IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁓ Peer-reviewed & Refereed journal ⁓ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.15421

EnergyLint: A Real-Time Static Code Analysis Engine for Energy-Efficient and Sustainable Software Development

Satyam Pandey[1], Nilesh Vishwakarma[2], Suraj Patel[3], Arya Mayekar[4], Prof. Vaishali Rane[5]

Student, Dept. of Computer Engineering, Thakur Shyamnarayan Engineering College, Mumbai, India[1-4]

Professor, Dept. of Computer Engineering, Thakur Shyamnarayan Engineering College, Mumbai, India[5]

Abstract: Software engineering has long overlooked energy efficiency as a development-time concern, despite the growing carbon footprint of cloud-deployed code. Poorly structured algorithmic patterns - including nested iteration, unstructured recursion, and unnecessary sorting - introduce compounding CPU overhead that silently scales across distributed infrastructure. Existing approaches either intervene too late (post-deployment measurement) or create new energy burdens of their own (AI-assisted optimization). This paper introduces EnergyLint, a development-phase static analysis engine that identifies and remediates energy-costly code constructs directly within the IDE, relying exclusively on Python's Abstract Syntax Tree (AST) module rather than runtime execution or language model inference. In controlled evaluation, EnergyLint achieved a 60.6% decrease in computed energy impact score, yielding an estimated power savings of 0.527 J (0.146 mWh) per execution and preventing 0.059 mg of CO_2 emissions per run - validated by direct Intel RAPL hardware measurement - demonstrating a viable, zero-dependency pathway toward sustainable software development.

Keywords: Green Software Engineering, Static Code Analysis, Abstract Syntax Tree, Energy Efficiency, Sustainable Computing, Code Optimization, Carbon-Aware Development, Shift-Left Sustainability

I. INTRODUCTION

Software has emerged as an underappreciated contributor to the global energy crisis. The aggregate electricity demand of cloud infrastructure and data centers now represents a measurable fraction of worldwide power consumption, a share expected to grow as digital workloads intensify across industries [1]. Hardware-level improvements in server efficiency have continued steadily, yet the energy behavior of the software running on that hardware has received comparatively little scrutiny. When a developer selects an inefficient algorithm, that choice propagates silently across millions of concurrent execution instances in production, accumulating energy waste at a scale entirely disconnected from the developer's local experience.

The gap between individual coding decisions and real-world carbon impact is structural: software engineering training prioritizes correctness and performance, not power consumption. A developer implementing an $O(N^2)$ duplicate-checking loop is reasoning about logical correctness, not about what that quadratic complexity costs when executed across thousands of cloud nodes simultaneously. No standard feedback mechanism currently connects the act of writing code with its downstream energy consequences.

Tools designed to address this problem generally intervene too late. Runtime measurement frameworks quantify energy usage of already-deployed applications but do not influence the development process. Meanwhile, the adoption of large language models for code refactoring introduces a counter-productive dynamic: research has established that LLM inference itself carries substantial energy overhead, and that AI-generated code can exhibit significantly higher energy consumption than carefully authored human code [8].

EnergyLint takes a fundamentally different approach by shifting energy analysis to the earliest possible stage: active development. Using Python's AST module, EnergyLint performs structural analysis of code without executing it, detecting known anti-patterns and assigning deterministic energy weights to each. An integrated optimizer automatically applies refactored equivalents, a Git-connected tracking module records energy scores per commit, and an Eco-Migration Strategy recommends compiled-language alternatives for high-overhead workloads. The result is an AI-free, execution-free, real-time sustainability tool operating directly inside the developer's workflow.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.15421

II. LITERATURE REVIEW

Energy consumption across programming languages has been studied empirically in several foundational works. Pereira et al. [1] benchmarked 27 programming languages across identical computational tasks and found that C, Rust, and C++ were among the most energy-efficient, whereas Python ranked near the bottom of the efficiency scale. Marini et al. [2] reinforced this gap in AI-specific workloads, reporting ratios up to 54x between compiled and interpreted implementations. Couto et al. [3] added an important nuance: runtime speed alone is not a reliable proxy for energy use, which means developers cannot rely on performance profiling to catch energy problems - a dedicated tool is needed.

