9	Incorrect arithmetic expression
`subsequences`	22	Missing function call
`to_base`	14	Swapped expressions

[3]All Python notebooks, CSV datasets, and the selected programs from the QuixBugs benchmark used in this study are publicly available as part of our replication package hosted on Zenodo: `https://doi.org/10.5281/zenodo.18791048`.

2.2 Experiment Variables

To answer RQ_1, RQ_2, and RQ_3, we consider the unit test generation approach as the independent variable. Accordingly, the three treatments of this variable are: two SLM-based approaches (via zero-shot prompting) and a conventional unit test generation tool. For RQ_1 and RQ_2, the dependent variable is the energy consumption, measured in Joules (J), incurred during unit test code generation. For RQ_3, the dependent variable is a proxy for the fault-detection capability of the generated test suites. This is quantified by assessing the effectiveness of the test suites in identifying faults present in faulty programs. Accordingly, the effectiveness of a test suite in identifying faults is *operationally defined* (VanderStoep and Johnson, 2008) as the proportion of faulty program versions for which the test suite includes at least one failing test case that reveals the injected fault. This metric is computed as the ratio between the number of faulty program versions successfully detected by the test suite and the total number of faulty program versions evaluated.

Energy consumption is a key dimension of software sustainability and is commonly quantified in Joules (J) or kilowatt-hours (kWh) (Castaño et al., 2023). To assess the energy demands of specific operations, profiling tools typically monitor the average energy consumption over a given time window. In the context of our experiment, we measured energy consumption during each execution using PowerJoular (Noureddine, 2022), a profiling tool that has been widely adopted in energy-related Software Engineering research due to its reliability and fine-grained measurement capabilities. PowerJoular tracks energy usage for both the CPU and GPU in Joules, enabling detailed insights into the computational cost of software processes.

For the conventional test generation approach, we recorded CPU energy consumption and utilization. In contrast, for the SLM-based approach, which leverages GPU-accelerated inference, we collected measurements of both CPU and GPU energy usage. These measurements, expressed in Joules, enabled a systematic comparison of the energy footprints of the two approaches under evaluation.

2.3 Prompting Configuration

As mentioned, for the SLM-based approaches, we adopted a zero-shot prompting strategy: the models were not provided with examples of previously generated test suites or feedback from earlier generations. For each subject program, the prompt consisted of a fixed instruction followed by the source code of the correct version of the program under test. The instruction asked the model to generate Python unit tests compatible with the `pytest` framework. The same prompt template was used for all programs and for both SLMs, with only the program-specific source code varying across executions.

Before being inserted into the prompt, each program was formatted as plain Python source code and delimited from the natural-language instruction using explicit textual markers. No semantic preprocessing, manual simplification, or program-specific rewriting was applied to the source code. Likewise, we did not provide the models with the faulty

version of the program, the expected fault location, or any information about the benchmark metadata. Therefore, the models generated tests solely from the correct implementation and the general instruction encoded in the prompt. After generation, the model outputs were parsed to extract executable Python test code. Outputs containing explanatory text or Markdown formatting were cleaned only to the extent necessary to isolate the generated test file; no manual improvement of assertions or test inputs was performed.

2.4 Data Analysis

Initially, our analysis is centered on applying descriptive statistics to the collected data in order to gain preliminary insights into the energy consumption and fault-finding effectiveness of the three approaches. Given that our primary objective is to address RQ_1 and RQ_2, we model the differences in energy consumption between the test data generation approaches using a Bayesian framework. In this expanded version of the study, we adopt a modeling strategy in which the posterior distributions derived from the execution of Phi in our previous study are formally incorporated to inform the prior distributions for Qwen. This strategy follows the formal logic of Bayesian updating (Kruschke, 2015). By carrying the posterior distribution from the earlier model forward as the prior for the present study, we embed knowledge from previous inferences into the probabilistic structure of the model and allow uncertainty from the earlier analysis to inform the new inference through a principled updating mechanism. The methodology adopted for this comparative analysis is detailed in the following subsection.

2.4.1 Bayesian Data Analysis

To investigate the problem posed by RQ_1 and RQ_2, we adopt Bayesian Data Analysis (BDA) rather than traditional frequentist methods, which, although prevalent in Software Engineering research (Boehm et al., 2005; Wohlin et al., 2012), have notable shortcomings. The frequentist approach to inference, particularly Null Hypothesis Significance Testing (NHST), tends to oversimplify complex, context-sensitive data into binary decisions based on arbitrary significance thresholds, often overlooking the uncertainty inherent in empirical observations (Nickerson, 2000; Gelman and Loken, 2014; Dienes and McLatchie, 2018). Moreover, issues such as p-hacking and an overemphasis on mean differences further undermine the reliability and expressiveness of frequentist inferences (Menzies and Shepperd, 2019).

In contrast, BDA emerges as a principled alternative that somewhat mitigates these limitations (Dienes and McLatchie, 2018). Rather than producing a binary outcome, BDA models variables as probability distributions and makes prior assumptions explicit. These assumptions are then updated in light of observed data using Bayes' Theorem, enabling a more nuanced and uncertainty-aware interpretation of the results.

Although BDA is conceptually straightforward, fully probabilistic models often yield analytically intractable expressions, which has hindered its widespread adoption for many years. Over the past decade, advancements in computational resources, coupled with the development of state-of-the-art Bayesian statistical tools, have significantly increased the accessibility of BDA. Building on this progress, statisticians have introduced systematic guidelines for conducting Bayesian modeling and inference (van de Schoot et al., 2021). In tandem, researchers have worked to tailor these methodologies to the context of empirical Software Engineering, offering domain-specific guidance to facilitate their adoption (Furia et al., 2021, 2022). As a result, although still gaining traction within the Software Engineering community, there has been a growing shift toward model and parameter estimation as an alternative to frequentist approaches for analyzing experimental data (Gigerenzer, 2004).

3 Results

We run our experiment on a machine with a 16-core AMD Ryzen 7 7435HS processor, 15.35 GiB of RAM, and an NVIDIA GeForce RTX 4050 Max-Q GPU. As previously mentioned, we executed the three test data generation approaches on the programs listed in Table 1, measuring their energy consumption (in Joules) and the time required to generate test code. Table 2 provides a summary of these results. Specifically, it reports, for each program–approach combination, the mean energy consumption, the peak (maximum) energy recorded during execution, and the total duration of the test generation process.

It is important to note a key distinction between the hardware usage profiles of the three approaches. In the case of Phi and Qwen, the majority of energy consumption is attributable to GPU usage, reflecting the computational demands of SLM inference. In contrast, Pynguin's energy expenditure is primarily associated with CPU activity, consistent with its design as a conventional test generation tool. The results in Table 2 would seem to suggest that this difference in hardware utilization has an effect on overall energy consumption. Specifically, Phi and Qwen tend to consume more energy across most programs, with average total energy consumptions of approximately 1,498 and 1,398 Joules, respectively, compared to approximately 958 Joules for Pynguin. Therefore, on average (i.e., mean), generating test code with Phi requires approximately 1.6 times more total energy than with Pynguin (approximately 1,498 and 958 Joules, respectively). Qwen exhibits a comparable energy usage pattern, with a mean total energy consumption of 1,398 Joules, which is approximately 1.46 times higher than that of Pynguin, and only slightly lower than Phi's.

When comparing mean energy consumption per execution, Phi also consumes significantly more energy than Pynguin: 70.91 per program versus 27.59, representing a ratio of roughly 2.6 to 1. Qwen follows the same trend, with a mean energy consumption of 75.36 per program, corresponding to an approximate 2.7 to 1 ratio relative to Pynguin. It is worth noting that this value is marginally higher than that observed for Phi under the same metric.

Overall, these results indicate that both SLM-based approaches (i.e., Phi and Qwen) consistently incur substantially higher energy costs than Pynguin, both in terms of total energy consumption and per-execution mean energy,

Table 2. Energy consumption summary per program for Phi, Pynguin, and Qwen.

