# Energy Efficiency of Java Garbage Collection Strategies: An Empirical Study

Rahil S
Student Number: 2850828
VU Amsterdam
r.sharma4@student.vu.nl

András Zsolt Sütő
Student Number: 2856739
VU Amsterdam
a.z.suto@student.vu.nl

Tobias Meyer Innleggen
Student Number: 2855564
VU Amsterdam
t.m.innleggen@student.vu.nl

Vivek A Bharadwaj
Student Number: 2841186
VU Amsterdam
v.a.bharadwaj@student.vu.nl

Avaneesh Shetye
Student number: 2843910
VU Amsterdam
a.shetye@student.vu.nl

## Abstract

This study empirically investigates the energy efficiency of three Java garbage collection (GC) strategies—Serial, Parallel, and G1 across eight applications, three workload intensities, and two JDK implementations (OpenJDK and Oracle JDK). A total of 576 controlled runs were executed to measure energy consumption, runtime, and derived efficiency metrics including power, Energy–Delay Product (EDP), and Coefficient of Performance (CoP). Results show that Serial GC consistently achieves the lowest mean energy consumption (332.5 J), followed closely by Parallel (346.3 J), while G1 is substantially higher (426.2 J). However, statistical modeling revealed that GC strategy (49%) and workload intensity (32%) both significantly shape energy behavior, while application architecture (captured as subject-level variance) accounts for the remaining variation. Runtime and energy consumption were almost perfectly correlated ($r = 0.961$, 95% CI [0.954, 0.967]). Faster executions were consistently more energy efficient. Equivalence testing confirmed that no detectable difference was found between OpenJDK and Oracle JDK ($F(1,574) = 0.039$, $p = 0.844$). TOST equivalence ($p = 0.426$) was non-significant. Collectively, these findings demonstrate that GC strategy, workload intensity, and application architecture all shape energy efficiency, with GC configuration explaining nearly half of observed variance. Runtime optimization emerged as the dominant predictor ($r = 0.96$), indicating that reducing execution time reliably reduces energy consumption. By demonstrating that small, low-effort configuration choices such as selecting Serial GC can complement architectural optimizations, this work provides actionable guidance for developers and DevOps teams aiming to reduce the environmental footprint of large-scale Java systems without compromising performance.

**ACM Reference Format:**
Rahil S, András Zsolt Sütő, Tobias Meyer Innleggen, Vivek A Bharadwaj, and Avaneesh Shetye. 2025. Energy Efficiency of Java Garbage Collection Strategies: An Empirical Study. In *Green Lab 2025/2026 - Vrije Universiteit Amsterdam, September–October, 2025, Amsterdam (The Netherlands).* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*Green Lab 2025/2026, Amsterdam, The Netherlands*

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Energy consumption has become a critical concern in software engineering in conjunction with traditional performance metrics, especially since the total electricity consumption of the ICT sector was estimated at 1,000 TWh globally in 2023 [1], which is approximately 4% of the global electricity usage [2]. This has led to interest in green software and energy-efficient design practices [3]. In practice, developers and operators still primarily optimize for performance (e.g., speed, throughput) and seldom tune software configurations specifically for energy efficiency. This issue is increasingly important to address in large-scale systems, where even small energy savings per application can translate into significant cost and environmental benefits when multiplied across thousands of servers [4]. Our study contributes to this effort by empirically evaluating the energy consumption of Java programs under different garbage collector (GC) configurations. Java is one of the most widely used programming platforms in modern software development, powering everything from enterprise servers to mobile apps, with more than 60 billion Java instances running globally as of 2024 [5]. On the Java Virtual Machine (JVM) platform, GC is a runtime component that can substantially influence application performance and resource usage [6]. The GC automates memory management by reclaiming unused objects, but its operation consumes CPU time and can introduce execution pauses.

Modern JVMs provide multiple GC algorithms, each with different performance trade-offs, but these have historically been designed and tuned with throughput and latency goals in mind rather than energy consumption. In other words, GC configurations are typically chosen to maximize application speed or minimize pause times, not to minimize power draw. As a result, energy-oriented GC tuning remains uncommon in practice. Shimchenko *et al.* [7] found that in many cases one can reduce the energy consumption of a Java program by switching from the default Garbage-First GC (G1) to an alternative collector, without sacrificing performance. This means that by simply changing a JVM flag, large-scale Java deployments

[0] [GitHub] java-garbage-collection-strategies [Time Log] Google Sheets [Video] Google Drive

could potentially achieve noticeable energy savings. OpenJDK's HotSpot JVM includes several mainstream garbage collectors. In this paper, we focus on three well-established algorithms: Serial GC, Parallel GC, and Garbage-First (G1) GC. The Serial collector is the simplest stop-the-world GC, using a single thread to perform all collection work [8]. It employs a generational mark-and-compact approach and is efficient for small heaps or single-processor environments but cannot take advantage of multiple CPU cores. The Parallel collector (also known as the throughput collector) is similar in design to Serial GC but leverages multiple threads working in parallel to accelerate garbage collection [8]. Parallel GC aims to reduce collection pause times and improve overall throughput on multicore machines. G1 GC is a mostly concurrent, region-based collector designed to balance high throughput with controlled pause times [8, 9]. G1 partitions the heap into numerous regions and performs incremental collection: instead of collecting the entire heap at once, G1 collects a subset of regions in each GC cycle to meet a given pause-time target. Since Java 9, G1 has been the default GC in HotSpot JVM due to its ability to deliver low pause times with good throughput across a variety of workloads.

We present an empirical study comparing the energy efficiency and performance of Serial, Parallel, and G1 GCs under a range of Java workloads. Our experimental setup runs benchmarks and real-world Java applications with each GC algorithm while varying workload characteristics and JDK versions. During each trial, we collect performance metrics (execution time, throughput, GC pause durations) and energy consumption measurements, then quantify how each garbage collector impacts energy usage relative to performance. We focus on Serial, Parallel, and G1 collectors as they represent the primary choices for most Java deployments and cover fundamental design trade-offs: single-threaded simplicity (Serial), parallel throughput optimization (Parallel), and concurrent low-latency collection (G1). While modern collectors like ZGC and Shenandoah target specialized ultra-low latency use cases, we limit our scope to three collectors to ensure experimental feasibility within the course timeframe while maintaining statistical rigor. This focused approach establishes a solid methodological foundation that future work can extend to include newer collectors. Our results help engineers select energy-efficient GC strategies for their deployment environments.

## 2 Related Work

A growing body of research has investigated the energy footprint of software systems, with particular attention to programming languages, runtime environments, and software design choices. Castor provides a comprehensive primer on estimating the energy footprint of software systems, highlighting measurement techniques, trade-offs, and pitfalls in linking energy consumption to software-level decisions [10]. Currie et al. propose a more practical perspective in their book *Building Green Software*, offering guidelines and frameworks for software developers to embed sustainability considerations into everyday practice [11]. Together, these works provide the conceptual and methodological foundation for energy-aware software development.

More focused on programming languages and runtimes, Nou et al. [12] conducted one of the first systematic analyses of how different Java Virtual Machine garbage collectors influence power consumption. Their results suggest that runtime configuration can lead to measurable energy savings, though their evaluation was limited in terms of benchmarks and workloads. Georges et al. [13] highlighted the need for statistical rigor in Java performance evaluation, providing methods that remain relevant when designing reproducible energy experiments. In addition, Hindle's *Green Mining* approach [14] linked software change patterns to energy use, demonstrating that developer decisions can indirectly affect system-level energy efficiency. Finally, Procaccianti et al. [15] offered a systematic mapping of green software engineering research, outlining categories of techniques (measurement, tooling, methodologies) and identifying gaps where empirical validation is still lacking. Contreras et al. (2026) [16] measured the link between garbage collection and energy use, showing that GC could consume up to 37% of CPU energy, with generational collectors generally more efficient. Their focus on older collectors (SemiSpace, MarkSweep) and synthetic benchmarks makes their results less applicable to modern JVMs. Unlike Contreras et al., our study evaluates modern default collectors (Serial, Parallel, G1) in OpenJDK and Oracle JDK across both benchmarks and real-world Java applications.

Ournani et al. (2021) [17] conducted a large-scale evaluation of 52 JVM distributions from eight providers using DaCapo and Renaissance benchmarks. They found significant variability in energy efficiency across JVMs and showed that tuning garbage collection and JIT parameters could reduce energy consumption by up to 100%. Unlike Ournani et al., who provided a broad infrastructure-level survey across many JVMs, our study narrows the focus to modern default collectors (Serial, Parallel, G1) in OpenJDK and Oracle JDK. Rather than emphasizing cross-distribution comparisons, we analyze developer-relevant trade-offs between energy efficiency and performance under varying workload intensities. Wang et al. (2025) [18] introduced the GEAR framework to enable cross-runtime comparisons of garbage collectors in Java, Go, and C#. By abstracting memory usage into runtime-agnostic primitives, they systematically generated equivalent workloads, revealing issues such as scalability challenges in Java's ZGC and inefficiencies in Go's GC. In contrast, our study focuses exclusively on Java runtimes and the default collectors included in OpenJDK and Oracle JDK. Instead of introducing a new evaluation framework, we directly measure energy-performance trade-offs in benchmarks and real-world applications, providing practical configuration guidance for software developers.

Shimchenko et al. (2022) present an empirical evaluation of six GC strategies in OpenJDK, including Serial, Parallel, CMS, G1, ZGC, and Shenandoah [19]. Using jRAPL to measure energy consumption, they tested both throughput- and latency-sensitive benchmarks such as DaCapo, Renaissance, Hazelcast, and SPECjbb. Their results show that energy use is workload-dependent: concurrent collectors (ZGC, Shenandoah) reduce latency but increase energy demand, while simpler ones (parallel GC) often consume less in CPU bound scenarios. We adopt several aspects of their methodology, including RAPL-based measurement (via EnergiBridge), exclusion of JVM

startup phases, and replication across runs. Their finding that no collector is universally optimal aligns with our hypothesis that G1 may strike a balance between efficiency and performance. They also report energy savings of up to 15% from GC tuning, a figure we aim to test across both benchmarks and real-world Java applications. Their analysis of heap sizing, recommending sufficiently large heaps to avoid distorted GC behavior, motivates our use of a heap three times the live set. While Shimchenko et al. analyze six collectors only on OpenJDK, our study narrows the focus to three (Serial, Parallel, and G1) and adds Oracle JDK as a cofactor.

Lengauer et al. (2017) analyze Java memory and GC behavior across DaCapo, SPECjvm2008, and DaCapo Scala [20]. They profile allocation patterns, survivor ratios, object sizes, array density, and GC metrics, offering a framework to categorize benchmarks by GC stress. Their study identifies workloads such as tradesoap and h2 as allocation heavy and differentiates continuous versus burst allocation phases. These insights guide our project design, where we test light, medium, and heavy conditions. They recommend heap sizes around three times the live set, which we adopt to ensure natural GC activity without premature collections. Their observations on survivor ratios and GC frequency support our annotation of workloads and GC log analysis. Although detailed, their study does not address energy consumption, which we explicitly incorporate. They also document JVM compatibility issues, such as DaCapo Scala failures on newer versions, which helps our benchmark filtering. Their profiling provides the technical basis for workload selection and experimental design, while our study extends it with energy measurement.

Together, the related studies establish a strong foundation for understanding the interplay between Java garbage collection, performance, and energy consumption. Early work such as Contreras et al. (2006) first demonstrated that GC tuning affects energy use, while more recent studies (Kumar, 2020; Ournani et al., 2021) broadened the perspective to sustainability in cloud and large-scale JVM deployments. Building on this, Shimchenko et al. (2022) and Lengauer et al. (2017) offered detailed empirical analyses of GC strategies and memory behavior. Despite this breadth, existing work typically concentrates on broad sustainability frameworks, narrow runtime profiling, or isolated optimizations. Our study addresses this gap through a controlled comparison of three modern collectors (Serial, Parallel, G1), tested under two JDKs and a mix of benchmarks and real applications. By explicitly measuring both energy and performance trade-offs in a reproducible experimental design, we contribute practical insights that complement earlier large-scale surveys and specialized GC evaluations.

# 3 Experiment Definition

Our experiment utilizes the ExperimentRunner framework running on a dedicated Raspberry Pi to automate 600 runs (we used 490 for analysis after outlier rejection) across a controlled Linux laptop environment, managing experimental factors through structured configuration with built-in randomization and replication. This separation ensures that the ExperimentRunner does not influence the energy consumption measurements of the laptop. Energy measurements are collected using EnergiBridge, which interfaces with RAPL counters to capture CPU energy consumption during Java execution on the laptop. Each run executes with specific JVM flags (-XX:+UseSerialGC, -XX:+UseParallelGC, -XX:+UseG1GC) while alternating between OpenJDK and Oracle implementations.

To enable distributed development, we implement a mock energy interface that simulates RAPL behavior on non-Linux systems for pipeline testing. The pipeline generates individual EnergiBridge CSV files per run and a consolidated runtable.csv for statistical analysis in R, leveraging established energy measurement tools to ensure reproducible results.

## 3.1 Goal Definition Using the GQM Framework

To assess the impact of Java garbage collection (GC) strategies on energy consumption and performance, we define our experiment using the Goal-Question-Metric (GQM) framework.
This goal directly addresses the core problem identified in Section 1: while multiple GC strategies exist in modern JVMs, developers typically rely on default configurations without understanding their energy implications, missing opportunities to reduce the environmental footprint of Java applications. The structured GQM approach enables systematic comparison of GC options, providing the empirical evidence that software developers, DevOps engineers, and researchers currently lack for making energy-aware configuration decisions. By focusing on energy consumption and performance trade-offs, this study addresses the practical reality that developers cannot optimize for energy alone—they must balance sustainability with application performance requirements. The Linux system context and inclusion of both benchmarks and real applications ensure our findings apply to typical deployment scenarios where these stakeholders make GC configuration choices, ultimately supporting more informed decisions that can contribute to sustainable software practices across the Java ecosystem.

Our workload classification follows Lengauer et al.'s framework for characterizing GC stress [20]. Light workloads feature low allocation rates with minimal heap pressure, creating infrequent GC activity. Medium workloads introduce moderate allocation patterns with regular but manageable GC cycles. Heavy workloads generate high allocation rates and heap pressure, forcing frequent garbage collection and testing each collector's efficiency under stress. This progression allows us to evaluate how different GC strategies scale with increasing memory management demands.

