FAST: Energy-Aware Test Prioritization for Fail-Early Software Testing

Dibyendu Gupta
TU Delft
D.T.Gupta@student.tudelft.nl

Noky Soekarman
TU Delft
N.P.Soekarman@student.tudelft.nl

Güngör Cem
TU Delft
C.Gungor@student.tudelft.nl

Leonidas Hadjiyiannis
TU Delft
L.Hadjiyiannis-1@student.tudelft.nl

Uddhav Pisharody
TU Delft
U.Pisharody@student.tudelft.nl

Abstract

Energy consumption is becoming an increasingly important concern in software engineering, as modern development workflows involve frequent test execution that can accumulate substantial computational cost. Reducing unnecessary energy use in software testing is especially valuable in regression scenarios, where developers repeatedly run existing test suites to obtain fast feedback after code changes. This paper presents FAST (***FA**il early in **S**oftware **T**esting*), a VSCode extension for JavaScript projects that measures the energy consumption of individual test cases and prioritizes them accordingly. The extension profiles existing tests, ranks them in ascending order of measured energy use, and executes them stopping at the first failure. The goal is to reduce wasted computation while also shortening the time developers wait for meaningful test feedback. Unlike full regression validation pipelines, which prioritize executing all tests, our approach targets developer-facing scenarios, where quickly detecting failures and reducing test execution cost are more important. We evaluate the approach using energy consumed before first failure, time to first failure, and overall feedback latency. Our results show that FAST does not consistently reduce total energy consumption or execution time across all runs, but can significantly reduce time to first failure when failing tests are present. Github Repository containing our VS Code extension: (https://github.com/SSE26/VSCode-extension-for-Test-Case-Analysis).

CCS Concepts

• **Software and its engineering → Software testing and debugging**; *Software libraries and repositories*; *Software performance.*

Keywords

Test Case Prioritization, Software Energy Consumption

ACM Reference Format:
Dibyendu Gupta, Noky Soekarman, Güngör Cem, Leonidas Hadjiyiannis, and Uddhav Pisharody. 2026. FAST: Energy-Aware Test Prioritization for Fail-Early Software Testing. In *CS4575-Q3-26: Sustainable Software Engineering, TU Delft*. ACM, New York, NY, USA, 9 pages.

CS4575-Q3-26, TU Delft

1 Introduction

The growing scale of modern software systems has led to a significant increase in computational resource usage, and consequently, energy consumption. CI/CD pipelines are on the rise and are becoming increasingly popular, requiring developers to frequently execute large test suites to validate code changes. While this process improves software reliability, it also introduces substantial and often overlooked energy costs. As software systems scale to industrial levels, where projects may include millions of test cases, these repeated executions can contribute to considerable energy waste.

Recent studies highlight the environmental impact of such automated workflows. For example, large-scale analyses of CI/CD ecosystems show that millions of workflow executions collectively result in significant carbon and water footprints, with estimates reaching hundreds of metric tons of CO_2 emissions annually [1]. These findings suggest that routine development practices, such as automated testing, can have non-trivial environmental consequences.

Regression testing in particular is a major contributor to this problem. In typical development workflows, test suites are repeatedly executed after incremental changes, even when only a small subset of tests is actually relevant to the modified code. This redundancy is further exacerbated by inefficiencies in test design. Recent empirical studies show that poorly structured tests, such as those containing test smells, can introduce additional unnecessary computation and energy overhead, with smelly test code consuming more energy than clean counterparts [2].

From an industrial perspective, inefficient testing practices also translate to substantial resource waste. Reports from industry indicate that running full regression suites for every code change can result in thousands of redundant test executions per week, significantly increasing both cloud costs and energy consumption. In contrast, adopting selective test execution strategies such as test impact analysis has been shown to reduce compute energy consumption by more than 50%, demonstrating the potential for more efficient testing workflows [3].

At the same time, developers are primarily interested in early feedback, specifically, identifying whether a failure has occurred, rather than completing the entire test suite. Running all tests to completion in such scenarios results in unnecessary computation, increased feedback latency, and higher energy consumption.

Test case prioritisation has traditionally been studied with the goal of detecting faults earlier in the execution order. However,

existing approaches largely focus on metrics such as code coverage, historical failure rates, or execution time [4], with limited consideration of energy consumption as the primary optimisation objective. Furthermore, recent empirical studies show that improvements in runtime do not necessarily translate to lower energy consumption, highlighting the need to explicitly consider energy as a separate optimization dimension [5]. Despite these advances, energy-aware testing remains insufficiently integrated into practical developer workflows, particularly in tools that support rapid, iterative development.

To address this gap, we propose FAST (*Fail Early in Software Testing*), a developer-centric approach for energy-aware test execution. FAST profiles individual test cases to estimate their energy consumption and prioritizes them accordingly, executing tests in ascending order of energy cost while stopping at the first failure. This design targets interactive development scenarios, where reducing both computation and feedback latency is critical.

Our approach is particularly relevant in large-scale industrial settings, where even small per-test energy savings can accumulate into substantial reductions when applied across extensive test suites. Unlike traditional regression testing pipelines designed for completeness, FAST focuses on efficient, developer-facing execution.