Program	Phi				Pynguin				Qwen			
	Mean	**Max**	**Total**	**Duration‡**	**Mean**	**Max**	**Total**	**Duration‡**	**Mean**	**Max**	**Total**	**Duration‡**
`find_first_in_sorted`	58.87	93.59	1,648.22	28	25.74	29.19	**1,595.75**	62	71.35	90.70	1,355.59	19
`find_in_sorted`	71.50	87.63	1,644.52	23	29.32	37.83	293.20	10	78.84	91.99	1,340.24	17
`flatten`	78.06	92.79	**2,029.48**	26	26.10	28.08	1,566.02	60	78.78	90.65	1,418.02	18
`get_factors`	74.33	91.99	1,412.20	19	27.30	29.86	273.00	10	73.69	89.96	1,326.36	18
`hanoi`	73.06	87.64	1,315.02	18	30.33	47.30	1,486.36	49	78.56	89.72	1,414.02	18
`is_valid_parenthesization`	59.35	86.78	771.52	13	28.25	47.22	988.86	35	74.16	90.26	1,260.80	17
`kheapsort`	74.88	90.65	1,497.51	20	27.15	43.23	**1,819.38**	67	73.46	90.56	**1,542.76**	21
`kth`	82.47	89.38	1,979.20	24	26.67	30.14	346.76	13	78.79	91.53	1,260.67	16
`lcs_length`	75.37	89.13	1,431.97	19	27.78	30.61	666.66	24	72.18	91.36	1,371.50	19
`lis`	75.61	93.03	**2,041.37**	27	26.77	29.49	160.64	6	71.47	90.20	1,357.99	19
`max_sublist_sum`	68.30	84.24	887.85	13	27.43	29.94	137.13	5	77.93	90.18	1,246.86	16
`minimum_spanning_tree`	75.82	88.32	1,668.02	22	23.25	27.02	837.09	36	79.04	90.85	1,501.82	19
`possible_change`	58.74	85.60	1,527.21	26	25.57	27.70	230.09	9	73.49	90.57	**1,543.33**	21
`sieve`	67.35	84.03	1,144.95	17	24.03	26.95	841.20	35	76.66	91.14	**1,533.23**	20
`sqrt`	60.13	84.98	1,022.21	17	24.15	28.34	1,497.10	62	72.18	90.19	1,443.61	20
`subsequences`	71.56	92.34	1,216.54	17	34.72	41.96	1,388.99	40	72.82	90.06	1,529.15	21
`to_base`	80.14	94.03	**2,243.81**	28	34.38	48.80	**2,165.93**	63	77.76	91.06	1,321.85	17
Mean	70.91	89.18	1,498.92	21	27.59	34.33	958.48	34	75.36	90.65	1,398.11	18.59
Median	73.06	89.13	1,497.51	20	27.15	29.94	841.20	35	74.16	90.57	1,371.50	19
Standard Deviation	7.63	3.39	419.43	4.95	3.19	8.06	660.08	22.89	2.99	0.61	103.07	1.66

‡ Duration is measured in seconds.

while Qwen exhibits slightly lower total energy than Phi but marginally higher mean energy consumption per program.

The standard deviations reported in Table 2 provide insight into the consistency of each approach across programs. Among the three alternatives, Qwen appears to be the most predictable in terms of both maximum energy consumption and execution time. Its standard deviation for maximum power is 0.61, substantially lower than that of Phi (3.39) and Pynguin (8.06). Similarly, its variability in execution duration is 1.66 seconds, compared to 4.95 seconds for Phi and 22.89 seconds for Pynguin. In addition, Qwen exhibits markedly lower dispersion in total energy consumption (103.07 Joules) than Phi (419.43 Joules) and Pynguin (660.08 Joules), further indicating a stable energy consumption profile across the evaluated programs.

Phi also demonstrates a relatively consistent pattern of behavior. Although its variability in maximum power and execution time is higher than that of Qwen, it remains considerably lower than that observed for Pynguin. This suggests that Phi's performance is reasonably stable across programs, though not to the same extent as Qwen. In contrast, Pynguin presents the highest standard deviations across all metrics, particularly in total energy consumption and execution time. This greater dispersion indicates that its energy consumption is more strongly influenced by the specific structural characteristics of each program. Taken together, these results suggest that Qwen exhibits the most stable behavior in terms of energy consumption.

For Phi, the programs with the highest total energy consumption (highlighted in Table 2) are `to_base` (2,243.81), `lis` (2,041.37), and `flatten` (2,029.48). These programs also correspond to some of the longest execution times observed for Phi, ranging from 26 to 28 seconds.

For Pynguin, the three highest energy-consuming programs are `to_base` (2,165.93), `kheapsort` (1,819.38), and `find_first_in_sorted` (1,595.75). Among these, `to_base` also appears in the top three highest energy-consuming programs for Phi, suggesting that it represents a particularly challenging case across different test generation paradigms. Moreover, these programs correspond to some of the longest execution times for Pynguin, notably 63 seconds for `to_base` and 67 seconds for `kheapsort`.

For Qwen, the programs with the highest total energy consumption (highlighted in Table 2) are `possible_change` (1,543.33), `kheapsort` (1,542.76), and `sieve` (1,533.23). Notably, `kheapsort` consistently appears among the three highest energy-consuming programs for both Pynguin and Qwen, suggesting that it represents a particularly demanding case across these two approaches. Differently from Phi and Pynguin, Qwen's highest energy values do not coincide with substantially longer execution times, as its durations remain tightly clustered between 17 and 21 seconds across all programs. This suggests that, for Qwen, elevated total energy consumption is less associated with prolonged runtime and more likely attributable to sustained computational intensity during inference.

Figure 1 presents pairwise per-program comparisons of total energy consumption across the three test generation approaches. In Figure 1 (a), the differences between Phi and Pynguin indicate that Phi consumes substantially more energy than Pynguin for the majority of programs. Phi incurred a higher energy cost, with the most substantial differences observed in `lis`, `kth`, and `find_in_sorted`. We surmise that these cases reflect programs where the inference process required by Phi's SLM architecture resulted in sustained GPU utilization. In contrast, Phi outperformed Pynguin in terms of energy efficiency for a smaller subset of programs, most notably `sqrt`, `kheapsort`, and `is_valid_parenthesization`.

Phi consumed more energy than Pynguin in 12 out of the 17 programs evaluated. Although this indicates that Phi tends to incur higher energy costs overall, the results also reveal notable variability in energy efficiency between the two test generation approaches. This variability suggests that energy consumption is program-dependent and may be influenced by the structural and computational characteristics of each subject program. These results are consistent with the descriptive statistics reported in Table 2, where Phi's average

total energy consumption exceeds that of Pynguin by a considerable margin.

Figure 1 (b) shows a more nuanced comparison between the two SLM-based approaches. For a subset of programs (i.e., `lis`, `kth`, `to_base`, `flatten`, and `find_first_in_sorted`) Phi consumes substantially more energy than Qwen, with differences exceeding 600 Joules in several cases and reaching over 900 Joules for `to_base`. These large positive gaps suggest that, for computationally demanding programs, Phi tends to incur higher total energy costs than Qwen. Conversely, there are several programs for which Qwen consumes more energy than Phi. Notable examples include `is_valid_parenthesization`, `sqrt`, `sieve`, and `max_sublist_sum`, where the differences are negative and range from approximately 300 to nearly 500 Joules in magnitude. These cases indicate that Qwen does not uniformly outperform Phi in terms of energy efficiency; rather, the relative performance depends on the specific structural characteristics of the program. Importantly, the magnitude of the positive differences generally exceeds that of the negative ones.

While Qwen outperforms Phi in several instances, the largest gaps favor Qwen, particularly in high-consumption programs such as `to_base` and `lis`. This suggests that although both approaches exhibit program-dependent variability, Phi appears more prone to spikes in energy consumption.

As highlighted in Figure 1 (c), Qwen consumes more energy than Pynguin in most cases. Programs such as `possible_change`, `lis`, `max_sublist_sum`, `get_factors`, and `find_in_sorted` exhibit energy gaps exceeding 1,000 Joules in favor of Pynguin. These large deviations suggest that Qwen incurs significantly higher total energy costs than the conventional test generation approach. Overall, the results suggest that Qwen generally requires more total energy than Pynguin across the evaluated programs.