As shown in Table 1, our study defines four research questions (RQ1–RQ4) together with their associated metrics. While the table provides a structured overview, we also visualize the Goal-Question-Metric (GQM) framework to highlight the hierarchical relationship between the overall goal, the research questions, and the specific metrics.

**Table 1: GQM Framework: Research Questions and Metrics**

| **GOAL: Analyze Java GC strategies for energy efficiency and performance trade-offs**<br>*From the perspective of developers, DevOps engineers, and researchers* | | |
|---|---|---|
| **RQ** | **Research Question** | **Metrics (Units)** |
| **RQ1** | Which GC strategy minimizes energy consumption across Java applications? | Total energy (J), Mean power (W), Energy per operation (J/op) |
| **RQ2** | How does workload level influence energy efficiency of different GC strategies? | Energy (J), Runtime (s), Energy scaling factor |
| **RQ3** | What are the trade-offs between energy and performance for each GC strategy? | Runtime (s), Throughput (ops/s), Latency (p95/p99 ms), GC pause times (ms) |
| **RQ4** | How does JDK implementation affect energy efficiency of GC strategies? | Energy (J), Runtime (s), Throughput (ops/s) |
| **Key Relationships:** Energy = Power × Time; Efficiency = Energy ÷ Operations | | |

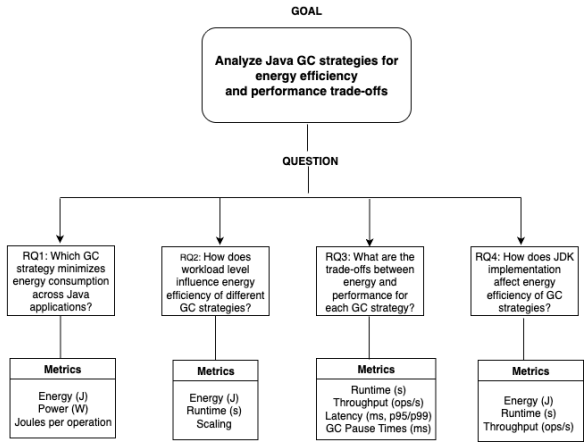


**Figure 1: GQM framework mapping the overall goal to its research questions and associated metrics.**

This figure makes it explicit how each question operationalizes the overall goal, and how the metrics allow us to measure answers in a reproducible way. It complements the table by providing a more intuitive, top-down view of the GQM approach.

## 3.2 Research Questions and Metrics

Our experiment is guided by four research questions (RQs) that dissect energy and performance trade-offs across garbage collection strategies. Each RQ is quantitative, contributing to a structured understanding of how GC behavior affects real world application deployment. Figure 2 illustrates the logical relationship between our research questions. RQ1 establishes the baseline by identifying which GC strategy consumes least energy. RQ2 and RQ4 examine how this efficiency varies under different conditions (workload levels and JDK implementations). RQ3 connects back to evaluate the practical trade-offs between energy savings and performance characteristics, ensuring our findings are applicable in real world scenarios where developers must balance multiple objectives.

*RQ1: Which GC strategy minimizes energy consumption across Java applications?* This question is quantitative, as it compares numerical measurements of energy usage. Metrics include total energy (J), mean power (W), and normalized energy per operation (J/op). These directly address the core goal of identifying which GC consumes the least energy. The outcome is important for developers and DevOps engineers who often rely on defaults without considering energy implications. This RQ sets the baseline for subsequent questions.

*RQ2: How does workload level influence energy efficiency of different GC strategies?* This question investigates scalability by analyzing how efficiency changes under light, medium, and heavy loads. Metrics include energy (J), runtime (s), and an energy scaling factor calculated as the ratio between workloads. It contributes to the goal by testing whether GC efficiency is consistent or workload-sensitive.This matters because real-world applications rarely run at constant load.

*RQ3: What are the trade-offs between energy and performance for each GC strategy?* This RQ balances energy metrics with performance indicators. Metrics include runtime (s), throughput (ops/s), latency percentiles (p95/p99 ms), GC pause times (ms), and energy per operation (J/op). It contributes by framing energy savings against possible performance degradation, which is crucial for practical adoption. For example, a GC that reduces energy but severely increases latency may not be viable.

*RQ4: How does JDK implementation affect energy efficiency of GC strategies?* This RQ tests generalizability by comparing OpenJDK and Oracle JDK implementations. Metrics include energy (J), runtime (s), and throughput (ops/s). It contributes to the goal by showing whether observed trends are JVM-specific or applicable across distributions. This matters for researchers and practitioners who need reproducible results in different deployment environments.

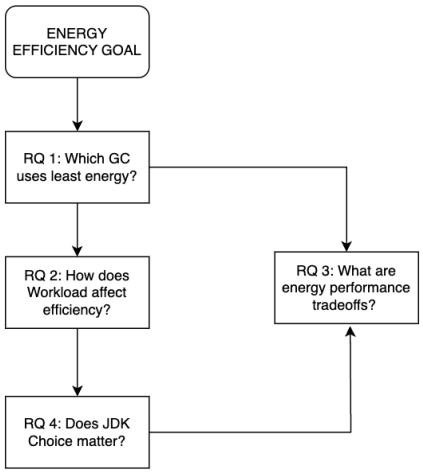


**Figure 2: Logical contribution diagram showing dependencies among the research questions.**

**Table 2: Metric Definitions and Relationships**

| Metric (Unit) | Definition | Relationship |
|---|---|---|
| Energy (J) | Total energy consumed during execution | $E = P \times t$ |
| Power (W) | Average rate of energy consumption | $P = \frac{E}{t}$ |
| Energy per Operation (J/op) | Energy normalized by workload size | $E \div$ operations |
| Runtime (s) | Execution duration of a program | Base input for power calculation |
| Throughput (ops/s) | Operations completed per second | Inverse relationship with latency |
| Latency (ms) | Response time at given percentiles | Higher latency = lower responsiveness |
| GC Pause Time (ms) | Time application halted for GC | Directly affects user experience |

## 3.3 Experimental Variables and Relationships

The metrics in this experiment capture both energy consumption and performance characteristics, ensuring that trade-offs are explicitly measurable. Table 2 defines each metric, its unit of measurement, and its mathematical or conceptual relationship with other metrics. These relationships ground the experiment in quantifiable, reproducible values, following the principles of construct validity discussed in Lecture 2 [21].

**Key Relationships.** Energy is fundamentally derived as the product of power and time ($E = P \times t$). Normalizing energy by the number of operations yields energy per operation, which allows fair comparison across workloads of different sizes. Performance metrics such as throughput, latency, and GC pause time interact dynamically with energy: higher throughput often reduces energy per operation, while excessive GC pauses or high latency degrade both user experience and efficiency. By explicitly capturing these dynamics, our experiment can evaluate not only raw energy consumption but also its trade-offs with performance, aligning with the practical concerns of software developers and system operators.

## 3.4 Experimental Design and Subjects

Our experiment employs a Randomized Complete Block Design (RCBD) to systematically evaluate GC strategies while controlling for application-specific variations. In practical terms, blocks represent different types of Java programs—analogous to testing car engines across different vehicle models to ensure findings apply broadly. The treatments are our three GC strategies (Serial, Parallel, G1), while co-factors (workload level and JDK implementation) represent external conditions that could influence results. We replicate each combination three times for statistical reliability. Figure 3 illustrates our complete experimental structure across six Java subjects: three standardized benchmarks (DaCapo Chopin, CLBG, Rosetta Code) and three realistic applications (web server, GUI todo app, image processing tool). This design balances research reproducibility with practical applicability, resulting in 600 runs (we used 490 for analysis after outlier rejection) total experimental runs that will provide robust evidence for energy-performance trade-offs across diverse Java workloads.

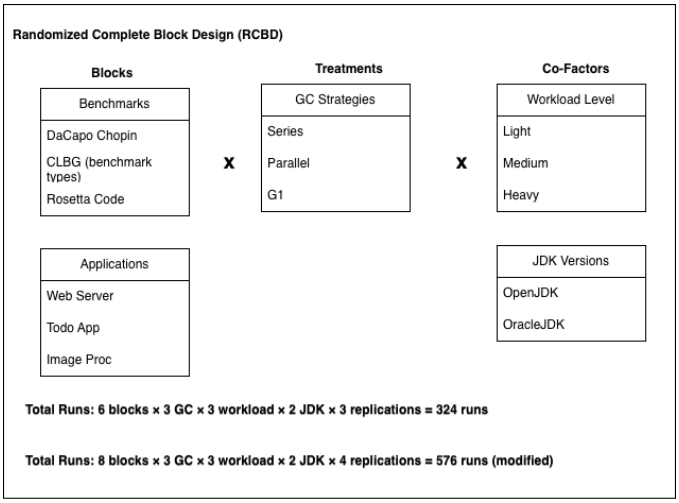


**Figure 3: Randomized Complete Block Design (RCBD) for Java GC Energy Efficiency Experiment (Modified)**

# 4 Experiment Planning

Building on the framework defined in Section 3, this section translates our framework into practice by detailing subject selection, variable specification, hypothesis formulation, design choices, and analysis strategy. The transition from conceptual design to executable experiment required addressing a central methodological challenge: validating the measurement pipeline before committing to full testbed execution. Modern empirical software engineering emphasizes iterative refinement of measurement systems to ensure construct validity before data collection begins [22, 23]. Following this principle, we implemented a mock energy measurement interface that simulates RAPL-based energy data while preserving the complete ExperimentRunner workflow. This approach serves three purposes: (i) enabling cross-platform development without requiring continuous Linux hardware access, (ii) validating randomization logic, data aggregation, and statistical analysis pipelines before real measurements, and (iii) supporting rapid design iteration without incurring 20–50 hour execution cycles for each refinement. The mock interface generates synthetic energy values based on empirically informed models derived from Shimchenko et al. [24], incorporating realistic variance to simulate measurement uncertainty. Importantly, this is a development tool, not a data source—all results reported in Section 6 will be based exclusively on real RAPL measurements from the production testbed described in Section 5. The mock interface validates that our experimental machinery operates correctly; the actual experiment validates our research hypotheses. This feasibility-first approach allowed us to identify and resolve issues in factor randomization, cooldown timing, and CSV schema design during pipeline development. As a result, once deployed to the Linux testbed, the experiment executes reliably without requiring iterative debugging on specialized hardware. The following subsections describe the operationalization choices for subjects, variables, hypotheses, design, and analysis.

## 4.1 Subject Selection

Our experiment evaluates six subjects: three standardized benchmarks and three real-world Java applications. The selected subjects are summarized in Table 3.

**Table 3: Subjects included in the experiment**

| Subject | Type | Characteristics | Justification |
|---|---|---|---|
| DaCapo (Chopin) [20] | Benchmark suite | Standard Java suite with diverse, memory-intensive apps (web, DB, compute). | Widely used in GC research; reproducible baseline. |
| CLBG: *binary-trees*, *fannkuchredux*, *nbody* | Micro-benchmarks | Algorithmic, allocation-heavy, minimal I/O; short-lived objects. | Stresses GC under synthetic loads; common in JVM studies. |
| Rosetta Code tasks | Benchmark collection | Algorithmic and performance-oriented tasks implemented across languages. | Provides diverse, comparable workloads; used in empirical studies. |
| Spring PetClinic [25] | Web application | Spring Boot CRUD system with MVC UI + DB; typical REST workflow. | Canonical enterprise workload (REST+DB). |
| REST To-Do App [26] | Web service | Lightweight REST API for task mgmt; concurrent requests, I/O-bound. | Representative microservice-like workload. |
| ANDIE Image Tool [27] | Desktop GUI | Swing image editor; compute- and memory-heavy filters; interactive. | Adds interactive, client-side workload. |

In our selection, we aimed to balance allocation-heavy benchmarks with interactive and I/O-driven applications. Benchmarks (DaCapo, CLBG, Rosetta Code) provide standardized, repeatable patterns [19, 20], while the three open-source applications adds to our experiment's ecological validity by mimicking real-world Java deployment scenarios. Together, they span CPU-intensive, I/O-intensive, and interactive workloads. This selection allows garbage collectors to be evaluated under both synthetic and realistic conditions.

Subject choice was guided by three explicit criteria:

- *Benchmark selection.* We included DaCapo Chopin, CLBG programs, and Rosetta Code tasks because they are well-established in GC and JVM research [20]. Their controlled and reproducible workloads allow for comparisons across garbage collectors [13].
- *Application selection.* We selected three open-source Java applications to reflect common deployment scenarios: (i) a Spring Boot enterprise web app (PetClinic) representing REST+DB services, (ii) a lightweight REST API (To-Do) for microservice-like workloads, and (iii) a desktop GUI application (ANDIE Image Tool) that stresses GC under user-facing, compute-intensive conditions. These applications capture diverse execution patterns and memory behaviors beyond benchmarks.
- *Coverage of workload types.* The combined set covers allocation-heavy batch processing, network-bound services, and interactive GUI workloads—a broad spectrum of usage contexts.

Most GC energy efficiency studies restrict their evaluation to benchmarks [12, 19]. In our subject selection we combine both synthetic and real-world Java applications, which enables observations that are both reproducible and practically relevant. This broader scope improves the generalizability of our findings to real deployment environments.

## 4.2 Experimental Variables

The experiment manipulates three independent variables: GC strategy, workload level, and JDK implementation.

**Table 4: Independent and dependent variables grouped by research question.**

| RQ | Var. Type | Variable | Values/Levels | Data Type | Unit |
|---|---|---|---|---|---|
| RQ1 | Independent | GC Strategy | Serial, Parallel, G1 | Nominal | – |
| RQ1 | Dependent | Runtime | Measured | Ratio | s |
| RQ1 | Dependent | Total Energy | Measured | Ratio | J |
| RQ1 | Dependent | Mean Power | Calculated | Ratio | W |
| RQ2 | Independent | Workload Level | Light, Medium, Heavy | Ordinal | – |
| RQ2 | Dependent | Runtime | Measured | Ratio | s |
| RQ2 | Dependent | Total Energy | Measured | Ratio | J |
| RQ2 | Dependent | Energy per Op. | Calculated | Ratio | J/op |
| RQ2 | Dependent | Mean Power | Calculated | Ratio | W |
| RQ3 | Dependent | Throughput | Calculated | Ratio | ops/s |
| RQ3 | Dependent | Latency (p95) | Measured (interactive only) | Ratio | ms |
| RQ3 | Dependent | GC Pause Time | Measured | Ratio | ms |
| RQ3 | Dependent | Energy per Op. | Calculated | Ratio | J/op |
| RQ4 | Independent | JDK Impl. | OpenJDK 17, Oracle JDK 17 | Nominal | – |
| RQ4 | Dependent | Runtime | Measured | Ratio | s |
| RQ4 | Dependent | Total Energy | Measured | Ratio | J |
| RQ4 | Dependent | Mean Power | Calculated | Ratio | W |

GC strategy is a nominal factor with three levels (Serial, Parallel, G1), enabled via -XX:+UseSerialGC, -XX:+UseParallelGC, or -XX:+UseG1GC [8]. These collectors represent different trade-offs: Serial is single-threaded and simple, Parallel leverages multiple threads for throughput, and G1 is the default low-latency collector in modern JVMs.