The main contributions of this paper are:

- A practical formulation of energy-aware test case prioritization for developer-facing workflows.
- The design and implementation of FAST, a VSCode extension that profiles and prioritizes tests based on estimated energy consumption.
- An empirical evaluation analyzing the impact of energy-aware prioritization on energy consumption and feedback latency across different testing scenarios.

The Github Repository containing our VSC Code extension [1] is public and mentions the guidelines for installing and running the extension.

2 Research Question

Under what conditions does energy-aware test case prioritization reduce the energy consumption of regression testing compared to standard execution strategies?

3 Related Work

This section surveys related work on regression testing and test case prioritization, energy consumption in software testing, and energy estimation techniques. These areas provide the basis for our work, which combines energy-aware reasoning with test prioritization in a fail-fast regression testing setting.

3.1 Regression Testing and Test Case Prioritization

Regression testing aims to ensure that software changes do not adversely affect previously validated functionality, but executing large test suites repeatedly can be expensive in both time and computational resources [6]. To address this problem, the test case prioritization (TCP) literature studies how to reorder test suites so that more valuable tests are executed earlier according to a given objective function [7]. Rothermel et al. formalized the TCP problem as the search for an ordering of a test suite that maximizes a chosen objective, and subsequent work has most often instantiated this objective in terms of earlier fault detection [7]. Elbaum et al. further showed that prioritization techniques can improve the rate of fault detection and thereby provide earlier feedback during regression testing, which is particularly useful when testing resources are limited or the execution process is interrupted [6].

Over time, TCP research has explored a wide variety of prioritization criteria, including structural coverage, execution history, requirement importance, model-based representations, and test cost [4]. Cost-aware regression testing has also been explored as an extension of classical TCP. For example, Indumathi and Madhumathi propose a regression testing approach that first prioritizes test cases using a genetic algorithm and then reduces the prioritized suite using an MFTS (Maximum Frequent Test Set) algorithm, explicitly aiming to lower regression testing cost while retaining coverage and fault-detection effectiveness [8]. Their prioritization procedure uses historical information including test cost, detected faults, and fault severity, and they evaluate the reordered suite using APFD (Average Percentage of Fault Detection). This line of work is important because it shows that TCP objectives have already been extended beyond pure coverage or fault detection toward execution-cost considerations. However, these approaches still treat cost mainly within a traditional regression-testing framework and often combine prioritization with suite reduction. Overall, this suggests that although cost-sensitive TCP has been recognized in the literature, it remains a relatively less developed direction than traditional coverage-prioritization [4].

More recent work has extended TCP toward continuous integration and data-driven settings, including techniques based on machine learning, reinforcement learning, failure history, and textual representations of tests. These approaches show that modern TCP can combine heterogeneous signals beyond structural coverage alone. However, they still predominantly optimize conventional objectives such as earlier fault detection, failure prediction, or related effectiveness measures [9]. In contrast, our work targets a different practical objective: reducing the energy spent during regression testing before the first observed failure, while keeping the full test suite available and avoiding suite reduction [10].

3.2 Energy Consumption in Software Testing

Software testing has increasingly been recognized as a relevant target for energy-aware software engineering, because repeated execution of large test suites can accumulate into a non-trivial energy footprint [10]. Prior work further argues that developers still lack practical tooling for making energy-aware decisions during development [11].

For example, Zaidman's work on the environmental impact of testing open-source Java projects shows that testing can impose a substantial annual energy cost, with the energy required to test a single project reaching levels comparable to powering an electric car for hundreds of kilometers [12]. Similarly, exploratory evidence on open-source projects shows that automated builds and tests can

[1] https://github.com/SSE26/VSCode-extension-for-Test-Case-Analysis

consume meaningful amounts of electricity at project scale, and that this impact varies considerably across systems and execution platforms.

Energy-aware software testing has recently been proposed as a research direction that explicitly aims to reduce the energy consumed during testing, either by adapting existing testing techniques or by introducing new energy-oriented ones. This literature highlights that energy efficiency introduces a distinct optimization objective alongside more traditional concerns such as reliability, coverage, and fault detection effectiveness. As highlighted in this emerging literature, reducing test energy consumption may require balancing environmental goals against other desirable software engineering properties, such as reliability, defect detection capability, and economic feasibility [10].

Related work has also explored energy reduction through test suite minimization. In particular, energy-directed test suite optimization treats energy as a first-class minimization criterion and shows that minimized suites can in some cases consume substantially less energy than suites produced by traditional minimization approaches. However, this line of work focuses on suite reduction rather than execution ordering of the full suite [13].

3.3 Energy Measurement and Estimation in Software Systems

A central challenge in energy-aware testing is to estimate software energy consumption in a way that is accurate enough to be meaningful while still practical for everyday software engineering use [14, 15] . Prior work has repeatedly noted that developers lack accessible, fine-grained tools for reasoning about software energy behavior, and that measurement-heavy approaches are often difficult to adopt in routine development settings [14, 16].

Many software-based energy estimation approaches build on the idea that processor power can be approximated from utilization-level signals, often using simplified power models rather than external metering hardware [17, 18].

Several tools follow this practical direction. GreenOracle proposes a resource-count-based prediction model motivated by the fact that direct energy measurement is expensive and often impractical for developers, while PETrA focuses on Android applications and shows that software-based estimation can provide useful application-level energy profiles without dedicated hardware instrumentation [16, 19]. Beyond mobile settings, EnergiBridge points toward cross-platform software-level energy measurement for general software systems, supporting Linux, Windows, and macOS across multiple CPU architectures [20].

