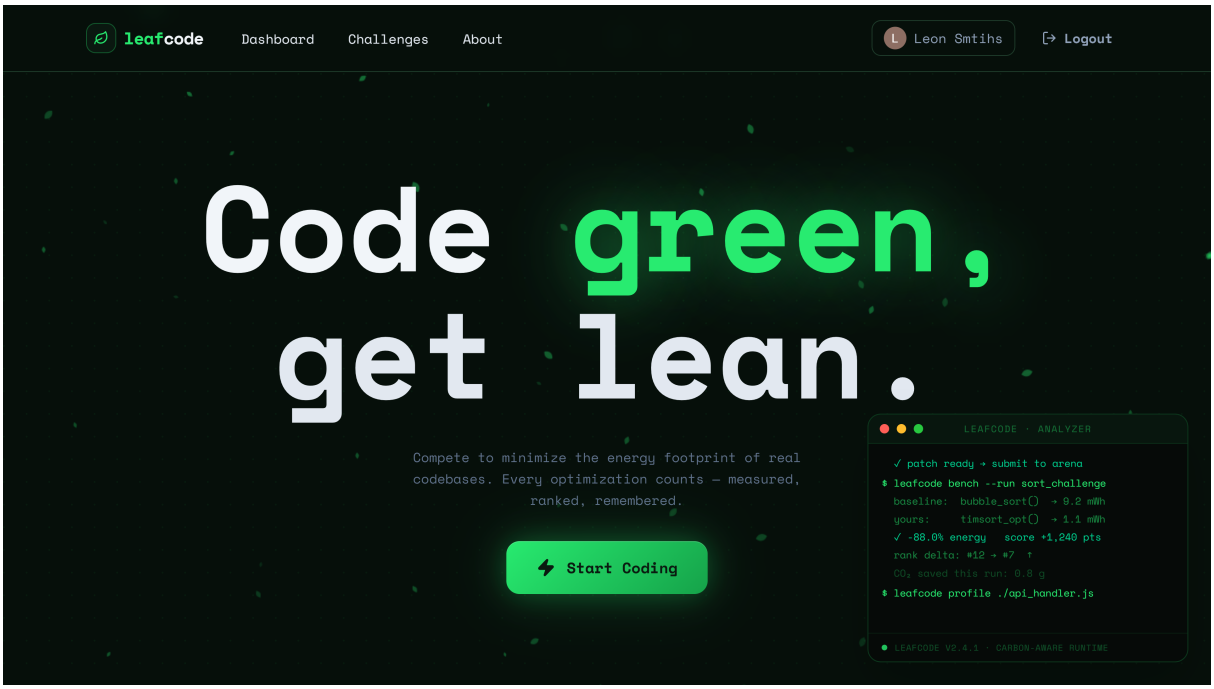
LeafCode

by

Ayush Khadka, Konstantinos Syrros, Medon Abraham,

Norah E. Milanesi & Job Stouthart

Delft University of Technology



Instructor: Luis Miranda da Cruz
Course: Sustainable Software Engineering
Date: Thursday 2nd April, 2026

TU Delft

Abstract

As the environmental impact of computing scales, traditional algorithmic platforms remain focused on time-memory optimization, neglecting energy efficiency. Various developers do not have the necessary skills to implement energy efficient code.We present *LeafCode*, a gamified learning platform designed to foster Green Software Engineering (GSE) practices and this is deployed using Vercel. *LeafCode* estimates energy using the `cloud-energy` framework. By ranking developer submissions based on actual Joule consumption, the platform shifts the optimization target from execution speed to carbon-awareness. Our evaluation indicates that this competitive, measurement-driven approach successfully bridges the gap between algorithmic proficiency and sustainable implementation, driving meaningful behavioral change in developers' architectural decision-making. *LeafCode* is accessible at `https://leafcode.konsyrros.me`.

Keywords: Sustainability, Green-Code, Coding Assistant, Challenges, RAPL

1. Introduction

1.1. Context

Within the software systems industry, the demand has increased massively. Cloud computing, distributed systems, and AI models consume massive amounts of resources and energy. The energy use of data centers has doubled in the past decade and will continue to increase in the upcoming decade [1]. Research shows that the energy efficiency of software is rarely considered in development, despite the fact that the behavior of software influences the energy consumption of the hardware and the overall sustainability of a system. Recent studies also show that energy related concerns are becoming more prominent within software engineering [2].

Only a few developers consider designing and implementing applications with the consideration of their energy usage. The reason most developers do not consider energy optimization is due to not understanding the decisions they make, which affect energy consumption, as well as lack of a tool to support them to improve or help modify their code. Software engineers can reduce energy consumption by considering design patterns, and certain implementations of an algorithm can be more energy-efficient than others [3]. As developers neglect energy efficiency, it later reflects on end users, as the software they use can increase energy consumption, leading to increased energy usage on their energy bills [4]. Based on this, it follows clearly that educating software developers about sustainable energy is the key to mitigating extreme energy consumption. A form of education such as a coding platform can not only help them improve their skills, but also design and implement efficient code.

1.2. Research Objectives and Solution

As mentioned above, the energy efficiency of software is rarely considered in the development lifecycle. Most coding platforms such as LeetCode and Hackerrank expect optimization in speed and correctness, but not in energy efficiency, leading to a lack of knowledge on the matter in most developers. Therefore, the objective is to develop a coding platform that promotes energy efficiency as the primary reward metric to increase Green Software Engineering literacy among developers, ensuring that they adapt their coding skills by writing energy efficient code. It aims to influence how developers reason with energy optimization, as well as encourage companies to recognize energy-efficient coding as a mandatory skill.

