On The Energy Consumption of Continuous Integration in Open-Source Java Projects

Robert Arntzenius
robertarntzenius@gmail.com
Delft University of Technology
Delft, The Netherlands

Xutong Liu
x.liu-14@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

***Abstract*—Continuous integration is essential for software quality, yet the energy footprint associated with its frequent execution has largely remained invisible. We provide the first comprehensive baseline of CI energy use through a large-scale study of 204 open-source Java projects with repeated measurements under Maven and Gradle. Our results show that energy use is highly skewed: while most projects consume energy modestly, a small number of "CI-intensive" systems can reach annual CI energy footprints of hundreds of kilowatt-hours, which is comparable to a quarter of an average EU household's electricity use. We further show that immediate, practical savings are possible: simply enabling dependency caching cuts energy by 30% on average in some Maven projects and by over 90% in some Gradle cases. These findings matter not only for individual developers, but also for large organizations that run thousands of builds. In those settings, even small inefficiencies can add up to very large energy costs. By exposing where energy is consumed and how to reduce it, our study establishes an actionable foundation for greener CI pipelines.**

***Index Terms*—energy consumption, continuous integration, software testing, sustainable**

I. Introduction

As we have grown accustomed to living in a software-filled world, we are also more and more relying on software for everyday tasks [1]. Because of our reliance on software, there is a greater need for the reliability of the software that we produce [2]. For example, it has been estimated that software failures in 2017 cost the economy $1.7 trillion [3]. Additionally, from an analysis of newspaper articles, Ko et al. estimate that software failures can be linked to the loss of 15,000 human lives [4]. In this light, the role of software quality assurance becomes ever more important.

To safeguard software quality, engineers employ approaches such as testing [5]–[8], code review [6], [9], [10], static analysis [11], [12], and build automation [13]–[16]. Many of these practices, particularly testing, static analysis, and build automation, are now routinely executed through continuous integration (CI) pipelines [13].

More generally, the Information and Communication Technology (ICT) sector itself has become a growing concern in the climate change debate. In 2020, ICT was estimated to consume about 15% of global electricity, with projections suggesting this figure could rise to 20% by 2025 [17], [18]. The environmental impact of this consumption depends on the *carbon intensity* of the electricity, i.e., the grams of CO_2 emitted per kilowatt hour (kWh) of electricity produced, which varies substantially depending on whether energy is generated from renewable resources or fossil fuels [19]. Overall, ICT accounts for roughly 2–4% of global greenhouse gas emissions [20]. While this share may seem modest compared to transportation (27%) and manufacturing (24%) [21], there is broad consensus that "urgent policy action and investment are needed to limit increases in energy use driven by increasing demand of ICT services" [22].

In software engineering research, initial evidence suggests that CI build processes can consume non-negligible amounts of energy [23]. This is particularly relevant given that 40% of the 34,000 most popular GitHub projects employ CI, with adoption rising to 70% among the top 500 projects [15]. Zaidman's study, however, was limited to 10 systems and a single measurement per system [23].

This paper extends these insights by conducting a large-scale study of the *energy requirements of building open-source Java projects*. We analyze 204 projects using an automated pipeline that enables repeated measurements, allowing us to characterize energy usage in a local CI environment. Our work is guided by the following research questions:

RQ1. To what extent do the build and test phases of CI contribute to energy consumption in open-source Java projects?

CI automates multiple activities in the development process, such as building and testing. While these activities are essential for software quality, they likely differ in their potential energy footprint. Understanding how much energy is spent in CI pipelines, and which phases dominate consumption, is crucial to assessing the sustainability of software development.

RQ2. How is energy consumption distributed across different phases of the CI pipeline?

Build pipelines are not monolithic: compilation, dependency resolution, and testing each impose different workloads. Identifying which phases act as energy hotspots enables us to move beyond aggregate numbers and pinpoint opportunities for optimization. However, little is known about how these costs manifest across a large and diverse set of open-source projects.

RQ3. How does dependency caching affect the runtime and energy usage of large Java projects?

Developers routinely use caching mechanisms in CI to speed up builds. However, the energy-saving effects of caching—beyond mere reductions in build time—remain underexplored. Investigating how caching impacts the runtime and energy efficiency of CI pipelines can reveal whether widely adopted practices also provide tangible environmental benefits.

Taken together, our findings provide the first large-scale view of energy usage in Java-based CI, demonstrating not only where energy is consumed, but also which practices can mitigate it. By raising awareness, our study contributes to ongoing calls for integrating sustainability considerations into everyday software engineering practice [24]–[26].