The appeal of these approaches lies in their deployability: they can often be integrated directly into software tooling and execution environments without the setup cost of external measurement infrastructure. At the same time, prior work also highlights their limitations, including reduced precision relative to hardware-based measurements, dependence on platform assumptions, and limited fine-grained attribution in many existing tools [14]. These trade-offs become particularly important when energy information is needed at the granularity of individual executions or tests, where the estimate must be lightweight enough to collect repeatedly while still stable enough to support optimization decisions [13, 14].

What appears underexplored is a practical approach that estimates the energy cost of individual test executions and uses that information directly for test prioritization in a fail-fast local development workflow [10, 11]. Our work targets this intersection by using lightweight software-level estimates to support energy-aware ordering at the granularity of individual test executions.

4 Problem Definition & Scope

The development of FAST (Fail Early in Software Testing) addresses the non-trivial energy costs associated with modern CI/CD pipelines, where large test suites are frequently executed to validate incremental code changes. In typical interactive developer scenarios, completing an entire test suite after a minor change often results in unnecessary computation and increased feedback latency, particularly when a developer only needs to know if a failure has occurred. FAST is designed specifically for this "fail-fast" local development workflow, aiming to reduce the energy expended before the first observed failure.

The scope of the current implementation is focused on JavaScript projects using the Node.js built-in test runner. It estimates energy consumption for individual test cases using a lightweight CPU-based proxy rather than specialized hardware to ensure cross-platform accessibility for developers without requiring administrative permissions. We focus on JavaScript because its standard runners often lack the integrated, input-aware test caching found in compiled language ecosystems like Java's Gradle or Rust's Cargo which naturally leverage build-system change detection to skip unchanged tests.

5 Extension Design

The VS Code extension we designed consists of three parts, selecting the files containing the desired tests, profiling the tests to get an estimate of the energy consumption, and running the tests efficiently using the calculated estimate.

5.1 Selecting tests

To save energy during the running of tests, one first needs to select the Javascript file(s) containing the tests. This can be done in two ways: Selecting individual files that are in a single folder, or selecting all the files within a folder by simply selecting the folder itself. When selecting an entire folder, the extension will recursively look through that folder for Javascript files. The extension will parse the test files and extract the test cases from the selected files. This can be used to select your whole test suite by selecting the root folder of the project.

5.2 Energy Estimation Model

Direct hardware energy measurement typically requires privileged access to processor-level interfaces such as Intel's Running Average Power Limit (RAPL)[2] counters or dedicated tools like EnergiBridge[3]. While such interfaces provide high-accuracy readings, they impose significant barriers to adoption: RAPL access requires kernel-level

[2] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

[3] https://github.com/tdurieux/EnergiBridge

permissions on Linux, is unavailable on non-Intel architectures, and is entirely inaccessible on Windows without a driver installation. For a developer-facing tool intended to run across heterogeneous machines without administrative setup for maximum accessibility, a purely software-based proxy is therefore necessary.

Theoretical grounding. The proxy we adopt is grounded in the principle of *energy-proportional computing*, first formalised by Barroso and Hölzle [18]. They observed that an ideal computing system should consume power in direct proportion to its utilization level: zero power at idle, and peak power under full load. Crucially, they identified the CPU as the hardware component that most closely approximates this ideal in practice, owing to decades of innovations in dynamic voltage and frequency scaling (DVFS) and clock gating. This proportionality provides the theoretical justification for using CPU utilization as a proxy signal for power draw.

Empirical basis. The proportionality assumption was empirically validated at scale by Fan, Weber, and Barroso [17], who demonstrated that server power consumption increases *linearly* with CPU utilization. Their measurements yielded the linear power model:

$$P(U) = P_{\text{idle}} + \left(P_{\text{peak}} - P_{\text{idle}}\right) \times U \tag{1}$$

where P_{idle} is the power drawn when the CPU is idle, P_{peak} is the power drawn at full utilisation, and $U \in [0, 1]$ is the instantaneous CPU utilisation fraction. This model has since been adopted as the standard reference for lightweight power estimation across the systems and green-software communities, accumulating over 1,500 citations.

Adopted formula. In the absence of at-wall power measurements, we substitute the manufacturer-specified Thermal Design Power (TDP) for P_{peak}. TDP is defined as the maximum sustained power the chip's cooling solution must be able to dissipate [21], and has been shown to serve as a reliable upper bound on average power draw under realistic workloads [22, 23]. The energy consumed by a process over an interval Δt is then estimated as:

$$E = \left(P_{\text{idle}} + U \times \text{TDP}\right) \times \Delta t \tag{2}$$

The four components of Equation 2 are as follows.

- U — the **CPU utilisation fraction** of the process under test, sampled over the test's execution window. Per-process CPU time is exposed cross-platform by the operating system, making this the only runtime measurement required.
- TDP — the **Thermal Design Power** of the host CPU in watts, as published by the manufacturer. It represents the maximum sustained power the CPU is designed to dissipate, and serves as the upper bound on power draw in the estimation. Read from the system's hardware information without any special permissions and stored as a static table.
- P_{idle} — the **idle baseline power**, representing the static electricity draw of the CPU at zero utilization. A zero-CPU system still dissipates power through leakage currents and peripheral components; omitting this term would systematically underestimate energy for short-running tests [24]. Following common practice in the green-software literature, we approximate P_{idle} as a fixed fraction of TDP (typically 10–15%) [22, 25].
- Δt — the **wall-clock duration** of the test execution, in seconds, converting instantaneous power into energy (joules).

