Energy Efficiency as a New Paradigm in Software Engineering

Pero Ranilović, Dražen Marinković
Pan-European University "Apeiron", Banja Luka, RS, BiH
{pero.d.ranilovic, drazen.m.marinkovic }@apeiron-edu.eu

Review paper
https://doi.org/10.7251/JIT2501045R UDC: 004.41:004.738.5

Abstract: This paper presents a comprehensive overview of energy efficiency as a modern engineering paradigm in software development. With growing demands on digital infrastructure and increasing energy consumption in information and communication technologies (ICT), optimizing software for energy performance has become a key quality requirement—alongside scalability, performance, and security. Drawing from international standards (such as ISO/IEC 25010), sustainability frameworks (e.g., Green Software Foundation), and relevant scientific literature, the paper analyzes how architectural choices, programming languages, software practices, and toolchains influence energy usage. It further highlights good engineering practices, comparative language benchmarks, and the integration of energy awareness into modern development workflows, such as DevOps and CI/CD pipelines. The aim is to raise awareness among software engineers and decision-makers about the importance of sustainable software design and to offer practical guidelines for building energy-conscious systems.

Keywords: CI/CD, DevOps, energy-efficient software, green software foundation, ISO/IEC 25010, programming languages, performance optimization, sustainable software engineering

Introduction

In the past decade, we have witnessed a significant increase in energy consumption within the field of information and communication technologies (ICT), driven by the rising use of mobile devices, data centers, smart systems, and the Internet of Things (IoT). Although hardware has become notably more energy-efficient, the overall energy consumption continues to grow—partly because software is still not designed with energy efficiency as a core principle. This phenomenon is known as the "rebound effect," where gains achieved in hardware efficiency are offset by increased demands placed on software systems [1].

Today, software not only governs the functionality of systems but also determines how energy is consumed across all layers of a system. Consequently, there is an emerging need for energy efficiency to be treated not as a secondary concern, but as a fundamental non-functional requirement of modern software—on par with security, scalability, and performance[2].

According to the ISO/IEC 25010 international standard, energy efficiency is defined as one of the sub-characteristics of performance efficiency in software. This requirement implies that the system should use a minimal amount of resources (including energy) to achieve the required functionality [3]. In this context, energy efficiency is not optional—it is an integrated component of overall software quality.

An increasing number of organizations and institutions, including the Green Software Foundation and IEEE, are advocating for the integration of energy optimization measures from the earliest stages of software development [4].

The aim of this paper is to provide a comprehensive overview of energy efficiency as a modern engineering paradigm in software development, drawing on relevant expert literature, technical standards, and current engineering practices. In addition, the paper includes selected analyses from contemporary scientific research to further support its conclusions.

The structure of the paper follows a logical progression—from defining theoretical and regulatory frameworks, through technical factors influencing energy consumption, to an analysis of tools, programming languages, and engineering practices that contribute to the development of energy-conscious software.

Standards and Theoretical Frameworks for Energy-Efficient Software

One of the foundational documents for defining software quality is the international standard **ISO/IEC 25010:2011**, which explicitly includes energy efficiency as a key subcharacteristic of performance efficiency, alongside system responsiveness and resource utilization[3]. According to the standard, energy efficiency refers to the degree to which a software product uses appropriate amounts of resources relative to the performance it delivers. This implies that efficient software should fulfill its functions while minimizing energy consumption—an especially important requirement in environments with limited computational capacity, such as embedded systems, mobile devices, and wireless sensor networks.

In addition to ISO standards, the field of energy-aware computing has seen increasing support through technical initiatives and environmental regulations. Notable among these are the **IEEE 1680.1** and **1680.2** standards, which address the environmental performance of electronic products, including software bundled with hardware (e.g., firmware, drivers). These standards provide guidance for evaluating the energy and environmental impact across the lifecycle of IT products [1] .

A more recent and influential initiative is the **Green Software Foundation**, which advocates for sustainable software engineering practices. Their principles emphasize energy measurability, design efficiency, data minimization, and climate impact awareness in software decision-making [4]. These concepts aim to embed sustainability into every phase of the software lifecycle—from design to deployment.

