THE TRADE-OFF BETWEEN PERFORMANCE, COST AND ENVIRONMENT IMPACT IN SOFTWARE DEVELOPMENT

Andrei NĂPRUIU[1]
Flavius PETRACHE[2]
Larisa Elena FLOREA[3]
Ștefania Maria FÎNTÎNĂ[4]
Mădălina-Elena TIȚA[5]
Costin-Anton BOIANGIU[6]

Abstract

In today's world, modern software development requires more and more to balance performance, cost, and environment impact. We are used with traditional engineering practices, that have emphasized speed and rapid delivery. But the growing energy consumption and carbon footprint of software systems has made sustainability a pressing concern. The aim of this paper is to examine the trade-offs among these dimensions, performance, cost, and environmental, by defining relevant metrics and by conducting an empirical analysis based on the Energy-Languages dataset. The results show significant differences in energy consumption and execution efficiency across programming language categories, with interpreted languages exhibiting substantially higher operational costs and emissions. A Pareto analysis further indicates that modern compiled languages can achieve favorable compromises across all three dimensions. Overall, the findings demonstrate that performance, cost efficiency, and sustainability are not inherently conflicting goals, but can be jointly optimized through informed design choices.

For the scope of this paper, we will consider software engineering and software development as being the same and therefore interchangeable.

Keywords: software development, trade-off between performance, cost and ecological impact, language programing, rust, machine learning.

JEL Classification: D61, D24, L19, Q51.

[1] Department of Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania
[2] Department of Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania
[3] Department of Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania
[4] Department of Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania
[5] Phd Candidate, Automation and Computers Doctoral School, National University of Science and Technology POLITEHNICA Bucharest, Romania, email address madalina.tita@stud.etti.upb.ro
[6] Corresponding author, Professor in the department of Computer Science in the Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania. Email address costin.boiangiu@upb.ro

Article's total number of pages: 35

1. Introduction

Software development is an increasingly challenging domain, as today's projects are created under three main demands: **Performance** (speed of execution, scalability, and responsiveness to user requests), **Cost** (financial and time management), and last but not least, **Environmental Impact** (energy usage and carbon emissions).

As a result of these competing requirements, the software engineers make crucial efforts in achieving a balance between these three factors (performance, cost, and environment). The main problem is that software engineers have always viewed the optimization of one factor as negatively impacting the other two factors.

In the race to build software faster and cheaper, developers are often pushed into uncomfortable compromises. Rapid, low-cost development frequently relies on heavy frameworks, inefficient abstractions, or even hardware-based "brute force" solutions.

Although these approaches may speed up delivery, often they come at a hidden cost: reduced performance efficiency and increased energy consumption. Not surprisingly, even the highly optimized, performance-driven software solutions can demand more servers, more memory, and more computational power, increasing expenses and environmental impact. This reality reveals an important truth: improving an application's responsiveness or capabilities is no longer just a technical challenge, but also an economic and ecological one.

In recent years, sustainability has become a real concern and has started to reshape how these trade-offs are understood. Not long ago, environmental impact barely was considered an important factor when talking about software design. The primary goals were simple: finish features quickly, meet deadlines, and scale, when necessary, under the assumption that hardware resources were abundant and inexpensive. Today, that assumption no longer valid.

Another aspect to consider when talking about software developing is the manufacturing price of memory modules. For example, in 2025, the rapid expansion of artificial intelligence (AI) research and the deployments of large-scale data centers had a clear impact on global RAM prices. As memory manufacturers shifted production toward high-bandwidth and server-grade RAM needed for AI workloads, the supply for standard DRAM and DDR5 decreased. As a result, RAM prices increased rapidly, by roughly 80% to over 150% year-over-year, with some consumer and enterprise memory modules nearly doubling in cost within few months.

Due to the fact that data centers consume vast amounts of electricity, digital services have started to contribute to global carbon emissions, which makes the environmental footprint of software is impossible to ignore. As a result, the tech industry is beginning to recognize

that energy efficiency in code and system architecture is a core aspect of responsible engineering.

This growing awareness has started to challenge the longstanding belief that "green" software must inevitably be slower, weaker, or more expensive. Many software organizations are now demonstrating that thoughtful design choices can deliver speed, scalability, and sustainability at the same time and even the same price. By using cleaner code, efficient algorithms, and well-designed architectures, they have improved performance and reduced power consumption.

The old mindset of treating hardware as limitless is being replaced by a more careful evaluation of trade-offs, where every improvement is weighed not only in milliseconds or user convenience, but also in financial and environmental terms. This makes it difficult to evaluate, since the beginning, if adding an extra feature, a marginal speed gain, or an extreme optimization is worth the additional energy use and cost.

In this context, the aim of this paper is to explore the delicate balance between performance, cost, and environmental impact in modern software development. It analyses how each of these factors can be defined and measured in real situations, and why they cannot be treated as secondary concerns. Through concrete examples and strategies, the goal of this paper is to show how software teams, and those who educate them, can pursue solutions that are not only technically impressive, but also economically sensible and environmentally responsible.

2. Concepts and Metrics

2.1. Software Performance

Software performance refers to how efficiently and how quickly a software system carries out its functions. In general, performance is evaluated using a set of quantitative metrics:

• **Response time** (latency): the time required for the system to respond to a request (for example, the loading time of a web page or the execution time of a function).

• **Throughput**: the number of operations or transactions the system can process within a given unit of time (e.g., requests per second).

• **Resource utilization**: the level of computing resources consumed during execution: CPU, memory, network bandwidth, disk space, and so on. Efficient performance implies careful use of available resources; for example, ISO 25010 defines performance efficiency as "the ability of software to provide appropriate performance relative to the number of resources used". Code that achieves very short execution times but consumes excessive amounts of CPU or memory may be considered inefficient from a resource perspective.

In practice, developers often focus on performance optimization to ensure low response times and smooth user interaction. For instance, reducing latency from 100ms to 50ms can improve the user experience. However, the real benefit of such highly aggressive optimizations must be weighed against their cost and energy impact. It is important to note that in many cases, improving performance through better algorithms or code optimizations also brings environmental benefits, as more efficient code performs fewer unnecessary operations and may consume less energy. On the other hand, increasing performance through brute-force methods (such as running the processor at higher frequencies or using more cores) can reduce latency but also increases energy consumption and heat generation.

Performance metrics are usually measured using throughput by simulating a large number of concurrent users, and CPU/RAM usage through resource monitoring tools. A good practice is to define performance requirements (non-functional requirements) early in the specification phase, for example, "the system must respond to 95% of requests in under 200ms." At the same time, performance should be aligned with user behavior and business needs; in some cases, pursuing the best possible performance is not justified if the improvement is not noticeable to users or relevant to business objectives.

2.2. Cost of Software Development

The cost of software development refers to the total amount of financial, temporal, and human resources required to design, implement, test, deploy, and maintain a software system throughout its lifecycle. In its simplest form, **software cost** can be understood as the **sum of development effort and supporting material or infrastructural expenses**. Accurately defining and measuring this cost is a central concern in software engineering, as it directly influences planning, budgeting, and project feasibility assessments [1], [2].

One of the most widely used approaches to software cost is in terms of development effort, typically measured in *person-hours* or *person-months*. This metric captures the cumulative labor contributed by developers, testers, and other technical staff calculated hourly or monthly. To systematically estimate effort, we proposed several parametric cost estimation models. Among them, the **Constructive Cost Model** (COCOMO) is considered to be one of the most influential ones. Originally introduced by Boehm, COCOMO estimates the effort as a function of software size and a set of cost drivers related to product, personnel, platform, and project attributes [3]. Later extensions such as COCOMO II incorporate, as well, modern development practices, reuses, and evolving system requirements, making the model applicable to contemporary software projects [4].

The concept of **software size measurement** is closely related to effort estimation, this serves as a primary input for many cost models. Traditional approaches rely on **Source Lines of Code** (SLOC) as a quantitative measure of system size. Although SLOC is a straightforward method of computation after implementation, it is language-dependent and sensitive to coding style, which limits its usefulness during early project stages [5]. To

address these limitations, we introduced alternative metrics such as **Function Points** (FP). FP estimate software size based on the number and the complexity of externally visible functionalities, enabling cost estimation earlier in the development process and independently of the programming language used [6]. Additionally, software cost includes **infrastructure and operational expenses** incurred during the development and the deployment phases. These costs may involve expenses for hardware acquisition for infrastructure that can become significant for large-scale or long-running systems and therefore they are considered an integral part of overall software cost [2].