The developed solution is called LeafCode [5], and is a coding challenge platform inspired by LeetCode which encourages developers to write energy-efficient software. The main goal is to encourage software developers to think about sustainability while designing algorithms and implementing them. It also aims to raise awareness of the environmental impacts of code and algorithms. The platform provides a set of programming challenges that users must solve by submitting their own implementations. The problem complexity varies, which allows developers to experiment with different design and implementation techniques at various levels.

2. Background

2.1. Green Software Engineering (GSE)

According to Hindle [4], Green Software Engineering (GSE) is considered an "umbrella" term that covers the study of how much power code uses (software energy consumption), the extraction of data/patterns to improve sustainability (Green Mining), the use of eco friendly computing (Green IT), and the minimization of environmental impact using computing (Sustainable Computing). Traditionally, development focuses on speed, cost and scalability but the GSE adds a new factor: sustainability. GSE has various principles which are energy efficiency, carbon aware-

ness, hardware longevity, sustainable architecture, life cycle management, measurement and transparency, and cultural transformation [6]. While Hindle focus on the methodological challenges such as hardware aware measurement, multiple version analysis and more, the GSE principles derived from the industry discussion can be explained using Hindles' work. Energy Efficiency refers to the reduction of unnecessary computation so that software decisions impact energy usage. Carbon Awareness refers to software adapting to carbon intensity. Hardware Longevity refers to maximizing the lifetime of hardware such that software needs to be sustainable and that developers have the responsibility towards hardware constraints. The principle of sustainable architecture refers to eco friendly scaling, modularity, and idle resource costs. The life cycle management refers to managing the software in the long run through using multi-version analysis which analyzes the energy consumption and determines regression of performance. The Measurement and Transparency principle refers to hardware based measurements such as energy. The principle of cultural transformation refers to how training engineers and educating them on sustainability.

2.2. Energy Measurement

A fundamental challenge in Green Software Engineering is the accurate measurement of energy consumption. The methods to measure energy are through hardware based measurement and estimation model measurements.

2.2.1. Hardware Measurement Method

Modern CPUs often expose on-chip energy counters that can be read by software for energy measurement [7]. The Running Average Power Limit (RAPL) is a hardware-based energy measurement tool which measures the electricity consumption of the CPU and other components due to the sensors integrated on the system chip. It exposes it to the OS through a model specific registers (MSR). It is integrated to modern Intel processors. The advantages of RAPL are that, unlike external power meters or power distribution units (PDUs), RAPL does not require additional hardware measurements. RAPL provides more details compared to PDUs as it monitors the internal components of the node. RAPL is more accurate compared to any estimation model. In order to extract the measurements, either a low level RAPL MSR needs to be used, or a higher level interface that is provided by the OS is required [8].

2.2.2. Energy Estimation Methods

In certain situations, direct hardware measurements can be unavailable, with researchers having developed energy measurement estimation tools using model-based approaches for such cases. Some of these estimation methods are using linear regression and simulations. For energy estimation, performance counters (PMCs) can be used with linear or piecewise regression to predict power based on CPU and memory activity. Simulation-based approaches use tools such as Wattch and SimplePower in order to estimate the CPU power or module energy, based on capacitance and input transition. The simulated data can be used as the input, and based on analytical power models, the power can be estimated [9].

3. Prototype

3.1. Chosen Tools

3.1.1. Vercel

In order to achieve fast, yet functional, deployment of LeafCode, we chose Vercel. Vercel is a cloud platform for building and deploying web applications and supports many features we deemed required for LeafCode. Besides hosting full-stack Next.js applications, which is our framework of choice, it provides additional services, such as Sandboxes for running untrusted code, perfect for our user submission assessments, blob storage for storing submitted code, as well as a PostgreSQL database in partnership with Neon, another cloud service provider. Vercel automatically builds and deploys the provided code and offers it around the world in a serverless manner for exceptional performance and availability [10].

3.1.2. Cloud Energy

While Vercel is great for flexible and performant deployments, it causes issues with hardware-based energy measurements due to its virtualized environment, which lacks direct access to the hardware sensor, about which more is explained in 3.4. On the basis of this, hardware-based measurement like RAPL cannot be used. To solve this, an estimation method can be used to estimate the energy. The tool is called "cloud energy", developed by Arne Tarara and Didi Hoffmann. Cloud Energy is used to estimate the server power usage, measured in watts, based only on a small number of inputs, such as CPU load, by using machine learning (ML) and statistical methods. It has two main model types, which are linear regression and the XGBoost model [11]. Linear regression is a simple statistical model that takes inputs such hardware specifications and outputs a rough power estimate. It provides a faster and easier estimation, but is less accurate. The XGBoost Model is an advanced ML model which is a gradient boosting model. This model is trained on a dataset called SPECPower. It can capture non-linear patterns between CPU load and power draw. This model has better accuracy in most cases. Cloud Energy allows developers and teams to understand the

amount of power their code draws, estimate the energy precisely, and helps optimize infrastructure as well as code for lower energy usage. The limitations and assumptions of this model are inherent in its machine-learning-based nature, and stem from the fact that it is trained on the SPECPower benchmark, which may not necessarily reflect real world hardware performance and energy usage in all cases. Cloud Energy works best for CPU-heavy workloads, and may increase in inaccuracy in lighter loads.

