On the Energy Footprint of Using a Small Language Model for Unit Test Generation

Rafael S. Durelli
Universidade Federal de Lavras
Minas Gerais, Brazil
rafael.durelli@ufla.br

Andre T. Endo
Universidade Federal de São Carlos
São Paulo, Brazil
andreendo@ufscar.br

Vinicius H. S. Durelli
Universidade Federal de São Carlos
São Paulo, Brazil
vinicius.durelli@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 a small language model (SLM) 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 an SLM (Phi-3.1 Mini 128k) and Pynguin, a purpose-built tool for unit test generation.

Method. We posed two research questions: *(i)* What is the difference in energy usage between Phi and Pynguin during the generation of unit test suites for Python programs?; and *(ii)* To what extent do unit test suites generated by Phi and Pynguin differ in their fault-finding effectiveness? To rigorously address the first research question, we employed Bayesian Data Analysis (BDA). For the second research question, we conducted a complementary empirical analysis using descriptive statistics.

Results. Our Bayesian analysis provides robust evidence indicating that Phi consistently consumes 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

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 [23]. 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 [3]. 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 [2, 11, 16, 21].

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 [29]. 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 [1, 26]. However, despite the transformative impact of models such as ChatGPT [25] 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 [18]. 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 [6], 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 [12]. 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 – and even preferable – from an energy consumption standpoint. This trend motivates the central research question of this study: 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, we set out to compare and evaluate an SLM, i.e., Phi 3.1 Mini 128k, and a traditional test case generation tool, i.e., Pynguin [19, 20], with respect to their energy consumption and overall effectiveness in generating test cases. We investigate whether the approach that demonstrates superior energy efficiency also produces effective test cases. This study contributes to the ongoing discussion on sustainable Software Engineering by assessing the energy impact of two distinct test case generation approaches when applied to a benchmark of faulty Python programs [17].

The remainder of this paper is organized as follows: Section 2 details the experiment design, including subject program selection, experimental variables, and the rationale behind employing a Bayesian approach to data analysis. Section 3 describes the experimental execution. Section 4 presents results from our Bayesian analysis, highlighting differences in energy consumption and fault-detection effectiveness between the two test code generation approaches. Section 5 discusses threats to the validity. Section 6 presents related work. Section 7 summarizes key insights and implications of our research, emphasizing sustainability considerations in automated Software Engineering practices.

2 Study Design

We set out to compare Pynguin with Phi 3.1 Mini 128k[1] in terms of their energy consumption and overall quality of the generated unit tests. Specifically, the primary objectives of this investigation are to examine their respective energy consumption profiles and assess the overall effectiveness of the resulting test suites. We hypothesize that employing zero-shot prompting with an SLM does not yield test suites 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 by energy consumption but also evaluates whether any additional energy expenditure results in demonstrably more robust test suites capable of detecting a greater number of faults.

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 have 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 [7, 20], which, while computationally intensive in their own right, have been refined over 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 probe into these trade-offs and provide empirical evidence for these assertions, we framed the following research question (RQ):

> **RQ_1:** What is the difference in energy usage between Phi and Pynguin during the generation of unit test suites for Python programs?

RQ_1 focuses on comparing the energy consumption of the two approaches during the generation of unit test suites for a set of Python programs described in Subsection 2.1. Complementing this analysis, we formulated a second 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_2:** To what extent do unit test suites generated by Phi 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 the programs from the QuixBugs benchmark [17], 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 experimemt, we focus exclusively on the Python implementations. A distinctive feature of QuixBugs is that its programs were originally written by participants of the Quixey Challenge [17, 31].

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

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 your experiment study, including their size in lines of code (LoC) and the type of fault present in each faulty version [31]. Further details on the programs can be found in the replication package of this study.[2] 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.

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

2.2 Experiment Variables

To answer RQ_1 and RQ_2, we consider the unit test generation approach as the independent variable. Accordingly, the two treatments of this variable are: *(i)* an SLM-based approach (via zero-shot prompting) and *(ii)* a conventional unit test generation tool. For RQ_1, the dependent variable is the energy consumption, measured in Joules (*J*), incurred during unit test code generation. For RQ_2, 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* [28] 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.

[2]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.15809327.

2.3 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 two approaches. Given that our primary objective is to address RQ_1, we focus on modeling the differences in energy consumption between the two test data generation approaches. The methodology adopted for this comparative analysis is detailed in the following subsection.