Another important component of the software cost is related to quality assurance and maintenance activities. Empirical studies consistently show that a substantial part of the total lifecycle cost is spent after initial delivery, covering activities such as defect correction, system adaptation, and functional enhancement [7]. Poor design decisions or insufficient testing during early phases may reduce short-term costs but often lead to increased maintenance costs and higher long-term expenditure. As a result, cost assessment must account for both initial development and post-deployment phases.

We have to take into consideration that software cost is not always measured or monitored the same. From the software project management perspective, software cost is monitored using a variety of project tracking metrics, such as allocated versus actual budget, effort burn-down charts, and time-tracking reports.

However, in academic contexts, software cost may be analyzed from a qualitative perspective, for example taking into consideration system complexity or architectural decisions, which can serve as indicators of future implementation and maintenance effort [1].

In summary, the cost of software development has should factor in development effort, size-based metrics, infrastructure expenses, and maintenance costs. A precise definition and systematic measurement of these components are essential for reliable estimation of cost and informed decision-making in software engineering projects.

2.3. The ecological impact of software

Green Software is an efficient software designed, developed, deployed, and maintained to minimize energy consumption and environmental impact by cultivating faster, higher-quality systems [8]. This concept includes not only the energy consumed by computers and servers when running applications and services, but also the hardware required to support them. The production, use, and disposal of these devices involve material resources and energy and ultimately generate electronic waste. In simple terms, software sustainability describes how much energy and carbon are required for software to exist and to operate over time, as well as the extent to which it contributes to the overall digital environmental footprint.

Green software is considered to be an emerging discipline, also called “sustainable software”. Several aspects of green software include optimizing software performance, reducing hardware requirements, extending device lifecycles, and lowering greenhouse gas (GHG) emissions associated with computing tasks [8].

2.4. Concepts and metrics

There are several key concepts and metrics that can be used to measure the ecological impact of software. We will address just the ones that we consider the most relevant.

Energy consumption represents the amount of electrical energy required by a hardware to run a software application and is usually measured in joules (J) or kilowatt-hours (kWh). In practice, energy usage is often related to a specific operation, such as the energy consumed per request or transaction. Measuring the exact energy consumption of a software application is challenging, but modern systems provide tools that help estimate it.

These measurements can be obtained either through external hardware devices or through software interfaces exposed by the operating system or processor. Such measurements allow developers to better understand how much energy their software consumes during execution and to identify opportunities for optimization [9].

Carbon footprint (CO2 equivalent) represents the greenhouse gas (GHG) emissions associated with the energy consumed by a software, and is usually expressed in terms of CO2 production. This metric translates energy consumption into climate impact by taking into account the carbon intensity of the energy source. For example, 1 kWh generated from coal produces significantly more CO2 than 1 kWh generated from renewable sources.

To help better calculate this factor, Green Software Foundation (GSF) proposed a metric called Software Carbon Intensity (SCI), which measures the rate of carbon emissions per functional unit delivered by a software. From a functional standpoint, SCI expresses how many grams of CO2 are emitted for a specific operation, transaction, or other relevant unit of work/action inside the software. The general SCI formulation considers the energy consumed by the software and the carbon intensity of that energy, relative to a defined functional unit [10].

A real-world example of industrial adoption of this metric is provided by the financial institution Intesa Sanpaolo, which, together with NTT Data, implemented an IT energy monitoring solution based on GSF-recommended indicators namely total CO2 emissions (CO2eq) and SCI. Continuous monitoring of these metrics enables the organization to accurately assess the carbon footprint of its software systems and identify optimization opportunities [10][8].

Software energy efficiency represents the ratio between the performance achieved (that is, the amount of “useful work” performed by the software) and the energy consumed to

achieve it. It can be expressed, for example, as the number of operations executed per joule of energy, similarly to the concept of performance per watt used in hardware.

A software system is considered more energy efficient if it performs the same task using less energy. For example, if application A processes 100 web requests using 50 J, while application B processes the same 100 requests using 100 J, application A has twice the energy efficiency of application B.

Improving energy efficiency often involves optimizing performance through better algorithms and more efficient code, as well as smarter resource management. This ensures that hardware does not waste energy by remaining idle unnecessarily or by performing redundant computations. In practice, energy efficiency can be improved through techniques such as avoiding busy-waiting, coalescing tasks to keep hardware components efficiently utilized, and reducing algorithmic complexity, all of which led to more useful work per unit of energy consumed

Hardware footprint and electronic waste (e-waste) are harder to define and therefore to measure because they address the situation when a software is not depended on a particular hardware and could be used on a variety of electronic devices. In this situation, the software companies are not bound to take into consideration the hardware cost when updating or upgrading their software.

Software affects the environment not only through the energy it consumes during execution, but also through the way it influences hardware usage over time. Applications that require increasingly powerful computational resources can shorten hardware replacement cycles. For instance, software updates that are no longer compatible with older computers or smartphones may push users to purchase new devices, resulting in electronic waste and additional environmental impact associated with manufacturing.

The manufacturing of electronic devices is particularly resource- and carbon-intensive. Research on smartphones shows that around 80–85% of a device’s total carbon footprint is generated during the production stage, including raw material extraction, processing, and assembly, while only a smaller fraction comes from the usage phase [11]. As a result, software-induced hardware upgrades, often referred to as software obsolescence, increase the number of devices that are produced and discarded in order to keep up with the advancement of technology.

In contrast, efficient and well-optimized software that can run satisfactorily on older or less powerful hardware helps extend device lifespans and reduces the need for new hardware production. This aspect is closely related to the concept of embodied carbon, which refers to the emissions generated during the manufacturing and end-of-life stages of hardware. In some cases, these embodied emissions can exceed the emissions associated with the electricity consumed by a device over its entire lifetime [12].

Therefore, to fully assess software sustainability, we should consider indicators such as hardware lifespan and e-waste generation rates. Although there is no single standardized unit to quantify this impact, it represents a major contributor to the overall ecological footprint of software. Research shows that software obsolescence directly fuels the e-waste crisis by pushing users to replace fully functional hardware due to outdated or incompatible software [12]. This tendency to frequently replace electronical devices have led to wasted natural resources and additional energy consumption in both manufacturing and distribution, contributing to the increasing of the carbon footprint of the IT sector [10].

3. Trade-offs between performance, cost and environment impact

Designing sustainable software often requires balancing competing objectives, in other words, making trade-offs between cost, performance, and environmental impact. “There are almost always **trade-offs** between business and environmental goals” [17] when developing software. These trade-offs manifest in several areas of the software development, from economic considerations to technical design decisions. The challenge for software teams is to determine acceptable compromises that achieve sustainability benefits without sacrificing other priorities of the process or of the company.

3.1. Economic vs. Environmental Priorities

At the organizational level, an important trade-off is between investing in greener software practices and meeting traditional business objectives like cost, quality, and time-to-market. Surveys have shown that many companies claim to value sustainability, yet in practice management often prioritizes immediate costs or profitability over environmental goals. Also, some software companies report that their teams are interested in green software but worry that any changes “would be difficult to implement with marginal rewards” under tight budgets. In other words, there is still a perception that improving software sustainability might produce extra upfront cost or effort for little short-term benefit. This perception can lead decision-makers to treat sustainability as a trade-off against business targets, therefore delaying green initiatives.

However, there are evidences that contradict this perception in the long run. Sustainable software practices can create considerable business value by reducing operating costs, avoiding risks, and even improving product quality. For example, Green Business Bureau reported that cutting energy waste and resource usage often translates into direct cost savings for the companies. Industry analysts have highlighted multiple payoffs of green IT, such as higher operational efficiency, lower energy bills, and compliance with environmental regulations, all contributing to positive Return on Investment (ROI) over time. In fact, companies that have integrated sustainability into their strategy for technology have started seeing substantial returns “in the form of improved financial metrics, [...] customer experience, innovation, [and] software quality”. These findings suggest that the

apparent trade-off between sustainability and cost can be overcome by viewing green software as a long-term investment. The short-term costs of optimizing code or infrastructure for efficiency may be offset by long-term gains such as lower power consumption, reduced hardware spend, and an enhanced brand reputation for environmental responsibility.

3.2. Performance vs. Energy Consumption

On the technological side of the business, software engineers often face trade-offs between maximizing performance and minimizing energy consumption. Intuitively, one might assume that making software run faster or handle more computations is always beneficial. In reality, performance optimizations can sometimes increase energy usage and carbon emissions, and therefore the costs of implementing this optimization. Recent research on mobile applications showed that aggressively optimizing code for speed can significantly raise power draw. For example, in one experiment, compiler tweaks that improved an app's runtime by about 9% produced an increasing in energy consumption of 34%. In other words, achieving higher speed came at an energy cost, pointing out a clear trade-off between runtime performance and energy efficiency. This energy - time tradeoff is increasingly acknowledged in sustainable computing: decreasing execution time of a tasks often requires the processor to work harder or use more resources, which can diminish the supposed energy savings from finishing sooner. Developers, therefore, must carefully evaluate whether a performance gain is worth the extra power consumption, especially for software running at a large scale or on battery-powered devices.