These efforts are further supported by **global policy frameworks**, such as the **United Nations Sustainable Development Goals (SDGs)**, particularly **SDG 9** (Industry, Innovation and Infrastructure) and **SDG 12** (Responsible Consumption and Production), which call for technological innovation aligned with environmental limits [2].

The concept of sustainability is now increasingly integrated into software engineering not only as a social responsibility but also as a strategic design objective. This is best exemplified in the growing adoption of **ESG (Environmental, Social, and Governance)** frameworks across the tech sector, where energy-efficient software contributes directly to the environmental pillar. Recent studies highlight how responsible AI, cloud infrastructure, and IoT systems are reshaping how developers incorporate energy considerations into software design from the outset [2].

As these frameworks and standards become embedded into standard development workflows, **energy efficiency is no longer a feature of innovation—it is a requirement of modern engineering responsibility**.

Factors Influencing Energy Consumption in Software Systems

Understanding what drives energy consumption in software systems is essential for engineers striving to build sustainable and resource-conscious solutions. Multiple interdependent factors shape how much energy is consumed—from system architecture and algorithm design to programming language, memory usage, and I/O behavior. Addressing these factors in a systematic way can substantially reduce the energy footprint of software.

One of the most fundamental influences lies in the **software architecture**. The decision between monolithic, microservice-based, or event-driven models has direct implications for energy use. Modular architectures often enable components to be independently paused or shut down during periods of low activity, thereby conserving power. However, microservices, while scalable and maintainable, introduce significant communication overhead, especially in distributed systems, leading to increased network traffic and energy usage [5].

Algorithmic efficiency is another central determinant. Efficient algorithms reduce the number of instructions executed by the processor, limiting both CPU cycles and memory accesses—two operations with high energy cost. For example, replacing linear search with binary search, or opting for a heap over

a simple list when managing priority queues, significantly improves energy use. Memory-aware designs, such as tiling in matrix operations or optimized caching, have also been shown to cut energy costs across scientific workloads [7].

The **programming language** used in a project also plays a notable role. A well-known study by Pereira et al. (2017) compared 27 programming languages across ten standard algorithms, measuring execution time, memory use, and energy consumption. The study revealed that compiled, low-level languages like **C and Rust** consistently outperform interpreted languages like **Python** and **JavaScript** in terms of energy efficiency [6]. This is due to lower abstraction layers, reduced runtime overhead, and more granular memory control.

Functional programming languages (e.g., Haskell, Erlang) offer benefits in concurrency but may incur higher memory use due to immutable data structures. Therefore, the **paradigm** must align with the performance and energy profile of the application domain.

Among the most overlooked yet impactful contributors to energy waste are **input/output operations and memory access patterns**. File reads and writes, frequent memory allocations, and repeated network calls can drastically inflate energy costs. For example, making uncached HTTP requests in a loop or executing poorly batched database queries can more than double energy use compared to optimized versions [5].

Additionally, metrics like **cache hits/misses**, **context switches**, and **CPU migrations** have been shown to correlate strongly with energy consumption in empirical studies across multiple workloads. These metrics should therefore be monitored as part of any serious energy profiling effort.

In sum, optimizing for energy efficiency involves careful selection of architecture, algorithm, language, and system-level operations. Each design decision reverberates through the energy consumption chain, and only a holistic view can ensure effective improvements.

Good Engineering Practices for Energy-Efficient Development

Energy efficiency in software is not achieved by accident—it is the result of deliberate and thoughtful engineering practices. From the earliest design stages to deployment and testing, integrating sustainability-oriented strategies can significantly reduce the total energy consumption of software systems. As with performance and scalability, addressing energy use is most effective when it is embedded from the beginning of the development lifecycle.