2.3.1 Bayesian Data Analysis. To investigate the problem posed by RQ_1, we adopt Bayesian Data Analysis (BDA) rather than traditional frequentist methods, which, although prevalent in Software Engineering research [30], have notable shortcomings. The frequentist approach, 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. Moreover, issues such as p-hacking and an overemphasis on mean differences further undermine the reliability and expressiveness of frequentist inferences [22].

In contrast, BDA emerges as a principled alternative that somewhat mitigates these limitations. 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.

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 [27]. In tandem, researchers have worked to tailor these methodologies to the context of empirical Software Engineering, offering domain-specific guidance to facilitate their adoption [8, 9]. 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 [10].

3 Experiment Execution

Energy consumption is a key dimension of software sustainability and is commonly quantified in Joules (*J*) or kilowatt-hours (kWh) [4]. 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 [24], 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.

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.

4 Results

As previously mentioned, we executed both 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 two approaches. In the case of Phi, the majority of energy consumption is attributable to GPU usage, which reflects the computational demands of an SLM during the inference phase. In contrast, Pynguin's energy expenditure is primarily associated with CPU activity, which aligns 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 the overall energy consumption: Phi tends to consume more energy across most programs, with an average total energy consumption of 1,498.92, compared to 958.48 for Pynguin.

On average (i.e., mean), generating test code with Phi requires approximately 1.6 times more total energy than with Pynguin (1,498.92 and 958.48, respectively). When comparing mean energy consumption per execution, Phi also consumes significantly more energy than Pynguin: on average, 70.91 per program versus 27.59, representing a ratio of roughly 2.6 to 1.

The relatively low standard deviations reported in Table 2 suggest that Phi is a more predictable alternative in terms of both maximum energy consumption and execution time. This indicates that Phi's behavior is relatively consistent across the programs in our sample. In contrast, the higher standard deviations observed for Pynguin imply greater variability, suggesting that its energy consumption is more sensitive to the specific characteristics of each program.

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. As for Pynguin, the three most energy-demanding programs are `to_base` (2,165.93), `kheapsort` (1,819.38), and `flatten` (1,566.02). Notably, `to_base` and `flatten` appear in the top three for both test generation approaches, suggesting that these programs pose substantial challenges to both SLM-based and conventional test generation approaches. Test case generation for these programs also resulted in the longest execution times for Pynguin.

Figure 1 presents the per-program energy consumption gap between Phi and Pynguin. Positive values (in gray) indicate cases where Phi consumed more energy than Pynguin, whereas negative values (in black) denote scenarios in which Pynguin consumed more energy than Phi. The data indicate that for the majority of programs in our experiment, Phi incurred a higher energy cost, with the most substantial differences observed in `lis` (+1,880.7), `kth` (+1,632.4), and `find_in_sorted` (+1,351.3). 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` (-474.9), `kheapsort` (-321.9), and `is_valid_parenthesization` (-217.3). As shown in Figure 1, 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.

To complement the aggregate view previously discussed in Table 2 and Figure 1, we now shift our focus to a finer-grained analysis of energy consumption patterns. While summary statistics such as means and standard deviations offer a useful overview, they may obscure important temporal dynamics [5]. In this more detailed analysis, we examine the *raw*, per-second energy usage measurements collected during the test generation process. This zoomed-in perspective allows for the identification of fluctuations that are not readily apparent in aggregated representations, thereby offering a more nuanced understanding of each tool's energy behavior. Figure 2 presents boxplots that compare the per-program energy consumption profiles of Phi and Pynguin. The boxplots suggest that Phi generally exhibits higher energy consumption across the board, as indicated by the elevated medians and wider interquartile ranges in most programs. Moreover, Phi's distributions display greater variability, with broader boxes and more widely spread whiskers, which suggests a less predictable energy profile. This trend is reinforced by the presence of numerous and more pronounced outliers in the Phi group, pointing to occasional spikes in energy usage. In contrast, Pynguin shows more stable behavior, with tighter distributions and fewer extreme values. Taken together, these findings suggest that while Phi benefits from GPU acceleration, it does so at the expense of both higher and more volatile energy expenditure.

4.1 Bayesian Estimation of Energy Consumption

As outlined in Subsection 2.3.1, 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. Consequently, we adopted a more informative, estimation-oriented framework grounded in Bayesian probability rather than frequentist inference. Instead of merely assessing whether the two groups differ, our analysis quantifies how different they are (i.e., we focus on estimating the effect size), thus providing richer and more substantively meaningful insight.