3.2. The User Interface

Dashboard, Leaderboard, and Coding Environment The website consists of a landing page that depicts "Code green, get lean", the platform's main slogan. It provides a short description of what LeafCode is about. In the Dashboard section, there is a message that welcomes the user and a button to take them to the challenges. Below, the user can find tables with statistics such as their ranking, score, and the challenges they have completed. On the right side there is a platform-wide leaderboard, showing all the users and their scores. It can be filtered by the programming language. When the user clicks on "Start Coding", it takes them to the Challenges page, which contains different challenges that the user can choose from. It can be filtered according to difficulty level, between Easy, Medium, and Hard.

After the user selects a challenge, they are brought to a page with the coding environment. It follows a structure similar to that of LeetCode in which the left side contains the description of the challenge and the concepts of the challenge, with the right side being comprised of a code editor. Users can submit their code in any programming language available for each challenge, submit it to the platform, and within seconds receive their energy measurements and scores.

The Stack LeafCode is built as a full-stack web application using a modern JavaScript/TypeScript ecosystem. The frontend and backend are unified under `Next.js 16` with the App Router, which enables a clean separation between server-rendered pages using React Server Components, and client-side interactive components with classic React. All code is written in `TypeScript` to ensure type safety across the entire codebase.

The user interface is built with `React` and styled using `Tailwind CSS`, with pre-built accessible components provided by `shadcn/ui`. The in-browser code editor is powered by `CodeMirror 6`, which provides syntax highlighting for all supported languages.

Authentication is handled by `Better Auth`, a session-based authentication framework that manages user registration, login, and session persistence, as well as social providers. Data is stored in a `PostgreSQL` database, accessed through `Prisma` as the ORM layer, which provides type-safe database queries generated from a declarative schema.

The execution backend (The Runner) is implemented in `Python 3.13` and leverages the `Cloud Energy` XGBoost model for energy estimation, with `psutil` monitoring CPU utilization in real-time. Code execution is handled through language-specific runtimes: `Python 3` and `Node.js` for interpreted languages, and `gcc` and `g++` for compiled C/C++ programs. The runner environment is sandboxed using a custom Vercel Sandbox configuration, for secure and isolated runs and grading. The entire platform is deployed on `Vercel` serverless infrastructure, which provides automatic scaling and world-wide CDN availability.

The full stack is summarized in Table 1.

Table 1: Technology stack used in LeafCode.

Layer	Technology
Framework	Next.js 16 (App Router)
Language	TypeScript
UI Components	React, Tailwind CSS, shadcn/ui
Code Editor	CodeMirror 6
Authentication	Better Auth
Database	PostgreSQL
ORM	Prisma
Execution Runtime	Python 3.13
Energy Estimation	Cloud Energy (XGBoost)
CPU Monitoring	`psutil`
Code Execution	Python 3, Node.js, gcc, g++

A visual representation of the stack can be seen in figure 1:

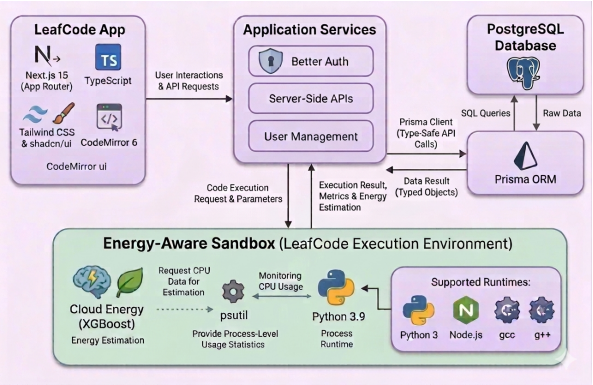


Figure 1: Architecture of LeafCode

3.3. Handling User Submissions

Once a user has completed the implementation of their solution to one of the challenges, they simply click the submit button in the coding environment. This triggers the code assessment workflow in the

backend of the application. The workflow consists of multiple steps, the first of which being the creation of a submission row in the database to keep track of submission metadata, as well as storing their code in blob storage.

Further, a sandbox environment is provisioned, along with our assessment framework, The Runner, to securely test the user's code both for correctness of the solution through various test cases (where available) and energy efficiency through probabilistic energy estimation. This process may happen immediately or slightly delayed, depending on the load on the system. Once completed, the results are handed back to our backend environment, to compute the overall score of the submission. Submissions that did not pass the correctness tests do not receive any points. Submissions that passed the correctness tests receive points according to their energy efficiency, according to the scoring algorithm, which takes into account the energy use as well as the difficulty of the challenge.

The score achieved is finally stored in the database with the submission attempt, and the user is notified of their result and added to the leaderboard. Further attempts are allowed, however previous attempts cannot be deleted or overwritten.

3.4. Energy Estimation: The Runner

Design Requirements and Implementation

The Runner is LeafCode's execution backend, responsible for running user-submitted code and measuring energy consumption. The system consists of two coordinated components: a TypeScript API route that handles HTTP requests and validates submissions, and a Python executor that performs the actual code execution, validation, and energy measurement, in a sandboxed environment.

Initial Approach: Hardware Level Measurement Our initial design attempted direct hardware level energy measurement using Intel's RAPL interface via the EnergiBridge wrapper. However, this approach proved incompatible with Vercel's serverless infrastructure. Vercel functions run inside Docker containers on AWS Lambda, which operate in virtualized environments lacking direct access to hardware sensors. Local testing confirmed that EnergiBridge functioned on bare metal systems but failed completely within containers, returning null readings, or produced inaccurate results.