Fitness for purpose. The model does not claim to produce absolute energy measurement counter, instead it provides a relative energy ordering of test cases. It provides *consistent rank ordering*: if two tests differ in CPU demand, the heavier one will reliably receive a higher estimated energy score, enabling the prioritization scheme. This relative adequacy of TDP-based estimation for ordering and optimization purposes has been confirmed across multiple independent evaluations of green-software tools [23, 26], and is further supported by empirical correlations between CPU runtime and energy consumption reported in the mobile computing literature [27].

5.3 Profiling Tests and Efficient Test Runs

To estimate the energy consumption of each individual test case, the extension must measure per-process CPU activity over the duration of that test's execution. This section describes how that profiling is carried out and how the resulting measurements are used to produce an energy estimate.

Per-process CPU sampling. Each test case is executed as a child process spawned by the extension host. Isolating tests into separate processes serves two purposes.

(1) It provides a clean measurement boundary: CPU time accumulated during one test cannot bleed into the sample of another.
(2) It mirrors the execution model of `node --test`, where each test file is treated as an independent unit.

Once a test process is running, the extension periodically checks how much CPU it is using via the `pidusage` library. This works on all major operating systems without any special permissions. These snapshots are averaged across the test's lifetime to get a single representative CPU utilization figure U, which is then plugged into Equation 2 alongside the test's total runtime Δt to produce the energy estimate for that test.

Handling measurement overhead. Polling itself consumes CPU cycles, which would inflate the energy estimate if not accounted for. We mitigate this in two ways. First, the sampling interval is set to 100 ms by default, a value chosen to be fine-grained enough to capture transient spikes in short-running tests while keeping the observer overhead negligible relative to the test process itself [28]. Second, the `pidusage` call is scoped strictly to the child process PID, excluding the extension host process from the sample, so any overhead incurred by the polling loop does not enter the estimate.

Energy-proportional ordering. Once all tests in a suite have been profiled, either from a fresh run or from a cache of prior measurements, their estimated energies are ranked in ascending order. The core hypothesis, consistent with the empirical findings of Fan et al. [17], is that a test demanding more CPU time will draw more power and therefore consume more energy for equivalent wall-clock durations. By scheduling lighter tests first, the suite reaches its first failing assertion sooner and with less total energy expended.

When a failure is detected, the extension aborts the remaining queue entirely, since continuing a suite with a known failure yields no actionable information. Because the most energy-intensive tests are deferred to the tail of the queue, they are precisely the ones most likely to be skipped meaning the largest energy savings occur exactly when a regression is present and the run is cut short.

5.4 Caching Test scores for repeated runs

To ensure that energy profiling remains efficient, the extension avoids the redundant overhead of measuring every test during every run. Instead, it utilizes a persistent per-workspace cache to store the energy consumption and duration of each test case. To keep these estimates accurate yet stable, the system uses a weighted rolling average. This approach smooths out potential outliers from a single run while ensuring the execution order for future runs is based on reliable, long-term data.

When a test is re-measured, the new data is integrated into the existing cache using specific weighting rules: unchanged tests use a 40/60 balance between old and new data, while modified tests shift to a 30/70 split to favour the most recent changes. By amortizing the profiling cost over multiple sessions, the extension provides a data-driven execution order that feels seamless within a standard development cycle.

5.5 Running the Tests

With an energy ordering established, the extension schedules and executes test files through Node.js's built-in test runner. This section describes the execution pipeline, from test discovery through to result reporting inside the editor.

Test discovery. The extension scans the open workspace (using VSCode workspace file-system API) for files that match the patterns recognized by `node –test`: files named `*.test.js`, `*.spec.js`, or residing in a `test/` or `tests/` directory [29]. Discovered files are grouped by directory to allow the user to run individual folders or the full suite from the editor's testing sidebar.

Ordered execution. Before invoking the runner, the extension sorts the discovered test files according to their cached energy estimates. Files with no prior energy record (for example, newly created tests) are appended to the end of the queue and assigned a profile run so that their energy is known for future invocations. The sorted list is then passed to `node –test` as an explicit file argument sequence:

```
node --test test-low-energy.js test-medium-energy.js
test-high-energy.js
```

Streaming output and fail-fast. As tests run, their results are streamed live into the VSCode testing UI rather than displayed all at once at the end. This means a developer can see passing and failing tests appear in real time without waiting for the full suite to finish. If a failing test is detected and fail-fast mode is enabled, the extension immediately stops the run. Since tests are ordered from least to most energy-intensive, the cheaper tests have already completed by the time a failure is likely to appear — so the results shown up to that point are still useful, and the remaining expensive tests are the ones that get skipped.

Result reporting. It displays a summary in the VSCode status bar showing the estimated total energy consumed by the run (in joules) alongside the number of tests passed and failed, giving developers immediate visibility into the energy cost of their test suite. This feedback loop is intended to make energy consumption a first-class metric in the development workflow, complementing existing metrics such as test duration and code coverage.