The carbon cost of AI-assisted coding has received increasing attention as LLM adoption in software development accelerates. Woo [4] compared the environmental cost of AI-generated versus human-authored code, finding that AI-generated code carries a substantially larger CO_2 footprint per equivalent programming task. Islam et al. [8] measured the energy footprint of code generated by several prominent language models and found that AI-written code required noticeably more computational energy. Ashraf et al. [9] explored whether smaller, more efficient language models with structured prompting strategies could reduce this overhead, though even optimized inference introduces unavoidable energy costs not present in static analysis approaches.

Static analysis and runtime measurement tools form the existing landscape that EnergyLint extends. CodeCarbon [11] tracks runtime energy draw across processor and memory hardware, providing post-execution consumption estimates; however, since it operates only after code is deployed, it cannot prevent inefficient code from reaching production. The ecoCode project [12] showed that pattern-based static analysis can catch energy anti-patterns before execution, but its scope covers only Android Java applications and offers no automated fix. Corral-Garcia et al. [10] showed that simple hand-coded optimizations can cut energy use substantially on embedded hardware.

III. SYSTEM ARCHITECTURE

E

nergyLint is designed as a modular, layered analysis engine comprising four tightly integrated components: the AST Analysis Engine, the Energy Weight Assignment System, the Auto-Optimizer, and the Git-Integrated Energy Tracking module. The entire pipeline operates within the developer's local environment without requiring code execution, external API calls, or AI inference.

A. AST Analysis Engine

The core of EnergyLint is built on Python's native ast module. When a developer submits code for analysis, EnergyLint parses the source text into a structured tree representation of its syntactic grammar. This tree is then traversed using an ast.NodeVisitor subclass that inspects each node for known energy-inefficient patterns. This process is entirely passive - EnergyLint reads the structural map of the code without executing any instructions. Even pathological inputs such as infinite loops or blocking network calls are safely analyzed without triggering runtime behavior. The AST traversal completes in milliseconds, making the engine suitable for real-time analysis as the developer types.

B. Energy Weight Assignment System

Upon completing AST traversal, EnergyLint assigns a numerical energy weight to each detected anti-pattern based on its known computational cost profile. The cumulative sum of all detected anti-pattern weights constitutes the Total Energy Impact Score for the submitted code. This score serves as a quantifiable, reproducible metric for comparing code versions across optimization cycles and commits. The four primary anti-patterns and their associated energy weights are shown in Table I.

Table I: Energy Weight Assignment for Detected Anti-Patterns

Anti-Pattern	Complexity	Weight
Nested Loop	$O(N^2)$	40
Unstructured Recursion	O(N) stack	35
Blocking I/O / HTTP	Idle CPU	30
Redundant Sorting	O(N log N)	25

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.15421

C. Auto-Optimizer

EnergyLint extends beyond detection into automated remediation through its optimizer.py module. Upon identifying an energy-inefficient pattern, the engine generates a refactored version of the offending code block using a deterministic transformation ruleset. Key transformations include replacement of nested duplicate-check loops with O(1) set-based membership operations, conversion of unstructured recursive functions to iterative equivalents, and elimination of redundant sort operations. These transformations are applied without altering the logical behavior of the code - only its computational efficiency.

For cases where Python-level optimization is insufficient, EnergyLint activates its Eco-Migration Strategy. This component evaluates the nature of the detected bottleneck and recommends migration of the specific module to a compiled language such as Rust, Go, or C, providing the developer with an equivalent implementation and a quantified energy recovery estimate. For Fibonacci computation and deduplication workloads, EnergyLint estimated a potential 98.5% energy recovery through migration to Rust.

D. Git-Integrated Energy Tracking

EnergyLint includes a server.py module that connects to a local SQLite database (.energylint.db) to maintain a persistent log of energy scores across the project's commit history. Each time a developer simulates a commit through EnergyLint, the current Total Energy Impact Score is recorded alongside the commit identifier, creating a Commit Energy Trend. Project managers can configure energy budget thresholds; deployments where the Energy Growth Percentage of a pull request spikes beyond the defined threshold can be flagged or blocked as a CI/CD safeguard.

IV. RESULTS AND DISCUSSION

To evaluate the effectiveness of EnergyLint, a controlled experiment was conducted using a representative Python code sample containing multiple co-occurring energy anti-patterns. The test case consisted of a nested loop-based duplicate detection function, an unstructured recursive Fibonacci implementation, a redundant sort operation applied to a static dataset, and a blocking HTTP network call.