Several well-established design principles have a particularly strong impact on reducing energy waste:

- **Minimizing complexity**: Simplified, clean code and well-structured logic reduce the computational overhead required for program execution. Reducing nested loops, redundant conditions, and unnecessary abstractions allows the processor to complete tasks with fewer operations and lower power demand.
- **Modularity**: Dividing software into smaller, independent modules enables more efficient control over component execution. Modules not currently in use can be unloaded or deactivated, particularly in mobile and embedded systems, thereby reducing background energy draw.
- **Data locality**: Ensuring that data is kept close to where it is processed (e.g., within the same memory hierarchy level or server node) significantly reduces the need for resource-expensive memory accesses and network requests. This practice not only reduces latency but also energy costs tied to I/O and communication operations [5].
- **Avoiding unnecessary computation**: Repetitive function calls, redundant loops, polling mechanisms without delays, and repeated data loading are common sources of waste. Optimizing these patterns—by introducing caching, memoization, and lazy evaluation—can lead to substantial reductions in CPU activity and memory usage.

Although these principles are not new, they are deeply rooted in software craftsmanship values. For instance, the well-known book *Clean Code* by Robert C. Martin emphasizes clarity, modularity, and simplicity—not with energy in mind, but for maintainability and robustness. Yet, their implementation naturally supports energy efficiency as a beneficial side effect [8].

Beyond good design, developers should be equally vigilant about eliminating **anti-patterns**—common but inefficient coding practices that contribute to unnecessary energy use. These include:

- Repetitive execution of logic within tight loops, especially when the logic can be precomputed or simplified;
- Repeatedly opening and closing files or database connections instead of reusing persistent sessions;
- Memory mismanagement, such as allowing object bloat or failing to deallocate unused memory, which leads to more frequent garbage collection and higher RAM usage.

Modern development environments offer powerful tools for detecting and addressing such inefficiencies. **Static code analyzers** (e.g., SonarQube, Pylint) and **profiling tools** (e.g., GreenScaler for Java, Intel VTune, VisualVM) allow developers to identify bottlenecks and measure how different segments of the code contribute to CPU usage, memory allocation, and energy drain [6].

Practical Examples of Energy-Aware Engineering

Research from the **Green Software Laboratory** and field studies [4] have identified several coding strategies that consistently reduce software energy consumption. These are summarized in the table 1 below:

Table 1. *Summary of Key Coding Practices for Energy-Efficient Software Development*

Practice	Description
Efficient algorithm selection	Replace costly operations (e.g., Bubble Sort) with optimized versions (e.g., Quick Sort).
Caching of results	Store computation results to prevent repeated expensive operations.
Lazy loading of components	Load modules or libraries only when they are actually required.
Data compression before transmission	Reduce data size before network transfer to lower bandwidth and CPU use.
Elimination of "busy wait" loops	Avoid while(true) loops that consume CPU without productive work.
Load-aware system scaling	Enable systems to downscale energy usage when demand is low.

Implementing these practices doesn't necessarily require a shift in tooling or technology stack. In many cases, teams can begin by including energy-related checks in code reviews, defining internal guidelines that promote resource awareness, and using profiling data as part of standard QA procedures.

By embedding such practices into development culture, teams not only improve performance but also directly contribute to the global effort of reducing the carbon footprint of digital infrastructure.

Comparative Analysis of Programming Languages in Terms of Energy Efficiency

Programming languages vary significantly in their energy efficiency, which depends not only on how code is written but also on how it is compiled, executed, and optimized. Key factors include the language's execution model (compiled vs. interpreted), memory management behavior, and compiler performance. When developing software for energy-constrained platforms—such as embedded systems, mobile devices, or large-scale servers—choosing the right programming language can substantially influence the overall energy footprint of an application.

Empirical Measurements of Energy Consumption by Language

One of the most comprehensive studies in this area is the work by Pereira et al. (2017), which analyzed 27 programming languages across ten common algorithmic tasks. The study measured execution time, memory usage, and energy consumption[6].

Table 2. *Comparative Analysis of Energy Efficiency, Execution Speed, and Memory Usage Across Programming Languages*

Language	Energy Consumption	Execution Speed	Memory Usage
C	Lowest	Fastest	Low
Rust	Very low	Fast	Low
Java	Medium	Moderate	High (GC overhead)
Python	High	Slow	Moderate
JavaScript	High	Slow	Moderate

Source: *Pereira et al., 2017*

These findings suggest that compiled, low-level languages like C and Rust are far superior in energy-critical applications (Table 2). C, due to its minimal runtime overhead and direct hardware access,

consistently ranks highest for efficiency. Rust, while higher-level and type-safe, still achieves near-C performance due to its powerful compiler optimizations and memory safety features without garbage collection.