6 Experiment Setup

To empirically validate the energy savings of the test ordering extension, a series of automated experiments were conducted using a realistic simulation of a developer's workflow. The experiments utilized the Node.js test suite[4] as a representative large-scale dataset, specifically targeting the parallel test directory. The VSCode version used for the experiment can be found on the feat/dataset-extraction branch[5] along with the raw results of the experiments (under scripts/experimental-batch-results-1/2/3 folders).

Experiment Design and Sampling. The experiment runner operates on a total of 20 runs. For each run, the system randomly samples 50 files from the dataset. To perform the experiment in an automatic way and avoiding human error, we created a script that runs the sampling of tests automatically. Each test file contains multiple test cases of varying size. The test suite also has naturally occurring failing test cases that makes the experimental setup as realistic as possible. This simulates the potential energy "waste" avoided by running cheaper, more likely-to-fail tests first. Since the extension estimates the energy consumption based on the state of the CPU, the amount of other programs running on the testing machine will be minimal. This means that only the automation script and VS Code will be running during the experiment.

System Warm-up and Interference Control. Each run consists of two distinct phases: a warm-up phase and a measured phase. The experiment calculates energy consumption by polling CPU usage at 100ms intervals, matching the extension's internal estimator logic. To ensure the integrity of these readings, the following controls were implemented:

- **Warm-up Iteration:** Each sampled set undergoes one initial warm-up iteration which is discarded from the final results. This is a critical control measure to ensure system stability. Initial runs often encounter "cold start" overheads, such as JIT (Just-In-Time) compilation and disk I/O latency. Discarding this run ensures that the measured data reflects the "steady-state" performance of the system.
- **Measured Iterations:** Following the warm-up, three measured iterations are performed on the same sampled set. During these iterations, the extension profiles all test cases to establish an energy baseline and then executes them using the efficient ordering strategy.
- **Cooldown Periods:** A 500 ms cooldown period is enforced between individual test cases to allow the CPU to return to an idle state and prevent thermal "bleed" between tests.

[4] https://github.com/nodejs/node
[5] https://github.com/SSE26/VSCode-extension-for-Test-Case-Analysis/tree/feat/dataset-extraction

Justification for Omitting the Rolling Average. While the VS Code extension itself utilizes a weighted rolling average (e.g., a 40/60 split) to stabilize test ordering across multiple user sessions, this feature was intentionally disabled for the experimental design. The primary goal of the experiment is to establish a consistent and independent baseline for each test file within a controlled environment. By avoiding the rolling average, each measurement iteration remains isolated from the data of previous runs. This ensures that the recorded energy consumption reflects the immediate physical cost of the code under test rather than any affect of historical data. This independence is essential for calculating accurate energy-saving percentages and ensuring that the observed efficiency gains are a direct result of the ordering strategy rather than artifacts of a persistent cache.

Environment and Installation. To replicate a production environment, the experiment was conducted with the extension compiled and installed as a VSIX package in the regular VS Code environment, rather than running within a development/debugging instance. This was facilitated by a custom installation script (npm run vsix:install), ensuring that the overhead of the extension itself is consistent with what an end-user would experience.

7 Results

The following section discusses results gathered from the experimental setup mentioned above.

7.1 Estimated Energy Savings

We evaluated the proposed ordering strategy on two batches of randomly sampled test subsets, where each run contained 50 random JavaScript test files drawn from a set of around 4000 tests. For each sampled subset, we compared a baseline execution order with an energy-aware order produced by profiling the tests and then sorting them according to estimated energy use.

Table 1 summarizes the estimated energy results. At the batch level, the first batch showed a slight increase in estimated energy for the efficient ordering, while the second batch showed a slight improvement on average. However, the overall picture: across all 20 runs, the mean estimated energy saving was 0.23 J, whereas the median saving remained negative at -0.13 J, and only 7 out of 20 runs favored the efficient ordering. This indicates that the proposed ordering does not provide a stable overall reduction in estimated energy consumption across all sampled subsets.

At the same time, the results also show where the tool is most useful. The largest improvement occurred in the only run that contained a failing test, which is consistent with the fail-first design of the approach: when a failure is encountered, executing lower-cost tests earlier can reduce the amount of work performed before termination. Similarly, the tool can still be useful when a developer operates under an energy budget, since profiling allows cheaper tests to be scheduled first and therefore enables more tests to be executed within the same budget. In contrast, when all tests pass and the full sampled suite is executed, reordering alone offers limited room for improvement.

Table 1: Estimated energy results. Lower values are better. Positive Δ indicates lower estimated energy for the efficient ordering.

Set	#Tests per Batch	Baseline μ (J)	Efficient μ (J)	Δ (J)	Med. Δ (J)	#Tests: Eff. < Bas.
Batch 1	10	**82.55**	82.77	-0.22	-0.32	2/10
Batch 2	10	51.86	**51.17**	0.69	-0.01	5/10
Overall	20	67.20	**66.97**	0.23	-0.13	7/20

7.2 Feedback Speed

We also evaluated feedback speed by comparing the total execution duration of the baseline and efficient orders. Table 2 shows that the overall effect was mixed: across all 20 runs, the mean duration saving was 0.06 s, while the median saving was -0.06 s, and only 5 out of 20 runs completed faster under the efficient ordering. This indicates that the proposed ordering did not provide a consistent overall speedup across sampled runs.