To complement the pairwise comparisons, Figure 2 provides an overview of the absolute per-program energy consumption across all three test generation approaches: the grouped bars in Figure 2 enable a direct comparison of absolute energy usage across all evaluated programs. Taken together, the broader trend indicates that the SLM-based approaches tend to incur higher energy costs compared to Pynguin.

3.1 Bayesian Estimation of Energy Consumption

As outlined in Subsection 2.4.1, NHST remains a prevalent method for comparing empirical samples, yet its practical limitations are well established: researchers must make a series of arbitrary decisions that seldom take the specific RQ into account, and the resulting p-value provides only indirect and frequently overstated evidence (Nickerson, 2000; Dienes and McLatchie, 2018). Consequently, we adopted a more informative, estimation-oriented framework grounded in Bayesian probability rather than frequentist inference. Instead of merely assessing whether the three test generation approaches differ, our analysis quantifies how different they are (i.e., we focus on estimating the effect size), thus providing richer and more meaningful insight.

The first step in Bayesian parameter estimation involves specifying a probability model that appropriately reflects the data-generating process under investigation. In our case, we selected the Student-t distribution to model the energy consumption associated with each unit test generation approach. This choice was informed by an initial examination of the results, which indicated the presence of potential outliers (Durelli et al., 2025). Compared to the normal distribution, the Student-t distribution offers increased robustness to outliers, thereby enhancing the reliability of our inferences.

In our previous study, we employed this modeling strategy to compare two approaches, Phi and Pynguin, following the framework proposed by Kruschke (2013). The model included one mean and one standard deviation parameter for each group, as well as a shared degrees-of-freedom parameter governing the shape of the t-distribution. This formulation allowed us to estimate differences in energy consumption while accounting for deviations from normality.

In the present follow-up study, we extend the analysis to include Qwen. Rather than specifying priors independently of previous findings, we adopt a Bayesian updating strategy grounded in the similarity between the SLM-based approaches to test generation. Specifically, because both Phi and Qwen are SLM-based approaches, we use the posterior distribution obtained for Phi in the previous analysis to inform the prior distribution for Qwen's parameters in the expanded model. This approach reflects the formal logic of Bayesian updating (Kruschke, 2015): information gained from one analysis is propagated forward as prior knowledge in a subsequent, related investigation. The new data associated with Qwen then update this informed prior to yield a posterior distribution that coherently integrates prior evidence with newly observed information. Specifically, our model incorporates seven parameters: one mean and one standard deviation for each group, as well as a *normality* parameter (i.e., a degrees-of-freedom parameter), which reflects the use of a t-distribution to account for potential deviations from normality in the data. Formally:

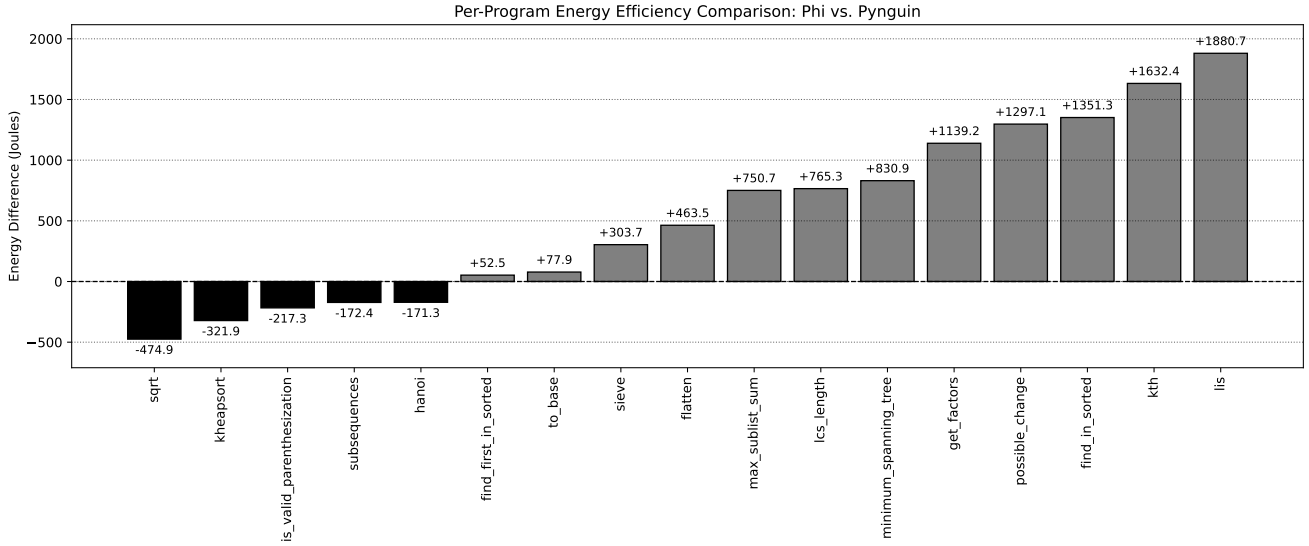
$$y^{(\text{Phi})} \sim \mathcal{T}(\nu,\, \mu_1,\, \sigma_1)$$

$$y^{(\text{Pynguin})} \sim \mathcal{T}(\nu,\, \mu_2,\, \sigma_2)$$

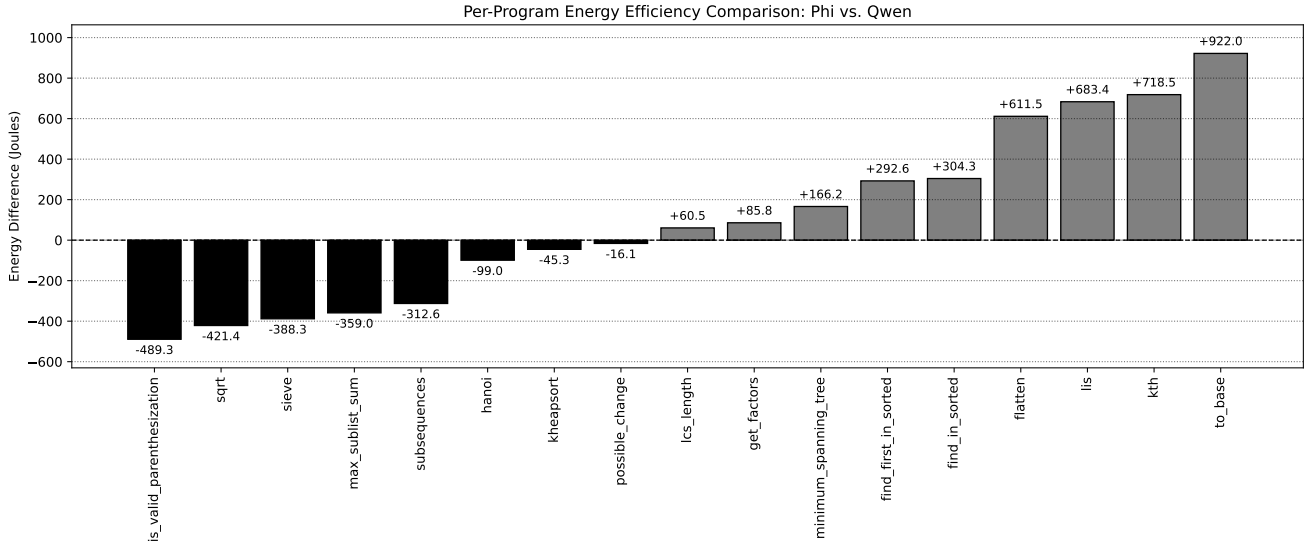
$$y^{(\text{Qwen})} \sim \mathcal{T}(\nu,\, \mu_3,\, \sigma_3)$$

where $\mathcal{T}(\nu, \mu, \sigma)$ denotes the Student-t distribution with location μ, scale σ, and degrees of freedom ν. The parameter ν controls the tail behavior of the distribution: large values of ν approximate a Gaussian distribution, whereas smaller values allow for heavier tails. As in the previous study, we assume a common degrees-of-freedom parameter across all approaches, thereby modeling tail behavior as a shared property of the energy measurements.