Languages such as Java and C# strike a balance by offering higher developer productivity through managed runtime environments. However, their memory usage tends to be higher, especially in long-lived applications where garbage collection processes introduce unpredictable spikes in CPU and memory activity [11].

In contrast, interpreted languages like Python and JavaScript perform the worst in terms of energy efficiency. Their dynamic typing, runtime interpretation, and rich—but heavy—standard libraries result in both slower execution and greater energy consumption [10].

Programming Paradigms and Their Energy Profiles

Besides language choice, the programming paradigm plays a vital role in determining energy behavior. Imperative languages such as C and Go allow fine-grained control over memory and computation, leading to predictable and efficient execution paths. Functional languages, such as Haskell or Erlang, often favor immutability and recursion, which can increase memory usage and stack depth—resulting in higher energy consumption unless carefully optimized [9].

Compiler behavior is also critical. For instance, enabling or disabling specific compiler optimizations can have a dramatic impact on energy consumption. In the case of Haskell, Kirkeby et al. (2024) found that disabling just a few of the Glasgow Haskell Compiler (GHC) optimizations led to significantly less efficient executables, both in terms of time and energy. Therefore, compiler configuration must be considered as part of language energy profiling—not all compilers are equal, and settings such as *-O2* or *-fno-** flags directly influence energy outcomes.

Furthermore, the compilation process itself—how and when code is translated—matters. Interpreted code or just-in-time compiled (JIT) code (like PyPy for Python) often leads to higher startup costs and runtime overhead, though dynamic recompilation techniques are improving these deficits over time [12].

Practical Implications for Language Selection

When selecting a language for an energy-sensitive system, developers must weigh several factors:

- Execution duration and frequency: For applications that run continuously (e.g., backend services), using efficient compiled languages (like C or Rust) can significantly reduce operational costs and environmental impact.
- Platform limitations: On devices with strict energy budgets (e.g., IoT sensors), interpreted languages are often unsuitable.
- Development priorities: In scenarios where rapid prototyping is more valuable than runtime efficiency, interpreted or semi-compiled languages may still be acceptable.

Nevertheless, energy efficiency should increasingly be considered a first-class requirement in system design. Balancing development speed with sustainable execution is becoming a defining challenge of modern software engineering.

Tools and Techniques for Measuring Energy Efficiency in Software

Measurability is the foundation of every meaningful optimization. In the context of energy-efficient software engineering, understanding how and where software consumes energy is critical for making informed design and development decisions. While energy consumption has traditionally been associated with hardware, today a broad ecosystem of tools and techniques is available to help engineers quantify and optimize the energy footprint of software—from the earliest coding stages to full deployment.

Some tools rely on direct hardware-based measurement using embedded sensors. For example, **Intel Power Gadget** enables precise monitoring of CPU energy use in real time on Intel platforms. These tools offer high measurement accuracy but are limited to specific hardware architectures, reducing their portability and broader applicability.

In addition to hardware-based tools, dynamic profiling solutions have become increasingly popular for capturing real-time energy behavior of software during execution. A notable example is **GreenScaler**, which automatically generates test cases to construct energy models of applications. This profiler helps developers detect energy regressions between soft-

ware versions and is especially useful in mobile or resource-constrained environments [13].

For embedded systems and IoT applications, where energy is a critical constraint, static analysis tools provide a different type of insight. These tools estimate energy consumption by analyzing the program's control flow and logic without needing to execute the code. For instance, **EnergyAnalyzer**, developed under the European TeamPlay project, uses worst-case execution time (WCET) techniques to estimate the energy cost of software components. This helps developers identify energy hotspots early in development, potentially even before full implementation[14].

Another line of work is focused on **static energy modeling** at the source-code level. Research by **Haj-Yihia and Ben-Asher (2017)** demonstrates how symbolic execution and path analysis can be used to predict energy usage across various CPU architectures. Their approach includes modeling memory usage and cache behavior, which are critical for accurate estimation of total energy cost [15]. While technically demanding, such tools offer valuable guidance during code optimization and compilation.