This paper is structured as follows: in Section II we present fundamental background information. Section III explains our experimental setup, while Section IV presents our results. In Section V we discuss our results, and in Section VI we relate our work to existing work. Section VII discusses threats to validity. We present our conclusions in Section VIII.

II. Background

A. CI Pipelines

CI has become a cornerstone of modern software development, automating activities such as compilation, testing, and static analysis to ensure continuous quality assurance [13]–[15]. A typical CI pipeline executes these tasks on dedicated infrastructure, often cloud-based services such as GitHub Actions, Jenkins, TravisCI, or CircleCI [15]. Because pipelines are triggered frequently—sometimes after every commit—CI builds represent a recurring workload whose cumulative energy cost may be substantial [23]. Understanding this cost is essential to assess the environmental footprint of software engineering practices [24].

CI pipelines are not monolithic: they consist of distinct phases such as dependency resolution, compilation, and testing. Each phase stresses hardware resources differently—network and I/O for dependency management, CPU for compilation, and CPU/memory for testing—making them natural candidates for detailed energy profiling [8]. Identifying which phases dominate energy consumption provides insights into where optimizations would be most effective.

Modern CI platforms further influence energy usage by offering caching mechanisms. Dependency caches, for example, allow build tools such as Maven and Gradle to reuse previously downloaded libraries and build artifacts across runs [13]. While primarily intended to reduce build times, caching may also reduce energy consumption by avoiding redundant I/O and computation. The extent of these benefits, however, has not been systematically quantified, making caching a key factor in our investigation [27].

B. Measuring energy consumption in software projects

Measuring software energy consumption has often been carried out using energy profilers, mainly due to their ease of use and accessibility. These profilers typically estimate energy consumption by monitoring the load of system components, such as the CPU, and applying estimation models [28], [29]. However, such tools have been shown to vary significantly in their estimates depending on the profiler used [30], which questions their reliability.

In contrast, hardware power monitors measure the actual power drawn by the entire system with high precision, providing more reliable measurements. While this introduces additional setup overhead, it also enables capturing energy consumption of components not directly visible to profilers, such as storage devices or network activity. Therefore, the choice between profilers and hardware monitors can be seen as a trade-off between ease of use and measurement accuracy.

Moreover, related work has demonstrated that software execution environments can also impact energy measurements. For instance, Santos et al. [27] showed that running software in Docker containers incurs a measurable energy penalty. This highlights the importance of carefully selecting both the measurement method and the execution environment when designing empirical studies of software energy consumption.

III. Experimental setup

A. Energy measurement setup

The objective of our experiment is to obtain reliable and repeatable measurements of CI energy consumption for a representative set of open-source projects. To this end, we designed a controlled measurement environment as follows.

Measurement procedure. Each CI build is measured five times to account for variability, with enforced waiting periods between runs as recommended in prior work [31]. Although some studies use more repetitions (e.g., 30 runs) to further reduce noise [32], [33], this would be prohibitively expensive in our setting, as our experimental campaign involves building 204 projects, each measured repeatedly. We therefore balance statistical robustness and practical feasibility by using five repetitions per build. We discuss consistency in Sections V-A and V-B.

Measurement device. We use an AVHzY CT-3 power monitor[1], which supports USB-C pass-through and USB connectivity to the host. The device reports a resolution of $0.0001\mathrm{V}$ and $0.0001\mathrm{A}$ with an accuracy of $0.1\% + 2d$, corresponding to an error of approximately 0.1% per reading. This precision allows us to capture the energy consumption of the entire system, including CPU, storage, and network components.

Hardware platform. Measurements are conducted on a local mini-PC (MinisForum EM680[2]) equipped with an AMD Ryzen 7 6800U CPU (8 cores, 16 threads) and 16 GB RAM. The device has no internal battery, which allows direct power intake measurement without interference from charging cycles.

Software environment. The experiments were conducted on Ubuntu 22.04.4 LTS (Linux 5.15.0-119, x86_64) with minimal background processes. Multiple Java versions (8, 11, 17, and 21, temurin distributions) were installed via *SDKMAN!*, together with Python2/3, and Docker (v26.1.1), to support

[1]https://store.avhzy.com/index.php?route=product/product&product_id=51, last visited 29/09/2025.