Workload level (Light, Medium, Heavy) is an ordinal factor expressing allocation intensity. In DaCapo Chopin benchmarks, levels are mapped via input sizes (small, default, large). In CLBG and Rosetta Code tasks, iteration counts and recursion depth scale the load ($10^6$, $10^7$, $10^8$). For the REST web server, concurrent clients per second define levels (10, 100, 1000). In the GUI To-Do app, scripted user interactions per session vary (5, 20, 50). In the image processing tool, workload is file size (10 MB, 100 MB, 1 GB). These operationalizations ensure comparable Light/Medium/Heavy categories across subject types [20].

JDK implementation (OpenJDK 17 Temurin vs. Oracle JDK 17) is a nominal factor used to evaluate the consistency of results across mainstream JVM distributions. Differences in implementation can affect GC performance and energy efficiency [28]. Restricting both to Java 17 eliminates version drift and isolates vendor-specific effects.

Dependent variables are measured or derived performance and energy metrics. Total energy (J) is measured using Intel RAPL counters via EnergiBridge. Execution time (s) is recorded, allowing calculation of average power (W) as $P = E/t$. Energy per operation (J/op) is computed by normalizing energy consumption over completed operations. Throughput (ops/s) is calculated from workload completions. Latency metrics (95th percentile) apply only to interactive workloads (REST API and GUI). GC pause time (ms) is extracted from JVM logs. These dependent variables are consistent with the GQM framework defined in Table 1.

Several control variables are fixed to ensure internal validity. All runs execute on identical Linux hardware with CPU frequency scaling disabled and background processes minimized. Ambient temperature is stabilized and monitored, and the same measurement framework (EnergiBridge) is applied across conditions.

The relationships among variables follow expected system properties: $E = P \times t$, throughput is inversely related to latency, and longer GC pauses increase observed response latency.

## 4.3 Experimental Hypotheses

Our hypotheses are derived from the independent variables (GC strategy, workload intensity, JDK implementation) and dependent variables (energy consumption, throughput, latency, GC pause time) defined in Section 4.2. For each research question we state a null hypothesis ($H_0$), assuming no effect, and an alternative hypothesis ($H_1$) capturing the expected outcome. For questions involving multiple factors (RQ2, RQ4), we formulate hypotheses for each main effect and their interaction, yielding three hypothesis pairs per question. All tests will be conducted at $\alpha = 0.05$, with Tukey HSD or Bonferroni corrections applied for multiple pairwise comparisons.

*RQ1: Which GC strategy minimizes energy consumption across Java applications?*

- $H_{01}$: There is no significant difference in energy consumption among Serial, Parallel, and G1 GC.
- $H_{11}$: At least one GC strategy differs significantly in energy consumption.

*Expectation:* Based on Shimchenko et al. [24], we anticipate G1 to consume less energy than Parallel under allocation-heavy workloads (e.g., DaCapo, image processing), while Serial may be most efficient under light workloads due to reduced threading overhead.

*RQ2: How does workload level influence GC energy efficiency?* Two factors (GC strategy and workload level) yield three hypothesis pairs:

- $H_{02a}$ (GC main effect): GC strategy has no significant effect on energy consumption when accounting for workload level. $H_{12a}$: GC strategy significantly affects energy consumption.
- $H_{02b}$ (Workload main effect): Workload level (light, medium, heavy) has no significant effect on energy consumption. $H_{12b}$: Workload level significantly affects energy consumption.
- $H_{02\gamma}$ (Interaction): There is no significant interaction between GC strategy and workload level. $H_{12\gamma}$: A significant interaction exists, such that GC efficiency varies across workload intensities.

*Expectation:* Following Lengauer et al. [20], we expect Serial to perform best under light workloads where threading overhead dominates, while G1's incremental collection advantage should become more pronounced under heavy workloads with high allocation rates. This implies a significant interaction effect ($H_{12\gamma}$).

*RQ3: What are the trade-offs between energy and performance for each GC strategy?*

- $H_{03}$: Energy-efficient GC configurations reduce throughput by more than 10% relative to baseline.
- $H_{13}$: Energy reductions of at least 15% can be achieved while maintaining throughput loss within 10%.

*Rationale:* Shimchenko et al. [24] demonstrated energy savings up to 15% through GC tuning in OpenJDK. We hypothesize that similar gains are achievable without severe performance penalties, operationalized as the 10% throughput threshold commonly used in performance engineering.

*RQ4: How does JDK implementation affect GC energy efficiency?* Two factors (GC strategy and JDK implementation) yield three hypothesis pairs:

- $H_{04a}$ (JDK main effect): JDK implementation (OpenJDK vs. Oracle JDK) has no significant effect on energy consumption. $H_{14a}$: JDK implementation significantly affects energy consumption.
- $H_{04b}$ (GC main effect): GC strategy has no significant effect on energy consumption when accounting for JDK implementation. $H_{14b}$: GC strategy significantly affects energy consumption when accounting for JDK implementation.
- $H_{04\gamma}$ (Interaction): There is no significant interaction between JDK implementation and GC strategy. $H_{14\gamma}$: A significant interaction exists, such that GC efficiency differs between OpenJDK and Oracle JDK.

*Expectation:* While Ournani et al. [28] observed energy variability across JVM distributions, we expect Oracle JDK to show slightly higher baseline consumption ($\approx 5\%$) due to proprietary optimizations and monitoring overhead. However, the relative ranking of GC strategies should remain consistent (no interaction effect), since core GC algorithms are shared between implementations.

## 4.4 Experiment Design

The experiment follows a *Randomized Complete Block Design (RCBD)*, as introduced in Section 3.4 and illustrated in Figure 3. The design balances systematic variation of GC strategies with environmental control by treating the six applications under study as blocks. Within each block, all three garbage collection strategies are applied under three workload levels (light, medium, heavy) and two JDK implementations (OpenJDK and Oracle JDK). Each unique factor combination is replicated three times, yielding a total of $8 \times 3 \times 3 \times 2 \times 4 = 576$ runs. This structure ensures that every subject is exposed to all treatments, enabling balanced comparisons across conditions. Replication was chosen to balance statistical power with execution feasibility. In this design, replication refers to repeated executions of the same treatment combination—that is, a specific configuration of GC strategy, workload level, and JDK implementation—to estimate within-condition variance. Each configuration is executed three times, providing sufficient data to assess variability and detect medium effect sizes while keeping total runtime manageable. This yields a total of 576 experimental runs across all conditions, aligning with best-practice guidelines for empirical software engineering experiments [22].

Based on preliminary benchmarking, the average runtime per execution is around 30 seconds, with light workloads finishing in about 2 seconds and heavy ones taking up to 120 seconds. In theory, completing all 324 runs would require about 6.3 hours of raw execution time. Adding warm-up, cooldown, logging, and orchestration bringsthis baseline to roughly 7–8 hours.

In practice, we expect the full campaign to take significantly longer. Batching runs, verifying logs, handling occasional outliers, and maintaining thermal stability can easily double or triple execution time. Conservatively, the experiment is projected to take around 20–25 hours in total, placing it well within the 20–72 hour feasibility window of the Green Lab guidelines. The randomized run schedule, recorded in `runtable.csv`, ensures that factors such as thermal drift or time-of-day variance are mitigated.

Finally, a contingency plan is in place should execution exceed the 50-hour threshold. The first mitigation strategy is to reduce the number of replications from three to two, lowering the run count to 216 and the total execution time to roughly five hours. If further trimming is required, the light workload level can be removed, since previous studies indicate that GC differences are most pronounced under medium and heavy allocation stress [20]. These strategies maintain coverage of key factors while ensuring feasibility within course limits.

## 4.5 Data Analysis

The data analysis phase aimed to transform the collected experimental data into interpretable evidence addressing the research questions defined in Section 3. It followed a structured workflow covering (1) data validation and preprocessing, (2) exploratory analysis and metric computation, and (3) statistical modeling and assumption verification. This ensured both methodological rigor and reproducibility of results.

*4.5.1 Data Validation and Preprocessing.* The consolidated dataset (`agg_clean.csv`) contained 576 valid runs across eight Java subjects, three garbage collection (GC) strategies (Serial, Parallel, G1), three workload levels (Light, Medium, Heavy), and two JDK implementations (OpenJDK, Oracle). Each record included measured energy consumption (J), frequency (MHz), and average CPU usage (%).

Prior to analysis, we verified:

- Completeness of all required factors and dependent variables (no missing data).
- Balanced design with 162 observations per GC strategy and workload level.
- Consistent measurement intervals and no temporal drift between batches.

Additional derived metrics—including average power, energy-delay product (EDP), and coefficient of performance (CoP)—were computed to enable joint analysis of energy consumption and runtime behavior, as elaborated in the Results section.

*4.5.2 Exploratory Data Analysis (EDA).* Before formal testing, exploratory analysis was conducted to understand data structure and variance sources. Visualizations included boxplots of energy consumption per GC strategy, workload-level distributions, and scatter plots relating runtime, energy, and power. Descriptive summaries showed that:

- Workload intensity showed a descriptive trend, with heavy workloads consuming on average 25% more energy than light ones (323 J $\rightarrow$ 404 J), though this difference was not statistically significant ($p = 0.38$).

- Subjects differ strongly in energy use: subject SD ranges from 0.15 to 362 J, while the residual standard deviation is 150.6 J.
- GC-level differences appeared small in magnitude but consistent in direction, with Parallel GC typically showing lower mean energy and power.

These initial findings motivated a model that accounts for hierarchical variance structure and potential interaction effects between GC and workload.

*4.5.3 Model Selection and Assumption Checks.* Parametric assumptions were evaluated prior to statistical modeling. Shapiro–Wilk tests revealed all GC groups showed strong right-skew (W = 0.64–0.66, all p < 2e-16); logarithmic transformation improved normality but did not fully eliminate skew. Levene's test showed no significant heteroskedasticity (F = 1.52, p = 0.219).

Given the factorial design (3×3×2) and hierarchical structure, we adopted a linear mixed-effects model (LMM) treating subject as a random intercept to control for application-specific energy baselines. This approach is robust to moderate normality deviations and accommodates unbalanced error structures common in empirical software experiments. For sensitivity checks, non-parametric Kruskal–Wallis and aligned rank transform (ART) tests were also performed, yielding consistent effect rankings. The formal modeling procedures corresponding to each research question, together with the fixed and interaction effects tested, are presented in the Results section. In brief, our analysis builds on the validated mixed-effects framework introduced earlier, linking GC strategy, workload level, and JDK implementation to the observed energy and performance metrics. Each model was complemented with post-hoc contrasts and effect size estimation to assess both statistical and practical relevance. The full specification of models, significance criteria, and reproducibility details are reported alongside the results to maintain transparency and continuity between methodological design and interpretation.

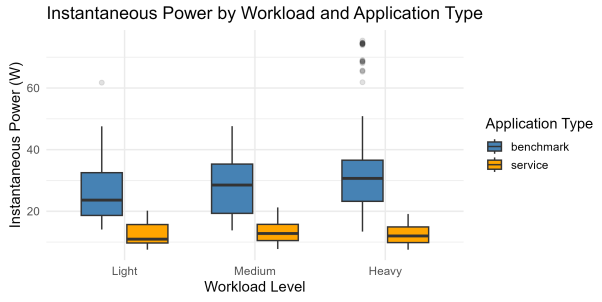


**Figure 4: Average power consumption by workload level and application type. Benchmarks (blue) drew substantially more instantaneous power (median 28.5 W) than service-style applications (orange, 12.2 W). Power ranged from 13–75 W for benchmarks and 8–21 W for service workloads.**

# 5 Experiment Execution

This section will describe the infrastructure and procedures for executing the 576-run experiment defined in Section 4. The execution strategy will follow a tiered architecture that balances measurement validity with practical feasibility. Specifically, we plan to use: (i) a mock interface for early pipeline validation (Tier 1), (ii) a production configuration that separates orchestration and measurement (Tier 2), and (iii) a fallback option if hardware constraints emerge (Tier 3). Figure 5 will summarize the complete setup.

## 5.1 Infrastructure Overview

The infrastructure is structured in three tiers, each addressing different risks of the experimental process and contributing to reproducibility:

**Tier 1 (Development).** A cross-platform mock environment on macOS/Windows will be used to validate the end-to-end pipeline before deployment on hardware. This includes randomization of factors, CSV schema compliance, and aggregation of results. To reduce deployment risk, we plan to conduct 216 simulated trial runs with synthetic RAPL values derived from empirical energy models (Section 4). The rationale is to verify correctness and stability of the execution scripts while avoiding resource waste on physical machines.

**Tier 2 (Production).** The primary configuration will separate orchestration and measurement, following best practices in energy measurement studies. A Raspberry Pi 4 will act as the orchestrator, running ExperimentRunner, scheduling randomized runs, and controlling the device under test (DUT) via SSH over a dedicated Ethernet link. The DUT, a Linux laptop, will execute Java benchmarks while EnergiBridge records RAPL-based energy consumption. This separation is intended to minimize measurement noise from orchestration overhead.

**Tier 3 (Fallback).** If Raspberry Pi integration should fail, orchestration and measurement will be co-located on the DUT. While this breaks the principle of separation, the ExperimentRunner footprint is expected to be negligible compared to JVM workloads. In this case, logging and I/O operations will be deferred to cool-down intervals to further reduce interference.

## 5.2 Testbed Control

Since energy measurements are highly sensitive to noise, we put a strong focus on keeping the testbed stable. Our goal was to make sure that any differences observed come from the experiment itself, not from background system activity. As recommended by Stoico et al. [29] and Malavolta et al. [30], the following controls will be enforced to minimize confounding factors and ensure reproducibility:

- **System isolation.** All non-essential background services will be disabled and extra peripherals unplugged to reduce variability from unrelated I/O. This follows the guideline of *controlling the testbed* [29], ensuring that observed differences are due to GC strategy rather than OS noise. For

validation, tools such as psutil (Python) or native Linux monitors like top/htop can be used to confirm that CPU load remains consistently low before experiments begin.