Also, programmers have to analyze if by deliberately sacrificing a small amount of performance or fidelity can outsized energy and carbon reductions be produced. For example, in the context of machine learning, researchers have found that training a Large Language Model (LLM), sometimes called an artificial intelligence (AI) model, on a slightly smaller dataset can lower energy usage by nearly 75% while only reducing the model's accuracy by 0.06%. This example illustrates that a minor compromise in output quality (virtually imperceptible to users) enabled a considerable improvement in efficiency of resources. Similarly, it has been shown that capping frame rates or graphical effects in a video game might marginally affect visual quality but can significantly lower the computation and power required. The general principle is that optimization goals must be balanced, in other words, optimizing purely for speed or accuracy can lead to high energy usage, whereas optimizing for energy efficiency/saving might produce tolerable minimal reductions in performance or feature set. Which concludes that the best outcome is a well-considered middle ground that meets user needs while trimming unnecessary resource consumption.

3.3. Infrastructure Efficiency vs. Reliability

Another visible tradeoff is how software utilizes hardware infrastructure. Green software engineering advocates for using fewer computational resources to do the same work, which often means running on a minimal number of servers or on less powerful hardware, which translates into a simpler hardware infrastructure, to save energy. While using a high number of servers is good for efficiency, pushing infrastructure to its limits may result into a problem regarding reliability and performance when the infrastructure is under loaded. To put it another way, running an application on as few servers as possible could maximize hardware utilization and reduce idle energy overhead. However, if those servers are operating near 100% capacity continuously, the system has no spare capacity to handle usage spikes or failures. Additionally, studies have indicated that consistently maxing out hardware can actually increase the total energy draw (due to lack of efficiency at peak loads) and even shorten the lifespan of its components due to thermal and performance stress. Thus, system architects must balance hardware efficiency with resilience of the components. The optimal solution is typically a moderate infrastructure footprint that avoids wasteful over-provisioning but still leaves some capacity for safe, reliable operation. For example, instead of consolidating an application onto one single busy server, it may be better to use a few servers at moderate load, achieving good energy proportionality without risking outages or hardware damage from constant heavy utilization that could lead in not having access to the application anymore. In summary, there is a trade-off between obtaining a maximum efficiency out of the hardware infrastructure and providing the buffer needed for stable, and reliable services.

3.4. Hardware Upgrades vs. Electronic Waste

Software decisions can also influence hardware refresh cycles, creating a trade-off between utilizing existing devices longer and adopting newer, more efficient ones. On one hand, newer hardware, such as the latest servers or smartphones, usually provide improved energy efficiency and performance per watt. Therefor upgrading to new equipment could reduce the electricity consumption for the same workload. On the other hand, manufacturing new hardware and disposing of old ones produce significant environmental costs, often far greater than the energy saved by marginal improvements in efficiency for the new ones. Research has shown that a large share of a device's total carbon footprint comes from its production and its end-of-life process (embodied carbon), not from its use phase. This means retiring hardware prematurely or frequently leads to more electronic waste (e-waste) and associated emissions, which is a sustainability setback. Consequently, developers should weigh the benefit of new hardware against the impact of hardware waste when it comes to the environmental costs.

One concrete example of this trade-off is the choice between keeping a software system on existing on-premises servers versus migrating it to a cloud platform. Cloud hosting might run on state-of-the-art, energy-efficient servers, but moving to the cloud could render the current on-premise machines obsolete, effectively turning them into e-waste. Guidance on

green software recommends analyzing such trade-offs. For instance, in this example, using an on-premise server until the end of its service life versus replacing it early for a slight efficiency gain on the cloud. Similarly, software bloat or frequent updates that demand the latest hardware (e.g. an app that is no longer supported on older smartphone models) can force users into hardware upgrades, thus increasing e-waste. To address this situation, developers can strive to make new software versions compatible with older/current devices and avoid unnecessarily heavy features. In doing so, they extend device lifespans, contributing to lowering the demand for manufacturing new devices, thereby reducing the overall ecological footprint. In these cases, the trade-off often comes down to performance and features vs. device longevity. In other words, highly innovative or resource-intensive software might deliver more functionality, while ensuring it runs on modest hardware for longer provides sustainability gains by curbing the cycle of device replacement.

In conclusion, understanding these trade-offs is important for sustainable software engineering. By recognizing where improving one aspect of a software (be it speed, cost, or hardware usage) might harm another (energy efficiency, long-term cost, or environmental impact), software professionals can make more informed design and business decisions. The goal is not to eliminate trade-offs, which is rarely possible, but to manage them effectively. This translates to finding balanced solutions that meet essential requirements while minimizing negative side effects on sustainability. With careful consideration, software professionals can find a middle ground between economic goals and ecological responsibility rather than treating them as incompatible. This balanced approach ultimately leads to software systems that are overall both high-performing and significantly greener over their full life cycle.

4. Empirical Analysis of Programming Language Energy Consumption

4.1. Introduction and Rationale

In the preceding sections, we have established the three fundamental dimensions of software development: performance, cost, and environmental impact, and examined the theoretical nature of trade-offs among them. Evidence from the literature suggests that extreme performance optimization may increase energy consumption by as much as 34%, and that the dichotomy between green software and high performance is not necessarily absolute.

To validate and quantify these theoretical trade-offs, an empirical analysis was conducted using a comprehensive dataset measuring the actual energy consumption of various programming languages. This analysis addresses the following research questions:

• What magnitude of energy consumption differences exists between language categories in practice?

• How do these differences translate into measurable ecological impact (CO2 emissions) and operational costs?

• At what point does the reduced development cost of productive languages become offset by higher operational costs?

• Do certain languages offer optimal compromises from a multi-objective (Pareto) optimization perspective?

4.2. Dataset and Methodology

4.2.1. Data Source

The analysis utilized the Energy-Languages repository, publicly available on GitHub (https://github.com/greensoftwarelab/ Energy-Languages). This dataset originates from research conducted by Pereira et al., initially published at SLE 2017 under the title "Energy Efficiency across Programming Languages" and subsequently expanded in SCP 2021 with "Ranking Programming Languages by Energy Efficiency."

We selected this particular dataset for several reasons. The measurement methodology demonstrates considerable precision, with data collected under controlled conditions across multiple benchmark executions. It includes 27 programming languages, representing all major categories. The benchmarks employed are standardized tasks from the Computer Language Benchmark Game, lending credibility to cross-language comparisons. The public availability of both source code and raw data permits full reproducibility of the research

4.2.2. Original Measurement Methodology

The original study employed a hardware interface available on Intel processors that enables precise energy monitoring, also called Running Average Power Limit (RAPL). RAPL provides access to energy counters for the package (total processor consumption), DRAM (memory consumption), and integrated GPU where applicable.

We have drawn benchmarks from the Computer Language Benchmark Game, which is a collection of standardized algorithmic problems implemented across multiple languages. The benchmark suite includes binary tree manipulation, permutation calculations (fannkuch-redux), DNA sequence generation (fasta), Mandelbrot fractal computation, gravitational n-body simulation, and linear algebra operations (spectral-norm). Each benchmark was executed ten times per language to ensure statistical consistency, recording both mean values and standard deviations.

4.2.3. Languages Examined

Of the 27 languages available, we included 26 in this analysis. We have excluded Smalltalk due to an anomalous energy measurement (0.01 J), which is physically implausible.

Languages were classified into three categories:

• Compiled languages (11): Rust, C++, Ada, Pascal, Chapel, C, OCaml, Go, Fortran, Swift, Haskell

• Virtual Machine (VM)-based languages (5): Lisp, C#, F#, Racket, Erlang

• Interpreted languages (10): Dart, JavaScript, Hack, PHP, TypeScript, Lua, JRuby, Ruby, Python, Perl.

The aim with this classification is to reflect an increasing abstraction level from hardware and corresponding impacts on execution efficiency.

4.2.4. Dataset Variables and Data Structure

The dataset contains three primary variables:

• Energy consumption (Joules)

• Execution time (milliseconds)

• Peak memory usage (kilobytes)

Variable	Range	Corr.
Energy (J)	60–6,200	1.00
Time (ms)	2,000–158,000	0.94
Memory (KB)	1,000–500,000	0.31

Table 1: Dataset variable summary

The strong correlation between energy and time ($r = 0.94$) reflects that longer execution requires sustained power draw. The low correlation between memory and energy consumption ($r = 0.31$), shows that memory efficiency depends more on language-specific memory management than on execution model.