[2]https://minisforumpc.eu/products/minisforum-em680-em780-refurbisched, last visited 29/09/2025.

diverse project requirements. A dedicated low-privilege user was used to control cached state and background activity. Docker was only employed when explicitly required by a project, as containerized execution has been shown to incur a measurable energy overhead [27].

B. Repository List

Our dataset builds on the collection of locally buildable Java projects by Khatami and Zaidman [34], which originally contained 202 Gradle and 457 Maven projects, each identified by a specific buildable version. As the dataset was constructed earlier, we revalidated all projects in our current environment. For each repository, we selected the most recent commit that successfully passed all GitHub workflows, ensuring up-to-date and buildable versions. To identify a compatible Java version, we sequentially attempted builds with Java 8, 11, 17, and 21 until success, avoiding reliance on potentially outdated configuration metadata. Following this process, our final dataset consists of 122 Gradle and 82 Maven projects. Detailed information can be found in our replication kit [35]. This set represents a diverse and actively maintained sample of Java systems with buildable CI configurations.

C. CI Setup

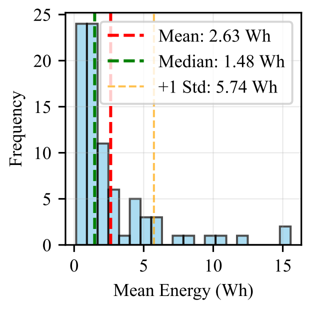
We simulate the CI pipeline using a bash script that performs three steps: (i) cloning the repository, (ii) compiling the application, and (iii) running the test suite. For Gradle, compilation and testing are executed via `./gradlew assemble` and `./gradlew check`; for Maven, via `./mvnw compile` and `./mvnw test`.

To ensure a clean state for each experiment, we use a cleanup script (`cleanup.sh`) that removes all cached files and directories, including project-level caches, Docker containers, and common home-directory caches (e.g., `.cache`, `.gradle`, and `.m2`). Only essential configuration files (e.g., `.sdkman`, `.ssh`, session-specific `.profile`, and shell configuration files) are preserved, and Docker is explicitly reset. This procedure ensures that all experiments start from an identical environment.

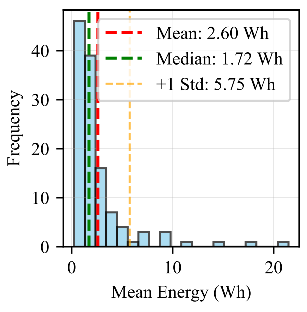
In addition to this main setup, we designed a variant to answer **RQ3**: *How does dependency caching affect the runtime and energy usage of large Java projects?* For this variant, we optionally preserved dependency caches in the home directory (i.e., `~/.gradle` and `~/.m2`) rather than delete these directories before every build, allowing Gradle and Maven to potentially reuse previously downloaded dependencies and build artifacts. This allows us to investigate the impact of dependency caching on runtime and energy consumption under conditions where cache reuse is possible.

D. Measurement framework

Measuring energy consumption with a power monitor device requires a system that controls this device and reads it out. We use a Windows PC as the controlling device: it sends commands to start the CI pipeline for a specific project on the MinisForum device and reads the power monitor while waiting for the process to finish. Throughout the experiment, constant communication between the systems is maintained. Messages and commands are exchanged sequentially, and the controller always waits for explicit confirmation before proceeding to the next step. The controller system can issue complex commands to the MinisForum, such as generating a build script that includes the relevant repository information. The MinisForum responds to these commands via the shellstream output using predefined code words. This communication structure also clearly delineates the different phases of the experiment, with specific messages indicating the start and end of each phase. Together, these interactions ensure a well-synchronized execution of the CI pipeline and accurate alignment between experimental phases and energy measurements.



(a) Maven projects



(b) Gradle projects

Figure 1: Total energy consumption distribution of Maven and Gradle projects

IV. Results

A. RQ1: How much energy is used by an open-source Java project during the build and test phases of CI?

To answer RQ1, we gathered and calculated the average energy consumption for each project. Figures 1a and 1b show the corresponding histograms, including mean and median values, for Maven and Gradle projects. From these results, we derive the following observations:

Skewed Distribution and High-Energy Outliers. Both Maven and Gradle projects exhibit strongly right-skewed CI energy distributions, where a small number of outliers substantially inflate the mean. For Maven projects, the mean energy consumption is 2.63 Wh versus a median of 1.48 Wh; for Gradle, the mean is 2.60 Wh and the median 1.72 Wh. This gap highlights the impact of extreme cases, e.g., Apache Iceberg consumes up to 21.47 Wh per build. As its CI configuration triggers at least fourteen build-and-test jobs per commit across multiple Java versions and operating systems, its annual CI energy consumption is estimated to exceed 442 kWh. Under the average 2023 EU electricity mix[3], this corresponds to approximately 213 kg of CO_2, exceeding the emissions of a round-trip flight between Amsterdam and Dublin.[4]

[3]According to Our World in Data; URL https://ourworldindata.org/electricity-mix

[4]According to Carbon Calculator; URL https://www.carbonfootprint.com/calculator.aspx

Table I: Characteristics of High Energy-Consuming Java Projects in CI

Project	Build Tool	Energy (Wh)	Key Characteristics	Reasons for High CI Energy Consumption
apache-dolphinscheduler	Maven	12.10	Workflow orchestration platform with complex module dependencies and integrations.	Multi-module builds and integration with heavy data systems increase build time and CI load.
spring-cloud-stream	Maven	15.47	Event-driven microservice framework relying on messaging middleware	Integration tests involving message brokers and asynchronous behavior prolong CI cycles.
apache-iotdb	Maven	15.56	Time-series database optimized for IoT workloads, with complex query and storage engines	Resource-intensive end-to-end tests simulating time-series ingestion and queries.
pravega-pravega	Gradle	14.10	Stream storage supporting high-concurrency segmentation and fault tolerance	CI includes multithreaded tests and failure-recovery simulations.
apache-groovy	Gradle	17.51	Dynamic JVM language supporting DSLs, scripting, and metaprogramming	Extensive runtime behavior tests and compatibility checks with Java increase CI time.
apache-iceberg	Gradle	21.47	Data lake table format supporting Spark, Flink, ACID transactions, and schema evolution	Integration tests across multiple compute engines and heavy Docker setups raise energy use.

Minimal Impact of Build Tool Choice. The average energy consumption between Maven and Gradle projects is nearly identical (2.63 vs. 2.60 Wh), and their median values are similarly close (1.48 vs. 1.72 Wh). As our sample did not favor either tool, the comparable distributions suggest that the choice of build tool alone does not significantly impact overall CI energy consumption. Instead, the differences are likely rooted in project-specific characteristics such as codebase size, test complexity, or dependency configurations.

High Energy Projects Share Common Characteristics. Analysis of the top three energy-consuming projects under Maven and Gradle (Table I) reveals common drivers of high CI energy usage. These projects have complex, multi-module architectures (e.g., workflow engines, databases, and data lake systems) and require tight integration with external ecosystems such as Zookeeper, Hadoop, or Flink. Their CI pipelines are dominated by extensive integration and stress tests, including high-concurrency workloads and multi-engine compatibility checks, which result in long-running, CPU-intensive builds. As these workflows rely on full integration tests and deployment simulations rather than unit tests, they are less amenable to standard optimization techniques such as caching or mocking.

Answer to RQ1. On average, CI builds of open-source Java projects consume about 2.6 Wh per run, with Maven and Gradle projects showing nearly identical energy profiles. While most projects have modest energy demands, a small number of CI-intensive systems — driven by extensive testing and ecosystem dependencies — consume much more and dominate the overall energy use.

B. RQ2:How is energy consumption distributed across different phases of the CI pipeline?

Fig. 2 presents the distribution of the percentage of energy consumption spent on the compilation phase during the entire build process for Maven and Gradle projects. We observe that:

Compilation and Testing as Even Contributors: On average, Maven projects spend 50.6% of their build energy on compilation and Gradle projects 54.2%, leaving nearly the other half to testing. This shows that compilation and testing contribute in roughly equal measure to CI energy consumption, making both phases important targets for optimization.

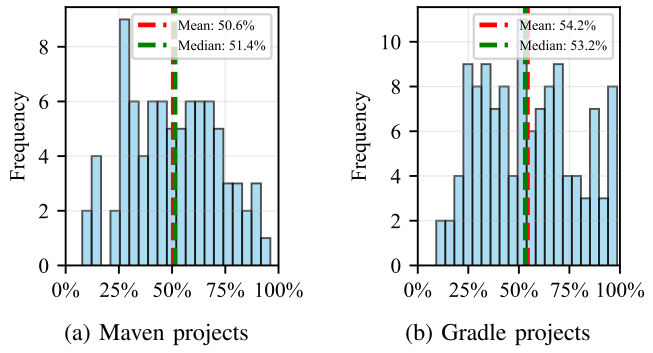


Figure 2: Histogram of percentage of energy spent on compiling among whole build process for Maven and Gradle projects

Limited Impact of Build Tools: Maven and Gradle projects show minimal differences in compilation energy consumption ratios, indicating that build tool type is not a decisive factor.