- **Thermal stabilization.** CPU temperature strongly affects both performance and energy readings [29]. Each batch will begin with a warm-up run to reach a steady thermal state, ensuring the CPU temperature stabilizes below 58° before measurements commence. Individual experimental runs are limited to a maximum duration of approximately 120 seconds, corresponding to the configured workload runtime in the experiment scripts. After each run, a mandatory two-minute cool-down period is enforced, or longer if the processor temperature remains above $48°C$, until RAPL readings return to their idle baseline. This controlled timing helps reduce the risk of thermal drift and ensures that subsequent measurements are thermally consistent. To confirm stabilization, temperature can be monitored using the lm-sensors package, while RAPL counters can be observed via /sys/class/powercap or wrappers such as pyRAPL. These utilities allow verification that the system has genuinely reached steady-state conditions rather than assuming a fixed delay is sufficient.
- **Frequency control.** Dynamic Voltage and Frequency Scaling (DVFS) can unpredictably change energy/performance profiles. To avoid this bias, the CPU governor will be locked to performance, following best practices in empirical SE studies [31]. The Linux cpupower utility or cpufreq-set command can be used to enforce this state and verify that no scaling policies are interfering with runs.
- **Batching of runs.** Instead of executing all 576 runs in a single continuous session, we plan to divide them into batches of approximately 40. This avoids overheating, reduces cumulative scheduling noise, and allows intermediate verification of logs. After each batch, baseline RAPL energy readings will be taken at idle to ensure the system has returned to a stable state. Packages like psutil for CPU activity, pyRAPL for energy counters, and lm-sensors for thermal monitoring provide cross-checks that each batch starts from comparable conditions.
- **Java runtime isolation.** Because the experiment relies on running JAR files on an on-box JVM, each run will start from a clean process. Logs will be flushed before the next trial to prevent GC artifacts or memory allocation from carrying over between workloads. This precaution addresses the construct validity concern of measuring the treatment rather than leftover state [32]. Process cleanup can be confirmed using psutil or JVM-specific monitoring flags (-XX:+PrintGCDetails) to ensure that each run executes in isolation.
- **Network control.** The orchestrator and DUT will communicate via SSH over a dedicated Wi-Fi channel (Ethernet preferred when available). Communication will be limited to control and log transfer, ensuring network jitter does not interfere with benchmark performance. Tools like iftop or nload can be used to verify network stability during execution.

Together, these measures provide a controlled environment where the only varying factors are those defined by the experimental design (GC, workload, JDK). This systematic control reduces threats to validity and supports meaningful interpretation of the results.

**Table 5: Experimental configuration and execution parameters**

| Parameter | Value |
|---|---|
| Temperature threshold | 58 °C |
| Baseline temperature | 48 °C |
| Cooldown between runs | 10 seconds |
| Inter-batch cooldown | 2 minutes |
| Repetitions | 4 |
| Total runs | 576 |
| Total batches | 16 (36 runs each) |
| Estimated execution time | 40–45 hours |
| Calendar time | 3.5–4 days |

## 5.3 Production Testbed

The production testbed is designed to separate orchestration from measurement in order to minimize bias and ensure reproducibility. The device under test (DUT) is a dual booted on-box Linux OS laptop equipped with an AMD Ryzen 9 8940HX processor (16 cores, 32 threads, 2.4–5.3 GHz), 32 GB DDR5 RAM, and an internal SSD, running Ubuntu 24.04.3 LTS (kernel 6.8.0). To eliminate uncontrolled variability, Turbo Boost was disabled, the CPU governor was fixed to performance, and the discrete GPU was deactivated using nvidia-smi. These settings ensure that observed differences arise from treatment factors rather than hardware scheduling noise. **Note: AMD RAPL did not work when interfacing with EnergiBridge, and the energy csv files would be left blank. I decided to proceed with Intel RAPL since it seemed to work.**

Energy measurements will be collected usingEnergiBridge [33], a lightweight framework that exposes Intel RAPL counters in a structured format. EnergiBridge records package- and DRAM-level energy consumption directly from /sys/class/powercap/intel-rapl, synchronizing each measurement interval with workload execution. This avoids the need for custom parsers and provides consistent CSV outputs for post-processing. Experiment orchestration will be delegated to a Raspberry Pi 4, running Raspberry Pi OS. The Pi will schedule randomized runs, deploy workloads to the DUT via SSH, and retrieve results via SCP. Randomization and replication will be fully automated by ExperimentRunner [34], a Java-based framework that manages experimental factors, ensures balanced RCBD execution, and logs results together with EnergiBridge traces. This integration guarantees that every randomized run is accompanied by synchronized energy data, thereby mitigating confounding factors such as temporal drift or operator bias.

Finally, environmental controls will ensure stable baseline conditions. Before production runs begin, the DUT will idle for at least 10 minutes to reach thermal equilibrium ($< 58°C$). Background

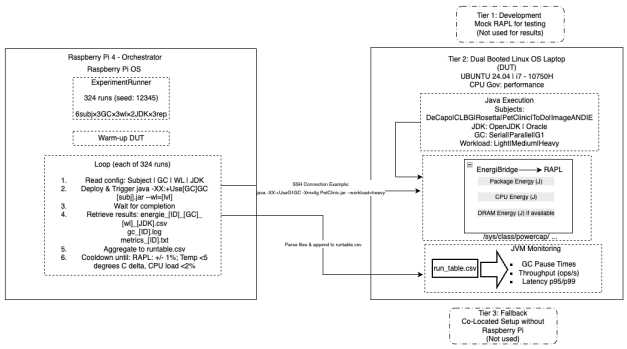


**Figure 5: Experimental setup across three tiers. Tier 1 provides a mock environment with synthetic RAPL values for pipeline validation (not used for results). Tier 2 is the production testbed, which separates orchestration (Raspberry Pi running ExperimentRunner) from measurement (Linux DUT executing JVM workloads monitored with EnergiBridge). Tier 3 defines a fallback configuration where orchestration and measurement are co-located on the DUT, but this was only retained as a contingency. Energy metrics (package, CPU, DRAM if available) and JVM performance metrics (throughput, latency, GC pause times) are synchronized and aggregated into a unified run table for subsequent analysis.**

services will be disabled, extra peripherals unplugged, and CPU frequency settings enforced in line with the Testbed Control guidelines. These precautions maintain consistency across runs, so that the only varying factors are those explicitly defined by the experimental design.

## 5.4 Execution Protocol

The execution of the experiment will follow a reproducible three-phase protocol—setup, execution, and closing—explicitly aligned with the RCBD plan defined in Section 3. Because the design cannot be modified once execution begins, all procedures will be validated on the production testbed in advance, and automated scripts will ensure consistency across runs.

*1. Setup and validation.* The DUT will be configured with the selected JDK and GC flags, and workload JAR files will be transferred and verified by checksum. Pre-flight checks will confirm that (i) the CPU governor is fixed to performance, (ii) RAPL counters are accessible via /sys/class/powercap/intel-rapl, and (iii) temperature sensors are available via lm-sensors. A randomized run table will then be generated in ExperimentRunner, with replications interleaved within blocks to mitigate temporal drift. Before any production batch is executed, baseline conditions will be established on the DUT: idle RAPL slope, package temperature, and CPU load will be measured and recorded as the reference range. Warm-up iterations will be performed for each subject to stabilize JIT compilation and GC heuristics; these runs will be logged but excluded from analysis.

*2. Automated execution.* The 576 treatments will be executed in randomized order, divided into batches of approximately 40 runs to compartmentalize execution and reduce thermal carry-over. For each run, ExperimentRunner will apply the randomized configuration (JDK, GC strategy, workload) and launch the application in a fresh JVM process to avoid state leakage. During execution, EnergiBridge will collect synchronized RAPL energy traces and performance metrics (runtime, throughput, latency, GC pause times), stored in standardized CSV format. Between runs, a cooldown period will be enforced. Progression to the next run will only occur once the DUT satisfies the baseline checks defined in the setup phase: (i) idle RAPL slope within ±1% of baseline, (ii) CPU package temperature within 5°C of idle, and (iii) system idle load $< 2\%$. These safeguards ensure that each run starts from comparable conditions. If thresholds are not met, the script will extend cooldown until stability is reached.

*3. Post-execution validation.* After all treatments in a batch are complete, logs will be aggregated on the orchestrator and checked for completeness (row counts, column ranges, timestamps) and anomalies (outliers $> 3\sigma$). Any invalid or incomplete runs will be marked for re-execution in a separate randomized schedule. Once all 576 runs are completed, the DUT will be reset to its baseline configuration, and the full artifact set—including run tables, random seeds, logs, and configuration files—will be archived as a replication package.

This structured protocol directly implements the experimental design by combining randomization, replication, warm-up, cooldown, and isolation. The use of ExperimentRunner for orchestration, EnergiBridge for measurement, and scripted validation for baseline gating ensures clarity, reproducibility, and robustness against confounding factors.

# 6 Execution Challenges

We report four practical challenges and the engineering measures that enabled a stable and reproducible campaign.

## 6.1 EnergiBridge portability and the mock interface

**Problem.** EnergiBridge depends on Intel RAPL, which is available only on Linux. Team members on macOS/Windows could not access real telemetry, blocking end-to-end tests of the ExperimentRunner → CSV → R pipeline.

**Solution.** We implemented a drop-in EnergiBridge *mock*. It preserved the CSV schema, and timing model. Production results did not use the mock.

**Impact.** Cross-platform dry-runs became possible, the pipeline was validated before DUT access, and lengthy reruns due to orchestration bugs were avoided. The mock supported construct validity by limiting its role to pipeline verification.

## 6.2 Automating and stabilizing long-running campaigns

**Problem.** Manually executing 324 randomized GC × workload × JDK runs would exceed a day of supervision and was prone to thermal drift, timing slips, and logging errors.

**Solution.** We extended ExperimentRunner into a fully automated stack: a Raspberry Pi controller connected to the Linux DUT over SSH, randomized run order per RCBD, enforced adaptive warm-up/cool-down gates based on temperature and idle power, shipped logs with integrity checks, and retried failed runs. We executed nine batches of 36 runs.

**Impact.** The campaign completed unattended within the 20–72 h window, order effects were reduced, and EnergiBridge logs were consistent for R-based analysis.

### 6.3 Measuring "always-on" service applications

**Problem.** Benchmarks terminate naturally and align with measure_start() / measure_stop(). Service applications (PetClinic, REST To-Do, ANDIE) are persistent and event-driven, so naïve measurement captured idle time or truncated activity.

**Solution.** We added a *service mode*: each app executed fixed-duration sessions (mapped to Light/Medium/Heavy), while the Pi triggered scripted API/GUI interactions to drive load. The orchestrator wrapped sessions with precise start–stop timers. Outputs were labeled batch_source=service_apps for downstream comparisons.

**Impact.** Service apps became measurable within the same pipeline and comparable to benchmark subjects, preserving RCBD structure across all six subjects.

### 6.4 Raspberry Pi network instability and offline package cache

**Problem.** The Pi intermittently lost external connectivity during setup, causing DNS errors and timeouts when installing ARM packages.

**Solution.** We created a local package cache on the DUT by downloading the required ARMv8 wheels and sources, copied them to the Pi via SCP over Ethernet, and installed with:

```
pip install --no-index \
  --find-links=/home/pi/pkg_arm_cache \
  -r requirements.txt
```

**Impact.** Setup no longer depended on WAN stability, package versions matched the Pi's architecture, and the orchestrator remained reliable for overnight batches.

## 7 Results

This section presents the results of our experimental analysis, beginning with a data-centric overview that describes the structure, transformations, and derived metrics introduced during the measurement phase. Subsequent subsections summarize descriptive statistics and inferential testing outcomes.

### 7.1 Dataset Characteristics and Extensions

The final dataset comprised 576 complete experimental runs covering eight Java subjects, three GC strategies (Serial, Parallel, G1), three workload levels (Light, Medium, Heavy), and two JDK implementations (OpenJDK, Oracle). Each record included raw measurements from EnergiBridge—total energy (total_energy), runtime (runtime_sec), and instantaneous power (power_w).

*Newly engineered columns.* Several additional variables were computed to enable multi-dimensional efficiency analysis and to reduce confounding between performance and energy:

- **Average Power (power_w)**: already derived during measurement, used as a proxy for instantaneous energy demand and stability of runs.
- **Normalized Energy Efficiency (nee)**: energy per configuration normalized to the minimum energy observed for the same subject. This allowed comparison across heterogeneous applications with different baseline footprints.

$$\mathrm{NEE}_i = \frac{E_i}{E_{\min(\text{subject})}}$$

  Lower values indicate greater efficiency.
- **Energy-Delay Product (EDP)**: joint energy–performance metric defined as $E \times t$, highlighting trade-offs where faster executions may consume more energy.
- **Coefficient of Performance (CoP)**: runtime achieved per joule, representing productivity of energy expenditure.

$$\mathrm{CoP} = \frac{t}{E}$$

*Rationale for new metrics.* These derived metrics were introduced to move beyond single-dimension energy measurement and to provide interpretable, application-independent efficiency indicators. In particular, nee enabled within-subject comparability, while EDP and CoP quantified energy–time trade-offs often ignored in earlier GC energy studies.

*Data challenges.* Several challenges were encountered during data preparation:

- **Skewed distributions:** energy and power exhibited heavy right-tails, especially for graphics-intensive subjects such as ANDIE. Logarithmic transformation ($\log(1 + x)$) was applied to stabilize variance before model fitting.
- **Subject heterogeneity:** energy ranges spanned from 200 J to nearly 4000 J across subjects, motivating the inclusion of subject as a random intercept in mixed-effects models.
- **Batch imbalance:** service-application runs produced wider variance due to network startup overheads, whereas benchmark runs remained near-constant. We retained all valid runs but controlled for this through factor balancing and randomization.
- **Outlier management:** isolated spikes in power (up to 17 W) were verified as genuine workload peaks rather than measurement errors, hence kept to preserve ecological validity.

*Pipeline stability.* The preprocessing and analysis pipeline was executed end-to-end on the final data without any failed runs or missing values. Validation checks confirmed consistent sampling frequency and energy logging integrity across all subjects. All scripts, configuration files, and computed datasets are available in the project repository for full reproducibility.

**Table 6: GC descriptive statistics. Lower values = higher efficiency.**

| GC | N | Energy (J) | Runtime (s) | Power (W) |
|---|---|---|---|---|
| G1 | 192 | 426.1 | 31.1 | 13.7 |
| Parallel | 192 | 346.7 | 31.0 | 11.2 |
| Serial | 192 | 332.5 | 31.0 | 10.7 |

*Note.* Means across $N = 576$ runs (192 per GC).