Adopted Solution: ML Based Estimation We adopted Cloud Energy [11], which estimates server power consumption using an XGBoost machine learning model trained on the SPECPower benchmark dataset. The model accepts hardware specifications (CPU frequency, core count, TDP, memory) and real time CPU utilization as input, outputting estimated power in joules. Energy is computed as:

$$E = \sum_{i=1}^{n} P(t_i) \cdot \Delta t$$

where $P(t_i)$ is the estimated power at sampling interval i, and $\Delta t = 0.5$ seconds. Our implementation monitors CPU utilization using Python's `psutil` library at 500ms intervals throughout code execution.

This approach enables deployment on serverless infrastructure while maintaining the ability to demonstrate relative efficiency differences between algorithms, which is the core objective of LeafCode.

Multi Language Support

The Runner supports four programming languages: Python, JavaScript, C, and C++. For interpreted languages (Python and JavaScript), code executes directly via their respective runtimes. For compiled languages (C and C++), the system first invokes `gcc` or `g++` with the `-O0` flag to disable optimization. This ensures energy measurements reflect the developer's algorithmic choices rather than compiler transformations, maintaining educational clarity. Compilation errors are caught and reported with diagnostic messages before execution attempts. Functionality tests are currently only supported in Python, and LeafCode currently has Python and C++ fully active.

Performance and Variance

The system leverages Vercel's serverless architecture where the XGBoost model loads on the first request (cold start, taking 3 to 4 seconds) and persists in memory for subsequent requests within the same sandbox lifecycle (approximately 2 minutes).

Energy estimation exhibits inherent variance due to system level CPU fluctuations from background processes. Our testing revealed typical variance of plus or minus 5 to 10 percent for identical code executions. For example, three consecutive executions of `print(42)` produced 8.84J, 8.85J, and 8.38J, representing a 5.2 percent difference.

We considered implementing three run averaging to reduce variance but decided against it due to the threefold increase in execution time (from 1 to 2 seconds up to 3 to 6 seconds), which would degrade user experience. This variance is acceptable for several reasons. Industry platforms like LeetCode exhibit similar execution time variance (10 to 20 percent) due to shared infrastructure. Algorithmic differences, which are the focus of optimization, far exceed measurement noise

and typically range from 50 to 200 percent. LeafCode's purpose is demonstrating relative efficiency, where variance has minimal impact on the educational value.

Technical Trade offs
The decision to use estimation over hardware measurement represents a pragmatic trade off prioritizing deployability and educational value over laboratory grade precision. This choice enabled universal deployment on any serverless platform without infrastructure costs, rapid iteration with updates deploying in seconds, and accessible development without requiring specialized hardware.

For LeafCode, we chose to prioritize building a functional platform that demonstrates relative efficiency over achieving perfect measurement accuracy with deployment constraints. For an educational platform focused on fostering sustainable coding awareness, demonstrating that recursive solutions consume more energy than iterative ones matters more than measuring the precise milliwatt difference.

4. Challenges

The LeafCode platform features a curated library of coding challenges, each designed to translate Green Software Engineering principles into practical, solvable scenarios. Each challenge presents a specific "trap" representing a common, energy-inefficient coding practice that developers must identify and overcome.

- **The Patience Trap:** This challenge focuses on proactive resource management. The trap is using a continuous CPU polling loop (busy waiting) to execute a time delay, which constantly drains system power. The sustainable solution requires yielding system resources by implementing a hibernate or sleep command.
- **The Turbo Trap:** Designed to teach concurrency scaling. The trap involves blindly applying multithreading to all payload sizes. Developers learn that booting multiple cores for small tasks incurs an unnecessary "energy tax," making sequential processing more efficient in those cases.
- **The Telemetry Router:** This problem highlights the importance of efficient search structures. The trap is relying on nested prefix scans or loops to match data strings. Implementing a Trie data structure drastically reduces redundant CPU cycles and overall energy consumption.
- **The Solar Flare Scanner:** Centered around eliminating redundant mathematical operations. The trap occurs when developers use repeated slicing or nested loops to calculate overlapping sums in an array. The optimal approach is to use a sliding window algorithm to minimize unnecessary addition operations.
- **The Spatial Locality Crisis:** This challenge addresses the need for memory architecture awareness. The trap is traversing a massive 2D grid column by column. Because computers physically store arrays sequentially in RAM, non-sequential access causes high cache misses and massive energy spikes. The green solution is sequential row-major traversal.
- **The Copy-by-Value Sinkhole:** Focused on minimizing data duplication. The trap is passing large objects by value into functions, which forces the hardware to waste energy copying data unnecessarily. The efficient fix is to pass the data by constant reference.
- **The Orbital Collision Engine:** This scenario tackles computationally heavy distance algorithms. The trap is running expensive mathematical functions, such as exact square roots, for every pair of objects on a map, regardless of their proximity. Developers must learn to optimize their logic to bypass heavy math for entities that are obviously too far apart.

A more specific implementation of each challenge can be found in Appendix A.

5. Evaluation

5.1. Comparative Analysis

To evaluate the unique value proposition of *LeafCode*, we conducted a comparative analysis against three industry-standard platforms that overlap with our functional domain: LeetCode [12], CodeCarbon [13], and Green Algorithms [14]. The comparison is structured around four pillars: primary metrics, user incentives, overarching end-goals, and methodological approach.