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

Program	Phi				Pynguin			
	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
find_in_sorted	71.50	87.63	1,644.52	23	29.32	37.83	293.20	10
flatten	78.06	92.79	**2,029.48**	26	26.10	28.08	1,566.02	60
get_factors	74.33	91.99	1,412.20	19	27.30	29.86	273.00	10
hanoi	73.06	87.64	1,315.02	18	30.33	47.30	1,486.36	49
is_valid_parenthesization	59.35	86.78	771.52	13	28.25	47.22	988.86	35
kheapsort	74.88	90.65	1,497.51	20	27.15	43.23	**1,819.38**	67
kth	82.47	89.38	1,979.20	24	26.67	30.14	346.76	13
lcs_length	75.37	89.13	1,431.97	19	27.78	30.61	666.66	24
lis	75.61	93.03	**2,041.37**	27	26.77	29.49	160.64	6
max_sublist_sum	68.30	84.24	887.85	13	27.43	29.94	137.13	5
minimum_spanning_tree	75.82	88.32	1,668.02	22	23.25	27.02	837.09	36
possible_change	58.74	85.60	1,527.21	26	25.57	27.70	230.09	9
sieve	67.35	84.03	1,144.95	17	24.03	26.95	841.20	35
sqrt	60.13	84.98	1,022.21	17	24.15	28.34	**1,497.10**	62
subsequences	71.56	92.34	1,216.54	17	34.72	41.96	1,388.99	40
to_base	80.14	94.03	**2,243.81**	28	34.38	48.80	**2,165.93**	63
Mean	70.91	89.18	1,498.92	21	27.59	34.33	958.48	34
Median	73.06	89.13	1,497.51	20	27.15	29.94	841.20	35
Standard Deviation	7.63	3.39	419.43	4.95	3.19	8.06	660.08	22.89

‡ Duration is measured in seconds.

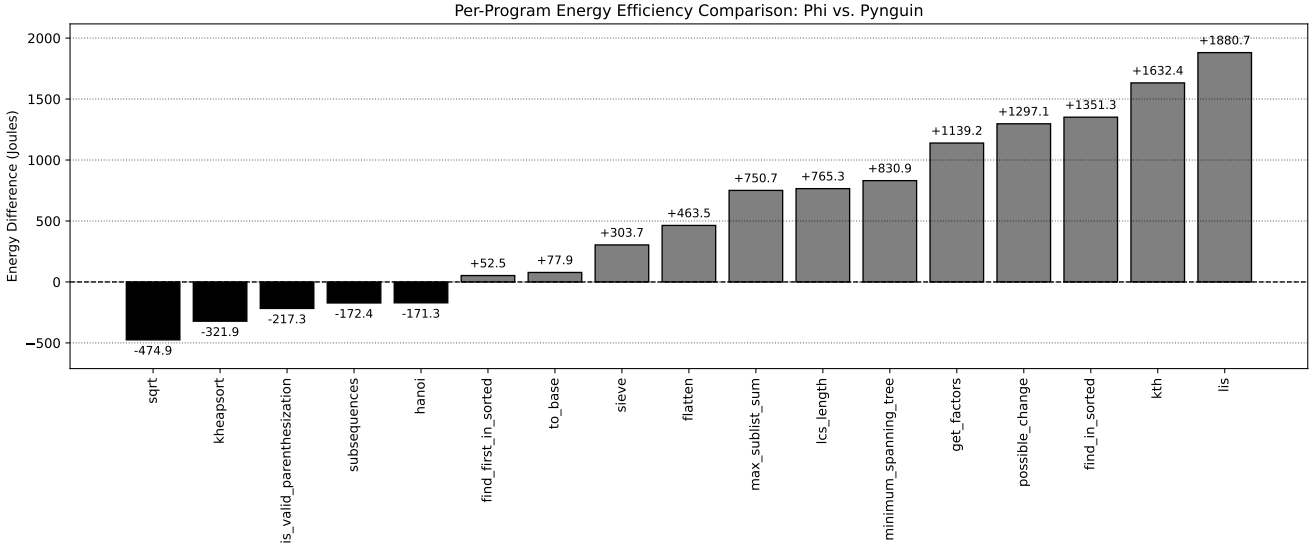

Figure 1: Per-program energy efficiency comparison between Phi and Pynguin.

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 (Figure 2). Compared to the normal distribution, the Student-t distribution offers increased robustness to outliers, thereby enhancing the reliability of our inferences. Specifically, our model builds upon the framework proposed by Kruschke [14] for comparing two groups. It incorporates five 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.