However, the clearest improvement appeared in the only run that contained a failing test. In that case, execution time decreased from 17.07 s in the baseline order to 15.02 s in the efficient order, corresponding to a saving of 2.05 s. This result is again consistent with the fail-first motivation of the tool: when a failure is present, prioritizing lower-cost tests can surface that failure earlier and reduce feedback time.

Table 2: Feedback-speed results. Lower values are better. Positive Δ indicates faster execution for the efficient ordering.

Set	#Tests per Batch	Baseline μ (J)	Efficient μ (J)	Δ (J)	Med. Δ (J)	#Tests: Eff. < Bas.
Batch 1	10	**23.56**	23.64	-0.08	-0.09	1/10
Batch 2	10	14.81	**14.62**	0.20	-0.01	4/10
Overall	20	19.19	**19.13**	0.06	-0.06	5/20

7.3 Example Output

Figure 1 presents an example output generated by the extension during normal use. In practice, the tool does not necessarily require a dedicated profiling run; instead, it collects per-test energy estimates during regular test execution and stores them in a cache. These cached values are then used to order subsequent test runs.

To keep the estimates up to date, the cache is updated using a weighted average. When a test case changes, the current observation is given a higher weight (0.7) and combined with the previous cached value (0.3). When the test case has not changed, the current observation is still incorporated, but with a slightly lower weight (0.6), while the remaining weight (0.4) is assigned to the cached estimate. This allows the ordering to adapt over time without being overly sensitive to short-term fluctuations.

The example output therefore focuses on the reordered execution sequence and the estimated energy values used to produce it, rather than displaying absolute or percentage differences between baseline and efficient runs. This makes the tool's behavior transparent to developers while reflecting its intended real-world use: learning from repeated executions and gradually improving the ordering of frequently run test suites.

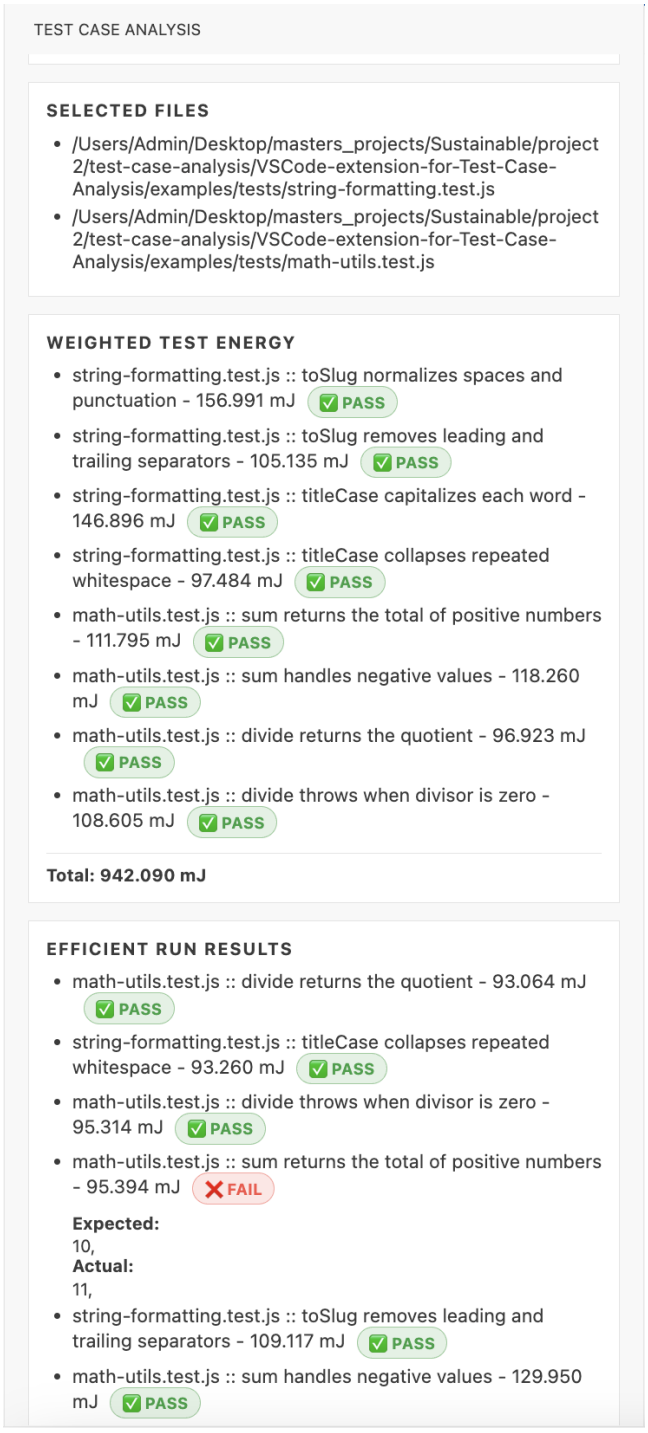

Figure 1: User interface of the FAST extension during test execution.

8 Discussion

The results show that the energy-aware ordering strategy has limited overall benefit on the sampled test subsets. In most runs, the efficient order did not clearly outperform the baseline, and the overall savings in both estimated energy and execution time were small. This means that reordering alone does not consistently improve test execution in the general case.