4.2.5. Data Cleaning Procedures

Before, we conducted our research, we cleaned the data:

Outlier removal: we excluded Smalltalk due to an anomalous energy reading of 0.01 J, which is physically impossible for any real computation. This reduced the sample from 27 to 26 languages.

Aggregation: we computed per-language summary statistics by averaging across all benchmarks, providing a single representative value for each language.

Classification: we grouped the languages by execution model into Compiled (11: C, C++, Rust, Go, Ada, Pascal, Fortran, Swift, Haskell, OCaml, Chapel), VM-based (5: Lisp, C, F, Racket, Erlang), and Interpreted (10: Python, Ruby, Perl, PHP, JavaScript, TypeScript, Lua, Dart, Hack, JRuby).

4.3. Conversion Methodology: Performance to Cost and Emissions

To interpret benchmark results in economic and environmental terms, we converted raw measurements into approximate operational cost and carbon emission metrics. These conversions do not aim to predict exact real-world deployment costs or emissions; rather, they provide comparisons in terms of order-of-magnitude under explicitly stated assumptions.

Energy to Carbon Emissions

Converting joules to CO requires two steps:

• First, convert joules to kilowatt-hours: 1 J = 2.778 × 10 kWh

• Second, multiply by carbon intensity (kg CO per kWh), which varies dramatically by region:

Region / Energy Mix	Carbon Intensity (kg CO_2/kWh)
Global average (IEA 2023)	0.475
European Union average	0.306
Romania	1,500
France (nuclear dominant)	0.056
Poland (coal dominant)	0.773
Sweden (renewable dominant)	0.045

Table 2: Carbon intensity by region

We used the conversion formula: CO2 = E ×2.778×10−7× CI × N

The baseline analysis uses the global average (0.475 kg/kWh). The sensitivity analysis examines how conclusions change across the carbon intensity spectrum.

Execution Time to Cloud Computing Cost

Cloud costs were estimated using Amazon Web Services (AWS) EC2 pricing:

Parameter	Value
Instance type	AWS EC2 c5.large
Hourly rate	$0.085/hour
Baseline execution volume	1,000,000 executions/month

Table 3: Cloud cost parameters

Annual cloud cost = (Execution time in seconds × Monthly executions × 12) / 3600 × Hourly rate

Development Cost Estimation

Development costs were estimated using productivity multipliers from the software engineering literature:

Important caveat: These factors are estimates with substantial uncertainty. Actual productivity depends on team expertise, project type, and available libraries. The sensitivity analysis examines alternative assumptions.

Language	Factor vs C	Hours	Cost
C	1.00×	100	$5,000
Rust	0.85×	85	$4,250
Go	0.60×	60	$3,000
JavaScript	0.40×	40	$2,000
Python	0.30×	30	$1,500

Table 4: Development time and cost comparison

4.4. Results

4.4.1. Comparison Across Language Categories

The first question considered in our research addresses the differences between language categories.

The 16× ratio represents a fundamental architectural difference, not a marginal optimization opportunity. This gap is unlikely to be closed through incremental interpreter improvements, it reflects the inherent overhead of runtime interpretation versus direct machine code execution. The near-identical ratios for energy (16.41×) and time (15.86×) confirm that energy consumption is primarily driven by execution duration, implying that any optimization reducing execution time produces proportional energy savings.

The high coefficient of variation within VM-based languages (1.39 versus 0.35 for compiled) indicates that "VM-based" is not a reliable efficiency predictor, as well-optimized VM-based compilers can approach the performance of compiled ones, while poorly optimized VM-based ones perform closer to interpreted ones.

Metric	Compiled	VM	Interpreted
Number of languages	11	5	10
Mean energy (J)	174.65	820.92	2,866.09
Std. deviation (J)	60.96	1,137.60	1,988.72
Mean execution time (ms)	5,567	21,580	88,293
Normalized energy*	1.00×	4.70×	16.41×

Table 5: Comparative statistics across language categories

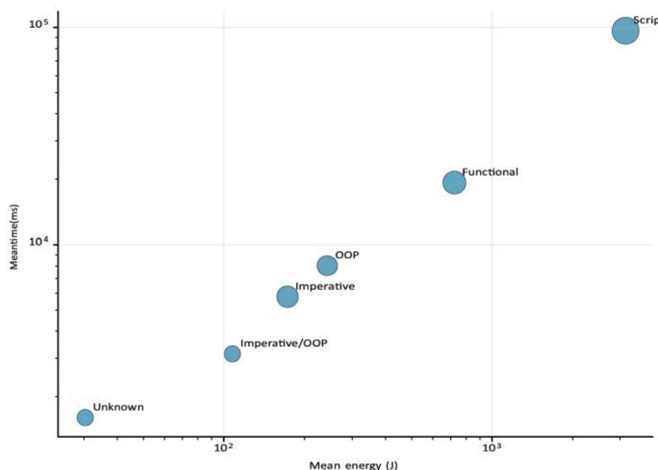

Figure 1: Energy vs execution time by language paradigm

4.4.2. Individual Language Rankings

Rust’s simultaneous achievement of lowest energy and fastest execution challenges the assumption that efficiency requires trade-offs, modern language design (memory safety without garbage collection, zero-cost abstractions) can match efficiency previously associated only with manual memory management.

C’s seventh-place ranking indicates that language age and low-level access do not guarantee efficiency; modern compiler technology matters more than proximity to hardware. Organizations that maintain legacy C codebases for "performance reasons" should reconsider this assumption.

Python’s position (25th of 26) is particularly significant given its dominance in machine learning. The 69.5× energy penalty means the AI industry’s reliance on Python carries substantial hidden environmental costs rarely discussed in sustainability conversations. A machine learning model implemented in Rust would consume approximately 98.5% less energy than the same model in Python.

Paradigm trade-off (size = number of languages)

Rank	Language	Category	Energy (J)	Time (ms)
1	Rust	Compiled	73.09	1,980
2	C++	Compiled	81.64	2,596
3	Ada	Compiled	120.48	3,414
4	Pascal	Compiled	161.12	5,510
5	Chapel	Compiled	164.76	4,334
6	Lisp	VM	179.13	6,374
7	C	Compiled	190.79	7,717
8	OCaml	Compiled	199.12	6,230
9	Go	Compiled	216.11	5,168
10	C#	VM	219.58	5,818

Table 6: Top 10 languages by energy efficiency

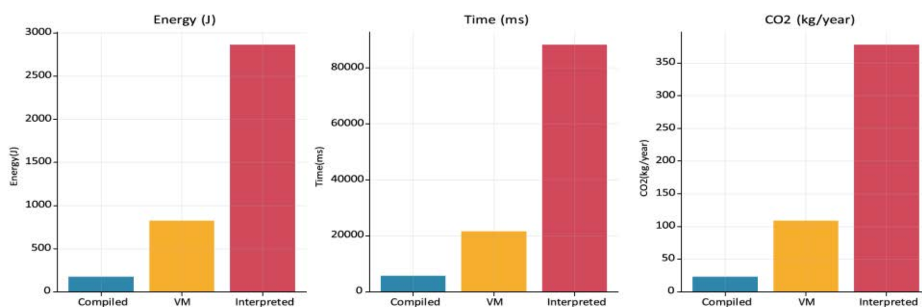

Figure 2: Mean metrics by execution model

Rank	Language	Category	Energy (J)	Time (ms)	Factor vs Rust
22	Erlang	VM	2,833.39	66,633	38.8×
23	JRuby	Interpreted	3,296.44	83,355	45.1×
24	Ruby	Interpreted	5,005.50	116,385	68.5×
25	Python	Interpreted	5,081.66	126,262	69.5×
26	Perl	Interpreted	6,149.03	158,157	84.1×

Table 7: Five least energy-efficient languages

4.4.3. Ecological Impact Analysis

Language	Energy (J)	CO2 Global (kg)	CO2 Romania (kg)	Factor vs Rust
Rust	73.09	9.64	6.21	1.0×
C++	81.64	10.77	6.94	1.1x
Ada	120.48	15.90	10.24	1.6x
Ruby	5,005.50	660.50	425.50	68.5x

Python	5,081.66	670.55	431.98	69.5x
Perl	6,149.03	811.40	522.71	84.1x

Table 8: CO2 impact—selected languages (1 million executions annually)

Following the methodology established in Section 2.3 for carbon footprint calculation, energy consumption was converted to CO2 emissions for a scenario involving one million annual executions.