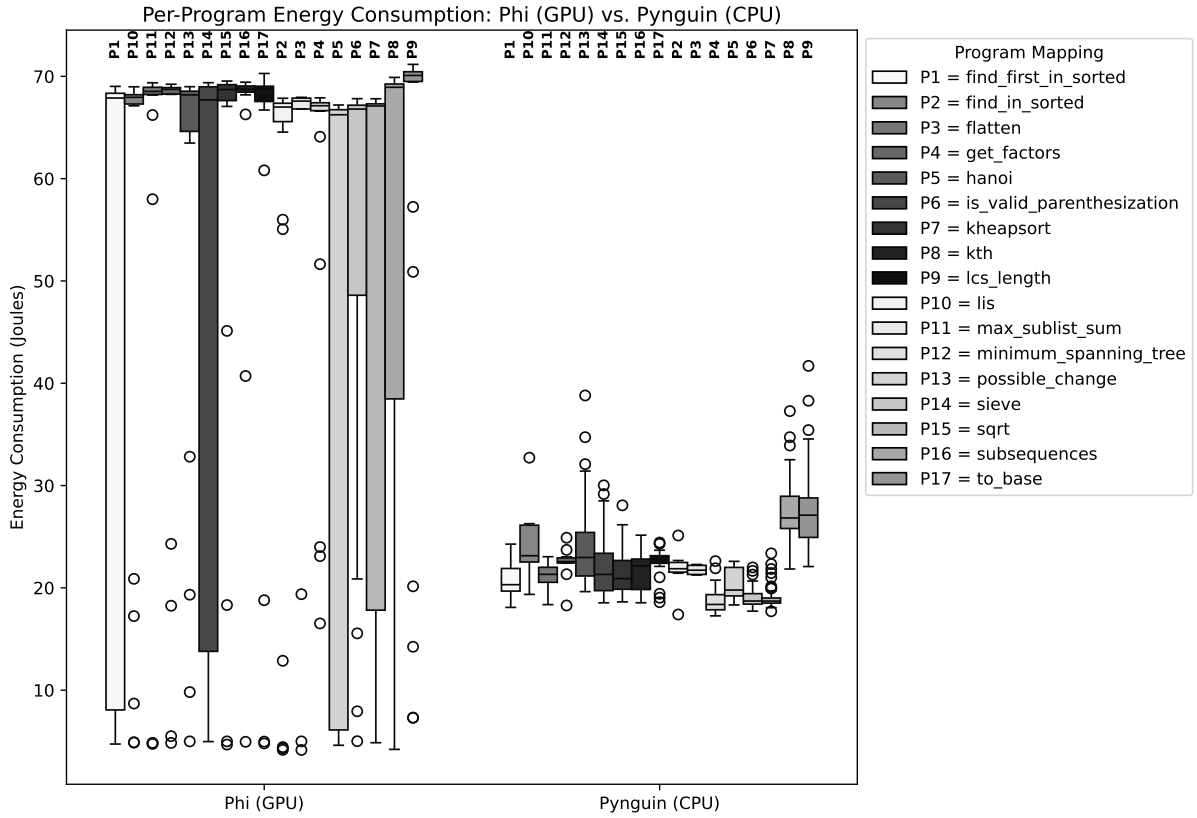

Figure 2: Boxplot comparison of per-program energy consumption between Phi (GPU-based) and Pynguin (CPU-based). Phi generally exhibits higher and more variable energy usage, with more pronounced outliers, indicating less predictable energy behavior across programs.

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

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

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 *normality* of the data: large ν yields a near-Gaussian distribution, whereas small ν allows for heavier tails. As a simplifying assumption, we let both unit test data generators share a single degrees-of-freedom parameter (ν), so that any heavy- or light-tailed behaviour is common across the two samples. We have separate parameters for the means μ_k ($k = 1,\ 2$) and standard deviations σ_k.

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

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

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

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) guarantees a positive scale but places minimal structure on σ_k. The exponential prior in (3) follows the recommendation of Kruschke [14], yielding high probability mass in the range where the t distribution mimics both Gaussian and heavy-tailed behavior.