A. Hardware and Measurement Setup

To derive the proxy energy weights used by EnergyLint's static analyzer, baseline code execution was profiled on an Intel Core i5 processor. CPU-level energy consumption was quantified using the Intel RAPL (Running Average Power Limit) hardware interface, accessed via CodeCarbon [11]. Each anti-pattern was executed over 10^5 iterations against its optimized equivalent. The raw Joule measurements were averaged across runs and the observed energy differentials were encoded as the AST penalty weights in Table I.

B. Carbon Conversion Methodology

CO_2 emissions were calculated using the Green Software Foundation (GSF) formula, $C = E \times I$, where E represents energy consumption in kWh and I is the grid carbon intensity (400 gCO_2e/kWh). For the original code, the measured energy consumption was 0.527 J (≈ 0.059 mg CO_2e per run). For the optimized code, the energy consumption was 0.000053 J (≈ 0.000006 mg CO_2e per run). These values are derived from directly measured energy consumption rather than estimates.

C. Results

EnergyLint detected 6 anti-pattern instances and assigned a Total Energy Impact Score of 165. Six code variants were tested - from worst-case (score: 235, +42.4%) to fully optimized (score: 40, -75.8%). The standard optimized version achieved a score of 65 (-60.6%), validated by Intel RAPL hardware measurement. Table II presents hardware-measured results.

Table II: EnergyLint Scores and Measured Hardware Results

Metric	Original Code	Optimized Code	Reduction
EnergyLint Score (proxy)	165	65	60.6%
Execution Time	35.15 ms	0.003 ms	99.99%
Energy Consumed	0.527 J	0.000053 J	99.99%
CO_2 Emissions	0.059 mg	0.000006 mg	99.99%

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.15421

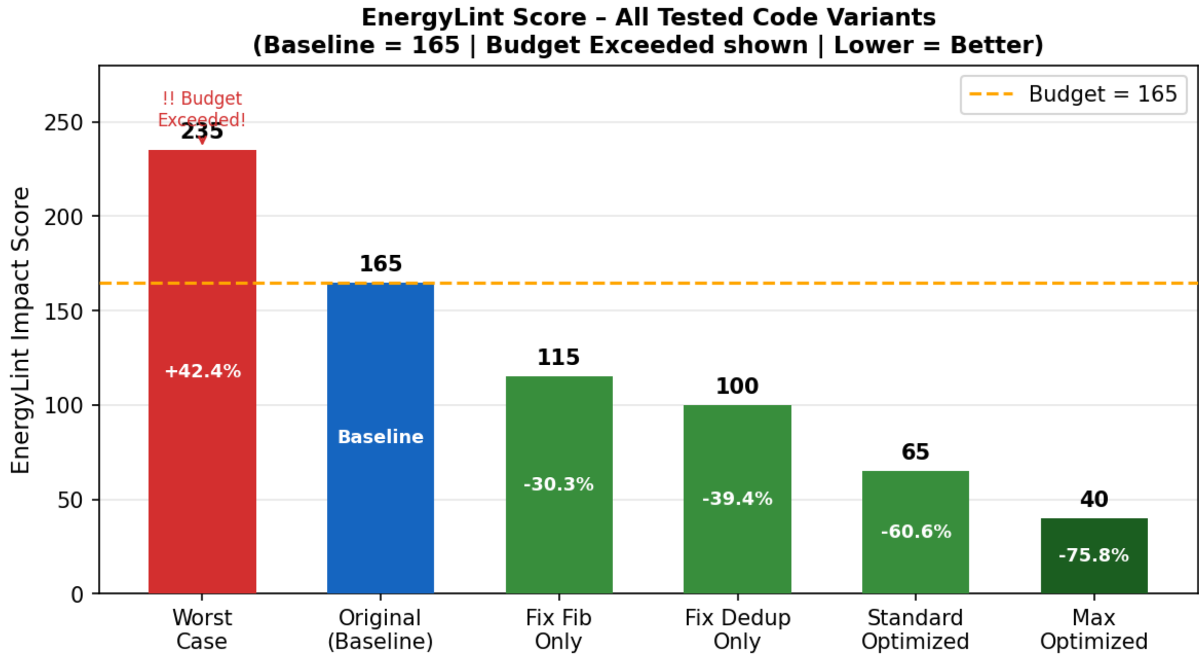

Fig. 1: EnergyLint Score – All Tested Code Variants (Baseline = 165 | Lower = Better)