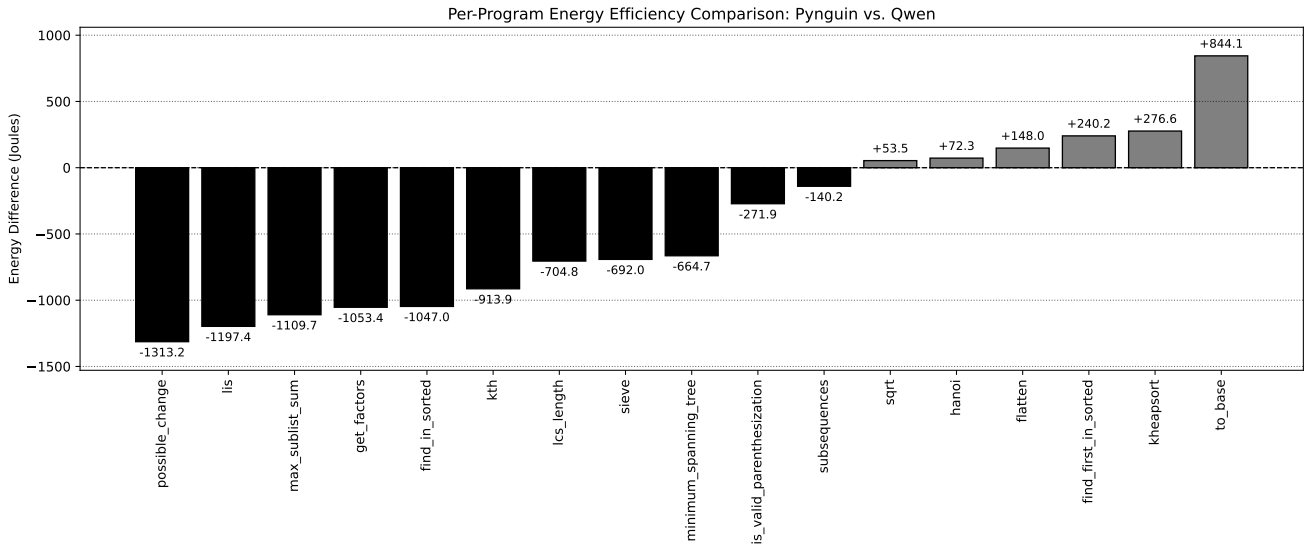
For Phi and Pynguin, we retain the weakly informative priors used in the original analysis:



(a) Phi vs. Pynguin



(b) Phi vs. Qwen



(c) Pynguin vs. Qwen

Figure 1. Per-program comparisons of total energy consumption across all pairwise combinations of approaches. Positive values (in gray) indicate that the first approach in each pairwise comparison incurred higher total energy consumption than the second, whereas negative values (in black) indicate the opposite.

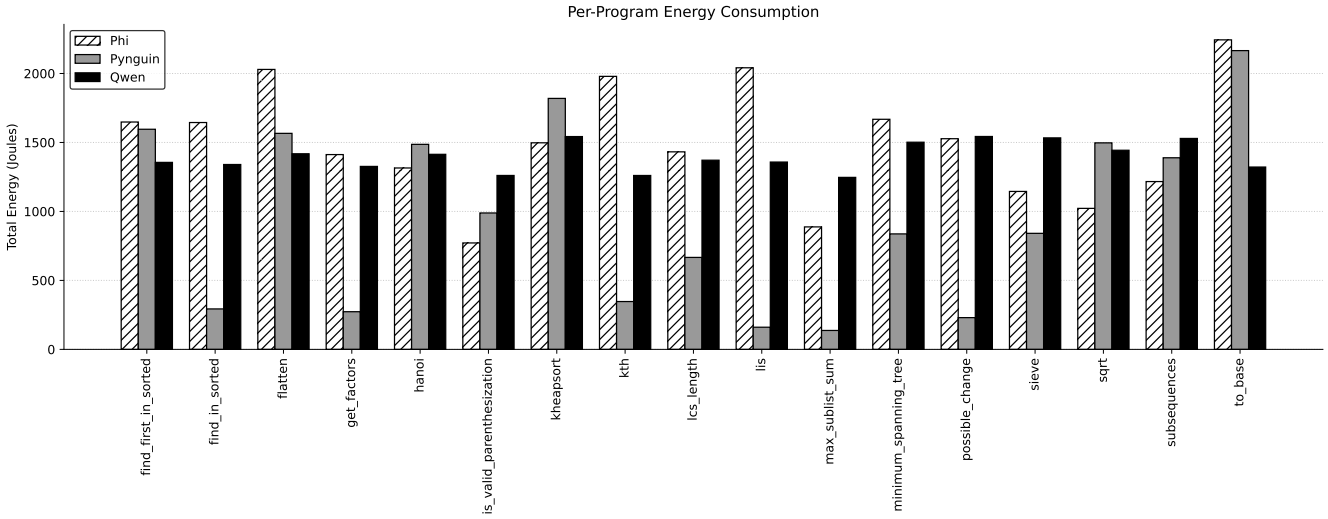


Figure 2. Per-program total energy consumption (in Joules) for Phi, Pynguin, and Qwen.

$$\mu_k \sim \mathcal{N}(\bar{x},\ 2s), \qquad \text{for } k \in \{1,\ 2\}, \tag{1}$$

$$\sigma_k \sim \mathcal{U}(1,\ 10), \qquad \text{for } k \in \{1,\ 2\}, \tag{2}$$

$$\nu \sim \text{Exp}(1/30) \tag{3}$$

In our previous study, we opted for weakly informative priors to reflect our limited prior knowledge. The normal prior in (1) is *diffuse* (i.e. with standard deviation $2s$) and thus non-informative with respect to either unit test code generator. The uniform prior in (2) guaranties a positive scale but places minimal structure on σ_k. The exponential prior in (3) follows the recommendation of Kruschke (2013), yielding high probability mass in the range where the t distribution mimics both Gaussian and heavy-tailed behavior.

However, for Qwen ($k = 3$), we adopt an informed prior derived from the posterior distribution of Phi obtained in the previous study. Formally:

$$\mu_3 \sim \mathcal{N}(\hat{\mu}_{\text{Phi}},\ \hat{\sigma}_{\mu,\text{Phi}}), \tag{4}$$

$$\sigma_3 \sim \mathcal{N}^{+}(\hat{\sigma}_{\text{Phi}},\ \hat{\sigma}_{\sigma,\text{Phi}}) \tag{5}$$

where $\hat{\mu}\text{Phi}$ and $\hat{\sigma}\text{Phi}$ in (4) denote posterior summaries from the previous analysis, and $\mathcal{N}^{+}$ in (5) indicates a truncated normal distribution enforcing positivity.

After fully specifying the model, we examined the posterior distributions of the stochastic parameters, as shown in Figure 3. These posteriors quantify the uncertainty in the mean and standard deviation of per-second energy consumption for each test generation approach. The model was implemented and sampled using `PyMC` (Abril-Pla et al., 2023)[4], a probabilistic programming framework for Bayesian inference.

As shown in Figure 3, the posterior distribution of the mean energy consumption for Phi is centered around 68, with a 94% highest density interval (HDI) approximately spanning [68.2, 68.5]. As for Pynguin, the posterior mean is substantially lower, centered near 21, with a 94% HDI of roughly [20.7, 21.2]. In the present follow-up analysis, the posterior for Qwen is centered near 68 as well, with a 94% HDI concentrated in a similarly narrow range (approximately [68.1, 68.5]). The posterior distributions for Phi and Qwen exhibit substantial overlap, while the posterior for Pynguin is located in a clearly distinct and lower range of values. This indicates that, given the model and observed data, the expected per-second energy consumption of the two SLM-based approaches is markedly higher than that of Pynguin.

The posterior distributions of the standard deviations, shown in the three lower panels of Figure 3, reveal a comparable pattern. For Phi and Qwen, the posterior means of the standard deviation are concentrated around 1.1 and 1.3, respectively, with relatively narrow 94% HDIs. In contrast, Pynguin exhibits a larger posterior mean standard deviation (approximately 1.8), with its corresponding 94% HDI shifted toward higher values. These results suggest that Pynguin's per-second energy consumption displays greater variability, whereas the SLM-based approaches appear more tightly concentrated around their respective means.