Strong Dependence on Project Type: To further investigate how compile energy varies across project types, we categorized subject projects based on their function (details in replication package [35]). Fig 3 shows the distribution of compile energy ratios.

Game-related projects exhibit the highest ratios (mean 85.4%), likely due to expensive compilation of graphics- and engine-heavy codebases combined with limited automated testing [36]. Communication/Social and Server/Infrastructure projects show similar trends. In contrast, Data Processing/Analytics and Development/Testing tools display much lower ratios (mean 35.2%), as energy consumption is often dominated

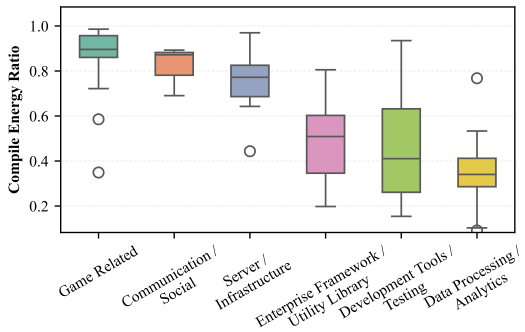


Figure 3: Distribution of compile energy ratios across project categories for Gradle projects

by test execution involving I/O, databases, or large-scale data processing. Enterprise frameworks and utility libraries are more heterogeneous, reflecting varied build sizes and testing strategies. Overall, these results indicate that compilation impact depends on project functionality and testing practices, suggesting the need for project-specific energy optimization.

Answer to RQ2. On average, compilation and testing each contribute about half of CI build energy, while dependency resolution plays a minor role unless caching is ineffective. Differences across project types outweigh those between Maven and Gradle: game-like and infrastructure projects are compilation-heavy, whereas data- and tool-oriented projects are testing-dominated. Consequently, optimization strategies should be tailored to project type.

C. RQ3: How does dependency caching affect the runtime and energy usage of large Java projects?

To examine the impact of dependency caching on energy consumption during CI process, we selected a set of relatively high energy consuming Java projects from our study projects for an extra experiment. The selection criterion was based on the average energy consumption in the uncached setting, from which we ranked the projects and chose the top nine Maven and Gradle projects. This allows us to focus on projects where dependency resolution might be costly and where caching might provide most benefits.

Tables II and III present a comparison between uncached (UC) and cached (C) builds. For each project, we report the average energy consumption in both settings, the absolute difference (Diff), and the relative change (Ratio). The last row shows the overall average across the selected projects.

For Maven-based projects (Table II), dependency caching consistently reduced energy consumption across all nine projects. The relative improvement ranges from modest reductions (e.g., -6.8% for `smallrye-smallrye-reactive-messaging`) to substantial savings (e.g., -82.7% for `shopizer-ecommerceshopizer`). On average, Maven projects experienced a 30.9% decrease in energy consumption, suggesting that caching provides benefits.

Gradle-based projects (Table III) displayed more variation. Some projects showed a dramatic decrease in energy consumption, such as `apache-groovy` (-97.7%) and `hibernate-hibernate-orm` (-92.8%). We attribute these large savings to the fact that, without caching, these projects spend substantial effort on repeatedly resolving and downloading dependencies. With caching enabled, this overhead is almost eliminated. Two projects (e.g., `typetools-checker-framework` and `broadinstitute-picard`) exhibited small regressions ($+2.3\%$ and $+2.6\%$, respectively). Despite these anomalies, the overall average for Gradle projects still reflected a sizable energy reduction of 38.7%.