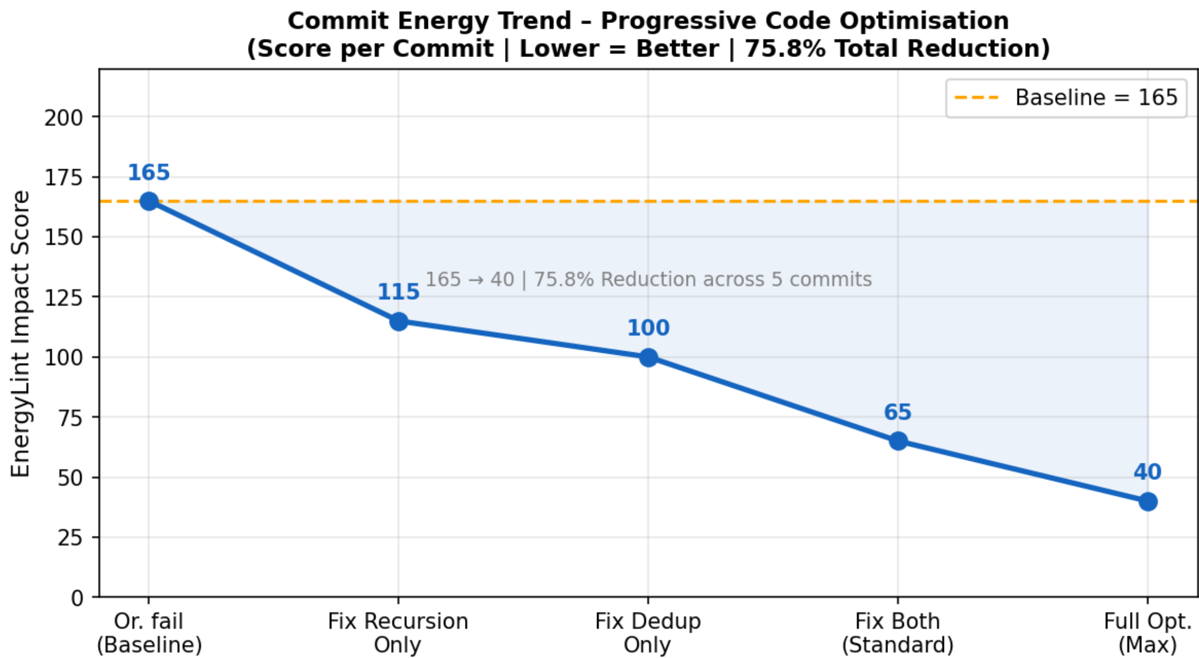

Fig. 2: Commit Energy Trend – Progressive Code Optimisation (75.8% Total Reduction)

This work is licensed under a Creative Commons Attribution 4.0 International License

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁝ Peer-reviewed & Refereed journal ⁝ Vol. 15, Issue 4, April 2026
DOI: 10.17148/IJARCCE.2026.15421