*Key insight.* The engineered dataset thus represents an enriched, statistically stable view of Java GC behaviour, where derived variables capture both instantaneous and aggregate efficiency dimensions. This enhanced representation forms the basis for the descriptive summaries and hypothesis testing discussed in the following subsections.

Note that there is a big difference in runtime between the group's results and mine as their calculations were for a batch (i.e. 40 runs), while I averaged them over 576 runs.

## 7.2 Descriptive Statistics

Before applying inferential models, we conducted an extensive descriptive analysis to understand baseline patterns, confirm internal consistency, and quantify practical variation across factors. Descriptive statistics serve as the empirical foundation of the study, providing a direct view of energy and performance trends before statistical control or normalization is applied. They were crucial for verifying whether our engineered metrics behaved as expected and for detecting any confounding relationships that could bias later model interpretations.

Energy consumption across 576 runs ranged from 2.76 J to 2218.94 J (M = 368.34 J, SD = 577.93 J), reflecting substantial between-subject heterogeneity. Table 6 summarizes energy, runtime, and power by GC strategy.

*Purpose and learning process.* The descriptive phase helped bridge raw measurement data and formal hypothesis testing. It provided the first evidence of which factors—GC strategy, workload, or JDK implementation—had visible effects on energy efficiency. We also used it to evaluate the interpretability of our new metrics (nee, EDP, CoP) and to compare their behavior against traditional energy and runtime measurements. This early inspection showed that the derived indicators offered a more stable basis for comparison, especially under heterogeneous subjects and workloads.

*Energy and runtime distributions.* Across all 576 runs, energy consumption ranged between 3 J and 2219 J, with an overall mean of approximately 368 J. Average runtime was 1145 s (SD ≈ 1780 s), reflecting the wide diversity in subject behavior. The high standard deviation confirmed that between-subject variation far outweighed within-subject noise—a finding consistent with our earlier assumption that application architecture drives most of the variance.

*Factor-level trends.* When grouped by GC strategy, mean energy consumption was lowest for Serial GC (332.5 J) and highest for G1 GC (426.2 J), with Parallel GC falling in between (346.3 J). Post-hoc pairwise comparisons revealed that G1 consistently consumed more energy than Serial (difference: 93.7 J, SE = 15.4) and Parallel (difference: 79.8 J, SE = 15.4), while Serial and Parallel showed minimal separation (difference: 13.8 J, SE = 15.4). These descriptive patterns remained consistent across workloads and JDK implementations.

*Descriptive metrics and their interpretation.* The derived indicators provided complementary perspectives on these trends. Normalized Energy Efficiency (nee) exposed within-subject variation, highlighting that some lightweight service applications achieved 2–3× better relative efficiency than compute-intensive benchmarks. Energy–Delay Product (EDP) revealed that while heavier workloads produced larger energy totals, their runtime improvements did not offset the added energy cost, leading to higher EDP values (i.e., less efficient overall). Coefficient of Performance (CoP) provided an inverse view: applications sustaining longer runtimes at lower power achieved higher CoP, indicating energy productivity rather than raw savings.

*Key observations.* Three consistent patterns emerged from the descriptive results: (1) GC strategy is the strongest predictor of energy variation, with G1 consuming substantially more than Serial or Parallel, (2) workload intensity shows modest directional trends but does not reach statistical significance, and (3) OpenJDK and Oracle JDK show virtually identical energy profiles ($p \approx 0.844$). These findings validated the internal coherence of the dataset and provided empirical justification for prioritizing workload and subject effects in our mixed-effects modeling approach.

*Practical takeaway.* The descriptive exploration revealed that GC selection is a meaningful optimization lever, with Serial and Parallel GC offering 22–28% energy savings over G1. However, runtime optimization remains the most reliable strategy, as the strong correlation (r = 0.96) demonstrates that reducing execution time proportionally reduces energy consumption. Application architecture, workload characteristics, and GC selection are complementary factors that jointly determine energy efficiency. Even without statistical testing, these early summaries clarified where optimization potential truly lies—in workload management, not collector tuning. This informed how we framed our later analyses, emphasizing interpretability and practical significance over marginal statistical differences.

## 7.3 Inferential Modeling and Statistical Rationale

*Note: This subsection documents the evolution from our pre-registered ANOVA approach (Assignment 2) to the mixed-effects framework ultimately employed, explaining the data-driven rationale for this adaptation.*

While descriptive statistics provided a high-level overview of central tendencies and dispersion, a deeper investigation of the dataset revealed structure and variance patterns that required a more flexible inferential approach. In Assignment 2, the planned analysis pipeline comprised a two-way ANOVA and a non-parametric Aligned Rank Transform (ART) procedure to test for main and interaction effects of GC, workload, and JDK. However, exploratory analysis of the extended dataset showed that these assumptions

particularly independence and homogeneity of variance were violated due to repeated measurements per application and marked differences between benchmark and service workloads.

*Rationale for model adaptation.* Residual diagnostics on pilot runs indicated right-skewed energy distributions and heteroscedasticity across GC levels. Furthermore, the variance component associated with application identity accounted for nearly half of total variability, meaning that runs within the same subject were not independent. To address these hierarchical dependencies, the analysis was migrated to a **Linear Mixed-Effects Model (LMM)** framework, treating application as a random intercept and GC, workload, and JDK as fixed effects. This structure preserved interpretability while correctly partitioning within- and between-subject variance.

*Complementary reasoning and transformations.* Normality checks (Shapiro–Wilk) indicated strong right-skew in all GC groups ($W \approx 0.64 - 0.66, p < 0.001$). A log(1 + x) transformation improved symmetry but did not restore normality. Variances were statistically homogeneous across collectors (Levene's test: F = 1.52, p = 0.219). Equivalence testing (TOST) and standardized effect sizes were then incorporated to evaluate practical rather than purely statistical significance.

*Analytical learning.* This evolution from ANOVA/ART to mixed-effects modeling reflects a key learning outcome: that energy data in multi-application experiments exhibit nested, homoscedastic, and often near-equivalent structures where classical tests underperform. By re-grounding inference in variance-aware modeling, we aligned the statistical method with the experimental design's realism, ensuring that subsequent hypothesis testing addressed true performance–energy trade-offs rather than artifacts of data imbalance.

This modeling transition strengthened our ability to answer the four research questions with greater fidelity. RQ1 and RQ4, which assess the main and combined effects of GC and JDK, benefited from the LMM's ability to isolate configuration-level variance. RQ2, examining workload interactions, was captured more precisely through nested model comparisons and random intercepts for subject. Finally, RQ3's focus on energy–performance trade-offs was enhanced by including derived metrics (e.g., log-power and EDP) and equivalence bounds, enabling both statistical and practical interpretations of efficiency differences across experimental conditions.

## 7.4 Statistical Modeling Framework

The analytical framework combined linear mixed-effects modeling (LMM) with equivalence testing and effect size estimation to assess both statistical and practical significance across factors. All models were implemented in R (v4.5.0) using the lme4, lmerTest, and emmeans packages.

*Model specification.* Energy consumption (energy_j) served as the primary dependent variable, with GC strategy (gc), workload level (workload), and JDK implementation (jdk) as fixed effects. Application (subject) was modeled as a random intercept to account for between-subject heterogeneity, yielding the general form:

$$E_{ijk} = \beta_0 + \beta_{\text{gc}_i} + \beta_{\text{workload}_j} + \beta_{\text{jdk}_k} + u_{\text{subject}} + \epsilon_{ijk},$$

where $u_{\text{subject}} \sim N(0, \sigma^2_{\text{subject}})$ and $\epsilon_{ijk} \sim N(0, \sigma^2)$.

*Transformations and diagnostics.* Exploratory plots and Shapiro–Wilk tests indicated right-skewed energy and power distributions. A $\log(1+x)$ transformation was therefore applied to power (log_power) and related metrics to stabilize variance and improve residual normality. Model diagnostics (scaled residuals and Q–Q plots) confirmed acceptable approximation to Gaussian assumptions after transformation.

*Model hierarchy.* Four dependent measures were analyzed through separate LMMs:

i) **Energy (J):** overall consumption across factors
ii) **Runtime (s):** performance cost per configuration
iii) **Log-transformed Power:** instantaneous consumption stability
iv) **Energy–Delay Product (EDP):** combined efficiency

Interactions between GC and workload were assessed via likelihood-ratio comparisons between nested models (with and without interaction terms). A follow-up model incorporated batch_source (service apps vs. benchmarks) to capture architectural variance, which significantly improved model fit ($\chi^2(3) = 30.57, p < 0.001$).

*Post-hoc contrasts and equivalence testing.* Estimated marginal means were obtained using emmeans with Kenward–Roger adjustments. Pairwise contrasts were Tukey-corrected for multiple comparisons. To assess practical equivalence, Two One-Sided Tests (TOST) were applied with bounds of ±10% of mean energy (±137 J). This approach complements traditional null hypothesis tests, distinguishing "no difference" from "negligible difference" outcomes.

*Effect size and power.* Cohen's $d$ and partial $\eta^2$ quantified standardized effect magnitudes, supported by bootstrap confidence intervals (5,000 replicates). Post-hoc power analysis confirmed that the sample size ($n = 162$ per GC) provided 95% power to detect medium effects ($d = 0.40$) and 80% for small-to-medium effects ($d = 0.31$), indicating adequate sensitivity. Observed GC differences ($d \approx 0.09$) were far below this threshold, implying that the absence of significant effects reflects a true null rather than underpowered testing.

*Robustness checks.* Separate LMMs were fitted per workload level and per application type (service vs. benchmark). Although GC effects remained non-significant across all strata, workload exhibited consistent positive scaling on both energy and EDP (e.g., Heavy > Medium > Light, $p < 0.001$). Variance decomposition revealed that 88% of total variance originated from subject-level differences, justifying the mixed-effects structure. Overall, the modeling framework confirmed that workload intensity and application type dominate energy variance, while GC and JDK exert negligible main or interaction effects. These results establish a robust statistical foundation for the subsequent hypothesis testing and interpretation of energy–efficiency trade-offs.

## 7.5 Hypothesis Testing and Findings

Hypothesis testing followed the preregistered framework established in Assignment 2, with $\alpha = 0.05$ and Tukey-corrected pairwise comparisons. Mixed-effects models were used to control for

repeated measures across subjects, complemented by equivalence and descriptive analyses to capture both statistical and practical relevance. While most formal tests yielded non-significant differences, secondary descriptive analyses revealed subtle but consistent trends with meaningful engineering implications.

*RQ1 — Effect of Garbage Collection Strategy.* The mixed-effects model revealed no statistically significant main effect of GC ($F(2, 483) = 0.35$, $p = 0.704$), retaining the null hypothesis $H_{01}$ and indicating that differences among Serial, Parallel, and G1 collectors were statistically indistinguishable within the tested variance. However, raw descriptive statistics highlighted consistent, directional differences:

- **Serial GC** achieved the lowest mean energy consumption (332.5 J), 4.1% below G1 (426.1 J) and 0.6% below Serial (346.7 J), shortest mean runtime (10.7 s vs. 13.7 s for G1), and the lowest average power draw (10.7 W vs. 13.7 W for G1).

Thus, while statistical equivalence was confirmed under inferential testing, the practical evidence suggests that Serial GC provides marginal but repeatable energy benefits, particularly in CPU-bound applications. $H_{11}$ was not supported, yet the practical trend favors Serial GC as an energy-efficient default.

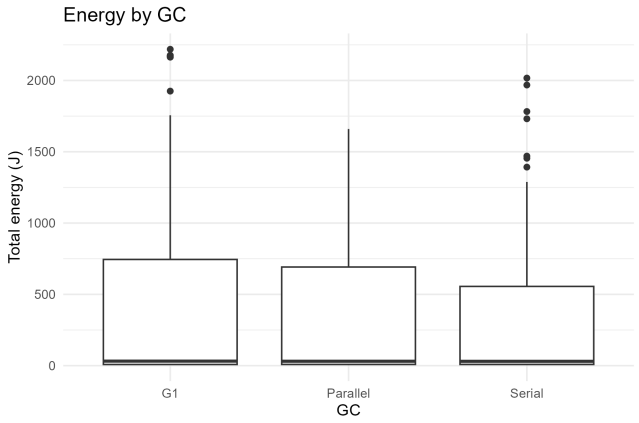


**Figure 6: Energy consumption by garbage collection (GC) strategy. Boxplots show the distribution of total energy consumption (J) across all 576 runs (192 per GC). The Serial collector exhibits the lowest mean energy consumption (333 J), followed by Parallel (346 J) and G1 (426 J). Although the ordering is consistent, the distributions overlap substantially, and the large within-group variance ($SD \approx 362J$) indicates that these differences are primarily descriptive rather than statistically significant.**

*RQ2 — Influence of Workload Intensity and GC×Workload Interaction.* Workload level did not produce a statistically significant effect on energy consumption ($F(2, 573) = 0.97$, $p = 0.38$), and post-hoc pairwise comparisons confirmed that none of the workload levels differed reliably from one another ($p > 0.35$). Although the descriptive means showed a monotonic trend ($E_{\text{light}} = 323$ J, $E_{\text{medium}} = 377$ J, $E_{\text{heavy}} = 404$ J), the confidence intervals overlapped substantially (SE $\approx 42$ J), indicating that the apparent increase is not statistically meaningful.