After fully specifying our model, we looked at the posterior distributions of the model's stochastic parameters, shown in Figure 3. The posteriors summarize the uncertainty in our estimates for the means and standard deviations of energy expenditure for both unit test generation approaches. The resulting model was implemented and sampled using `PyMC`,[3] a probabilistic programming framework for Bayesian inference. As shown in Figure 3, the posterior mean energy consumption for Phi is centered around 68.3, while for Pynguin it is considerably lower, around 21. The corresponding 94% Highest Density Intervals (HDIs) [15], [68.2, 68.5] for Phi and [20.8, 21.2] for Pynguin, indicate that the distributions are sharply peaked and well-separated, with virtually no overlap. In the Bayesian framework, an HDI represents the range within which the parameter values fall with a specified probability (here, 94%) [15], such that every value inside the interval has higher credibility than any value outside it. Unlike frequentist confidence intervals, HDIs allow for direct probability statements about the parameters.

Phi shows a narrower posterior distribution for the standard deviation with a posterior mean of approximately 1.1 and a 94% HDI

[3] https://www.pymc.io/welcome.html

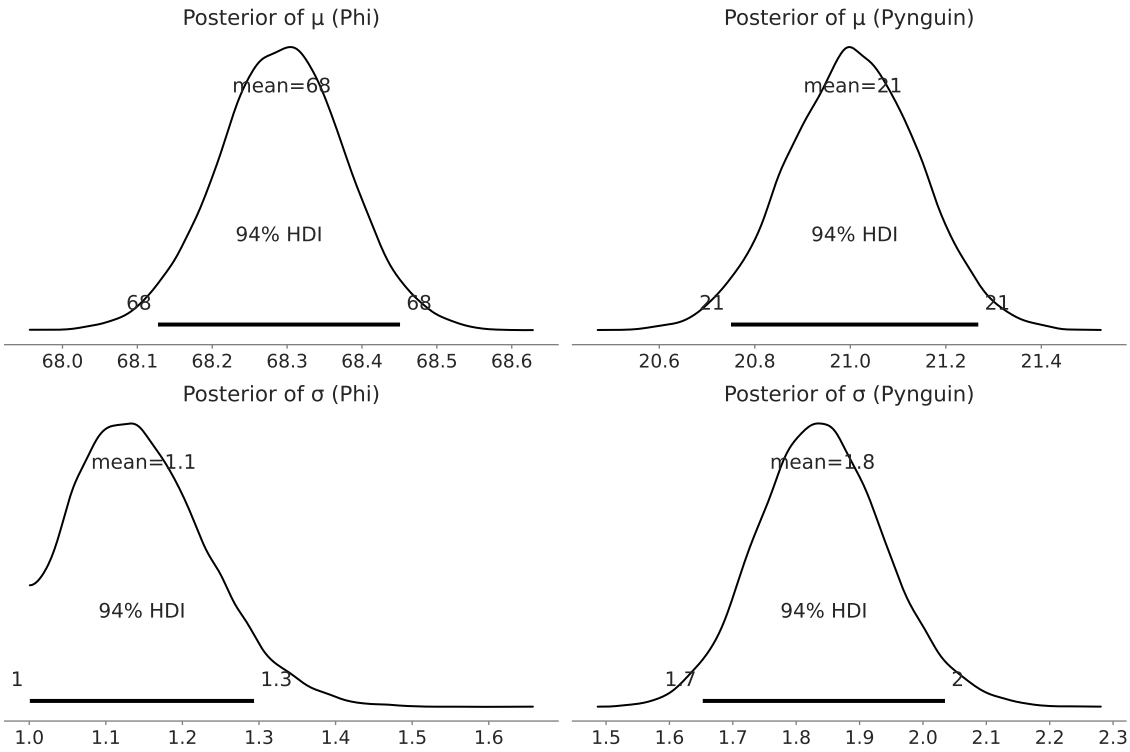

Figure 3: Summarized posterior distributions for the parameters of the energy consumption model.

of [1.0, 1.3], suggesting more consistent energy usage. In contrast, Pynguin's standard deviation is higher, with a posterior mean of 1.8 and HDI of [1.7, 2.0], indicating greater variability in its energy consumption profile. A summary of the posterior distributions is provided in Table 3.

Table 3: Summary of the posterior distributions.