Within the Bayesian framework, the 94% HDI represents the interval containing the most credible 94% of posterior mass, such that every value inside the interval has a higher posterior density than any value outside it (Kruschke, 2015). Differently from frequentist confidence intervals, HDIs allow direct probabilistic statements about parameter values conditional on the model and observed data. Accordingly, the clear separation between the posteriors of the SLM-based approaches and Pynguin reflects strong posterior evidence of systematic differences in mean energy consumption, while the near-complete overlap between Phi and Qwen suggests that their expected energy usage is practically indistinguishable under the present modeling assumptions (Figure 4). A summary of the posterior distributions is provided in Table 3.

Looking at the differences in Figure 5, we examine posterior contrasts in the mean per-second energy consumption across the three test generation approaches. As in the previous study, zero was used as a reference value, allowing us to assess the proportion of posterior mass lying above or below zero for each comparison. In this follow-up analysis, the inclusion of Qwen enables a reassessment of the previously reported differences between Phi and Pynguin

[4] `https://www.pymc.io/welcome.html`

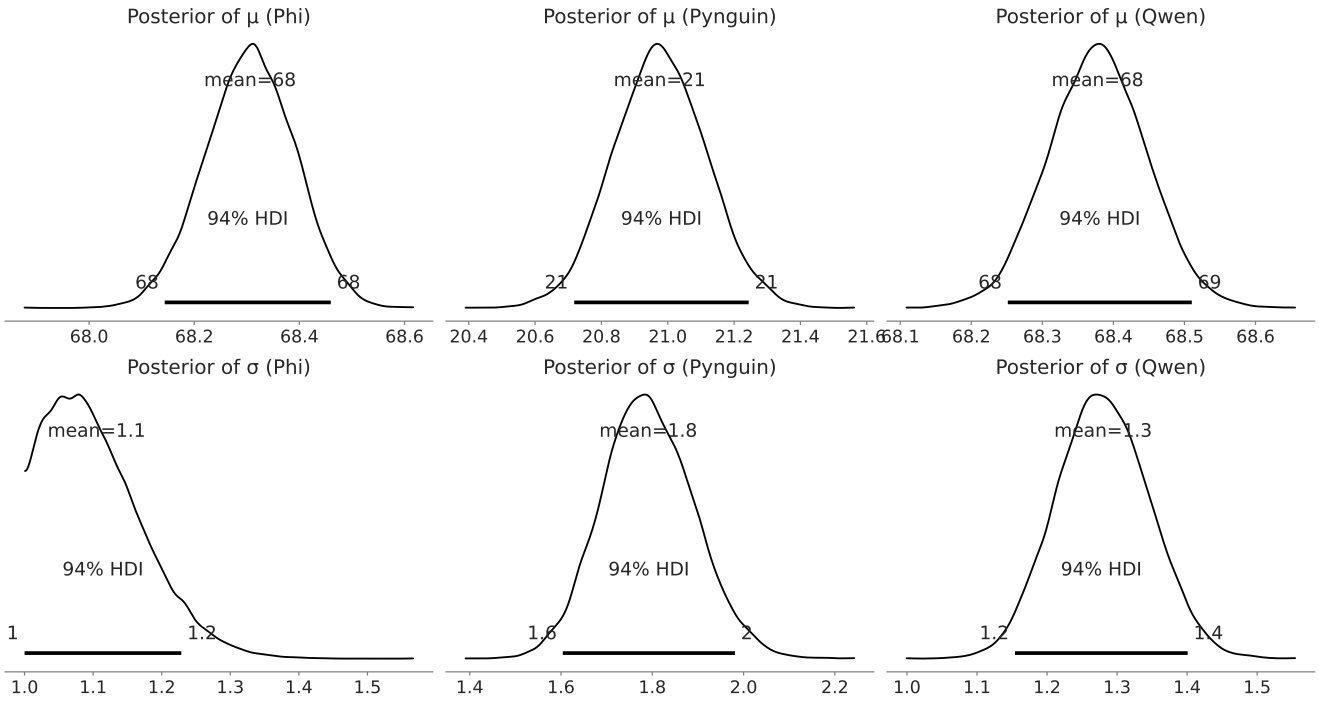


Figure 3. Summarized posterior distributions for the parameters of the energy consumption model.

(Kifetew et al., 2025) while situating Qwen within the same inferential framework.

For the *difference in means* ($\mu_{\text{Phi}} - \mu_{\text{Pynguin}}$), the posterior distribution remains tightly concentrated well above zero, with a mean difference of approximately 47 units and a 94% HDI of [47, 48]. Virtually all posterior mass lies above zero ($0.0\% < 0 < 100.0\%$), providing strong posterior evidence that Phi consumes more energy per second than Pynguin. To quantify the standardized difference, we computed Hedges' g, a bias-corrected effect size appropriate for unequal sample sizes, defined as:

$$g = d \times \left(1 - \frac{3}{4(n_1 + n_2) - 9}\right), \tag{6}$$

where d denotes Cohen's d and n_1, n_2 are the respective sample sizes. We derived a posterior distribution for d using the weighted pooled standard deviation and subsequently applied the small-sample correction to obtain Hedges' g. The posterior mean of Hedges' g is approximately 3.147, indicating a very large standardized effect. Thus, even after adjusting for sample size, Phi's energy consumption substantially exceeds that of Pynguin.

The comparison between Qwen and Pynguin indicates a nearly identical pattern. The posterior distribution of $(\mu_{\text{Qwen}} - \mu_{\text{Pynguin}})$ is clearly separated from zero, with a mean difference again close to 47 units and a 94% HDI excluding zero entirely. All posterior mass lies above zero, and the corresponding Hedges' $g \approx 3.588$ indicates a very large standardized effect size. These results suggest that the elevated energy consumption observed for Phi relative to Pynguin is not specific to that model, but rather appears to characterize SLM-based test generation more broadly.

Table 3. Summary of the posterior distributions for the three test generation approaches.

Parameter	Mean	SD	HDI 3%	HDI 97%
μ_{Phi}	68.294	0.086	68.134	68.453
μ_{Pynguin}	21.007	0.137	20.744	21.257
μ_{Qwen}	68.307	0.085	68.142	68.472
σ_{Phi}	1.148	0.087	1.000	1.295
σ_{Pynguin}	1.842	0.101	1.654	2.032
σ_{Qwen}	1.302	0.079	1.150	1.450

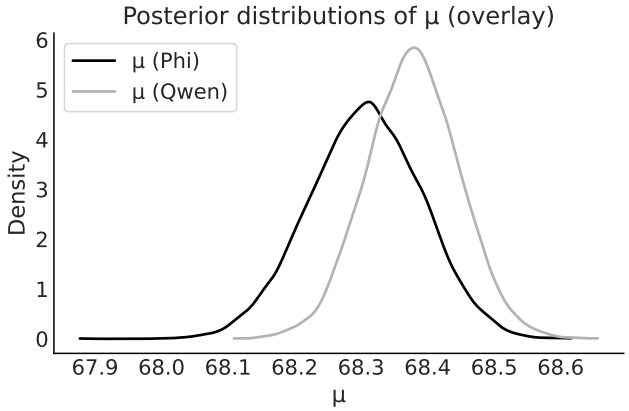


Figure 4. Posterior density overlay of the mean per-second energy consumption (μ) for Phi and Qwen.

In contrast, the comparison between Phi and Qwen reveals no meaningful difference. The posterior distribution of $(\mu_{\text{Phi}} - \mu_{\text{Qwen}})$ is centered near zero (i.e., mean ≈ -0.072), with a 94% HDI spanning both negative and positive values. Approximately 73.9% of the posterior mass lies below zero and 26.1% above zero, indicating no clear directional dominance. The associated Hedges' $g \approx -0.003$ is effectively negligible, suggesting that the two SLM-based approaches exhibit comparable mean energy consumption.