More and more organizations are integrating these tools into their CI/CD workflows. By tracking energy metrics along with traditional KPIs like performance and security, energy efficiency becomes an embedded part of quality assurance. For example, GreenScaler can flag inefficient changes in new code commits, while static analysis tools help developers configure compilers or detect early inefficiencies.

Despite this progress, several challenges remain. Many tools are platform-specific and rely on non-standard metrics, making cross-platform comparison difficult. Furthermore, most solutions focus exclusively on CPU consumption, neglecting other critical components such as GPUs, memory buses, and network cards.

Looking ahead, the development of hybrid tools that combine static and dynamic analysis, along with standardized models for energy reporting, will be essential. Such advancements would not only improve precision but also allow engineers to compare results across platforms and programming environments.

Ultimately, the ability to measure energy use in software is no longer a luxury—it is a professional necessity. Engineers equipped with the tools and knowledge to assess their code's energy impact are better positioned to make sustainable, efficient, and forward-thinking design decisions in the digital age.

Integrating Energy Efficiency into the Software Development Lifecycle

As the awareness of sustainable engineering grows, software energy efficiency must be embedded not only in the product but also in the process. The integration of energy metrics into the software development lifecycle (SDLC) is becoming an emerging best practice, particularly within Agile and DevOps frameworks.

Modern development teams rely heavily on Continuous Integration and Continuous Delivery (CI/CD) pipelines to automate software testing and deployment. These automated pipelines are now evolving to include **energy profiling and optimization checkpoints**. By integrating tools like GreenScaler and static energy analyzers into CI/CD, teams can continuously track and optimize energy usage during software builds and releases [16].

This integration doesn't stop at tooling. Some organizations have begun defining **"green" acceptance criteria** as part of Agile user stories, ensuring that new features must meet both functional and energy efficiency requirements. This cultural shift promotes shared ownership of sustainability goals, aligning developers, testers, and operations teams under the same value system [17].

From a strategic standpoint, the most effective teams embed energy-awareness into three stages:

1. **Pre-development planning**: Estimating the energy cost of alternative implementation paths and choosing the most efficient.
2. **Development and testing**: Using profilers and simulators to test energy consumption during code changes.
3. **Post-deployment monitoring**: Logging real-time energy metrics to dashboards, much like performance logs or error tracking.

Some DevOps pipelines now include feedback loops where **energy regressions trigger alerts**, just like failing tests. In one case study, applying this principle led to a 22% reduction in overall cloud infrastructure costs simply by removing a memory-intensive module that previously went unnoticed in traditional code reviews [18].

Organizations are also increasingly integrating energy analysis into **code quality dashboards**, using metrics such as energy per transaction, energy per test suite, and deployment energy impact. These indicators provide clarity and accountability, enabling software teams to monitor their impact over time without disrupting their existing workflows [19].

Adopting energy-conscious practices within SDLC not only reduces environmental impact but also optimizes performance, infrastructure utilization, and cost. As tooling and awareness continue to mature, integrating energy metrics into Agile and DevOps processes is transitioning from a novelty to a necessity.

Experimental Validation of Energy-Efficient Coding Practices in .NET

To support the theoretical findings and recommendations outlined in this paper, a series of simple experiments were conducted using the .NET platform (C# language) in a local development environment. The goal was to observe and compare CPU resource utilization and execution time for different software operations. This practical component aimed to demonstrate how various implementation choices—particularly algorithm efficiency, I/O operations, and parallelization strategies—can influence energy-related metrics, even in small-scale desktop applications.