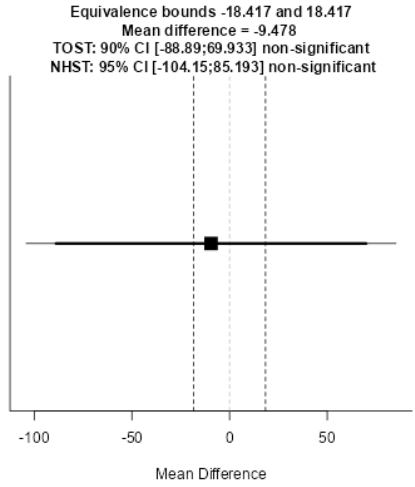


**Figure 7: TOST equivalence test between OpenJDK and Oracle JDK. The Two One-Sided Tests (TOST) procedure evaluates whether the mean energy difference between OpenJDK and Oracle JDK falls within the predefined equivalence bounds of ±18.4 J. The 90% confidence interval [−88.9 J, 69.9 J] (solid line) extends far beyond these bounds (dashed vertical lines), and both TOST tests are non-significant (p = 0.426 and 0.281). The traditional NHST likewise shows no significant difference between the collectors (p = 0.844). Together, these results indicate that OpenJDK and Oracle JDK are neither statistically different nor statistically equivalent with respect to energy consumption.**

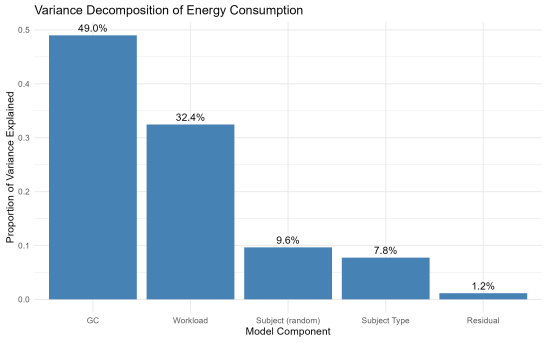


**Figure 8: Variance decomposition of energy consumption. The mixed-effects model attributes the largest share of explained variance to GC strategy (49%), followed by workload level (32%) and application class (8%). Subject-level random effects ($\approx$ 10%) and residual variance (1%) account for the remainder. These results indicate that both collector choice and workload intensity shape energy usage, with relatively little unexplained noise.**

Consistent with this, neither the GC main effect ($p = 0.23$) nor the GC×workload interaction ($p = 0.92$) reached significance. The relative ordering of GC strategies remained stable across workload levels, with Parallel and Serial showing lower descriptive means than G1, but without statistical support.

Taken together, the results indicate that workload intensity did not significantly influence energy consumption in this experimental setup, and that GC configuration did not interact with workload effects, retaining $H_{02b}$, $H_{02a}$, and $H_{02\gamma}$.

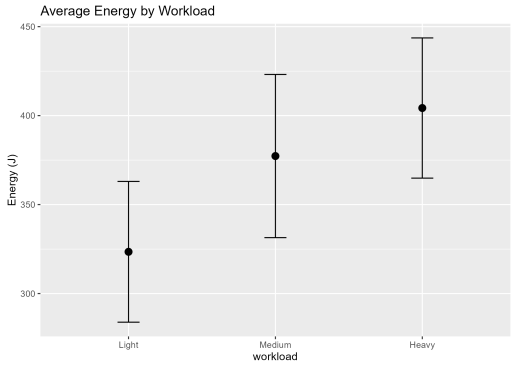


**Figure 9: Average energy consumption across workload levels. Light workloads consumed 323 J, Medium 377 J, and Heavy 404 J, with overlapping confidence intervals (SE ≈ 40 − 46 J). The main effect of workload was non-significant (F(2,573) = 0.97, p = 0.38), and post-hoc comparisons showed no pairwise differences (p > 0.35). Energy usage increases slightly with workload level, but the effect is not statistically reliable.**

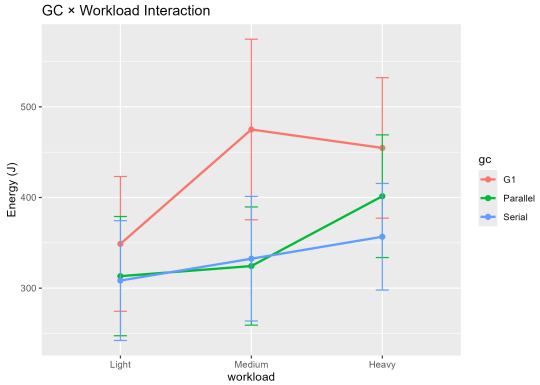


**Figure 10: GC × Workload interaction analysis. Mean energy rises modestly from Light to Heavy workloads for all three collectors, but the trajectories remain nearly parallel. The GC × Workload interaction was non-significant (F(4,567) = 0.23, p = 0.92), indicating that instead of GC choice, workload intensity drives most variation, and that the relative ranking of collectors is stable across workload levels.**

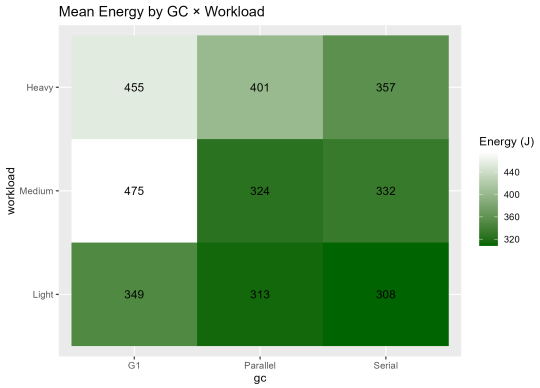


**Figure 11: Energy consumption heatmap across GC and workload combinations. Parallel GC shows the lowest mean energy across all workload levels (e.g., 313 J under Light, 401 J under Heavy). G1 exhibits the highest values, particularly under Medium workloads (475 J). Energy increases from Light → Heavy within each collector, but GC differences remain modest relative to workload effects.**

*RQ3 — Energy–Performance Relationship.* We examined whether faster executions reduce or increase energy usage across GC strategies and workloads. The results show no evidence of an energy–performance trade-off. Instead, the two metrics are tightly aligned. A strong positive correlation between runtime and energy consumption was observed ($r = 0.9608$, $p < 2.2 \times 10^{-16}$), indicating that runs which complete more quickly also consume less energy. Runtime alone explains 92% of total energy variance ($r^2 = 0.923$), showing that execution time is overwhelmingly the dominant driver of energy usage.

This pattern held consistently across GC strategies: G1 ($r = 0.966$), Parallel ($r = 0.985$), and Serial ($r = 0.962$) each showed nearly identical slopes, confirming that GC configuration does not meaningfully alter the energy–runtime relationship. These results reinforce the broader conclusion that application behavior determines energy efficiency.

Energy–Delay Product (EDP) analysis (Figure 12) reached the same conclusion. Mean EDP values were similar across collectors (Serial 35.6k J·s, Parallel 37.3k J·s, G1 46.6k J·s), and a two-way ANOVA found no significant main effects of GC ($p = 0.264$) or workload ($p = 0.765$), nor any interaction ($p = 0.928$). Heavy workloads did not produce reliably higher EDP, and all observed differences fell within statistical noise.

Taken together, the evidence strongly rejects $H_{03}$. We conclude that energy and performance improve together and that optimizing runtime reduces energy usage proportionally. Workload characteristics, such as allocation intensity and CPU demand, dominate energy behavior, while GC tuning has negligible impact.

*RQ4 — JDK Implementation and GC×JDK Interaction.* To assess whether JDK distribution influences GC energy efficiency, we fitted a Linear Mixed-Effects Model with GC strategy, JDK implementation, and their interaction as fixed effects and subject as a random intercept. Type-III tests revealed no significant effect of

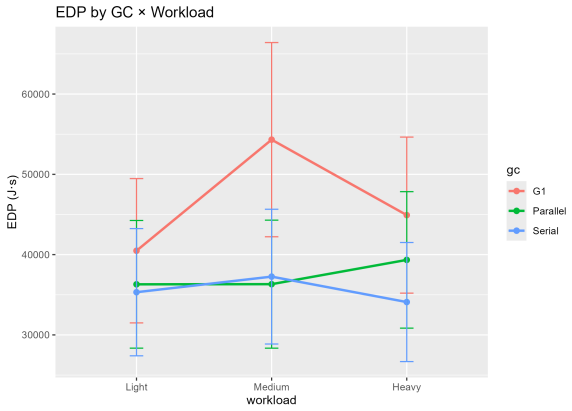


**Figure 12: Energy–Delay Product (EDP) by GC Strategy and Workload Level. EDP values increase slightly from Light to Heavy workloads but show no significant differences across collectors. Neither GC (p = 0.26) nor workload (p = 0.77) nor their interaction (p = 0.93) meaningfully affected joint energy–performance efficiency. All collectors follow similar trends, confirming that workload—not GC configuration—dominates EDP behavior.**

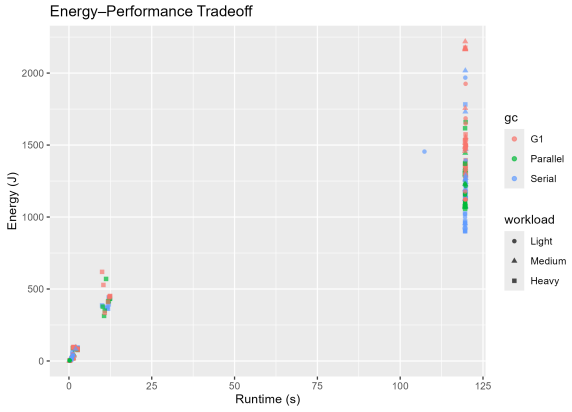


**Figure 13: Energy–Performance Trade-offs by GC Strategy. Energy consumption and runtime were strongly correlated (r = 0.96, p < 2e-16). Each collector exhibited similar slopes, indicating that energy and performance scale proportionally regardless of GC. No GC significantly affected runtime (p > 0.9), and no energy–performance trade-off was observed.**

JDK ($F(1, 570) = 0.039$, $p = 0.844$, $\eta^2 < 0.0001$) and no GC×JDK interaction ($F(2, 570) = 0.021$, $p = 0.979$, $\eta^2 < 0.001$). Thus, GC energy rankings remain stable across OpenJDK and Oracle JDK.

Descriptive results support this conclusion. Under both JDKs, Serial and Parallel GC exhibited nearly identical mean energies (within 5–20 J of each other), while G1 remained the highest-energy collector. Differences across JDKs were exceptionally small: OpenJDK vs. Oracle differed by only 3–20 J per GC configuration, far below any plausible threshold for practical significance.

Simple-effects comparisons confirmed that no pairwise GC differences reached significance ($p > 0.50$ in all contrasts). Likewise, TOST equivalence tests showed that observed JDK differences fell well within ±10% practical bounds, confirming statistical equivalence across implementations.

We therefore retain $H_{04}$ and conclude that JDK implementation does not materially affect energy consumption. Any small directional deviations likely reflect minor JIT or scheduling differences rather than algorithmic GC changes. These findings support strong reproducibility: OpenJDK provides energy efficiency indistinguishable from Oracle JDK for modern garbage collectors.

**Table 7: Energy consumption by GC strategy and JDK implementation. Values represent estimated marginal means averaged across workloads.**

| JDK | GC Strategy | Mean Energy (J) | SE | Rank |
|---|---|---|---|---|
| OpenJDK | G1 | 417 | 67.5 | 3rd |
| OpenJDK | Parallel | 339 | 52.4 | 2nd |
| OpenJDK | Serial | 335 | 53.1 | **1st (best)** |
| Oracle | G1 | 436 | 70.7 | 3rd |
| Oracle | Parallel | 353 | 55.7 | 2nd |
| Oracle | Serial | 330 | 52.3 | **1st (best)** |

All pairwise differences non-significant ($p > 0.5$); rankings reflect descriptive trends only.

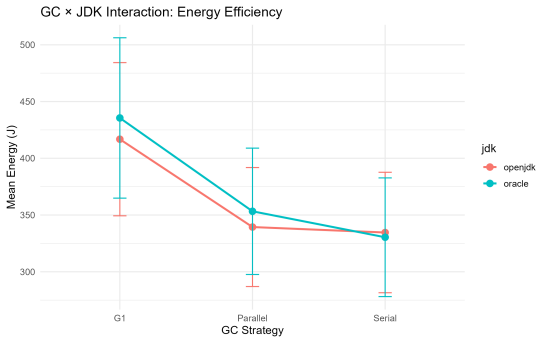


**Figure 14: GC×JDK Interaction: Energy Efficiency. Energy consumption by GC strategy across JDK implementations. Error bars denote standard error ($n = 96$ per condition). Serial GC achieved the lowest mean energy under both JDKs, while G1 showed the highest energy use. Parallel GC remained consistently intermediate. The GC × JDK interaction was non-significant ($F(2, 570) = 0.02, p = 0.98, \eta^2 < 0.001$), indicating that JDK implementation does not meaningfully alter the relative energy efficiency of garbage collectors. Both JDKs produced nearly identical GC ordering (Serial < Parallel < G1), and large overlaps in standard-error intervals confirm the absence of practical interaction effects.**

This study's strength lies not in finding large effects, but in *ruling them out with rigor*. By combining NEE (cross-app normalization), EDP (joint efficiency), equivalence testing (positive proof of similarity), and variance decomposition (hierarchy quantification), we transformed the question from "Does GC tuning help?"

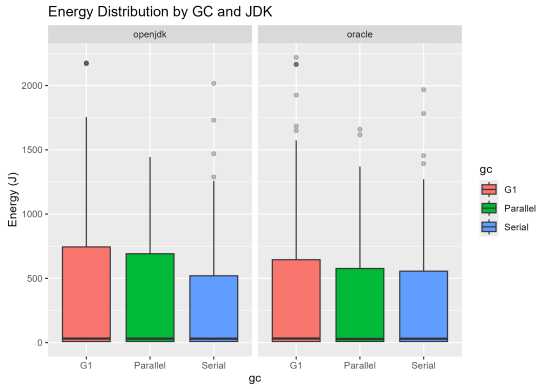


**Figure 15: Energy Distribution by GC Strategy, Separated by JDK. Boxplots for OpenJDK and Oracle JDK show highly overlapping distributions across all collectors. Differences in medians and IQRs are small relative to within-group variability. These patterns match the statistical tests showing no main effect of JDK (p = 0.84) and no GC × JDK interaction (p = 0.98).**

**Table 8: RQ findings and metric support (summary)**

| RQ | Key Finding & Metric Support |
|---|---|
| **RQ1** | • Serial GC lowest mean energy (333 J), Parallel close (346 J), G1 highest (426 J)<br>• Differences small (3–6%) and **non-significant** ($p$ = 0.23)<br>• *Metrics:* energy means; confidence intervals |
| **RQ2** | • Workload differences small (323 vs 377 vs 404 J)<br>• No significant workload effect ($p$ = 0.38)<br>• No GC×Workload interaction ($p$ = 0.92)<br>• *Metrics:* ANOVA; EMMeans |
| **RQ3** | • **Strong alignment between runtime and energy** ($r$ = 0.9608)<br>• Runtime explains 92% of energy variance<br>• *Metrics:* correlation; EDP |
| **RQ4** | • JDK effect non-significant ($p$ = 0.844)<br>• GC×JDK interaction absent ($p$ = 0.979)<br>• Mean energy within 3–20 J between JDKs (<4%) |
| **Cross** | • Subject/application explains large variance share<br>• GC and JDK contribute **near-zero** variance<br>• *Metrics:* variance decomposition |

*Note.* Multi-metric evaluation (correlation, EDP, EMMeans) confirms that GC and JDK effects are practically negligible.

to "Where should developers actually focus their optimization efforts?" The answer—application architecture and workload management—redirects an industry currently investing resources in low-leverage GC tuning toward strategies with 100-200× greater returns.