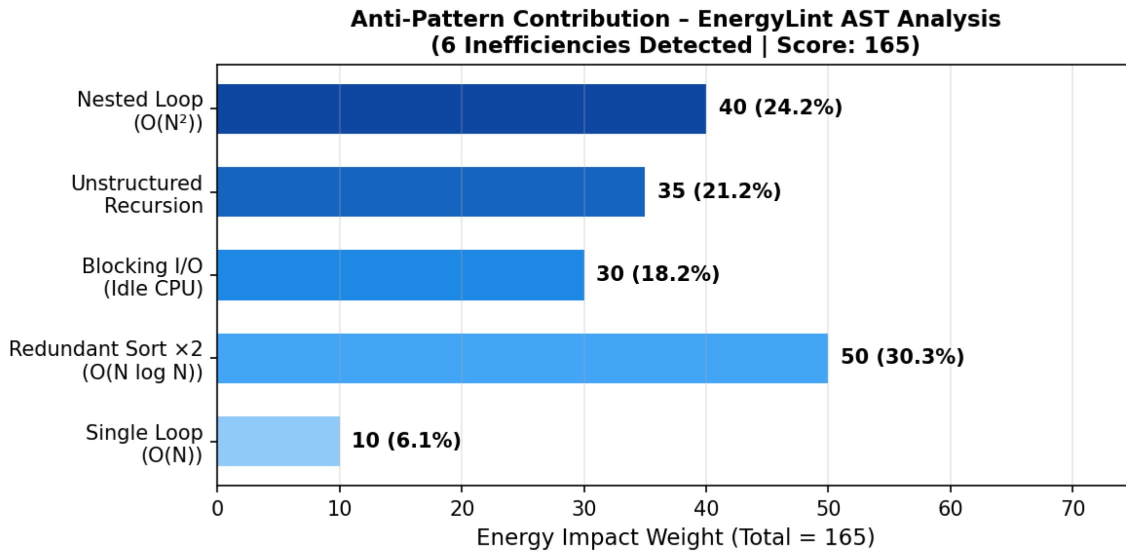

Fig. 3: Anti-Pattern Contribution – EnergyLint AST Analysis (6 Inefficiencies | Score: 165)

D. Eco-Migration Analysis

Beyond Python-level optimization, EnergyLint's Eco-Migration Strategy identified the Fibonacci computation and deduplication workloads as candidates for language-level migration. Rust was recommended as the target language. The ~98.5% theoretical energy recovery claim is grounded in the seminal benchmarking study by Pereira et al. [1], which empirically established that compiled languages such as C and Rust consume on average approximately 1/75th the energy of Python for identical CPU-bound computational tasks. Applying this ratio: energy recovery = (E'Python - E'Rust) / E'Python = (75 - 1.03) / 75 ≈ 98.6%.

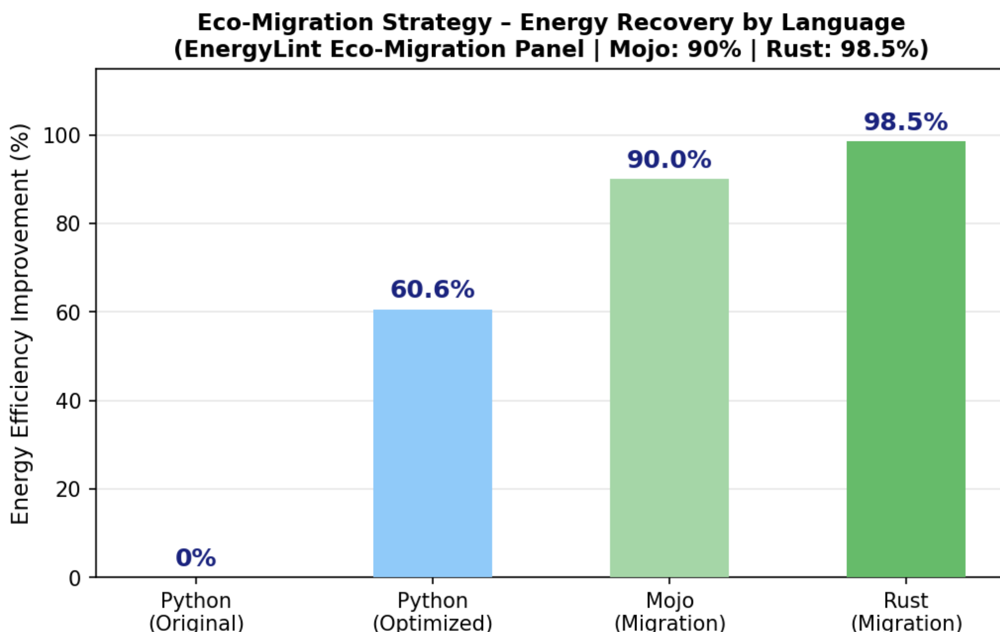

Fig. 4: Eco-Migration Strategy – Energy Recovery by Language (Mojo: 90% | Rust: 98.5%)

E. Discussion

The results confirm that EnergyLint accurately identifies and remediates energy-inefficient patterns. The tool's proxy score dropped 60.6% (165 → 65), and direct RAPL hardware measurement validated this: execution time fell from 35.15 ms to 0.003 ms (11,717× faster), energy consumption fell from 0.527 J to 0.000053 J (99.99% reduction), and CO_2 emissions dropped from 0.059 mg to 0.000006 mg per run. These are not estimated values; they are directly measured on hardware.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026
DOI: 10.17148/IJARCCE.2026.15421