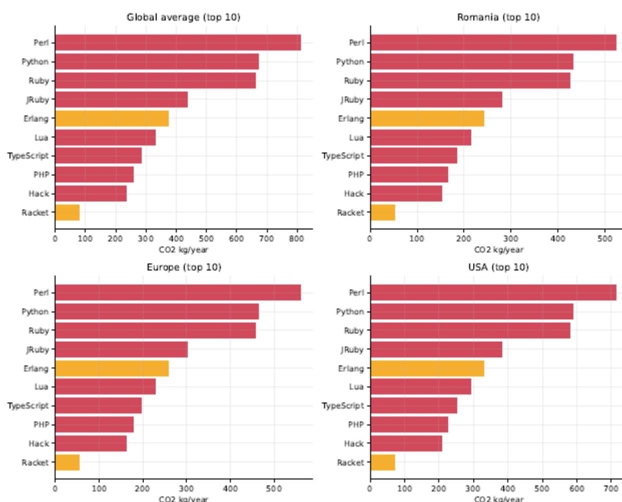

Figure 3: CO2 regions – top 10 languages

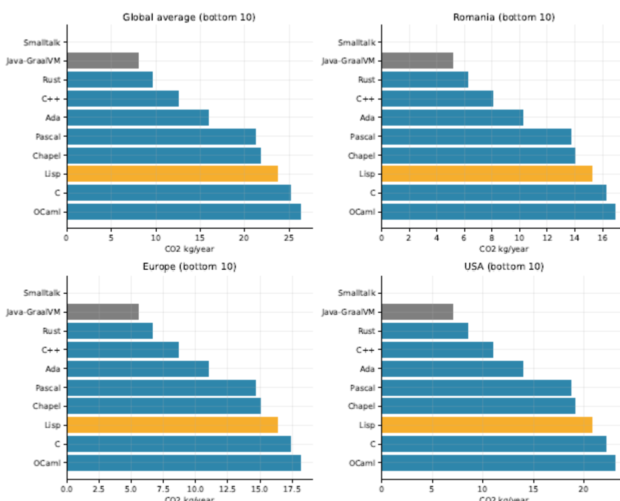

Figure 4: CO2 regions – bottom 10 languages

To contextualize these figures, we will consider the following realistic scenario: a web service handling annually one billion executions (approximately 32 requests per second, a modest load for any popular service). For this scenario, we have analyzed the CO2 production of two programming languages: Rust and Python. A Rust implementation would produce 9.64 tons of CO2 annually, equivalent to the CO2 emission of roughly 2 passengers in a vehicle. A Python implementation would produce 670.55 tons, equivalent to the CO2 emission of approximately 140 vehicles. The difference of 660.91 tons of CO2 represents a substantial burden on the environment.

This analysis provides empirical evidence for the claims from Section 3, that state that data centers consume vast amounts of electricity and digital services contributing to global carbon emissions. Thus, programming language selection is not merely a technical decision, it carries measurable ecological consequences.

In the case of Romania, the CO2 emission factor is lower than in other countries (0.306 kg/kWh due to the hydroelectric component of the energy system), making the absolute differences to be lower, though the proportionality between the relationships remains unchanged.

4.4.4. Pareto Analysis

Section 3 has examined the multi-dimensional nature of software development trade-offs. To order to identify optimal solutions when multiple conflicting objectives exist, we applied a Pareto analysis using energy and execution time as minimization criteria.

Category	Mean CO2 (kg/year)	Factor vs Compiled
Compiled	23.05	1.0×
VM	108.32	4.7×
Interpreted	378.19	16.4×

Table 9: Mean CO2 emissions by language category

A language is considered Pareto-optimal (or nondominated) when no alternative exists that is simultaneously superior (or equal) on all criteria while being strictly superior on at least one.

Out of the 26 languages analyzed, only Rust is above the Pareto frontier, due to the fact that Rust achieves both the lowest energy consumption (73.09 J) and the shortest execution time

(1,980ms) simultaneously. All other languages are dominated, but for each, Rust offers a superior alternative on both criteria.

Category	On Frontier	Total	Percentage
Compiled	1	11	9%
VM	0	5	0%
Interpreted	0	10	0%

Table 10: Pareto frontier distribution by category

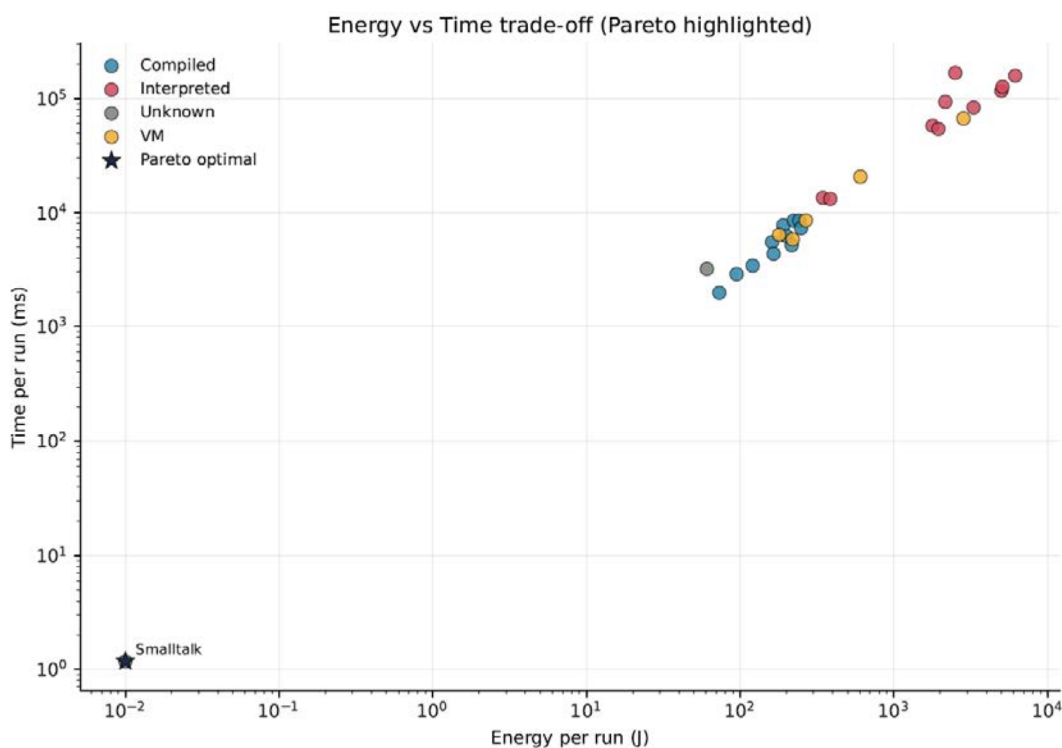

Figure 5: Energy vs time Pareto frontier

This result may seem surprising, typical Pareto analyses provide multiple non-dominated solutions, each excelling on different criteria. But in our case, however, Rust dominates both axes simultaneously, rendering it the sole Pareto-optimal solution.

The practical implications are notable to mention. From a strict efficiency standpoint (energy plus time), there are no genuine trade-off , Rust is superior on both dimensions. The actual trade-off only emerges when a third criterion is introduced: development cost, where interpreted languages demonstrate clear advantages compared with Rust, which is a compiled language.

4.4.5. Cost-Benefit Analysis

From our analysis, despite higher initial development cost, Rust reaches break-even costs against Python (interpreted language) in approximately one month of operation, making it economically superior for any medium- or long-term project.