The rationale for measuring CPU usage stems from the fact that processor time is one of the most energy-intensive resources in computing systems. By monitoring CPU load during execution of selected methods, we obtain a proxy for energy consumption. Although these experiments do not measure energy in joules, they provide meaningful insights into computational efficiency, which strongly correlates with energy use in real-world scenarios. The implementation of the measurement logic is shown in Figure 1, where part of the source code demonstrates the usage of the Stopwatch class for execution time and the TotalProcessorTime property for calculating CPU usage. The testing was performed on a personal computer equipped with the following specifications:

- Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- RAM: 16 GB
- Operating System: Windows 10
- Development Framework: .NET 8 (C#)

The testing was done using the Stopwatch class from the System.Diagnostics namespace to measure execution time, and the TotalProcessorTime property from the current process to calculate CPU usage percentage (Figure 1). The measurements included:

- **Algorithm comparison**: A naïve implementation of the Bubble Sort algorithm was compared with the optimized built-in Array.Sort() method on arrays of 100,000 elements.
- **I/O operations**: Two standard methods for reading large text files were compared — File.ReadAllLines() vs. File.ReadLines() — to assess how different memory-loading strategies affect performance.
- **Parallel vs. serial execution**: LINQ-based data processing was executed in both serial and Parallel.ForEach configurations to investigate the tradeoff between parallelism and CPU usage.

```
{
    // SORTING TEST
    var rand = new Random();
    var array1 = Enumerable.Range(0, 100000).OrderBy(x => rand.Next()).ToArray();
    var array2 = (int[])array1.Clone();

    Measure("Bubble Sort", () => BubbleSort(array1));
    Measure("Array.Sort", () => Array.Sort(array2));

    // FILE I/O TEST
    string filePath = "data.txt";
    if (!File.Exists(filePath))
    {
        File.WriteAllLines(filePath, Enumerable.Range(0, 100000).Select(i => $"Line {i}"));
    }

    Measure("File.ReadAllLines", () => File.ReadAllLines(filePath));
    Measure("File.ReadLines", () => File.ReadLines(filePath).ToList());

    // LINQ SERIAL VS PARALLEL
    var numbers = Enumerable.Range(1, 100000).ToArray();

    Measure("Serial LINQ", () =>
    {
        var result = numbers.Select(x => Math.Sqrt(x)).ToArray();
    });

    Measure("Parallel LINQ", () =>
    {
        var result = numbers.AsParallel().Select(x => Math.Sqrt(x)).ToArray();
    });
}
```

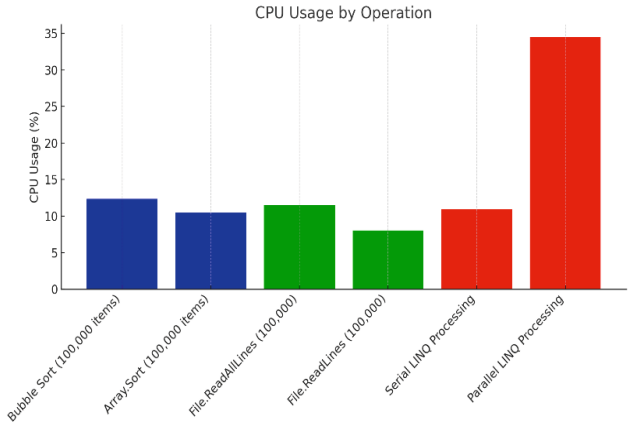
Figure 1. *Part of source Code for Experimental Measurements*

These experiments, though relatively simple and time-limited, were chosen to illustrate the real impact of code design decisions on resource consumption. They were executed under consistent conditions and without background processes, ensuring reliability of the results. The summarized results of all tests are presented in Table 3, showing both CPU usage percentage and execution time for each operation. Additionally, Graph 1 visualizes CPU usage by operation type, with color-coded categories to emphasize differences across algorithmic, I/O, and data processing domains.

These findings clearly demonstrate that more "optimized" or built-in solutions tend to consume fewer resources, while parallel execution, although potentially faster in theory, may introduce overheads that reduce energy efficiency for moderate workloads. The experiments underscore the importance of performance-conscious design choices and support the thesis that energy efficiency should be considered a core concern in everyday software development. The results were systematically recorded and summarized in the following table:

Table 3. *Comparative Analysis of Energy Efficiency, Execution Speed, and Memory Usage Across Programming Languages*

Operation	CPU Usage (%)	Time (ms)
Bubble Sort (100,000 items)	12.35%	42,434
Array.Sort (100,000 items)	10.45%	7
File.ReadAllLines (100,000)	11.49%	16
File.ReadLines (100,000)	8.03%	24
Serial LINQ Processing	10.93%	35.74
Parallel LINQ Processing	34.50%	101.89



Graph 1. *CPU Usage by Operation*

These experimental results, although limited in scope, offer compelling validation for the theoretical principles discussed throughout the paper. They highlight how even basic implementation choices can substantially affect CPU utilization and performance. Integrating such lightweight measurement strategies into standard development workflows can help teams build more energy-conscious software without requiring complex infrastructure or tools. Future research and practice should focus on expanding this methodology with more precise energy metering tools and broader test coverage across different platforms and workloads.