LeetCode represents the pedagogical baseline for algorithmic practice. Although it successfully employs gamification and a vast library of challenges, its optimization targets are strictly limited to execution time and memory footprint. It lacks any dimension of environmental impact or energy awareness.

CodeCarbon provides an empirical measurement framework via a Python library. It is highly effective for retrospective carbon tracking, particularly in high-compute fields such as machine learning. However, it is a diagnostic tool rather than a learning platform; it lacks the structured challenges and competitive feedback loops necessary to drive proactive behavioral change in general software development.

Green Algorithms offers a theoretical approach to sustainability. It provides calculators that estimate footprints based on hardware specifications and runtime assumptions. Although it serves as an excellent resource for high-level project auditing, it does not provide the real-time code-level feedback required for developers to iterate on specific implementation logic.

The LeafCode Advantage As illustrated in Table 2, *LeafCode* bridges the gap between these solutions. Unlike LeetCode, our metrics are based on hardware-level energy consumption. Unlike CodeCarbon, we wrap these metrics in an interactive, challenge-based environment. Finally, unlike Green Algorithms, our feedback is derived from actual code execution via the `cloud-energy` tool. This synthesis allows *LeafCode* to transition from passive carbon reporting to active sustainable education, encouraging a "green-first" mindset in developers during the implementation phase, rather than as an afterthought.

5.2. User Evaluation: Narrative Scenarios

To evaluate the pedagogical impact of *LeafCode*, we present four representative narratives based on the platform's actual challenge library A. These stories illustrate how specific coding tasks influence developer behavior and facilitate the internalization of Green Software Engineering (GSE) principles. From the GSE principles, we derived four core technical principles, which are concurrency scaling, eliminating redundancy, memory architecture awareness, and proactive resource management in order to translate the GSE principles into principles closely related to the challenges.

Narrative 1: Concurrency Scaling (Ravi) 2 Ravi, a performance-oriented backend engineer, selects the **"The Turbo Trap"** challenge. The task requires applying a smoothing algorithm to telescope signals that vary in size from 10 to 10,000,000 frequencies. Ravi initially uses the `multiprocessing` library to utilize all CPU cores for every batch. Upon submission, the platform's feedback reveals that while his execution time was low, the energy consumption for small payloads was disproportionately high.
Outcome: Ravi refactors his solution to use a conditional threshold, processing small signals on a single thread to avoid the "energy tax" of booting up multiple cores. He successfully internalizes the principle of **Concurrency Scaling**.

Narrative 2: Eliminating Redundancy (Lena) 3 Lena, a student motivated by sustainability, attempts the **"The Solar Flare Scanner"** challenge. The goal is to find the maximum radiation absorbed in any continuous block of k seconds within a dataset of 1,000,000 points. Lena's initial approach uses a nested loop that recalculates the sum for every window. After viewing the high energy consumption results and the hint regarding redundant additions, she implements a sliding window algorithm.
Outcome: By reducing unnecessary mathematical operations, Lena dramatically lowers total CPU cycles and Joule consumption. This experience directly increases her GSE literacy by shifting her reasoning from passing tests to minimizing algorithmic carbon intensity. Furthermore, by achieving a high efficiency score, Lena builds a verifiable profile that demonstrates her ability to adapt standard coding skills into energy-efficient implementations.

Narrative 3: Memory Architecture Awareness (Marcus) Marcus, an experienced engineer, investigates **"The Spatial Locality Crisis"**. He must calculate the sum of elevation points in a massive 2D grid. The challenge's hint asks how a computer physically stores a 2D grid in RAM. Marcus compares two implementations: one traversing by columns and another by rows. The *LeafCode* profiling demonstrates a significant energy disparity due to cache misses in the non-sequential version.
Outcome: This narrative highlights the impact of **Memory Architecture** on power draw. Marcus learns that even "correct" logic can be energy-inefficient if it ignores spatial locality.

Narrative 4: Proactive Resource Management (Ji-woo) 8 Ji-woo, a junior developer, takes on **"The Patience Trap"**. She must program a Mars Rover to wait for a specific delay before transmitting data. Her first attempt uses a constant CPU polling loop to check the time. The platform provides a warning that constant polling drains the battery. She replaces the loop with a system sleep command.
Outcome: Ji-woo learns the importance of **Resource Management**. This challenge fundamentally changes her reasoning with energy optimization, moving her toward a mindset of resource conservation. By mastering this hardware-aware logic, Ji-woo demonstrates the exact skill adaptation that LeafCode aims to promote, signaling to companies that developers with high platform rankings possess the specialized literacy needed for sustainable infrastructure.

6. Results

The main result of this project is the successful development and deployment of the LeafCode prototype. We showed that it is possible to build a serverless web application that estimates code energy usage in real time. Through our custom backend, called The Runner, the platform securely runs user code in isolated

Table 2: Comparative Matrix of Coding and Sustainability Platforms

Feature	LeetCode	CodeCarbon	Green Algorithms	LeafCode
Primary Metric	Time & Memory	Estimated CO_2	Theoretical Carbon	Energy (Joules)
Incentive	Career Prep	Compliance/Report	Research Auditing	Behavioral Change
Approach	Gamified Coding	CLI/Library Tool	Web Calculator	Gamified Learning
End-Goal	Algorithmic Skill	Carbon Tracking	Carbon Estimation	Green Awareness
Measurement	Software-level	Runtime Estimation	Formula-based	Hardware-level (RAPL)

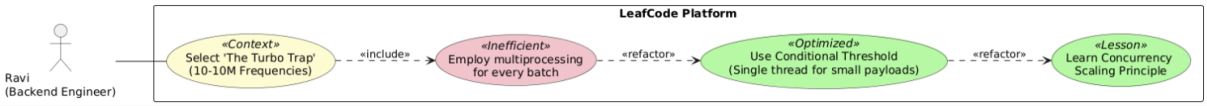


Figure 2: Narrative 1 - Ravi & Concurrency Scaling

sandboxes. The platform checks the code and then provides users with clear feedback on their estimated energy usage in Joules.