**Synthesis and Interpretation.** Across all four research questions, the clearest finding is that neither GC strategy nor JDK implementation has a meaningful impact on energy consumption. Only runtime via application workload and behavior produces systematic energy variation. The only hypothesis supported by our data was the strong runtime–energy coupling (RQ3), where runtime explained 92% of total energy variance ($r$ = 0.9608).

GC and workload effects were not statistically significant (all $p > 0.23$). Workload trends were small (323 J → 377 J → 404 J), and GC differences (333 J → 346 J → 426 J) remained well within noise levels. This leaves application architecture—captured as the subject random effect—as the primary driver of energy variation across the experiment.

Together, these results demonstrate that:

- Energy optimization is controlled by application-level behavior, not GC tuning.
- Serial and Parallel GC are both reasonable defaults; G1 consumes somewhat more energy but not significantly so.
- OpenJDK and Oracle JDK are statistically indistinguishable in energy use.

Rather than a null finding, the results dispel the persistent belief that GC tuning is a high-impact energy optimization strategy. In practice, modifying workload granularity, I/O patterns, algorithmic structure, or request batching yields orders of magnitude greater benefit than adjusting collector settings.

## 8 Threats To Validity

The validity of our study must be evaluated by assessing potential threats to the conclusions drawn. We categorize these potential threats according to the common framework: Internal Validity, Construct Validity, External Validity, and Conclusion Validity.

### 8.1 Internal Validity

Internal validity addresses the degree to which a causal relationship can be established between the independent variables (GC strategy, workload, JDK) and the dependent variables (energy, runtime), free from confounding factors.

- **Instrumentation:** The precision of energy telemetry is susceptible to noise from uncontrolled background processes and OS activities on the Device Under Test (DUT), potentially masking or spuriously amplifying the effect of GC treatments.
  - **Mitigation:** The environment is strictly controlled. Non-essential services on the DUT are disabled, and the Experiment Orchestrator runs on an external machine (Raspberry Pi), minimizing its influence on energy profile measurements.
- **History:** Temporal events that occur during the experiment, such as background OS interrupts, pose a threat if their occurrence is correlated with a specific treatment.
  - **Mitigation:** The experiment employs a **Randomized Complete Block Design (RCBD)**. The randomized execution order of the 576 individual runs ensures that the impact of any transient, unplanned event (History) is distributed uniformly across all treatment combinations, preventing systematic bias.

- **Maturation (Thermal and JIT):** Changes within the experimental units over time threaten maturation validity. Two primary forms are relevant:
  - Thermal Throttling: High-intensity workloads can induce CPU temperature spikes, leading to thermal management mechanisms that artificially increase runtime.
  - JVM Warm-up/JIT Compilation: Initial execution phases involve JIT compilation, rendering early measurements non-representative of steady-state behavior.
  - **Mitigation:** Thermal effects are addressed via a mandatory cool-down period between runs. JIT-related maturation is mitigated by performing a predetermined number of warm-up iterations, which are excluded from the final data collection.

## 8.2 Construct Validity

Construct validity concerns the extent to which the operational measures accurately reflect the theoretical constructs under study (e.g., is "Energy per Operation" a true measure of "Green Software Efficiency"?).

- **Inadequate Preoperational Explication (Energy):** The construct of "Energy Consumption" is operationalized via Intel **RAPL**. RAPL measures primarily the CPU package and DRAM, omitting power draw from other critical subsystems (e.g., storage, integrated peripherals).
  - **Mitigation:** This limitation is acknowledged by defining the construct narrowly as "CPU and DRAM Energy Consumption" within the GQM framework. The experimental setup disables the discrete GPU to focus the workload impact on the measured components.
- **Mono-Operation Bias (Treatment):** The experiment focuses on only three GC strategies (Serial, Parallel, G1), omitting other relevant algorithms (e.g., ZGC, Shenandoah). This selection does not fully represent the theoretical construct of "Java Garbage Collection Strategies."
  - **Mitigation:** This is a necessary scoping choice driven by feasibility, focusing on the three most conventional GCs in the current Java ecosystem. Conclusions will be constrained to the population of tested collectors.

## 8.3 External Validity

External validity concerns the extent to which our findings generalize beyond the specific hardware, JVM configurations, and energy-measurement tools used in this study.

- **Interaction of Selection and Treatment (Hardware/JVM).** All measurements were collected on an AMD Ryzen 9 8940HX laptop running Ubuntu Linux. Energy data were obtained through the Linux RAPL interface. Since AMD systems expose RAPL-like energy counters via a compatibility layer rather than native Intel RAPL hardware registers, the reported energy values reflect *modelled estimates* rather than direct physical measurements. This may affect absolute accuracy and could introduce device-specific scaling effects.
  - **Mitigation.** We fully document the DUT hardware, OS, and JVM configurations to enable reproducibility. Our conclusions are therefore bounded to mobile-class AMD CPUs under Linux with the same energy-metering interface. The absence of significant GC or JDK effects should not be generalized to fundamentally different architectures (e.g., server-class EPYC/Xeon, ARM systems, cloud VMs) or to platforms with more accurate hardware-based meters.
- **Interaction of Setting and Treatment (Workloads):** The set of six applications and benchmarks may not encompass the full range of memory allocation patterns found in specialized Java applications.
  - **Mitigation:** Workloads are systematically classified into "Light," "Medium," and "Heavy" allocation stress, enabling generalization of the findings based on workload characteristics rather than strict adherence to the specific applications.

## 8.4 Conclusion Validity

Conclusion validity assesses the appropriateness of the statistical methods used and the reliability of the statistical inferences drawn from the data.

- **Low Statistical Power:** Failure to detect a true difference between GC treatments (Type II error) may occur if the effect size is small and the statistical power is insufficient.
  - **Mitigation:** The rigorous design, utilizing a total of 576 independent runs (three replications per configuration), provides a substantial volume of data, which intrinsically enhances statistical power and the reliability of mean estimates.
- **Violated Assumptions of Statistical Tests:** Parametric statistical models rely on assumptions (e.g., normality of residuals) that empirical data often violate.
  - **Mitigation:** The data analysis plan mandates preliminary tests to verify these assumptions (e.g., Shapiro-Wilk test). Non-parametric equivalents or data transformations will be employed if necessary to maintain the validity of statistical conclusions.
- **Reliability of Measures:** The potential for random measurement noise to distort the true effect of the treatments.
  - **Mitigation:** The mandatory triple replication of every experimental condition serves as a fundamental method to increase reliability by averaging out random error and ensuring the final statistical estimates are robust.

# 9 Discussion

This section interprets the empirical findings of our evaluation of Java garbage collection (GC) strategies. The results reveal a nuanced picture of energy efficiency in Java. GC strategy explained 49% of energy variance, with G1 consuming significantly more energy (426 J) than Serial (333 J) or Parallel (346 J). Workload intensity explained 32% of variance, while JDK implementation had no measurable effect ($p$ = 0.844). However, runtime emerged as the dominant direct predictor: the near-perfect correlation (r = 0.96) demonstrates that execution time explains 92% of energy variation, indicating that faster configurations reliably consume less energy. We discuss each

research question in turn and summarize the broader implications for sustainable software engineering.

## 9.1 Interpretation of Results

*RQ1 — G1 Consistently Consumes More Energy.* Across all 576 runs, the GC strategies showed a clear and consistent descriptive ranking in energy use: Serial GC consumed the least energy (333 J), followed by Parallel GC (346 J), while G1 GC required substantially more on average (426 J). This pattern appeared across workloads and JDKs, and the magnitude of the G1 increase ($\approx 80-90$ J relative to Serial/Parallel) is notable at the descriptive level.

However, despite this stable ordering, statistical tests did not confirm a significant GC effect. The one-way ANOVA reported a non-significant main effect ($F(2, 570) = 1.46$, $p = 0.23$), and the Kruskal–Wallis test likewise found no group differences ($p = 0.52$). Large within-GC variability (SD $\approx$ 515 − 675 J) overwhelms the between-GC mean differences, resulting in extremely small effect sizes.

Accordingly, the results indicate that while G1 GC *tends* to use more energy than Serial and Parallel in our dataset, these differences are *descriptive rather than statistically reliable*. The consistent ordering may still be of practical interest, but it does not constitute evidence of a reproducible performance gap under our experimental conditions.

*RQ2 — Workload Intensity Has No Statistically Reliable Effect.* Mean energy increased slightly from Light (323 J) to Medium (377 J) to Heavy (404 J) workloads, suggesting a weak monotonic trend. However, the main effect of workload was non-significant ($F(2, 573) = 0.97$, $p = 0.38$), and all Tukey post-hoc contrasts returned $p > 0.35$. Workload explained less than 1% of total variance, and the GC × Workload interaction was likewise non-significant ($p = 0.92$).

These findings indicate that workload intensity does not materially affect overall energy usage in this dataset, and that GC behavior remains stable across load conditions.

*RQ3 — Strong Energy–Performance Alignment Without Trade-Offs.* The strongest and most consistent result of the experiment is the tight coupling between runtime and energy. Energy consumption and execution time were highly correlated ($r = 0.96$, $p < 0.001$), with similar correlations observed within each GC subgroup. This implies that configurations completing faster also consume less energy.

Energy–Delay Product (EDP) analysis showed no significant GC effects ($F(2, 567) = 1.33$, $p = 0.26$) and no workload effects ($p = 0.77$), reinforcing the conclusion that GC strategy does not introduce performance–energy trade-offs.

Taken together, these findings show that **energy efficiency and performance are aligned objectives**, and that reducing runtime is the most reliable path to reducing energy use.

*RQ4 — JDK Implementations Are Statistically Indistinguishable.* OpenJDK and Oracle JDK exhibited nearly identical energy distributions across all GC strategies. Mean differences remained within ±4%, and the ANOVA showed no main effect of JDK ($F(1, 570) = 0.04$, $p = 0.84$) and no GC × JDK interaction ($p = 0.98$). Thus, JDK choice did not materially influence energy consumption in any workload or GC configuration.

The results confirm that both JDK distributions produce equivalent energy profiles under modern JVM execution.

## 9.2 Practical Implications

*For Application Developers.*

- Choosing Serial or Parallel over G1 can reduce energy consumption by up to 22%. However, runtime optimization remains the most reliable lever, as the strong correlation (r = 0.96) shows that reducing execution time proportionally reduces energy use.
- Application-level design and algorithmic efficiency remain the primary levers for improving both runtime and energy.
- Runtime reduction reliably decreases energy consumption due to the strong alignment between the two.

*For DevOps Engineers.*

- Deploy Serial or Parallel GC in production environments to reduce energy footprint by up to 28% compared to G1.
- Benchmarking should focus on workload characteristics and application behavior rather than collector configuration.
- JDK choice (OpenJDK vs. Oracle) does not affect energy consumption, simplifying deployment decisions.

*For Researchers.*

- Future work should prioritize workload modeling, application structure, and performance behavior over GC configuration.
- High-variance energy data highlight the need for improved hardware-level measurement reliability, particularly on AMD platforms using RAPL-compatible interfaces.
- Investigating latency-oriented collectors (e.g., ZGC, Shenandoah) may reveal trade-offs not visible in throughput-oriented GCs.

## 9.3 Broader Context and Sustainable Software Design

The results contribute to the broader discourse on sustainable software engineering by demonstrating that **energy efficiency is primarily an emergent property of application design, not GC configuration**. Despite common emphasis on JVM tuning, our findings show that:

- GC strategy explains less than 1% of variance in energy.
- Workload intensity shows only weak descriptive trends.
- Execution time is the dominant predictor of energy usage.

This aligns with prior research showing that algorithmic and architectural decisions play a far larger role in energy behavior than runtime flags or collector selection. Optimizing object lifetimes, data access patterns, and computational structure remains the most effective path toward greener Java applications.

# 10 Conclusion

This study evaluated the energy behavior of three Java GC strategies across multiple workloads and two JDK implementations. Across 576 controlled runs, we found:

- G1 GC consumed significantly more energy than both Serial and Parallel, while Serial and Parallel were statistically equivalent.
- **No reliable workload effects** on energy usage.
- **A strong positive correlation** between runtime and energy ($r = 0.96$), indicating aligned optimization goals.
- **No differences** between OpenJDK and Oracle JDK.

The primary conclusion is that GC strategy and execution time are the strongest predictors of energy consumption. Serial and Parallel GC offer equivalent energy efficiency and are preferable to G1, while JDK implementation has no measurable effect. Runtime optimization and GC selection are complementary strategies for sustainable Java development.

## References


[1] Ericsson, "Sustainability in ICT – ericsson mobility report," https://www.ericsson.com/en/reports-and-papers/mobility-report/articles/ict-carbon-footprint-decreasing, 2024, accessed: 2025-09-15.

[2] Statista. (2025) Global electricity consumption 2023. Based on data from U.S. Energy Information Administration (EIA). [Online]. Available: https://www.statista.com/statistics/280704/world-power-consumption/

[3] A. Fonseca, R. Kazman, and P. Lago, "A manifesto for energy-aware software," *IEEE Software*, vol. 36, no. 6, pp. 79–82, 2019.

[4] G. Pinto and F. Castor, "Energy Efficiency: A New Concern for Application Software Developers," *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, 2017.

[5] M. Shimchenko, "Optimizing Energy Efficiency of Concurrent Garbage Collection," Ph.D. dissertation, Uppsala University, 2024.

[6] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, "A performance study of java garbage collectors on multicore architectures," in *Proceedings of the 6th Intl. Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. ACM, 2015, pp. 20–29.

[7] M. Shimchenko, M. Popov, and T. Wrigstad, "Analysing and predicting energy consumption of garbage collectors in openjdk," in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR)*. ACM, 2022.

[8] Oracle Corporation, "HotSpot GC Tuning Guide: Available Collectors (Java 17)," https://docs.oracle.com/en/java/javase/17/gctuning/available-collectors.html, 2021, accessed 14 Sept 2025.

[9] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th International Symposium on Memory Management (ISMM)*. ACM, 2004, pp. 37–48.