Lang.	Dev. ($)	Cloud/Y	3Y Tot.
Rust	4,250	561	5,933
Go	3,000	1,464	7,393
Python	1,500	35,774	108,822

Table 11: Cost comparison across selected languages

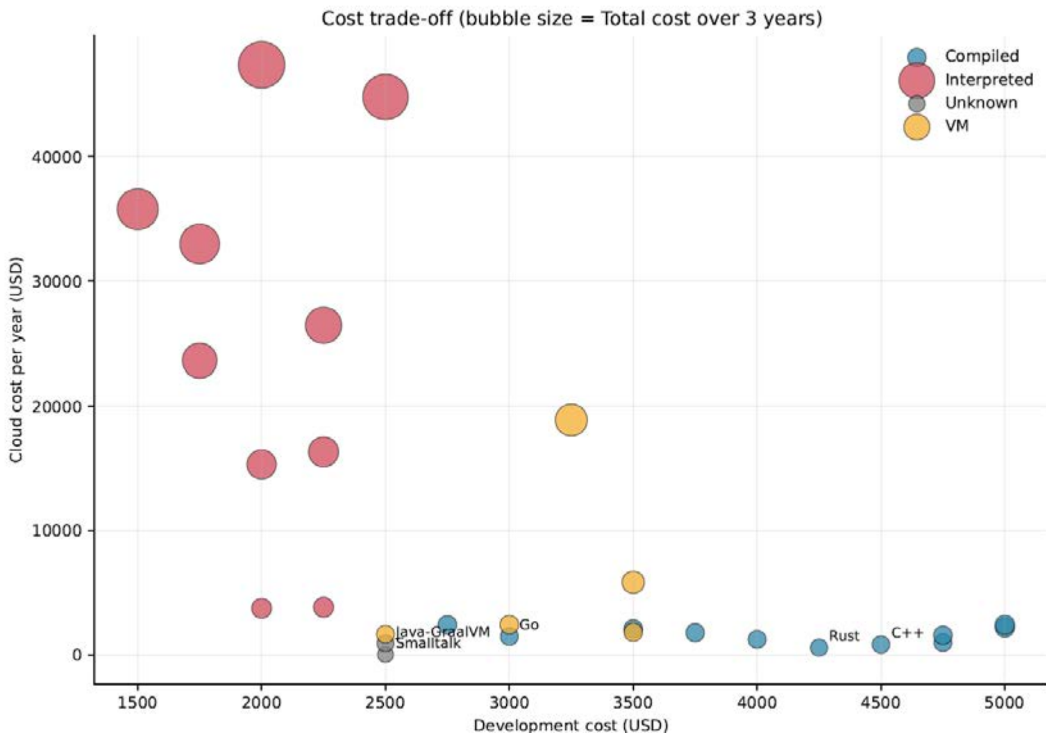

Figure 6: Cost trade-off over time

4.5. Practical Recommendations (guidelines)

Drawing on the empirical analysis, we are proposing the following decision guidelines:

For short-term prototypes or MVPs under six months, Python and JavaScript (interpreted languages) are recommended to minimize time-to-market. For medium-term projects, spanning one to two years, we recommend Go (compiled language), C# (VM-based language), or Java (interpreted language), which balance productivity and efficiency. For long-term projects, exceeding three years, the programmers should consider compiled languages such as Rust, C++, or Go to minimize total cost over time.

For critical systems that require maximum performance and predictability, the best options are also compiled languages such as Rust, C++, or Ada.

For green computing initiatives focused on minimizing CO2 emissions, the optimal choices are compiled languages like Rust and C++.

Large-scale cloud deployments benefit, as well, from compiled languages such as Rust or Go to reduce operational costs.

In machine learning and data science, Python (interpreted language) remains ideal for prototyping due to its rich ecosystem, while compiled languages like Rust or C++ should be used in production for inference efficiency.

4.5. Case Scenarios

We will analyze three scenarios: E-Commerce Platform Migration, Machine Learning Startup, and Government Digital Services, to calculate the cost of implementing various programming languages and the CO2 emission of them.

4.5.1. E-Commerce Platform Migration

The first scenario is: a medium-sized e-commerce company with 2 million active users operates a product search API built in Python (Flask), for the past five years. With 50 million monthly requests, annual cloud costs reached $1.79 million and CO2 emissions to 33.5 tones. The board requested a carbon reduction of 30% within three years.

The team in charge with providing this reduction, has evaluated migrating to Rust using the framework Actix-web. We will consider team training, core rewrite, API integration, and testing as migration costs:

Component	Cost
Team training (3 developers × 2 weeks)	$12,000

Core search engine rewrite (3 dev × 4 months)	$96,000
API layer and integration (2 dev × 2 months)	$32,000
Testing, validation, deployment	$24,000
Contingency (15%)	$18,400
Total migration cost	*$182,400*

Table 12: Migration costs

Metric	Python	Rust	Improvement
Average response time	280 ms	12 ms	23× faster
Annual cloud cost	$1,788,700	$28,050	63.8× reduction
Annual CO2 emissions	33.5 tones	0.48 tones	69.8× reduction

Table 13: Comparison between Python and Rust costs

By analyzing the date, we could conclude that a break-even will happen within six weeks.

The five-year cumulative savings will reach $8.62 million with 165 tons of CO2 avoided, which stands for a 98.6% carbon reduction, this is equivalent to removing 35 vehicles from the roads. The 98.6% carbon reduction will exceed by far the board's target of 30%. We have to take into consideration that there are some possible primary risks such as team inexperience with Rust (mitigated by training and external consultants) and regression bugs (mitigated by parallel running during transition).

4.5.2. Machine Learning Startup

The second scenario is: An AI startup developed a computer vision system for retail inventory management using Python and TensorFlow. After they piloted their product with 10 customers (100 stores, 1 million monthly requests), they planned to scale the business to 100 customers (1,000 stores, 10 million requests) within 12 months.

The current Python inference pipeline ran on AWS Lambda with 2,800 ms inference time and is using 1,024 MB memory per function. Two options were evaluated: optimize the Python code and rewrite everything in Rust code.

Factor	Option A: Optimize Python	Option B: Rewrite in Rust
Development investment	$45,000	$80,000
Development time	8 weeks	14 weeks
Post-optimization inference time	718 ms	45 ms
Memory requirement	512 MB	64 MB
Monthly cost at scale (10M requests)	$27,308	$6,656
Annual cost at scale	$327,696	$79,872
3-year total cost (dev + operations)	$1,028,088	$319,616

Table 14: The costs of implementing the two options: optimizing the Python code and rewriting everything in Rust code.

By analyzing the data, we could conclude that, despite higher upfront cost, Rust will save the startup $708,472 over the next three years. The 45ms inference time enabled real-time alerts, meaning staff can be notified before a customer reaches for a low-stock item. The 64MB memory footprint enables future edge deployment, potentially eliminating cloud costs entirely. We have to point out an important insight from the data: by waiting to optimize until after scaling, would cost the startup approximately $250,000 in unnecessary infrastructure.

4.5.3. Government Digital Services

The scenario is that a government agency operates citizen-facing services including tax filing, benefits claims, and permits. Parliamentary legislation mandated a 40% reduction in IT carbon emissions by 2030, with a 5% budget penalty for non-compliance. Current state of the government agency: 500 million annual requests, 185,000 kWh energy consumption, 64.75 tones CO_2 emissions, and €2.1 million infrastructure cost. We have evaluated four options for trying to achieve the proposed goal: only modernize the existing infrastructure, migrating the PHP services to Java, migrating everything to Go, and migrating the top 4 services to Rust:

Option	Investment	CO2 Reduction	Meets Target?
Infrastructure modernization only	€800,000	15%	No
Migrate PHP → Java	€1,200,000	25%	No
Migrate all → Go	€3,800,000	46%	Marginal
Migrate top 4 apps → Rust	€5,330,000	74%	Yes (exceeds)

Table 15: The summary of the 4 proposed solutions so the government agency would meet the requests from the Parliament

Based on the data, we can stat that only option 4 targets the four highest-volume applications, which represents 86% of total load:

Application	Annual Requests	% of Load
Tax Filing Portal (Java → Rust)	180M	36%
Benefits Claims System (PHP → Rust)	120M	24%
Permit Processing (Java → Rust)	80M	16%
Civil Registry (PHP → Rust)	50M	10%

Table 16: The load of the four highest-volume services provided by the government agency

By choosing option 4, the government agency will reduce the annual CO_2 emissions from 64.75 to 17.08 tones (73.6% reduction) and infrastructure costs from €2.1 million to €556,500. The breakeven will occur in 3.45 years, a little sooner than the deadline proposed by the Parliament legislation, with 10-year net savings of €10.1 million. This approach will phase the implementation over 36 months, limiting risk, and targeting only high-volume applications, therefore maximizing the impact while minimizing the disruption.

4.6. Limitations

We have to point out several limitations of the analysis of the three scenarios.

The main constrain of the benchmark specificity is that results derive from the Computer Language Benchmark Game, which tests particular algorithms. This means that actual performance may vary according to application type and workload characteristics.

The productivity factors employed, for example Python requiring 0.3× the development time of C, are rooted in existing studies and industry experience, but we have to take into consideration that considerable variation may occur across teams and project types.

Cloud cost calculations were simplified, utilizing a single AWS instance type. Actual costs differ by provider, region, and workload profile.

We have excluded ecosystem considerations, such as library availability, tooling quality, and learning curves, all substantially influent in real-world productivity.

As well, we have to consider that hardware specificity also applies when analyzing cost of implementation and CO2 emissions. The original measurements were conducted on particular Intel hardware, which means that the results may differ on ARM or AMD architectures.