We also created a library of coding challenges. These challenges turn theoretical Green Software Engineering ideas into practical problems that users can solve. They cover important concepts such as concurrency scaling, memory awareness, and resource management.

To show that the platform actually works, screenshots are included of the user interface below. Figure 5 shows the live coding environment. Lastly, Figure 6 shows how our backend estimates energy and returns the Joule measurements to the user.

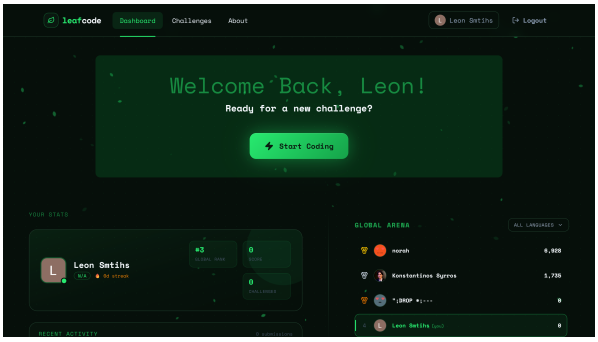


Figure 5: Dashboard of LeafCode

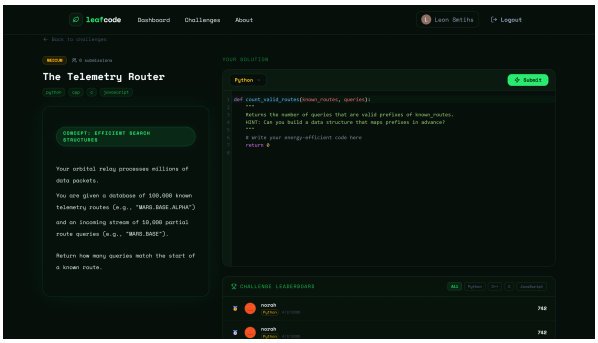


Figure 6: The Telemetry Router Challenge in LeafCode

LeafCode is accessible at `https://leafcode.konsyrros.me`.

7. Limitations

While LeafCode implements the base functionality of our proposed platform, it comes with limitations. The

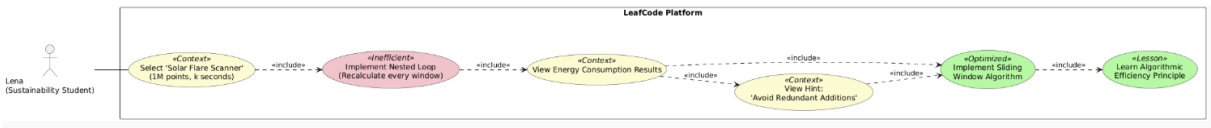


Figure 3: Narrative 2 - Lena & Eliminating Redundancy

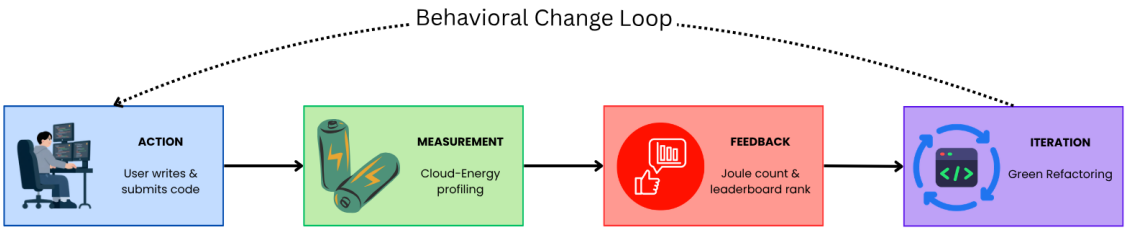


Figure 4: The LeafCode Behavioral Change Loop

Table 3: Persona-Impact Matrix: GSE Principles Internalized via Challenges

Persona	Challenge Source	GSE Principle Internalized
Ravi	The Turbo Tra	**Concurrency Scaling:** Balancing parallel overhead against payload size.
Lena	Solar Flare Scanner	**Redundant Operations:** Minimizing CPU cycles through efficient data structures.
Marcus	Spatial Locality Crisis	**Memory Architecture:** Optimizing data access patterns for cache efficiency.
Ji-woo	The Patience Trap	**Resource Management:** Avoiding active polling to preserve system power.

first and foremost is the limited selection of programming languages supported. Only four programming languages are supported. The implementation of more languages can attract more developers to program and critically develop their sustainable solutions. Further, the functionality tests are currently only supported in Python, with the other languages measuring energy on any code provided, no matter its correctness. Another limitation is that the energy usage of the code is not truly measured, but rather estimated from the CPU usage. This may pose inaccuracies between the given result and actual energy efficiency, though it is expected to correspond to some extent. This limitation is tied to the hardware and cloud architecture used. A limitation that potentially degrades user experience is using Vercel Serverless Functions, which causes the website to have cold start delays of around 3 seconds. Further, the energy consumption is estimated from the CPU utilization samples at 500ms intervals, in which short execution spikes may not be captured, leading to approximation errors. Finally, a user's previous submissions are not deleted or overwritten, which could allow users to accumulate repeated scores.