CONCLUSION

The integration of energy efficiency into software engineering marks a significant evolution in how digital systems are conceived, developed, and maintained. No longer relegated to low-level hardware concerns or experimental projects, energy-aware programming has become a critical aspect of responsible, modern software development. As demonstrated in this paper, energy efficiency must be treated as a first-class non-functional requirement—alongside performance, scalability, and security—especially as computing ecosystems grow increasingly complex and resource-intensive.

Global standards such as ISO/IEC 25010 and initiatives like the Green Software Foundation have laid a strong foundation for embedding sustainability into engineering processes (ISO/IEC 25010, 2011), (Green Software Foundation, 2022). Software design choices—from programming languages and data structures to CI/CD pipeline configurations—play a decisive role in shaping the energy profile of applications. Moreover, empirical studies have underscored the tangible impact that these decisions have on resource consumption across different execution environments [6].

Despite promising advances, several challenges remain. Tooling for real-time and fine-grained energy measurement is still fragmented and not yet standardized across platforms. Educational curricula have yet to fully integrate sustainable software practices, leaving a knowledge gap among new developers. Industry adoption is also uneven—particularly among small and medium-sized enterprises (SMEs)—due to the perceived overhead of incorporating energy metrics into workflows [20].

Looking forward, future research and practice should aim to address these gaps through:

- **Standardized tooling**: Developing cross-platform tools and APIs for measuring and visualizing energy usage in a consistent and vendor-neutral way.
- **Developer education**: Introducing energy-aware programming as a core module in software engineering education, supported by interactive labs and gamified challenges.
- **Policy integration**: Encouraging government and enterprise procurement policies to prioritize energy-aware software products.

- **Holistic frameworks**: Building unified models that integrate energy metrics into quality assurance, compliance, and continuous integration pipelines[18].

A simple experimental case study presented in this paper has further illustrated how basic code-level decisions—such as algorithm choice, file handling, or parallelization—can lead to measurable differences in CPU usage and execution time, reinforcing the importance of energy-conscious design.

The road to widespread adoption of energy-aware software engineering will require continued collaboration across academia, industry, and policy makers. However, the potential benefits—both ecological and economic—make this a worthy and necessary pursuit. Energy-efficient software is not only about conserving watts; it's about building a more sustainable digital future. It is also important to note that the availability of high-quality research on this topic remains limited, and future studies should focus on developing standardized methodologies and tools for measuring software energy efficiency.

References

[1] Grosskop, K., & Visser, J. (2013). Energy Efficiency Optimization of Application Software. Advances in Computers, 88, 199–241.

[2] Lee, S. U., Fernando, N., Lee, K., & Schneider, J.-G. (2024). A Survey of Energy Concerns for Software Engineering. Journal of Systems and Software, 210, 111944.

[3] Trichkova-Kashamova, E. (2021). Applying the ISO/IEC 25010 Quality Models. 2021 12th National Conference (ELECTRONICA).

[4] Qiang, Y., Che Pa, N., & Ismail, R. (2024). Sustainable Software Solutions: A Tool Integrating Life Cycle Analysis and ISO Quality Models. International Journal on Advanced Science, Engineering and Information Technology.

[5] Rajan, A., Noureddine, A., & Stratis, P. (2016). A Study on the Influence of Software and Hardware Features on Program Energy. ESEM 2016.

[6] Pereira, R., Couto, M., Ribeiro, B., & Saraiva, J. (2017). Energy Efficiency Across Programming Languages. ACM SIGPLAN Conference on Software Language Engineering.