Our results provide strong posterior evidence that SLM-based approaches to unit test generation (i.e., Phi and Qwen)

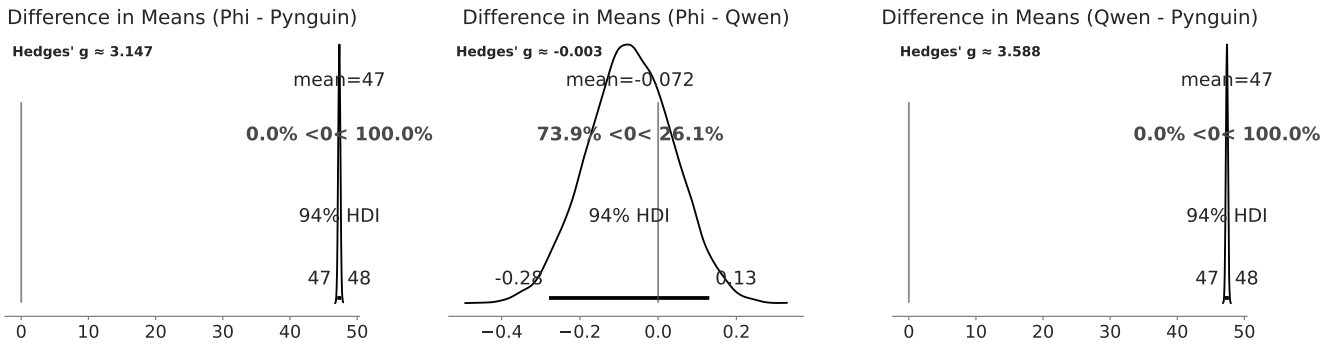


Figure 5. Posterior distributions of the pairwise differences in mean per-second energy consumption among Phi, Pynguin, and Qwen. Each panel displays the posterior distribution of the mean difference along with the corresponding Hedges' g. The results indicate that both Phi and Qwen consume substantially more energy than Pynguin, whereas the difference between Phi and Qwen is practically negligible.

incur substantially higher energy costs than a lightweight, purpose-built alternative (i.e., Pynguin). The posterior distributions of the pairwise mean differences indicate that both Phi and Qwen consistently consume markedly more energy per second than Pynguin, with very large standardized effect sizes (Hedges' $g > 3$ in both comparisons). In contrast, the posterior comparison between Phi and Qwen reveals no meaningful difference in energy consumption, suggesting that the additional energy cost is characteristic of the SLM-based paradigm rather than of a specific model implementation.

Key insights for RQ$_1$: our findings suggest that the additional energy cost associated with SLM-based unit test generation is substantial. *From a sustainability perspective, our results underscore the importance of explicitly evaluating the energy footprint of language-model-based approaches before their adoption in Software Engineering workflows.*

When examining the results in light of RQ$_2$, we assess whether the energy cost associated with SLM-based unit test generation varies meaningfully across different SLM architectures. Our findings indicate that the energy expenditure exhibits minimal variation between the two evaluated SLMs (Phi and Qwen). The posterior distribution of the mean difference between Phi and Qwen is centered near zero, with a 94% HDI spanning both positive and negative values and substantial posterior overlap. The corresponding standardized effect size (Hedges' $g \approx -0.003$) is negligible, indicating no practically meaningful difference in mean energy consumption between the two models.

Key insights for RQ$_2$: our findings suggest that, within the scope of our experimental setup, the additional energy demand observed for SLM-based test generation appears largely invariant to architectural differences between the evaluated models. In other words, the increased energy cost would seem to be attributable primarily to the GPU-intensive inference paradigm itself rather than to specific design choices of a given SLM architecture.

3.2 A Brief Look Into the Fault-Finding Effectiveness of the Test Suites

Table 4 summarizes the fault-finding effectiveness of the test suites generated by Phi, Pynguin, and Qwen across all subject programs. The test suites produced by Phi and Pynguin were generated during the first phase of our experiment. For this expanded version of our study, Qwen was incorporated to enable a broader comparative analysis of effectiveness across LLM-based and conventional automated test generation approaches.

For each test generation approach and subject program, the table reports the number of test cases that did not reveal the fault (✗), the number that successfully uncovered it (✓), the total number of generated tests, and the percentage of fault-revealing cases (% ✓). In the context of our study, the latter provides a normalized measure of effectiveness, allowing for comparisons across approaches despite differences in test suite size.

Overall, when comparing Phi and Pynguin, Phi achieved a higher percentage of fault-revealing tests in 10 out of the 17 programs, whereas Pynguin outperformed Phi in the remaining 7. Despite this distribution, Pynguin exhibited a higher average fault-detection rate across programs (66.7%) compared to Phi (58.8%). Pynguin also achieved perfect effectiveness (100%) in `find_first_in_sorted` and `sqrt`, whereas Phi did not reveal the fault in either case. Conversely, Phi demonstrated notably high effectiveness in `to_base` (100%) and `kth` (90.0%), and its generated test suites were often larger, which may increase the likelihood of exposing subtle faults.

Qwen achieved the highest fault-detection rate in 5 of the 17 programs (including `find_in_sorted`, `flatten`, `hanoi`, `lis`, and `max_sublist_sum`), and matched the best-performing approach in `find_first_in_sorted`. However, Qwen exhibited 0% fault-detection effectiveness in several programs, including `kth`, `minimum_spanning_tree`, `to_base`, and `subsequences`, and achieved comparatively low effectiveness in others.

Table 5 summarizes the overall fault-detection effectiveness of the test suites generated by each approach across all programs. Effectiveness is computed as the proportion of generated test cases that successfully reveal a fault. Both Phi (70.8%) and Pynguin (70.3%) achieved nearly identical aggregate effectiveness. Despite differences in their generation

paradigms, their overall ability to produce fault-revealing tests is comparable when results are pooled across programs. In contrast, Qwen (51.2%) demonstrates substantially lower aggregate effectiveness. Although Qwen generated a comparable number of total test cases (86) relative to Phi (106), a smaller proportion of these tests revealed faults. This suggests that, in general, Qwen's generated tests are less effective at exposing faults.

A closer examination of the results at the level of individual programs reveals substantial heterogeneity in fault-detection effectiveness across approaches. Although the aggregate statistics (Table 5) indicate that Phi and Pynguin achieved comparable overall effectiveness and outperformed Qwen at generating fault-revealing test cases, the program-level analysis demonstrates that these differences are not consistent across all programs.

First, there are programs for which all approaches achieve moderate to high effectiveness. Specifically, in `flatten`, `hanoi`, `lis`, and `max_sublist_sum`, each approach achieves comparatively high fault-detection rates. These results suggest that certain programs may be more amenable to automated test generation, potentially due to characteristics such as comparatively simple input domains or fault conditions that are triggered under common execution paths.

Second, several programs resulted in pronounced differences between approaches. Qwen, in particular, achieves 0% effectiveness on multiple programs, including `kth`, `minimum_spanning_tree`, `to_base`, and `subsequences`. In these instances, although test cases were generated, none revealed the faults. This pattern indicates that Qwen's test generation strategy may be less effective for certain classes of algorithmic problems. Nevertheless, challenging cases are not limited to Qwen. Phi generated no tests for `kheapsort` and `sqrt`. Similarly, Pynguin achieved 0% effectiveness on `find_in_sorted` and `sieve`. These instances highlight that neither LLM-based nor Pynguin performed well across all program types.

Collectively, these program-level patterns suggest a meaningful interaction between the test generation approach and intrinsic program characteristics. Structural properties (e.g., algorithmic complexity, recursion depth, and input constraints) are likely to influence the effectiveness of generated test suites. Accordingly, we recommend that future investigations into the fault-revealing capabilities of LLM-based test generation approaches refrain from modeling fault-detection effectiveness as a fixed property of a given generation paradigm. Instead, effectiveness should be analyzed within a hierarchical modeling framework that explicitly accounts for between-program variability. Such an approach provides a statistically principled basis for inference by accommodating heterogeneity across programs and yielding more stable estimates of approach-level effects through partial pooling. In addition, hierarchical modeling facilitates the estimation of program-level difficulty parameters, thereby allowing for a more nuanced understanding of which categories of programs present greater challenges to automated test generation techniques and the extent to which these challenges vary across approaches.

Key insights for RQ$_3$: Phi and Pynguin achieve comparable overall fault-detection effectiveness (approximately 71%), whereas Qwen exhibits lower overall effectiveness (approximately 51%). However, no approach consistently outperforms the others across all subject programs. The findings indicate that fault-finding effectiveness is strongly context-dependent and influenced by the interaction between the test generation approach and intrinsic program characteristics. Thus, differences in effectiveness should not be interpreted as inherent advantages of a given approach, but rather as outcomes shaped by program-specific factors.