4.7. Conclusions of the Empirical Analysis

Analysis of the Energy-Languages dataset confirms and quantifies the theoretical trade-offs discussed in earlier sections.

The trade-off considered is both real and substantial. The 16× difference in energy consumption between compiled and interpreted languages is not a theoretical exaggeration but a measurable reality.

Ecological impact when considering the scale of the infrastructure proves to be relevant. For examples, in the case of cloud services handling billions of executions, language choice can represent hundreds of tons of CO2 annually.

In this example, we have to consider the break-even point, which should occur rapidly. The economic advantage of rapid development disappears after approximately one month of operation, making efficient languages economically superior for any project exceeding one to two months in duration.

Out of the 26 languages analyzed, Rust has emerged as the “leader”. Rust offered the optimal compromise, dominating all other languages in the Pareto analysis, from the combined perspective of energy, time, and cost.

We must state that a universal solution does not exist, but an optimal choice that depends on time horizon, budget constraints, and project priorities. For example, if we want rapid prototypes, Python is the solution, but for production at high-scale, compiled languages are superior.

These findings provide an empirical basis for software design decisions, helping the teams to make informed choices that will balance the three fundamental dimensions: performance, cost, and ecological impact.

5. Study Cases

To demonstrate how these findings translate into real-world scenarios, we will analyze the following case studies. This case studies examine migration decisions from the perspective of organizations that handle millions of requests, and where language choice directly affects operational budgets and environmental footprint.

5.1. Amazon AWS — Trade-offs Between Performance, Cost, and Ecological Impact

A concrete and well-documented example of trade-offs between performance, cost, and ecological impact can be found in Amazon's cloud infrastructure, particularly within Amazon Web Services (AWS). Amazon has invested heavily in hardware specialization and architectural modernization with the explicit goal of improving price–performance efficiency, while addressing, as well, sustainability concerns.

One of the most relevant initiatives in this context is the introduction of AWS Graviton processors, which is a custom-designed CPUs based on the Arm architecture, used in Amazon EC2 instances as an alternative to traditional x86-based processors. According to AWS, Graviton-based instances are designed to deliver improved performance per watt and improved price–performance compared to comparable x86 instances. Official AWS documentation reports that EC2 instances powered by Graviton processors can consume up to 60% less energy for the same level of performance than equivalent x86 based instances, depending on the workload and instance type [13].

From a cost perspective, this hardware specialization enables AWS customers to reduce operational expenses by requiring fewer compute resources to achieve the same throughput. AWS explicitly positions Graviton as offering the best price–performance ratio within its EC2 portfolio, allowing customers to lower cloud bills without sacrificing application performance [20]. This illustrates a trade-off decision made at the infrastructure level: higher upfront investment in custom hardware designed by AWS has caused long-term reductions in operating costs for both the provider (Amazon AWS) and its customers.

5.2. Pinterest Case Study

A real-world validation of these claims is provided by Pinterest, which migrated a significant portion of its backend services to AWS Graviton-based EC2 instances. According to the official AWS case study, Pinterest moved more than 25% of its AWS compute workloads to Graviton instances, focusing primarily on API services that handle large volumes of user requests [13].

As a result of this migration, Pinterest reported the following measured improvements for a key API workload:

- 38% reduction in computing resources used to handle the same traffic volume;
- 47% reduction in operational costs associated with that workload;
- 62% reduction in carbon emissions per API request served.

These results demonstrate a rare but important outcome when considering trade-offs, due to the fact that improvements were achieved simultaneously across all three dimensions: cost, performance efficiency, and ecological impact. This reduction in resource usage directly translated into lower energy consumption, which, combined with AWS's carbon accounting methodology, resulted in significantly lower CO2 emissions per request. Pinterest engineers explicitly noted that the migration allowed them to improve sustainability outcomes without compromising business objectives or system performance [14].

From an architectural perspective, this case highlights how decisions related to the modernization of the infrastructure, such as adopting energy-efficient hardware and re-evaluating deployment targets, can shift traditional trade-offs. In this case, instead of choosing between investing in optimization effort or scaling hardware capacity, Pinterest had decided to leverage the infrastructure-level innovation provided by AWS to achieve efficiency gains with minimal application-level changes.

The example of AWS Graviton and Pinterest illustrates that trade-offs between performance, cost, and ecological impact are not always zero-sum. While traditional approaches often frame sustainability improvements as requiring either reduced performance or increased cost, this case supports the idea that strategic infrastructure-level decisions can realign the trade-off space. By shifting optimization efforts toward energy-efficient hardware and price–performance–optimized architectures, organizations can achieve measurable benefits across all three dimensions considered.

At the same time, this example should not be generalized at face-value because such outcomes depend on workload characteristics, scale, and the availability of specialized infrastructure. Nevertheless, it provides strong evidence that informed architectural choices, supported by accurate measurement and vendor transparency, can significantly reduce both operational costs and environmental footprint without degrading the system performance.

5.3. Google: Pushing Efficiency in Software and Data Centers

Google has been a leader in investing in software and infrastructure efficiency to reduce data center energy usage without compromising service performance for years.

A recent example, circa 2024–2025, comes from Google's AI division: by optimizing its AI model stack, primarily the Gemini model and the related infrastructure, Google achieved extraordinary gains in efficiency.

Over a 12-month span, the median energy consumption per AI text prompt and the associated carbon footprint per prompt dropped by factors of 33× and 44× respectively. In other words, due to software and hardware optimizations, the energy required for an average AI query (response to a prompt) is now roughly equivalent to just nine seconds of watching television. This median energy consumption decreasing has carried on to the computation cost and to the CO2 emissions per user query, even as the quality and the complexity of AI responses have increased. Also, Google has reported that in 2024 its data center energy-related emissions fell by 12% compared to the previous year, despite a 27% growth in overall electricity consumption due to higher demands. This indicates that cleaner energy sources and software efficiency improvements have "absorbed" the impact of growing workloads, allowing Google's services to expand without a proportional rise in carbon footprint.

Another important factor in Google's efficiency strategy is architectural optimization in service delivery. Google pioneered a large-scale container orchestration with its internal cluster manager called Borg, which later inspired the open-source Kubernetes system. The goal of Borg is to maximize the utilization of Google's vast server fleet by "intelligently" scheduling and packing workloads on machines, thereby avoiding idle resources. Borg can co-locate multiple tasks (from different applications or services) on the same server in isolated containers, using techniques like resource overcommitment and priority-based scheduling, all while maintaining reliability.

At Google's scale, even a small increase in average server utilization can translate into massive savings, as they them report "even small improvements in utilization translate to millions of dollars in savings". In other words, by doing "more work with fewer servers", Borg reduces the number of machines needed for a given amount of computation, directly cutting operational costs and energy usage per unit of work.

One analysis of Google's Borg clusters, however, revealed that in practice the average CPU and memory usage of a cluster still hovers around only 50–60% of capacity, highlighting room for further optimization. Even so, this approach of "squeezing" more computing workload for each server has been essential in lowering Google's infrastructure costs and environmental footprint over time.

5.4. Meta (Facebook): Switching to a More Efficient Engine with HHVM

Meta (formerly Facebook) has similarly invested in software efficiency improvements, especially given its heavy use of PHP in web services. A high-profile example is Meta's development of the HipHop Virtual Machine (HHVM), which is an alternative high-performance runtime for PHP. HHVM was designed to execute PHP code far more

efficiently than the standard Zend PHP interpreter by using just-in-time (JIT) compilation. Introduced around 2013, Facebook's HHVM has demonstrated substantial speedups, internal benchmarks showed it running typical PHP workloads about 2× to 4× faster than PHP 5 (the then-standard PHP engine). By accelerating the code at the language runtime level, Meta aimed to serve more users with the same hardware or to reduce the number of servers required to handle existing traffic.

The real-world impact of HHVM can be seen in the experience of Box, a large cloud storage company that collaborated with Facebook to adopt HHVM.

Box's platform was largely built on PHP, and they undertook a migration from regular PHP engine to HHVM to improve performance and efficiency. The results were significant: in initial tests, an important API endpoint at Box ran over 4x faster on HHVM compared to running on the standard PHP interpreter. Over the course of a careful year-long migration, Box rolled out HHVM across most of its production servers. This led to an average drop of more than 50% of CPU utilization, essentially the servers' CPU load was cut in half for the same traffic. In other words, after HHVM, Box could handle the same user demand with roughly half the CPU resources it previously required, effectively doubling their capacity without adding new hardware.

Most important, this efficiency gain translated directly into cost and energy benefits. Using at half of the capacity of CPU meant Box needed far fewer server cycles to perform the same work, which corresponded to fewer physical servers and less power consumption in the data center, therefor reducing the power consumption and the CO2 footprint.