[7] Lopes, G. (2019). A Study on the Energy Efficiency of Matrix Transposition Algorithms.

[8] Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

[9] Kirkeby, M. H., Santos, B., Fernandes, J. P., & Pardo, A. (2024). Compiling Haskell for Energy Efficiency: Empirical Analysis of Individual Transformations. Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing.

[10] Georgiou, S., Kechagia, M., & Spinellis, D. (2017). Analyzing Programming Languages' Energy Consumption: An Empirical Study. Proceedings of the 21st Pan-Hellenic Conference on Informatics.

[11] Kim, S., Tomar, S., Vijaykrishnan, N., Kandemir, M., & Irwin, M. (2004). Energy-Efficient Java Execution Using Local Memory and Object Co-location. IEE Proceedings - Computers and Digital Techniques, 151(1), 33–42.

[12] Li, Y., & Jiang, Z. (2019). Assessing and Optimizing the Performance Impact of the Just-in-Time Configuration Parameters – A Case Study on PyPy. Empirical Software Engineering.

[13] Chowdhury, R. R., Borle, N., & Ernst, R. (2018). *GreenScaler: Training software energy models with automatic test generation*. Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops.

[14] Wegener, S., Nikov, S., & Tverdokhlebov, A. (2023). *EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate Software Energy Consumption*. Proceedings of the 2023 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems.

[15] Haj-Yihia, M., & Ben-Asher, Y. (2017). *Software Static Energy Modeling for Modern Processors*. Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition.

[16] Raheem, A., Osilaja, A. M., Kolawole, I., & Essien, V. E. (2024). Exploring continuous integration and deployment strategies for streamlined DevOps processes in software engineering practices. *World Journal of Advanced Research and Reviews.*

[17] Gowda, P. G., Stanley, N. S. A., & Joyce, J. E. (2024). DevOps Dynamics: Tools Driving Continuous Integration and Deployment. *2024 IEEE International Conference on IT, Electronics and Intelligent Communication Systems.*

[18] Johnson, O. B., Olamijuwon, J., Samira, Z., Osundare, O. S., & Ekpobimi, H. O. (2024). Developing advanced CI/CD pipeline models for Java and Python applications. *Computer Science & IT Research Journal.*

[19] Dachepally, R. (2021). CI/CD Pipelines: Best Practices for Modern Enterprises. *International Journal of Scientific Research in Engineering and Management.*

[20] Samira, Z., Weldegeorgise, G. T., Okeke, C. N., & Shonubi, K. A. (2024). *CI/CD model for optimizing software deployment in SMEs*.

Received: May 7, 2025
Accepted: May 25, 2025

ABOUT THE AUTHORS

Pero Ranilović was born in Prijedor, Republic of Srpska, Bosnia and Herzegovina. He graduated from Grammar School "Petar Kočić" in Novi Grad. He earned his Bachelor's degree in Programming and Software Engineering from the Faculty of Information Technology at Pan-European University "APEIRON" in Banja Luka, where he also completed his Master's degree. Since 2018, he has been employed as a Software Developer at Lanaco. In addition, he works as a Teaching Assistant at Pan-European University "APEIRON". He is currently pursuing his PhD studies in the field of Information Technology at Pan-European University "APEIRON".

Dražen Marinković was born in 1978. He received his M.Sc. degree in 2015 and Ph.D. degree in 2020, both from the Faculty of Informatics at the Pan-European University Apeiron in Banja Luka, specializing in computer and informatics engineering. He is currently an associate professor at the Faculty of Informatics, Pan-European University Apeiron. His research interests include data science, computer networks, and related fields in modern computing technologies.

FOR CITATION

Pero Ranilović, Dražen Marinković, Energy Efficiency as a New Paradigm in Software Engineering, *JITA – Journal of Information Technology and Applications, Banja Luka*, Pan-Europien University APEIRON, Banja Luka, Republika Srpska, Bosna i Hercegovina, JITA 15(2025)1:45-54, (UDC: 004.41:004.738.5), (DOI: 10.7251/JIT2501045R), Volume 15, Number 1, Banja Luka, June (1-80), ISSN 2232-9625 (print), ISSN 2233-0194 (online), UDC 004