4 Threats to Validity

A potential threat to internal validity arises from the overhead introduced by the profiling tool, which may act as a confounding factor. Specifically, we cannot rule out the possibility that the tool used to profile energy consumption may have inadvertently affected the energy expenditure measurements. However, it is worth noting that the overhead introduced by PowerJoular (Noureddine, 2022) is relatively low and is typically considered acceptable for most profiling and energy usage analysis experiments. Future replications should consider complementary measurement strategies, such as external power meters or repeated calibration runs, to further assess the robustness of the measurements.

Another threat to internal validity concerns the hardware configuration used in our experiment. The experiments were executed on a single machine equipped with a multi-core CPU and a GPU. This configuration is particularly relevant because the evaluated approaches rely on different computational resources: Pynguin primarily executes on the CPU, whereas the SLM-based approaches rely heavily on GPU-accelerated inference. Consequently, part of the observed difference in energy consumption may reflect the interaction between each approach and the underlying hardware architecture, rather than only intrinsic differences between the test generation techniques. For instance, a different GPU, CPU, cooling configuration, or power-management policy could affect the absolute energy measurements and, potentially, the magnitude of the observed differences. Thus, our results should be interpreted as evidence obtained under a specific hardware setup, not as hardware-independent estimates of energy consumption.

A potential threat to external validity lies in the evaluation of a single conventional test generation tool for Python. This may limit the generalizability of our findings to other unit test generation tools or programming languages. However, Python is one of the most widely used programming languages in both academia and industry. Moreover, we contend that focusing on a well-established language offers a practical and representative context for assessing the capabilities of contemporary test generation approaches.

Another potential threat to external validity lies in the use of a single benchmark. Our evaluation was conducted on a subset of Python programs from the QuixBugs bench-

Table 4. Fault-finding effectiveness of test suites generated by Phi, Pynguin, and Qwen.

Program	Phi				Pynguin				Qwen			
	✗	✓	Total	% ✓	✗	✓	Total	% ✓	✗	✓	Total	% ✓
`find_first_in_sorted`	7	2	9	22.2%	0	6	6	100.0%	0	6	6	100.0%
`find_in_sorted`	3	5	8	62.5%	3	0	3	0.0%	2	6	8	75.0%
`flatten`	1	6	7	85.7%	1	2	3	66.7%	0	6	6	100.0%
`get_factors`	2	7	9	77.8%	1	2	3	66.7%	3	1	4	25.0%
`hanoi`	5	3	8	37.5%	1	2	3	66.7%	1	3	4	75.0%
`is_valid_parenthesization`	6	4	10	40.0%	1	4	5	80.0%	1	0	1	0.0%
`kheapsort`	0	0	0	0.0%	1	3	4	75.0%	1	0	1	0.0%
`kth`	1	9	10	90.0%	1	4	5	80.0%	8	0	8	0.0%
`lcs_length`	0	0	0	0.0%	1	4	5	80.0%	2	4	6	66.7%
`lis`	4	3	7	42.9%	1	1	2	50.0%	5	11	16	68.8%
`max_sublist_sum`	3	2	5	40.0%	1	1	2	50.0%	1	4	5	80.0%
`minimum_spanning_tree`	1	5	6	83.3%	1	3	4	75.0%	9	0	9	0.0%
`possible_change`	2	5	7	71.4%	1	2	3	66.7%	4	2	6	33.3%
`sieve`	2	5	7	71.4%	2	0	2	0.0%	2	0	2	0.0%
`sqrt`	0	0	0	0.0%	0	6	6	100.0%	2	1	3	33.3%
`subsequences`	1	6	7	85.7%	1	3	4	75.0%	4	0	4	0.0%
`to_base`	0	6	6	100.0%	2	2	4	50.0%	7	0	7	0.0%

mark (Lin et al., 2017). Although QuixBugs is widely used in program repair and software testing research, the programs are relatively small algorithmic tasks and may not fully represent the characteristics of larger, industrial, object-oriented, or framework-dependent Python systems. In addition, we did not use all programs available in the benchmark because some programs posed practical difficulties related to timeouts and integration into our experimental framework. Thus, our findings may not generalize to all QuixBugs programs, to other Python benchmarks, or to software systems with different structural characteristics.

Table 5. Aggregate fault-detection effectiveness across all programs.

Approach	✓	Total Tests	Effectiveness (%)
Phi	75	106	70.8%
Pynguin	45	64	70.3%
Qwen	44	86	51.2%

External validity is also limited by the set of test generation approaches considered in this study. We evaluated one conventional unit test generation tool for Python and two SLMs. Although Pynguin is a relevant and purpose-built tool for automated unit test generation in Python, other conventional tools, search-based configurations, symbolic execution engines, or hybrid approaches may exhibit different energy and effectiveness profiles. Similarly, Phi and Qwen do not represent the full design space of language-model-based test generation. Different model sizes, quantization strategies, decoding parameters, prompt templates, inference frameworks, or hardware backends could lead to different results. Accordingly, our conclusions should be interpreted as applying to the specific combination of tools, models, prompts, benchmark programs, and execution environment evaluated in this study.

One threat to validity stems from the inherent variability in the outputs produced by the SLMs used in our experiments. Although an identical prompt was applied uniformly across all subject programs, the generated outputs exhibited inconsistencies in format and level of completeness across cases. Given that the prompt and experimental procedure were held constant, these variations cannot be attributed to differences in the experimental configuration. Rather, they reflect a broader limitation of prompt-based code generation (i.e., output characteristics may vary even under controlled and repeatable conditions, potentially affecting the consistency of the resulting test suites).

A threat to construct validity concerns the use of energy consumption as the main proxy for sustainability. Energy usage is an important and measurable dimension of computational sustainability, but it does not fully capture the broader environmental impact of a test generation approach. For instance, our analysis does not estimate carbon emissions. Additionally, we focus on the energy consumed during test suite generation and do not model downstream costs, such as test execution and maintenance of generated tests. Therefore, our findings should be interpreted as evidence about operational energy consumption during generation, rather than as a complete life-cycle assessment of sustainability.

Another construct validity threat relates to the operationalization of fault-finding effectiveness. In our study, effectiveness is measured by whether the generated test suites reveal faults in the faulty versions of the selected programs. This metric captures an important aspect of test suite quality, but it does not encompass other relevant dimensions, such as branch coverage, mutation score, readability, maintainability, or the effort required to integrate the generated tests into a development workflow. Consequently, an approach that performs well according to our effectiveness metric may still generate tests that are difficult to maintain or less useful in practice. Conversely, an approach with lower fault-finding effectiveness in our benchmark may still produce tests that are valuable under different quality criteria.

Although we compared energy consumption across approaches and subject programs, our study was not structured to support causal inference. Specifically, we did not isolate

specific program-level or model-level factors that may contribute to variations in energy consumption. As a result, while our analysis quantifies differences across approaches, it does not allow us to attribute these differences to specific structural characteristics of the programs (e.g., input size, control-flow complexity, recursion depth) or to internal properties of the test generation approaches (e.g., prompt sensitivity).

5 Related Work

Recent research by Kifetew et al. (2025) investigated the energy consumption of automated test generation using the EvoSuite tool for Java programs. They assessed various algorithms implemented within EvoSuite, measuring energy consumption during both test generation and execution phases. The study revealed significant variability in energy consumption across different test generation algorithms, notably influenced by the cyclomatic complexity of the programs under test. Additionally, their findings indicated that manually written test cases often consumed more energy compared to automatically generated ones without necessarily providing higher code coverage. In contrast to our research, Kifetew et al. focused on comparing various algorithms within a conventional test code generation tool for Java. Our study probes into energy consumption specifically associated with using a SLM (Phi-3.1 Mini 128k) for unit test generation in Python. Furthermore, while Kifetew et al. employed classical statistical methods (i.e., frequentist) to analyze energy consumption data, our approach leverages BDA, providing nuanced insights into energy consumption differences. Thus, we argue that our research contributes to the ongoing investigation into sustainable software testing practices by evaluating the environmental implications of employing language models versus conventional test generation tools.