Parameter	Mean	SD	HDI 3%	HDI 97%
μ_{Phi}	68.293	0.086	68.128	68.451
μ_{Pynguin}	21.005	0.138	20.750	21.268
σ_{Phi}	1.149	0.085	1.001	1.294
σ_{Pynguin}	1.842	0.101	1.653	2.035

Looking at the group differences in Figure 4, we observe strong evidence of distinctions between Phi and Pynguin in both the mean and standard deviation of energy consumption. For these comparisons, we used zero as a reference value, allowing us to assess how much of the posterior mass lies above or below zero. For the *difference in means* ($\mu_{\text{Phi}} - \mu_{\text{Pynguin}}$), the posterior distribution is tightly concentrated well above zero, with a mean of approximately 47 and a 94% HDI of [47, 48]. Notably, approximately 100% of the posterior mass lies above zero, which provides strong evidence that Phi tends to consume more energy per second than Pynguin. To quantify the standardized difference in energy consumption between Phi and Pynguin, we computed Hedges' g, a bias-corrected effect size suitable for samples of unequal size, which is defined as:

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

where d is Cohen's d, and n_1, n_2 are the sample sizes of the two groups. In this case, we derived a posterior distribution for d using the weighted pooled standard deviation and adjusted it to obtain Hedges' g.[4] As shown in Figure 4, the mean value of Hedges' g is approximately 3.14, indicating a very large effect size. This suggests that Phi's energy consumption is substantially higher than Pynguin's, even after adjusting for differences in sample size. A Hedges' g of 3.14 implies that the group means differ by more than three pooled standard deviations, which is a strong and practically meaningful difference.

The *difference in standard deviations* ($\sigma_{\text{Phi}} - \sigma_{\text{Pynguin}}$) also suggests a credible effect (Figure 4). The posterior has a mean of −0.69, with a 94% HDI of [−0.93, −0.45], and nearly all of the posterior mass lies below zero. This implies that Phi's energy consumption is not only higher, but also tends to be more consistent (i.e., less variable) than that of Pynguin. These results provide strong evidence of systematic differences between these unit test code generation tools: while Phi appears to be more energy-hungry, its energy usage tends to be somewhat relatively stable across runs, whereas Pynguin is more energy-efficient on average, though its energy consumption is more variable.

[4]We computed Hedges' g using the correction shown in (4) to account for unequal sample sizes and avoid bias in estimating the population effect size.

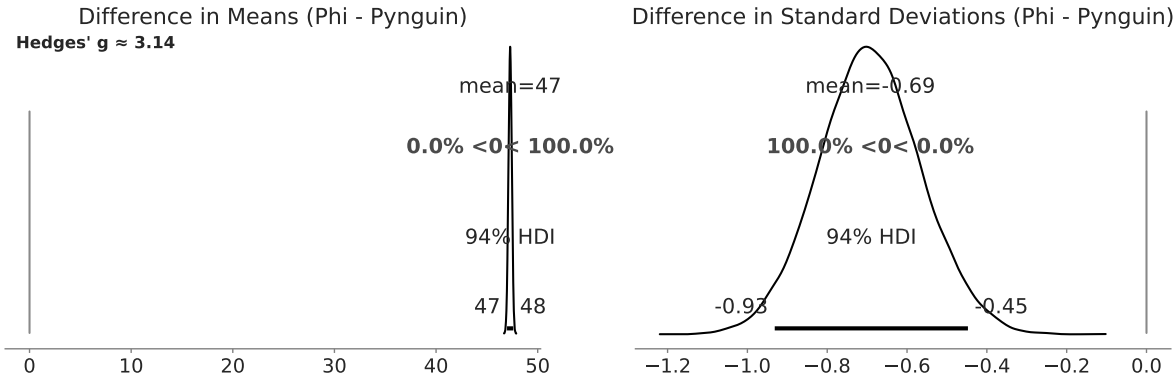

Figure 4: Posterior distributions of the differences in mean and standard deviation of energy consumption between Phi and Pynguin. The left panel shows the difference in means and Hedges' g, indicating that Phi consumes significantly more energy.

Key insights for RQ$_1$: our results demonstrate strong statistical evidence that Phi, despite being an SLM, consistently incurs significantly higher energy costs than Pynguin, a conventional unit test generation tool. Specifically, the posterior distributions obtained through our Bayesian analysis indicate a substantial effect size, underscoring the magnitude of Phi's increased energy expenditure. **These insights underscore critical considerations for the sustainable adoption of language-model-based approaches to test code generation, highlighting the need for careful evaluation of their energy footprint in practical Software Engineering scenarios.**

4.2 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 and Pynguin across all subject programs. These test suites were generated during the first phase of our experiment, which focused on measuring the energy consumption of Phi and Pynguin. 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 (% ✓). This percentage serves as a normalized measure of effectiveness, allowing for a fair comparison between the tools regardless of differences in test suite size.