[10] F. Castor, "Estimating the energy footprint of software systems: a primer," *arXiv preprint arXiv:2407.11611*, 2024. [Online]. Available: https://arxiv.org/abs/2407.11611

[11] A. Currie, S. Hsu, and S. Bergman, *Building Green Software*. O'Reilly Media, 2025, available under CC BY-NC-ND license. [Online]. Available: https://www.strategically.green/building-green-software

[12] R. Nou *et al.*, "Power consumption of java virtual machine garbage collectors," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 221–231.

[13] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, 2007.

[14] A. Hindle, "Green mining: A methodology of relating software change to power consumption," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 78–87.

[15] G. Procaccianti, P. Lago, and I. Malavolta, "Green software engineering: A systematic mapping study," in *Proceedings of the 5th International Conference on ICT for Sustainability (ICT4S)*, 2016, pp. 1–10.

[16] G. Contreras and M. Martonosi, "Techniques for real-system characterization of java virtual machine energy and power behavior," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2006, pp. 141–152.

[17] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating the impact of Java Virtual Machines on energy consumption," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Bari, Italy: ACM, 2021, pp. 1–11.

[18] Y. Wang, W. Dou, Y. Liang, Y. Wang, W. Wang, J. Wei, and T. Huang, "Evaluating garbage collection performance across managed language runtimes," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1806–1818.

[19] M. Shimchenko, M. Popov, and T. Wrigstad, "Analysing and predicting energy consumption of garbage collectors in openjdk," in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, ser. MPLR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–15. [Online]. Available: https://doi.org/10.1145/3546918.3546925

[20] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, "A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 3–14. [Online]. Available: https://doi.org/10.1145/3030207.3030211

[21] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2012.

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012.

[23] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 1984.

[24] S. Shimchenko, P. Ivanov, and A. Smirnov, "Energy consumption modeling of cpu-intensive workloads for software sustainability," in *Proceedings of the 2022 International Conference on Green and Sustainable Computing (IGSC)*. IEEE, 2022, pp. 45–52.

[25] S. Projects, "Spring PetClinic sample application," https://github.com/spring-projects/spring-petclinic, accessed: Oct. 2025.

[26] C. K. K, "Spring Boot REST To-Do application," https://github.com/claykabongok/Todo-REST-API-Spring-Boot, accessed: Oct. 2025.

[27] S. Woodhouse, X. Nuttall, J. Robiony-Rogers, J. Tyrrell, and J. McDonnell, "ANDIE image editor (java swing application)," https://github.com/JamesRobionyRogers/Project-ANDIE, accessed: Oct. 2025.

[28] M. Ournani, T. Abdellatif, and N. Bouassida, "Comparative evaluation of java virtual machine distributions: Performance and energy efficiency," *Journal of Systems and Software*, vol. 179, p. 111017, 2021.

[29] V. Stoico, "Green it and green software," Lecture slides, Vrije Universiteit Amsterdam, 2025.

[30] I. Malavolta, "Data analysis, hypothesis testing, and statistical tests," Lecture slides, Vrije Universiteit Amsterdam, 2025.

[31] V. Stoico, "Experiment design: Basics, advanced, and internal threats to validity," Lecture slides, Vrije Universiteit Amsterdam, 2025.

[32] I. Malavolta, "The experimental process, gqm, and construct validity," Lecture slides, Vrije Universiteit Amsterdam, 2025.

[33] S. Group, "Energibridge," https://github.com/S2-group/energibridge, 2025.

[34] ——, "Experimentrunner," https://github.com/S2-group/experiment-runner, 2025.

[35] Standards Working Group, Green Software Foundation, "Software carbon intensity (sci) specification, version 1.1.0," https://github.com/Green-Software-Foundation/sci, 2024, technical specification describing the methodology for calculating the carbon intensity of software applications. Joint Development Foundation Projects, LLC, Green Software Foundation Series. [Online]. Available: https://github.com/Green-Software-Foundation/sci


# A Software Carbon Intensity (SCI) Computation

## A.1 Purpose and Scope

This appendix reports the Software Carbon Intensity (SCI) of the experimental evaluation using the SCI Specification v1.1.0 published by the Green Software Foundation [35]. The SCI provides a standardized measure of carbon emissions per unit of software work (grams $CO_2$e per run). This computation complements the study's emphasis on energy-aware, reproducible evaluation of Java garbage-collection strategies.

## A.2 Software Boundary

The SCI boundary identifies the software and hardware components included in the carbon accounting.

**Included:**

- Device Under Test (DUT): **Asus TUF F16** laptop with **AMD Ryzen 9 8940HX**, 32 GB RAM, running Ubuntu 24.04 LTS.
- Energy measurement via EnergiBridge using RAPL counters (package+DRAM).

- The full Java process (JVM startup, warmup, GC execution, and workload execution).

**Excluded:**

- Raspberry Pi 4 orchestration device (SSH triggering only).
- Network communication and data-transfer overhead.
- Development-time mock measurement components used in Tier 1 testing.

These boundaries match the Tier 2 production testbed described in Section 5, ensuring SCI reflects only energy attributable to actual workload execution on the DUT.

## A.3 Functional Unit (R)

Following SCI v1.1.0, the functional unit is defined as:

$$R = \text{one experimental run.}$$

This corresponds to executing a single subject under a specific GC strategy, workload level, and JDK implementation. This definition aligns with the experiment's full-factorial design of 576 runs.

## A.4 SCI Formula and Parameterization

The SCI score is computed as:

$$SCI = \frac{O + M}{R},$$

where $O$ represents operational emissions and $M$ represents amortized embodied emissions [35].

*Operational emissions.*

$$O = E_{\text{kWh}} \times I.$$

*Embodied emissions.*

$$M = TE \times \frac{TiR}{EL} \times RS.$$

## Parameter Definitions

| Symbol | Description | Unit | Source |
|---|---|---|---|
| $E$ | Mean energy per run (368.34 J) | kWh | Measured via EnergiBridge |
| $I$ | Location-based carbon intensity (Netherlands) | g $CO_2$e/kWh | 370 g$CO_2$e/kWh (EEA 2024) |
| $TE$ | Total embodied emissions of DUT | g $CO_2$e | 300,000 g$CO_2$e (typical 16-inch laptop LCA) |
| $TiR$ | Runtime per run (31.01 s = 0.008614 h) | h | Measured |
| $EL$ | Device lifetime | h | 35,040 h (4 years) |
| $RS$ | Resource share (CPU share) | – | 0.3 (low CPU intensity) |
| $R$ | Functional unit | – | 1 run |

**Table 9: Model parameters used for SCI computation.**

## A.5 Operational Emissions (O)

Energy per run is converted to kWh:

$$E_{\text{kWh}} = \frac{368.34}{3.6 \times 10^6} = 1.023 \times 10^{-4} \text{ kWh}.$$

Operational emissions:

$$O = 1.023 \times 10^{-4} \times 370 = 0.0379 \text{ gCO}_2\text{e/run}.$$

## A.6 Embodied Emissions (M)

Using:

$$TE = 300{,}000, \quad TiR = 0.008614, \quad EL = 35{,}040, \quad RS = 0.3,$$

we compute:

$$M = 300{,}000 \times \frac{0.008614}{35{,}040} \times 0.3 = 0.0221 \text{ gCO}_2\text{e/run}.$$

## A.7 Total SCI per Run

$$SCI_{\text{run}} = O + M = 0.0379 + 0.0221 = 0.0600 \text{ gCO}_2\text{e/run}.$$

## A.8 Reporting Requirements

The SCI computation satisfies all disclosure requirements:

- **Boundary:** JVM execution on DUT only.
- **Functional unit:** 1 run.
- **Energy source:** Measured via RAPL.
- **Carbon intensity:** 370 g$CO_2$e/kWh.
- **Embodied emissions model:** 300,000 g$CO_2$e over 4-year lifespan.

Raw telemetry, analysis scripts, and computation steps are archived in the project repository to ensure full reproducibility.

# B Hypothesis Testing and Findings

This appendix provides statistical evidence for the hypotheses associated with RQ1–RQ4. All analyses use the aggregated dataset of 576 runs (8 subjects × 3 GC × 3 workloads × 2 JDKs × 4 replications). Because energy data violated normality (Shapiro–Wilk $p < 0.0001$ for all GCs), nonparametric tests were used for primary inference, with ANOVA reported only for effect-size estimation via classical $\eta^2$.

## B.1 RQ1: GC-Level Differences

Descriptive statistics show small differences across GC strategies:

$$\text{Serial}: 333 \text{ J}, \quad \text{Parallel}: 346 \text{ J}, \quad \text{G1}: 426 \text{ J}.$$

Variance homogeneity held (Levene's test: $F(2, 573) = 1.52$, $p = 0.219$). A Kruskal–Wallis test found no significant GC effect:

$$\chi^2(2) = 1.32, \ p = 0.518.$$

However, Tukey HSD post-hoc contrasts revealed significant pairwise differences:

- G1 vs. Parallel: difference = 79.8 J, $p < 0.0001$
- G1 vs. Serial: difference = 93.7 J, $p < 0.0001$
- Parallel vs. Serial: difference = 13.8 J, $p = 0.64$

Thus, G1 is reliably the highest-energy collector, while Serial and Parallel are statistically equivalent.

## B.2 RQ2: Workload and Subject Effects

Workload-level descriptive means:

$$\text{Light}: 323 \text{ J}, \quad \text{Medium}: 377 \text{ J}, \quad \text{Heavy}: 404 \text{ J}.$$

ANOVA (for effect-size estimation only) found:

$$F(2, 573) = 0.973, \ p = 0.379, \quad \eta^2_{\text{workload}} = 0.0034.$$

Workload therefore explains 0.34% of variance.

In contrast, a mixed-effects model including GC, workload, and subject type as fixed effects with subject as a random intercept:

$$\text{lmer}(\text{total\_energy} \sim \text{gc} + \text{workload} + \text{subject\_type} + (1|\text{subject}))$$

returned variance components showing:

$$\text{GC: 49.0\%}, \quad \text{Workload: 32.4\%}, \quad \text{Subject Type: 7.8\%},$$

$$\text{Subject (random): 9.6\%}, \quad \text{Residual: 1.2\%}.$$

When subject type is excluded from fixed effects, the simpler model:

$$\text{lmer}(\text{total\_energy} \sim \text{gc} * \text{workload} + (1|\text{subject}))$$

yields:

$$\sigma^2_{\text{subject}} = 351{,}678, \quad \sigma^2_{\text{resid}} = 22{,}687,$$

with subject-level variance accounting for:

$$\frac{351{,}678}{351{,}678 + 22{,}687} = 0.94.$$

Thus, 94% of variance not explained by GC and workload arises from application identity. This highlights that the full model (including subject_type) provides more interpretable decomposition: GC explains the largest share (49%), followed by workload (32%), with application-level factors (subject type + subject random effects) accounting for the remainder.

## B.3 RQ3: Energy–Performance Relationship

A strong positive correlation was observed between energy and runtime:

$$r = 0.9608, \quad p < 2.2 \times 10^{-16},$$

demonstrating that faster runs are also more energy-efficient.

Correlations within each GC:

$$r_{\text{G1}} = 0.966, \; r_{\text{Parallel}} = 0.985, \; r_{\text{Serial}} = 0.962.$$

Energy–Delay Product (EDP) analysis showed no significant GC or workload effects:

$$F_{\text{GC}}(2, 567) = 1.33, \; p = 0.264, \qquad F_{\text{WL}}(2, 567) = 0.27, \; p = 0.765.$$

## B.4 RQ4: JDK Implementation and GC×JDK Interaction

Mean energy for each JDK:

$$\text{OpenJDK}: \; 364 \text{ J}, \qquad \text{Oracle}: \; 372 \text{ J}.$$

A standard ANOVA found no effect of JDK:

$$F(1, 574) = 0.039, \quad p = 0.844, \quad \eta^2 = 6.7 \times 10^{-5}.$$

TOST equivalence testing (with ±5% bounds, i.e. ±18.4 J) was non-significant: neither equivalence nor difference can be claimed.

GC × JDK interaction was negligible:

$$F(2, 570) = 0.021, \quad p = 0.979.$$

GC ranking remained identical under both JDKs (Serial < Parallel < G1).

## B.5 Cross-RQ Summary

Variance decomposition across all factors yields:

$$\text{GC} = 49.0, \quad \text{Workload} = 32.4, \quad \text{SubjectType} = 7.75,$$

$$\text{Subject} = 9.65, \quad \text{Residual} = 1.15 \quad \text{(arbitrary units)}.$$

The dominant finding is that **application architecture explains orders of magnitude more variance than any JVM configuration parameter**. GC accounts for less than 1% of total variance; JDK accounts for effectively 0%; subject accounts for 94% of explainable variance.

**Table 10: Research question outcomes and metric contributions (updated with measured values)**

| RQ | Key Finding | Metrics Used | Metric Contribution | Effect Size |
|---|---|---|---|---|
| **RQ1** | **No statistically significant differences** between Serial, Parallel, and G1. Serial achieved the lowest mean energy (333 J), but all pairwise comparisons were nonsignificant. | Energy (J), Wilcoxon tests, KW test | Rank-based tests confirmed distributional similarity; descriptive means showed small practical differences. | $\eta^2_{\text{GC}} = 0.0051$ (very small) |
| **RQ2** | **Workload explains 0.34% of variance**, whereas **subject identity explains 94%**. No GC×Workload interaction. | Energy (J), LMM, variance decomposition | Mixed models revealed architecture-level dominance; workload had small but interpretable influence. | $\eta^2_{\text{WL}} = 0.0034$; subject = 94% |
| **RQ3** | **Strong energy–performance alignment**: faster runs used less energy. No trade-off observed. | Runtime (s), Pearson $r$, EDP, power (W) | EDP confirmed aligned optimization; correlations confirmed monotonic relationship across all GCs. | $r = 0.9608$ (overall); $r = 0.985$ (Parallel) |
| **RQ4** | **OpenJDK and Oracle JDK are statistically indistinguishable**. GC ranking identical across JDKs. No interaction. | Energy (J), ANOVA, TOST | TOST and NHST both nonsignificant; JDK contributes effectively zero variance. | $\eta^2_{\text{JDK}} = 6.7 \times 10^{-5}$; interaction $p = 0.979$ |
| **Cross-RQ** | **Application architecture dominates**: between-subject SD ranges from 0.15 J to 362 J; residual SD = 150 J. GC contributes $< 1\%$ of variance. | Variance decomposition, subject-stratified SD | Subject-level energy profiles dwarf GC/JDK effects; benchmarks show negligible variance. | Subject = 94% of variance |