A simple reason is that, when all tests pass, the same tests are still executed. Only the order changes. Because the total amount of work stays almost the same, there is only limited room to reduce either runtime or estimated energy consumption. Small differences between runs may also be affected by normal measurement noise, such as warm-up effects or other variation in the execution environment. This is an empirical contribution towards the academic community displaying the order of test case execution based purely on energy consumption as a metric doesn't yield greater efficiency in test case execution.

At the same time, the results show a situation in which the tool is more useful. The clearest improvement appeared in the run that contained a failing test. In that case, the efficient order surfaced the failure earlier and reduced the work done before termination. This matches the intended fail-first design of the extension. In practice, this may be the most important use case, since earlier failure detection can help developers receive feedback faster and avoid unnecessary computation.

The approach may also still be useful when developers work under a limited time or energy budget. Even if the total cost of a full passing run does not change much, the profiling information makes it possible to run cheaper tests first. This can help execute more tests within the same budget and may therefore still be valuable in realistic development settings.

Overall, the findings show that the benefit of FAST is context-dependent. The extension does not provide a stable improvement for every sampled run, but it appears more promising in situations where failures are present or where only part of the test suite can be executed.

9 Limitations

While the project demonstrates that test-case analysis and energy-aware execution can be integrated into a sustainable VS Code extension, several limitations remain. These limitations mainly concern the range of supported technologies, the approach to test discovery and execution, and the accuracy of the energy estimation model. Recognizing these constraints is important for correctly interpreting the results and identifying opportunities for future improvement.

First, the current implementation only supports JavaScript test files. As a result, projects written in other programming languages fall outside the scope of the tool. This restricts its applicability in mixed-language or non-JavaScript codebases and limits its usefulness in more diverse development environments.

Second, although the test command can be configured, the extension is primarily designed around Node's built-in test runner. Testing frameworks that use different execution models or produce different output formats may require additional adaptation before they can be effectively integrated.

Third, test discovery is limited because it relies on regular expression matching. At the moment, the implementation only detects direct test(...) and it(...) calls. More advanced or indirect patterns—such as dynamically generated tests, wrapper functions, aliases, nested abstractions, or framework-specific syntax, may not be recognized correctly. As a result, some test cases may be missed during analysis.

Finally, the energy estimation model only considers CPU-related energy usage. It does not account for other hardware resources such as memory usage, disk activity, GPU utilization, or background system processes. Therefore, the reported energy values should be interpreted as approximations rather than precise measurements of total system energy consumption.

10 Conclusion and Future Work

This paper presented FAST, a VS Code extension for energy-aware test execution in local development workflows. The extension estimates per-test energy usage using a lightweight CPU-based proxy, caches these estimates across runs, and reorders tests from lower to higher estimated cost while supporting fail-fast execution. In this way, FAST brings energy-aware test case prioritization into a practical developer-facing setting rather than treating it only as an offline regression-testing problem.

Our empirical results show that the overall effect of the proposed ordering is mixed when considering all sampled runs together. Across the full experiment, the method did not produce a stable overall reduction in estimated energy consumption or execution duration. At the same time, the largest improvement appeared in the only run containing a failing test, which is consistent with the intended fail-first design of the approach. This suggests that the main value of FAST lies not in guaranteeing uniform savings on every run, but in supporting scenarios where failures are likely or where developers need to operate under a limited energy budget.

Taken together, these findings suggest that energy-aware test ordering is a promising direction for sustainable software tooling, but that its benefits are highly context-dependent. Rather than claiming that the approach improves every test run, our study shows that this kind of strategy can be integrated into everyday developer tooling and can be useful in the right scenarios. We hope FAST can provide a basis for future work on practical energy-aware testing support in software engineering.

These findings also point to several directions for future work. First, future work should examine the profiling overhead of the approach more systematically and determine after how many repeated runs its cost is offset by later savings. Although this overhead is not expected to dominate in repeated developer workflows, understanding its practical impact would make the approach easier to assess in real-world use.

Second, the current energy model can be extended and validated further. At present, FAST measures a CPU-based proxy of execution cost rather than full system energy usage. While this is sufficient for lightweight developer-facing ordering decisions, it does not capture memory, disk, network, GPU, or background-process activity. Future work could therefore compare the current proxy against hardware-based energy measurements such as RAPL or external metering tools, and explore process-tree accounting to better capture all work triggered by a test run.

Finally, future work could explore broader applicability and more adaptive prioritization strategies. This includes extending FAST to additional testing ecosystems, improving how tests are identified and tracked over time, and combining cached energy estimates with other useful signals such as failure history or code changes. Such extensions would help make the tool more flexible and more effective across a wider range of development settings.

References

[1] Nuno Saavedra, Alexandra Mendes, and João F Ferreira. Environmental impact of ci/cd pipelines. *arXiv preprint arXiv:2510.26413*, 2025.

[2] Md Rakib Hossain Misu, Jiawei Li, Adithya Bhattiprolu, Yang Liu, Eduardo Santana de Almeida, and Iftekhar Ahmed. Test smell: A parasitic energy consumer in software testing. *Information and Software Technology*, 181:107671, 2025.