8. Future Work

Taking the above limitations into account, we have considered some potential solutions to be implemented in the future. Further programming languages can be implemented in order to attract more developers and instill a sustainable mindset when developing code. As the pedagogical impact of the platform was demonstrated through the use of narrative scenarios, future work can validate it through actual longitudinal empirical studies with real users, which can help analyse whether users retain their green "mindset". Functionality tests for other programming languages need to be implemented, and different potential solutions need to be tested to achieve the use of more accurate energy measurement tools, such as EnergiBridge. Finally, new features such as tutorials on how to code sustainably can be implemented to further help developers and provide a green mindset to existing developers, as well as new ones coming in the field.

9. Conclusion

There has been an increase in computing demand and data energy consumption throughout the years, but developers are rarely trained to consider the impact of energy on their implementation. The objective of the study is to ensure that it promotes energy efficiency as a primary metric to increase GSE literacy among developers. It ensures that developers reason with energy optimization, and encourages companies to engage developers to have green code implementation. This study proposes a coding platform called LeafCode in order to demonstrate that sustainability can be integrated directly into algorithmic education through measurable competitive feedback. LeetCode

follows the GSE principles in terms of its design, and to estimate the energy, Cloud Energy was used. Vercel was chosen for the deployment of the complete, end-to-end stack of LeafCode. The Prototype demonstrated that it is feasible to integrate energy estimation into a serverless web application, when combined with external code runners. The Runner architecture, multi-language support, and the sandboxed execution environment ensure that the energy consumption becomes a measurable estimated metric. The Evaluation demonstrated that LeafCode offers better qualities compared to platforms such as LeetCode, CodeCarbon, and Green Algorithms. Several narrative scenarios were added to illustrate how developers internalize sustainability principles through the challenges.

References

[1] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, Feb. 2020. DOI: 10.1126/science.aba3758. [Online]. Available: https://doi.org/10.1126/science.aba3758.

[2] C. Calero, M. Polo, and M. Á. Moraga, "Investigating the impact on execution time and energy consumption of developing with spring," *Sustainable Computing: Informatics and Systems*, vol. 32, p. 100 603, 2021, ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2021.100603. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537921000913.

[3] I. Manotas, L. Pollock, and J. Clause, "Seeds: A software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Technical Track / Research Paper, Hyderabad, India: ACM, May 2014, —.

[4] A. Hindle, "Green software engineering: The curse of methodology," in *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016) FOSE Track: Leaders of Tomorrow: Future Of Software Engineering*, Invited but peer-reviewed, Osaka, Japan: IEEE Computer Society, 2016, pp. 529–540. DOI: 10.1109/SANER.2016.60. [Online]. Available: https://softwareprocess.es/pubs/hindle2016SANERFOSE-green-software-engineering.pdf.

[5] LeafCode Team, *Leafcode - green code challenges*, https://leafcode.konsyrros.me/, Accessed: April 2, 2026, 2026.

[6] Arnia Software. "Green software engineering for a sustainable future. "[Online]. Available: https://www.arnia.com/green-software-engineering-for-a-sustainable-future/#Implementing_Green_Software_Practices.

[7] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing," *Energies*, vol. 12, no. 11, p. 2204, 2019. DOI: 10.3390/en12112204. [Online]. Available: https://www.mdpi.com/1996-1073/12/11/2204.

[8] G. Raffin and D. Trystram, "Dissecting the software-based measurement of cpu energy consumption: A comparative analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 1, pp. 96–107, 2025. DOI: 10.1109/TPDS.2024.3492336.

[9] E. García-Martín, C. F. Rodrigues, G. Riley, and H. Grahn, "Estimation of energy consumption in machine learning," *Journal of Parallel and Distributed Computing*, vol. 134, pp. 75–88, 2019, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.07.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518308773.

[10] Vercel Inc., *Vercel: Develop. preview. ship.* https://vercel.com/, 2026.

[11] green-coding-solutions, *cloud-energy: Energy estimation model for cloud servers (github repository)*, 2025. [Online]. Available: https://github.com/green-coding-solutions/cloud-energy.

[12] LeetCode, "Leetcode - the world's leading online programming learning platform," 2026, Accessed: 2026-03-28.

[13] V. Schmidt, K. Goyal, G. Joshi, M. Sarda, et al., "Codecarbon: Track and emit co2 from computing," 2026, Accessed: 2026-03-28.

[14] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Measuring the environmental impact of computing," 2026, Accessed: 2026-03-28.

A. LeafCode Challenge Library

This section details the specific coding challenges implemented in the *LeafCode* prototype, categorized by difficulty and core Green Software Engineering (GSE) concepts.

A.1. The Patience Trap

Difficulty: Easy
Concept: Resource Management 7]
Story: You are programming the Ares Mars Rover's communication relay. The orbiter will be in position in exactly `delay_ms` milliseconds 8, 9]. You must wait exactly that long to save battery, then encrypt and return the transmission payload 10].
Warning: Constant CPU polling will drain the battery!