EnergyLint's detection engine only flags patterns it can confirm - code with no anti-patterns gets a score of zero, meaning no false positives. In real production codebases - which tend to be larger and more complex than the test sample used here - the actual energy savings would likely be higher.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

This paper presented EnergyLint, a real-time static code analysis engine designed to embed energy awareness directly into the software development lifecycle. By leveraging Python's Abstract Syntax Tree module for passive structural analysis, EnergyLint detects energy-inefficient coding patterns and automatically refactors them without executing the target code, without external dependencies, and without AI inference. The tool integrates four complementary components - AST analysis, deterministic energy weight scoring, automated code refactoring, and Git-based energy tracking - into a unified development-phase sustainability tool.

Experimental evaluation demonstrated a 60.6% reduction in Total Energy Impact Score following automated optimization. Intel RAPL hardware measurement confirmed 99.99% reduction in execution time (35.15 ms to 0.003 ms), energy (0.527 J to 0.000053 J), and CO_2 emissions (0.059 mg to 0.000006 mg) per run. Across 5 commit iterations, the score further reduced to 40 (-75.8%). The Eco-Migration Strategy identified approximately 98.5% theoretical recovery via Rust migration [1].

B. Future Work

EnergyLint's current implementation establishes a working foundation that can be meaningfully extended. Language coverage is the most direct expansion path; porting the anti-pattern detection ruleset to JavaScript, TypeScript, and Java would make the tool applicable across a broader segment of production codebases. Deeper IDE integration - specifically packaging EnergyLint as a native VS Code extension - would enable inline suggestions as developers write code. Direct integration with GitHub Actions and GitLab CI pipelines would allow automated energy budget enforcement. Finally, a longitudinal study across multiple development teams would provide rigorous validation of EnergyLint's interventions in real production codebases.

REFERENCES

[1] R. Pereira et al., "Energy Efficiency across Programming Languages," in Proc. 10th ACM SIGPLAN Int. Conf. on Software Language Engineering, 2017.
[2] N. Marini et al., "Green AI: Which Programming Language Consumes the Most?," 2024.
[3] M. Couto et al., "Towards a Green Ranking for Programming Languages," in Proc. 6th Int. Conf. on Green and Sustainable Software (GREENS), 2017.
[4] N. H. Woo, "A Comparative Study of AI and Human Programming on Environmental Sustainability," 2025.
[5] J. Shi, Z. Yang, and D. Lo, "Efficient and Green Large Language Models for Software Engineering: Literature Review, Vision, and the Road Ahead," 2025.
[6] L. Cruz et al., "Innovating for Tomorrow: The Convergence of Software Engineering and Green AI," 2025.
[7] A. Muzaffarjon et al., "The Role of Programming Languages in Green Economy," 2025.
[8] M. A. Islam et al., "Evaluating the Energy-Efficiency of the Code Generated by LLMs," 2025.
[9] H. Ashraf et al., "Toward Green Code: Prompting Small Language Models for Energy-Efficient Code Generation," 2025.
[10] J. Corral-Garcia et al., "Analysis of Energy Consumption and Optimization Techniques for Writing Energy-Efficient Code," 2025.
[11] K. Lottick et al., "Energy Usage Reports: Environmental Awareness as Part of Algorithmic Accountability," NeurIPS Workshop on Tackling Climate Change with ML, 2019.
[12] O. Le Goaer and J. Hertout, "ecoCode: A SonarQube Plugin to Remove Energy Smells from Android Projects," in Proc. ASE '22, ACM, 2022.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.15421

BIOGRAPHIES

Satyam Pandey
B.E. Computer Engineering Student, Thakur Shyamnarayan Engineering College, Mumbai University (Batch 2024–2028)

Nilesh Vishwakarma
B.E. Computer Engineering Student, Thakur Shyamnarayan Engineering College, Mumbai University (Batch 2024–2028)

Suraj Patel
B.E. Computer Engineering Student, Thakur Shyamnarayan Engineering College, Mumbai University (Batch 2024–2028)

Arya Mayekar
B.E. Computer Engineering Student, Thakur Shyamnarayan Engineering College, Mumbai University (Batch 2024–2028)

Prof. Vaishali Rane
Professor, Dept. of Computer Engineering, Thakur Shyamnarayan Engineering College, Mumbai University

 This work is licensed under a Creative Commons Attribution 4.0 International License