[3] Craig Risi. Green it in testing and quality engineering: Driving sustainability through smart quality practices. https://www.craigrisi.com/post/green-it-in-testing-and-quality-engineering-driving-sustainability-through-smart-quality-practices, 2023. Accessed: 2026-04-02.

[4] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, September 2013. ISSN 0963-9314. doi: 10.1007/s11219-012-9181-z. URL https://doi.org/10.1007/s11219-012-9181-z.

[5] José Miguel Aragón-Jurado, Abdul Ali Bangash, Bernabé Dorronsoro, Karim Ali, Abram Hindle, and Patricia Ruiz. Does faster mean greener? runtime and energy trade-offs in ios applications with compiler optimizations. *Sustainable Computing: Informatics and Systems*, page 101166, 2025.

[6] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, February 2002. ISSN 0098-5589. doi: 10.1109/32.988497. URL https://doi.org/10.1109/32.988497.

[7] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000. ISSN 0163-5948. doi: 10.1145/347636.348910. URL https://doi.org/10.1145/347636.348910.

[8] C. P. Indumathi and S. Madhumathi. Cost aware test suite reduction algorithm for regression testing. pages 869–874, 2017. doi: 10.1109/ICOEI.2017.8300829.

[9] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Softw. Engg.*, 27(2), March 2022. ISSN 1382-3256. doi: 10.1007/s10664-021-10066-6. URL https://doi.org/10.1007/s10664-021-10066-6.

[10] Roberto Verdecchia, Emilio Cruciani, Antonia Bertolino, and Breno Miranda. Energy-aware software testing. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 101–105, 2025. doi: 10.1109/ICSE-NIER66352.2025.00026.

[11] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Spelling out energy leaks: Aiding developers locate energy inefficient code. *J. Syst. Softw.*, 161(C), March 2020. ISSN 0164-1212. doi: 10.1016/j.jss.2019.110463. URL https://doi.org/10.1016/j.jss.2019.110463.

[12] Andy Zaidman. An inconvenient truth in software engineering? the environmental impact of testing open source java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, AST '24, page 214–218, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705885. doi: 10.1145/3644032.3644461. URL https://doi.org/10.1145/3644032.3644461.

[13] Ding Li, Cagri Sahin, James Clause, and William G. J. Halfond. Energy-directed test suite optimization. In *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, GREENS '13, page 62–69. IEEE Press, 2013. ISBN 9781467362672.

[14] Felix Rieger and Christoph Bockisch. Survey of approaches for assessing software energy consumption. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*, CoCoS 2017, page 19–24, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355216. doi: 10.1145/3141842.3141846. URL https://doi.org/10.1145/3141842.3141846.

[15] Sung Une Lee, Niroshinie Fernando, Kevin Lee, and Jean-Guy Schneider. A survey of energy concerns for software engineering. *J. Syst. Softw.*, 210(C), April 2024. ISSN 0164-1212. doi: 10.1016/j.jss.2023.111944. URL https://doi.org/10.1016/j.jss.2023.111944.

[16] Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 49–60, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901763. URL https://doi.org/10.1145/2901739.2901763.

[17] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International*

Symposium on Computer Architecture, ISCA '07, pages 13–23. ACM, 2007. doi: 10.1145/1250662.1250665.

[18] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007. doi: 10.1109/MC.2007.443.

[19] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: a software-based tool for estimating the energy profile of android applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 3–6. IEEE Press, 2017. ISBN 9781538615898. doi: 10.1109/ICSE-C.2017.18. URL https://doi.org/10.1109/ICSE-C.2017.18.

[20] June Sallou, Luís Cruz, and Thomas Durieux. Energibridge: Empowering software sustainability through cross-platform energy measurement, 2023. URL https://arxiv.org/abs/2312.13897.

[21] Intel Corporation. Measuring processor power: TDP vs. ACP. Intel White Paper, 2011. URL https://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf.

[22] Loïc Lannelongue, Jason Grealey, and Michael Inouye. Green algorithms: Quantifying the carbon footprint of computation. *Advanced Science*, 8(12):2100707, 2021. doi: 10.1002/advs.202100707.

[23] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019. URL https://arxiv.org/abs/1910.09700.

[24] Green Software Foundation. SCI guidance: Performance engineering based energy estimation. https://sci-guide.greensoftware.foundation/E/PerformanceEngineeringBased/, 2023. Accessed: 2025.

[25] Kadan Lottick, Silvia Susai, Sorelle A. Friedler, and Jonathan P. Wilson. Energy usage reports: Environmental awareness as part of algorithmic accountability. In *NeurIPS Workshop on Tackling Climate Change with Machine Learning*, 2019. URL https://arxiv.org/abs/1911.08354.

[26] Loïc Lannelongue and Michael Inouye. How to estimate carbon footprint when training deep learning models? A guide and review. *arXiv preprint arXiv:2306.08323*, 2023. URL https://arxiv.org/abs/2306.08323.

[27] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 92–101. IEEE, 2013. doi: 10.1109/ICSE.2013.6606555.

[28] Caleb Tripp et al. A beginner's guide to power and energy measurement for machine learning. Technical Report NREL/TP-6A20-91518, National Renewable Energy Laboratory, 2025. URL https://docs.nrel.gov/docs/fy25osti/91518.pdf.

[29] Node.js Project. Test runner — Node.js documentation. https://nodejs.org/api/test.html, 2024. Accessed: 2025.