Overall, Phi achieved a higher percentage of fault-revealing tests in 10 out of the 17 programs, whereas Pynguin outperformed Phi in 7. However, Pynguin achieved a higher average fault-detection rate across all programs: 66.7% compared to Phi's 58.8%. Pynguin also had multiple cases in which all tests uncovered the fault in the subject program: `find_first_in_sorted` and `sqrt`. Phi failed to reveal the fault in those two programs. On the other hand, Phi showed strong performance in the following subject programs: `to_base` (100%) and `kth` (90.0%), and its test suites tended to be larger in many cases, potentially increasing the chance of triggering subtle faults.

It is worth noting that in three cases (`kheapsort`, `lcs_lenght`, and `sqrt`) Phi did not produce any valid test cases, as indicated by a total of zero test cases in Table 4. Upon manual inspection, we found that Phi generated outputs that resembled pseudocode or natural language explanations rather than executable Python code. This issue occurred despite the use of the same prompt across all subject programs.

Figure 5 shows a grouped stacked bar plot that gives an overview of test cases generated by Phi and Pynguin for each subject program. Each par of bars corresponds to a program in our experiment, the height of each bar represents the total number of generated test cases, which are subdivided into two parts: non-fault-revealing test cases (in light gray) and fault-revealing test cases (in dark gray). This bar plot provides a comparative view not only of the fault-finding effectiveness of each tool but also of the relative size of the test suites they generated.

Key insights for RQ$_2$: Phi and Pynguin show variability in fault-finding effectiveness across the evaluated programs. On average, **Pynguin showed slightly superior fault-detection performance (66.7%) compared to Phi (58.8%).** However, **Phi generated more test cases on average, which could potentially increase the likelihood of detecting subtle faults in more complex scenarios. Moreover, Phi achieved particularly high fault-detection rates in specific cases** (e.g., `to_base` and `kth`), **indicating scenarios where its approach can yield highly effective test suites**.

5 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 [24] is relatively low

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

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

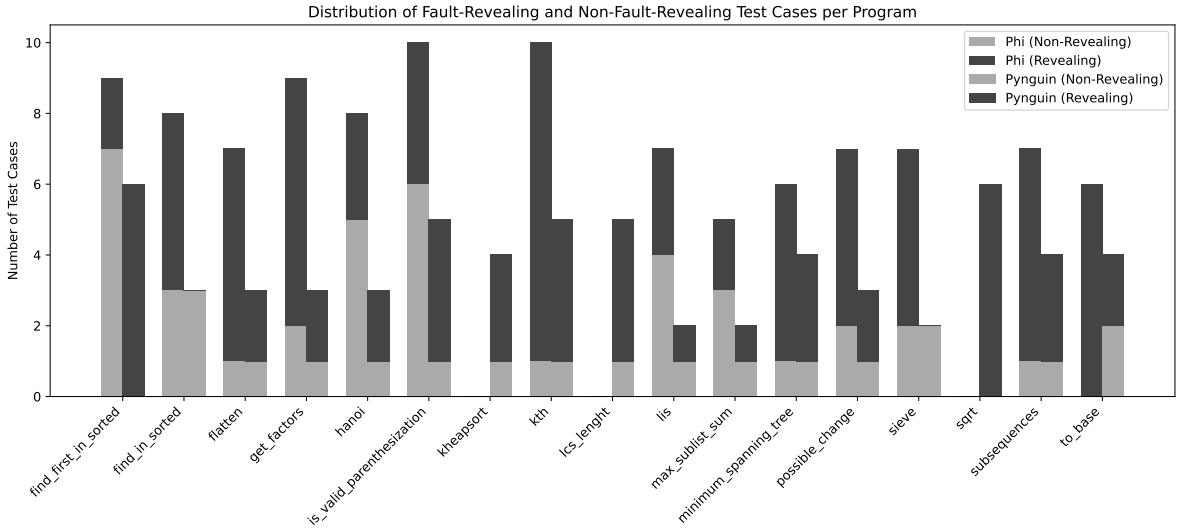

Figure 5: Fault-revealing and non-fault-revealing test cases generated by Phi and Pynguin for each subject program.

and is typically considered acceptable for most profiling and energy usage analysis experiments.

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. A similar limitation applies to our use of a single SLM. Other language models may differ in their energy consumption and fault-detection effectiveness, and as such, relying on a single SLM also constrains the extent to which our results can be generalized.