The company noted that this gave them "over twice our current frontend capacity for free", providing significant savings in server procurement, power consumption, and data center space. At the same time, end-users experienced faster response times due to the performance boost. This was a rare situation with no trade-off, where both speed and cost-efficiency have improved.

Furthermore, HHVM opened the door for engineers at Facebook/Box to use Hack, Facebook's statically-typed PHP dialect, and other advanced features. Hack's static typing, along with HHVM's other capabilities, like built-in profilers and an option to precompile code, help developers write more efficient and maintainable code.

In summary, Meta's transition to a more efficient execution engine with HHVM demonstrates that software-level optimization can lead to substantial improvements in scalability and sustainability. By adopting a faster runtime, companies such as Facebook and Box have achieved higher performance rates and have reduced their numbers of machines (and electricity) required to serve their workloads, thereby cutting costs and lowering the environmental impact of their operations without sacrificing service quality.

6. Conclusions

We have examined, in this paper, the complex trade-offs between performance, cost, and ecological impact in modern software development, reinforcing the idea that these dimensions are deeply interconnected rather than being independent concerns. While traditional software engineering has often prioritized performance and rapid delivery, in the detriment of lowering CO2 emissions. The growing energy consumption of digital systems and the increasing awareness of environmental sustainability demand a more balanced and responsible approach.

This paper has established a conceptual framework for understanding sustainable software engineering through various approaches that relate to performance metrics, development and operational costs, and ecological indicators such as energy consumption, carbon footprint, and hardware lifecycle impact. The analysis emphasized that optimization choices made at the software level can significantly influence not only execution efficiency, but also long-term economic and environmental outcomes.

The empirical analysis based on the Energy-Languages dataset has provided concrete evidence that programming language choice has also a substantial effect on energy usage, execution time, and operational cost. The results have demonstrated that interpreted languages produce significantly higher energy consumption and runtime overhead compared to compiled languages, leading to noticeable higher cloud costs and carbon emissions at a high-scale.

Moreover, the Pareto analysis revealed that modern compiled languages, particularly Rust, can achieve favorable compromises across all three dimensions, performance, cost, and ecological impact, therefor challenging the assumption that sustainability necessarily requires to sacrifice performance or to increase costs.

These findings were reinforced by the real-world case studies presented, from large-scale industry actors such as Amazon Web Services, Google, and Meta. They stated that infrastructure-level innovations, energy efficient hardware, and optimized runtime systems would deliver simultaneous improvements in performance efficiency, cost reduction, and environmental impact. These examples illustrate that sustainability-oriented decisions, when guided by accurate measurement and informed architectural choices, can realign traditional trade-offs rather than increase them.

Overall, the findings of this paper suggest that sustainable software development is not a constraint, but an opportunity. Software professionals can build systems that are efficient, economically viable, and environmentally responsible over their entire lifecycle, by simply integrating performance engineering, cost awareness, and ecological responsibility into early design decisions. As software continues to play an increasingly central role in global energy consumption, such integrated approaches will be essential for aligning technological progress with long term sustainability goals.

References

[1] P. Bourque and R. E. Fairley - *Guide to the Software* - Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, 2014.

[2] R. S. Pressman and B. R. Maxim - *Software Engineering: A Practitioner's Approach*, 8th ed., McGraw-Hill, 2015.

[3] B. W. Boehm - *Software Engineering Economics* - IEEE Transactions on Software Engineering, vol. 10, no. 1, pp. 4–21, 1984.

[4] B. W. Boehm et al. - *Software Cost Estimation with COCOMO II* - Prentice Hall, 2000.

[5] A. J. Albrecht - *Measuring Application Development Productivity* - IBM Applications Development Symposium, 1979.

[6] ISO/IEC 20926:2009, Software and Systems Engineering — Software Measurement.

[7] M. Lehman - *Programs, Life Cycles, and Laws of Software Evolution* - Proceedings of the IEEE, vol. 68, no. 9, pp. 1060–1076, 1980.

[8] https://testrigor.com/blog/what-is-green-software/ testRigor - What is Green Software - 19-04-2026.

[9]. https://www.nttdata.com/global/en/insights/reports/intesa-sanpaolo NTT Data - Measuring the IT Carbon Footprint for Intesa Sanpaolo - 19-04-2026.

[10] https://preview.methodology.scope3.com/software_carbon_intensity Green Software Foundation - Software Carbon Intensity Specification - 19-04-2026

[11]. https://www.fairplanet.org/story/smartphone-pollution-electronic-waste/ FairPlanet - *A Closer Look at Smartphone Pollution* - 19-04-2026.

[12]https://pollution.sustainability-directory.com/question/how-does-software-obsolescence-affect-sustainability/ Sustainability Directory - *How Does Software Obsolescence Affect Sustainability?* - 19-04-2026.

[13] https://aws.amazon.com/ec2/graviton/graviton-sustainability/ Amazon Web Services - *Sustainability with AWS Graviton* - 19-04-2026.

[14]https://aws.amazon.com/solutions/case-studies/pinterest-graviton-case-study/ Amazon Web Services - Improving Sustainability and Price Performance Using AWS Graviton with Pinterest - 19-04-2026.

Bibliography

P. Bourque and R. E. Fairley - *Guide to the Software* - Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, 2014.

R. S. Pressman and B. R. Maxim - *Software Engineering: A Practitioner's Approach*, 8th ed., McGraw-Hill, 2015.

B. W. Boehm - *Software Engineering Economics* - IEEE Transactions on Software Engineering, vol. 10, no. 1, pp. 4–21, 1984.

B. W. Boehm et al. - *Software Cost Estimation with COCOMO II* - Prentice Hall, 2000.

A. J. Albrecht - *Measuring Application Development Productivity* - IBM Applications Development Symposium, 1979.

ISO/IEC 20926:2009, Software and Systems Engineering — Software Measurement.

M. Lehman - *Programs, Life Cycles, and Laws of Software Evolution* - Proceedings of the IEEE, vol. 68, no. 9, pp. 1060–1076, 1980.

https://testrigor.com/blog/what-is-green-software/ testRigor - What is Green Software - 19-04-2026.

https://www.nttdata.com/global/en/insights/reports/intesa-sanpaolo NTT Data - Measuring the IT Carbon Footprint for Intesa Sanpaolo - 19-04-2026.

https://preview.methodology.scope3.com/software_carbon_intensity Green Software Foundation - Software Carbon Intensity Specification - 19-04-2026

https://www.fairplanet.org/story/smartphone-pollution-electronic-waste/ FairPlanet - *A Closer Look at Smartphone Pollution* - 19-04-2026.

https://pollution.sustainability-directory.com/question/how-does-software-obsolescence-affect-sustainability/ Sustainability Directory - *How Does Software Obsolescence Affect Sustainability?* - 19-04-2026.

https://aws.amazon.com/ec2/graviton/graviton-sustainability/ Amazon Web Services - *Sustainability with AWS Graviton* - 19-04-2026.

https://aws.amazon.com/solutions/case-studies/pinterest-graviton-case-study/ Amazon Web Services - Improving Sustainability and Price Performance Using AWS Graviton with Pinterest - 19-04-2026.

https://github.com/greensoftwarelab/Energy-Languages - R. Pereira et al. - *Energy Efficiency across Programming Languages* - 19-04-2026.

A. Noureddine et al. - *Investigating the Impact of Software Design Patterns on Energy Consumption* - Sustainable Computing: Informatics and Systems, 2025.

M. Darvish, C. Archetti, L.Coelho - *Trade-offs between environmental and economic performance in production and inventory-routing problems* - International Journal of Production Economics, Volume 217, , Pages 269-280, November 2019.

https://green-forum.ec.europa.eu/news/how-calculate-full-environmental-impact-products-and-organisations-2024-06-24_en - European Commission - How to calculate the full environmental impact of products and organisations - 19-04-2026.

A. Núñez, C. Andrés, M. G. Merayomg - Optimizing the Trade-offs Between Cost and Performance in Scientific Computing - Procedia Computer Science, Volume 9, Pages 498-507, 2012.

S. Barney, K. Petersen, M. Svahnberg, A. Aurum, H. Barney - *Software quality trade-offs: A systematic map* - Information and Software Technology Volume 54, Issue 7, Pages 651-662, July 2012.

Theresia Ratih Dewi Saputri, Seok-Won Lee - *Analyzing Trade-offs for Sustainability Requirements: A Decision-Making Process* - CEUR Workshop Proceedings (CEUR-WS.org), Volume 2541, ISSN 1613-0073, 2019.