Listing 1: Starter Code: Resource Management

```
def wait_and_transmit(api, delay_ms, payload_string):
    """
␣␣␣␣Waits␣for␣the␣specified␣delay_ms,␣then␣encrypts␣and␣returns␣the␣payload.
␣␣␣␣"""
    # Write your energy-efficient code here
    pass
```

A.2. The Turbo Trap

Difficulty: Hard
Concept: Concurrency Scaling
Story: A deep space telescope has captured an array of signal frequencies. You must apply a heavy mathematical smoothing algorithm to every frequency 26, 27]. The payload sizes vary wildly: sometimes 10 signals, and sometimes 10,000,000 28].
Hint: Is booting up 4 CPU cores always the most energy-efficient choice?

Listing 2: Starter Code: Concurrency Scaling

```
import multiprocessing

def process_signals(frequencies):
    """
␣␣␣␣Applies␣a␣mathematical␣smoothing␣function␣to␣a␣list␣of␣frequencies.
␣␣␣␣"""
    # Write your energy-efficient code here
    pass
```

A.3. The Telemetry Router

Difficulty: Medium
Concept: Efficient Search Structures
Story: Your orbital relay processes millions of data packets. Given a database of 100,000 known telemetry routes and a stream of 10,000 partial route queries, return how many queries match the start of a known route.
Hint: Can you build a data structure that maps prefixes in advance?

Listing 3: Starter Code: Efficient Search

```
def count_valid_routes(known_routes, queries):
    """
␣␣␣␣Returns␣the␣number␣of␣queries␣that␣are␣valid␣prefixes␣of␣known_routes.
␣␣␣␣"""
    # Write your energy-efficient code here
    return 0
```

A.4. The Solar Flare Scanner

Difficulty: Medium
Concept: Redundant Operations
Story: Your satellite has recorded 1,000,000 seconds of solar radiation data. Find the maximum radiation absorbed in any continuous block of exactly k seconds.
Hint: Do you really need to add up all the numbers again to find the next sum?

Listing 4: Starter Code: Redundant Operations

```
def max_radiation_window(radiation_data, k):
    """
␣␣␣␣Finds␣the␣maximum␣sum␣of␣any␣contiguous␣sub-array␣of␣length␣'k'.
␣␣␣␣"""
    # Write your energy-efficient code here
    return 0
```

A.5. The Spatial Locality Crisis

Difficulty: Medium
Concept: Memory Architecture
Story: You are processing a 2D topographical map represented as a massive grid. Calculate the sum of all elevation points.
Hint: How does the computer physically store a 2D grid in RAM?

Listing 5: Starter Code: Memory Architecture

```
long long calculateElevation(const std::vector<std::vector<int>>& grid, int size) {
    /*
     * Calculates the total sum of all points in the 2D grid.
     */
    // Write your energy-efficient code here
    return 0;
}
```

A.6. The Copy-by-Value Sinkhole

Difficulty: Easy
Concept: Data Duplication
Story: The central server receives massive user-profile objects from edge nodes. Write a function that returns true if a user's suspicion score is over 90.
Warning: How much energy does it take to hand this object to the function?

Listing 6: Starter Code: Data Duplication

```
bool evaluateProfile(UserProfile profile) {
    /*
     * Evaluates the profile.
     */
    // Write your energy-efficient code here
    return false;
}
```

A.7. The Orbital Collision Engine

Difficulty: Hard
Concept: Distance Algorithms
Story: Tracking 20,000 asteroids in a 2D plane, find how many pairs are dangerously close (distance $< D$).
Hint: math.sqrt() is heavy. Do you really need the exact distance between asteroids on opposite sides of the map?

Listing 7: Starter Code: Distance Algorithms

```
import math

def count_collisions(asteroids, dangerous_distance):
    """
␣␣␣␣Returns␣the␣total␣number␣of␣asteroid␣pairs␣within␣the␣dangerous_distance.
␣␣␣␣"""
    # Write your energy-efficient code here
    return 0
```

B. Figures Library

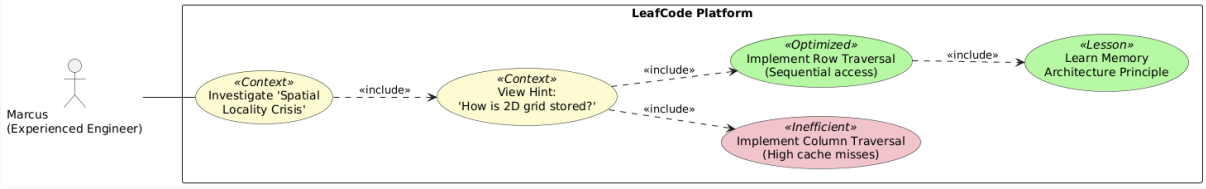


Figure 7: Narrative 3 - Marcus & Memory Architecture Awareness

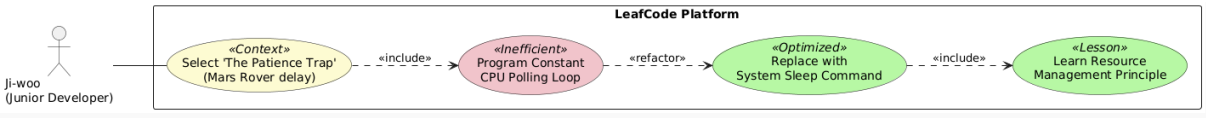


Figure 8: Narrative 4 - Ji-woo & Proactive Resource Management