One threat to validity arises from the variability in the outputs produced by the SLM we used in our experiment. Although the same prompt was applied uniformly across all subject programs, the language model failed to generate valid test code for three cases, instead producing something that resembles pseudocode. As no changes were made to the prompt or experimental procedure, it is difficult to attribute these issues to any identifiable variation in the setup. This underscores a broader limitation of prompt-based code generation: output quality can vary unpredictably, even under controlled and repeatable conditions.

6 Related Work

Recent research by Kifetew et al. [13] 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.

7 Concluding Remarks

Motivated by sustainability concerns, we set out to investigate the energy footprint of employing an SLM (i.e., Phi) for automated unit test generation. To the best of our knowledge, this is the first study to directly compare the energy consumption and fault-detection effectiveness of an SLM against a conventional unit test generation tool. By comparing Phi and Pynguin, a purpose-built unit test generation tool, our research underscores the potential energy-related drawbacks of employing language models for specialized Software Engineering tasks. The primary contribution of our paper revolves around the insights derived from addressing RQ_1, which investigates the difference in energy consumption between Phi and Pynguin when generating unit test suites. Leveraging BDA, our results demonstrate strong statistical evidence that Phi, despite being an SLM, consistently incurs significantly higher energy costs than Pynguin. This result raises important questions about the sustainability and environmental footprint associated even with smaller-scale language models, challenging the assumption that SLMs offer a universally energy-efficient alternative to their larger counterparts. Consequently, our findings highlight the necessity for Software Engineering researchers and practitioners to carefully consider the energy expenditure and sustainability implications when adopting language-model-based automation approaches, particularly in comparison to specialized, domain-specific tools.

REFERENCES

[1] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. 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)*. ACM, NY, USA, 221–232.

[2] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder K. Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.

[3] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[4] Joel Castaño, Silverio Martínez-Fernández, Xavier Franch, and Justus Bogner. 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)*. ACM, NY, USA, 1–12.

[5] Noel Cressie and Christopher K. Wikle. 2011. *Statistics for Spatio-Temporal Data*. John Wiley & Sons, Hoboken, NJ. 624 pages.

[6] Yi Ding and Tianyao Shi. 2024. Sustainable LLM Serving: Environmental Implications, Challenges, and Opportunities : Invited Paper. In *IEEE 15th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 37–38.

[7] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[8] Carlo A. Furia, Robert Feldt, and Richard Torkar. 2021. Bayesian Data Analysis in Empirical Software Engineering Research. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1786–1810.

[9] Carlo A. Furia, Richard Torkar, and Robert Feldt. 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 (2022).

[10] Gerd Gigerenzer. 2004. Mindless Statistics. *The Journal of Socio-Economics* 33, 5 (2004), 587–606.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. *ACM SIGPLAN Notices* 40, 6 (2005), 213–223.

[12] Mladan Jovanović and Mark Campbell. 2024. Compacting AI: In Search of the Small Language Model. *Computer* 57, 8 (2024), 96–100.

[13] Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2025. On the Energy Consumption of Test Generation. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*. 360–370.

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

[15] John K. Kruschke. 2015. *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan* (2nd revised ed.). Academic Press. 776 pages.

[16] Kiran Lakhotia, Phil McMinn, and Mark Harman. 2010. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software* 83, 12 (2010), 2379–2391.

[17] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 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)*. ACM, NY, USA, 55–56.

[18] Alexandra S. Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. 2023. Estimating the carbon footprint of BLOOM, a 176B parameter language model. *The Journal of Machine Learning Research* 24, 1 (2023).

[19] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 168–172.

[20] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (2023), 36.

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

[22] Tim Menzies and Martin Shepperd. 2019. "Bad smells" in software analytics papers. *Information and Software Technology* 112 (2019), 35–47.

[23] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley. 256 pages.

[24] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments*. Biarritz, France.

[25] OpenAI. 2025. ChatGPT. https://chat.openai.com. Large language model.

[26] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105.

[27] Rens van de Schoot, Sarah Depaoli, Ruth King, Bianca Kramer, Kaspar Märtens, Mahlet G. Tadesse, Marina Vannucci, Andrew Gelman, Duco Veen, Joukje Willemsen, and Christopher Yau. 2021. Bayesian statistics and modelling. *Nature Reviews Methods Primers* 1 (2021).

[28] Scott W. VanderStoep and Deidre D. Johnson. 2008. *Research Methods for Everyday Life: Blending Qualitative and Quantitative Approaches*. Jossey-Bass. 352 pages.

[29] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer. 236 pages.

[31] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021).