6 Concluding Remarks

Motivated by sustainability concerns, we set out to investigate the energy footprint of employing SLMs for automated unit test generation. In this extended version of our prior study, we broadened the scope of the comparison by incorporating an additional SLM (i.e., Qwen) alongside Phi and Pynguin. This expanded design enables a more comprehensive assessment of both energy consumption and fault-detection effectiveness across LLM-based and conventional automated test generation approaches.

The main contribution of our study lies in the insights derived from addressing RQ_1, which examines differences in energy consumption among Phi, Qwen, and Pynguin. BDA, our results provide strong statistical evidence that both SLMs incur substantially higher energy costs than the conventional tool. In particular, posterior estimates indicate that Phi consistently consumes more energy than Pynguin across subject programs. Posterior estimates also suggest that Phi and Qwen exhibit comparable energy consumption levels, with overlapping uncertainty intervals, while both consistently consume significantly more energy than Pynguin across subject programs. These findings reinforce the conclusion that even smaller-scale language models introduce non-negligible energy overhead when compared to purpose-built test generation tools.

With respect to RQ_3, our analysis further demonstrates that differences in fault-detection effectiveness are program-dependent. Although Phi and Pynguin achieve comparable aggregate effectiveness, Qwen attains lower overall effectiveness, primarily due to pronounced variability across programs. It is worth noting that no single approach consistently dominates across all programs in our sample.

Our findings challenge the assumption that SLM-based automation necessarily offers a cost-effective alternative to specialized tools, at least from an energy perspective. At the same time, the variability observed in effectiveness underscores the need for more nuanced evaluation frameworks that account for contextual factors.

As a direction for future work, we encourage further investigation into the qualitative properties of the generated test suites, including coverage, maintainability, redundancy, and readability. Such investigation should be conducted while explicitly incorporating energy consumption as a constraint, thereby enabling a multi-objective evaluation of efficiency and test suite quality. A balanced assessment along these dimensions is essential for determining the practical viability of language-model-based approaches in sustainable Software Engineering contexts.

Future studies should assess the impact of different prompting strategies on both fault-finding effectiveness and energy consumption. For instance, few-shot prompting, chain-of-thought-style test planning, prompt templates that explicitly request boundary-value and exception-oriented tests, or prompts that constrain the number and structure of generated test cases may yield more effective test suites without necessarily increasing inference cost. Similarly, future work should examine whether iterative prompting strategies, in which failed or low-coverage tests are refined through feedback, lead to meaningful improvements in fault detection and whether such improvements justify the additional energy expenditure. Another promising direction concerns hybrid test generation approaches. Rather than treating SLMs and conventional tools as competing alternatives, future work could investigate pipelines in which lightweight tools such as Pynguin are used as the default generation mechanism, while SLMs are invoked only when conventional generation fails to cover relevant branches or expose faults. Such hybrid strategies may reduce unnecessary model inference while still exploiting the semantic flexibility of SLMs in cases where traditional search-based or heuristic techniques are less effective. Future work should also explore energy-aware model selection and resource-aware test generation. This includes comparing models of different sizes, quantization levels, inference backends, decoding parameters, and hardware configurations under fixed energy, time, or cost budgets. An energy-aware test generation framework could dynamically select between conventional tools, smaller models, and larger models depending on the expected difficulty of the program under test and the available resource budget. Such a framework would move beyond reporting energy consumption after the fact and instead treat energy as a first-class optimization criterion during test generation.

Acknowledgments

This work was partially supported by the National Council for Scientific and Technological Development – CNPq, Brazil (grants nr. 444311/2024-6 and 406941/2025-4).

References

Abdullin, A., Derakhshanfar, P., and Panichella, A. (2025). Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 221–232, NY, USA. ACM.

Abril-Pla, O., Andreani, V., Carroll, C., Dong, L., Fonnesbeck, C. J., Kochurov, M., Kumar, R., Lao, J., Luhmann, C. C., Martin, O. A., Osthege, M., Vieira, R., Wiecki, T., and Zinkov, R. (2023). PyMC: A modern and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science*, 9(e1516).

Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and Mcminn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001.

Anthropic (2025). Claude. `https://www.anthropic.com/index/claude`. Large language model.

Boehm, B., Rombach, H., and Zelkowitz, M. (2005). *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili*. Springer, Berlin, Heidelberg.

Castaño, J., Martínez-Fernández, S., Franch, X., and Bogner, J. (2023). Exploring the carbon footprint of hugging face's ml models: A repository mining study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, NY, USA. ACM.

Dienes, Z. and McLatchie, N. (2018). Four reasons to prefer bayesian analyses over significance testing. *Psychonomic Bulletin & Review*, 25(1):207–218.

Ding, Y. and Shi, T. (2024). Sustainable llm serving: Environmental implications, challenges, and opportunities : Invited paper. In *IEEE 15th International Green and Sustainable Computing Conference (IGSC)*, pages 37–38. IEEE.

Durelli, R., Endo, A., and Durelli, V. (2025). On the energy footprint of using a small language model for unit test generation. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*, pages 55–64, Porto Alegre, RS, Brasil. SBC.

Fontes, A., Gay, G., de Oliveira Neto, F. G., and Feldt, R. (2023). Automated support for unit test generation. In Romero, J. R., Medina-Bulo, I., and Chicano, F., editors, *Optimising the Software Development Process with Artificial Intelligence*, Natural Computing Series, pages 179–219. Springer.

Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291.

Furia, C. A., Feldt, R., and Torkar, R. (2021). Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, 47(9):1786–1810.

Furia, C. A., Torkar, R., and Feldt, R. (2022). Applying bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3).

Gelman, A. and Loken, E. (2014). The statistical crisis in science: Data-dependent analysis—a "garden of forking paths"—explains why many statistically significant comparisons don't hold up. *American Scientist*, 102(6):460–466.

Gigerenzer, G. (2004). Mindless Statistics. *The Journal of Socio-Economics*, 33(5):587–606.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223.

Jovanović, M. and Campbell, M. (2024). Compacting ai: In search of the small language model. *Computer*, 57(8):96–100.

Kifetew, F., Prandi, D., and Susi, A. (2025). On the energy consumption of test generation. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 360–370.

Kruschke, J. K. (2013). Bayesian estimation supersedes the t test. *Journal of Experimental Psychology: General*, 142(2):573–603.

Kruschke, J. K. (2015). *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan*. Academic Press, 2nd revised edition.

Lakhotia, K., McMinn, P., and Harman, M. (2010). An empirical investigation into branch coverage for c programs using cute and austin. *Journal of Systems and Software*, 83(12):2379–2391.

Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. (2017). Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH, pages 55–56, NY, USA. ACM.

Luccioni, A. S., Viguier, S., and Ligozat, A.-L. (2023). Estimating the carbon footprint of bloom, a 176b parameter language model. *The Journal of Machine Learning Research*, 24(1).

Lukasczyk, S. and Fraser, G. (2022). Pynguin: Automated unit test generation for python. In *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 168–172.

Lukasczyk, S., Kroiß, F., and Fraser, G. (2023). An empirical study of automated unit test generation for python. *Empirical Software Engineering*, 28(2):36.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156.

Menzies, T. and Shepperd, M. (2019). "bad smells" in software analytics papers. *Information and Software Technology*, 112:35–47.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. Wiley, 3rd edition.

Nickerson, R. S. (2000). Null hypothesis significance testing: A review of an old and continuing controversy. *Psychological Methods*, 5(2):241–301.

Noureddine, A. (2022). Powerjoular and joularjx: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments*, Biarritz, France.

OpenAI (2025). ChatGPT. `https://chat.openai.com`. Large language model.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

van de Schoot, R., Depaoli, S., King, R., Kramer, B., Märtens, K., Tadesse, M. G., Vannucci, M., Gelman, A., Veen, D., Willemsen, J., and Yau, C. (2021). Bayesian statistics and modelling. *Nature Reviews Methods Primers*, 1.

VanderStoep, S. W. and Johnson, D. D. (2008). *Research Methods for Everyday Life: Blending Qualitative and Quantitative Approaches*. Jossey-Bass.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

Ye, H., Martinez, M., Durieux, T., and Monperrus, M. (2021). A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171.