Table II: Comparison of Uncached and Cached Performance Across Maven Projects

Project	**UC**	**C**	**Diff**	**Ratio**
apache-iotdb	15.6	12.1	-3.5	-22.2%
spring-cloud-spring-cloud-stream	15.5	7.4	-8.0	-51.9%
apache-dolphinscheduler	12.1	9.5	-2.6	-21.6%
fabric8io-kubernetes-client	10.6	9.4	-1.2	-11.0%
apache-incubator-shenyu	9.8	7.3	-2.6	-26.0%
smallrye-smallrye-reactive-messaging	8.4	7.8	-0.6	-6.8%
apache-dubbo	7.5	5.3	-2.3	-29.9%
googlecloudplatform-spring-cloud-gcp	6.0	4.4	-1.6	-26.4%
shopizer-ecommerce-shopizer	5.9	1.0	-4.9	-82.7%
Average	10.2	7.1	-3.0	-30.9%

Table III: Comparison of Uncached and Cached Performance Across Gradle Projects

Project	**UC**	**C**	**Diff**	**Ratio**
apache-iceberg	21.5	21.2	-0.2	-1.1%
apache-groovy	17.5	0.4	-17.1	-97.7%
pravega-pravega	14.1	13.5	-0.6	-4.5%
typetools-checker-framework	9.2	9.4	0.2	2.3%
broadinstitute-picard	9.1	9.3	0.2	2.6%
consensys-teku	9.1	7.9	-1.2	-13.2%
openrewrite-rewrite	7.7	1.4	-6.3	-82.0%
odpi-egeria	7.5	2.9	-4.6	-61.9%
hibernate-hibernate-orm	6.9	0.5	-6.4	-92.8%
Average	11.4	7.4	-4.0	-38.7%

Answer to RQ3. Overall, enabling dependency caching improves the runtime and energy efficiency of CI builds in our study, although the magnitude of improvement varies across projects. Caching remains a practical mechanism for greener pipelines, but it comes with trade-offs, such as additional storage or potential cache invalidation.

V. Discussion

A. Consistency of energy consumption measurements in Gradle projects

To validate the reliability of our experimental findings, we analyze the consistency of the energy measurements. The more consistent the results across independent runs, the greater confidence we can place in the reliability of the data.

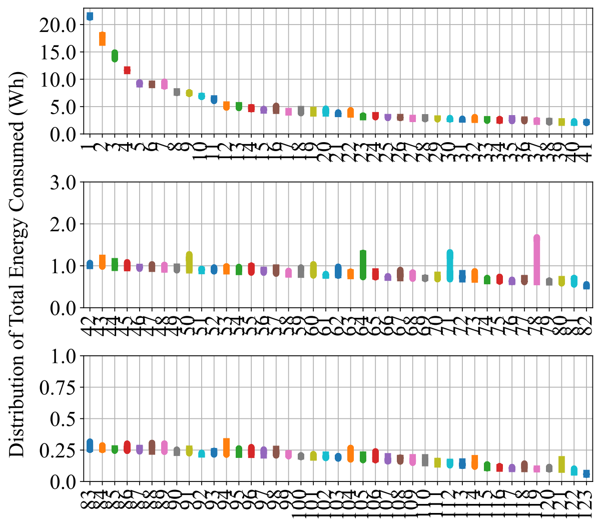


Figure 4: Total energy consumption from Gradle projects ordered by energy intensity.

Fig. 4 presents the distribution of energy consumption across Gradle projects, with each project executed five times and ordered according to its mean energy consumption. For brevity, project names are replaced by numerical identifiers, and the full mapping between identifiers and project names is available in our replication kit [35]. As shown in Fig. 4, the spread of data points for most relatively small-scale projects typically falls between $< 0.1\text{Wh}$ and 0.5Wh, while for larger-scale projects the spread generally ranges between $< 0.1\text{Wh}$ and 1Wh. A notable exception is the Apache Lucene project (number 78 in Fig. 4), whose spread extends from 1.2Wh to 3.4Wh. Overall, the relatively narrow variation across repeated runs demonstrates the consistency of the energy measurements, suggesting that the results are reproducible and not substantially influenced by random noise.

B. Consistency of energy consumption measurements in Maven projects

Similar to Section V-A, this subsection analyzes the consistency of energy consumption measurements for Maven projects. Fig. 5 shows the distribution of energy consumption across projects, each measured five times and ordered by mean energy consumption. For brevity, projects are labeled with numerical identifiers; the full mapping is provided in the replication kit [35].

Compared to Fig. 4, Maven projects exhibit substantially lower measurement consistency. In addition to higher variability in energy consumption, we observed practical issues such as long delays or failures during dependency downloads, often due to timeouts. To assess whether dependency retrieval contributes to these inconsistencies, we conducted an additional experiment on a subset of Maven projects whose *pom* configurations allow caching to be disabled. The experiment differed from the original setup only in that dependencies were cached in advance by preserving the `.m2` directory, thereby eliminating download-related delays during subsequent builds. The results are shown in Fig. 6.

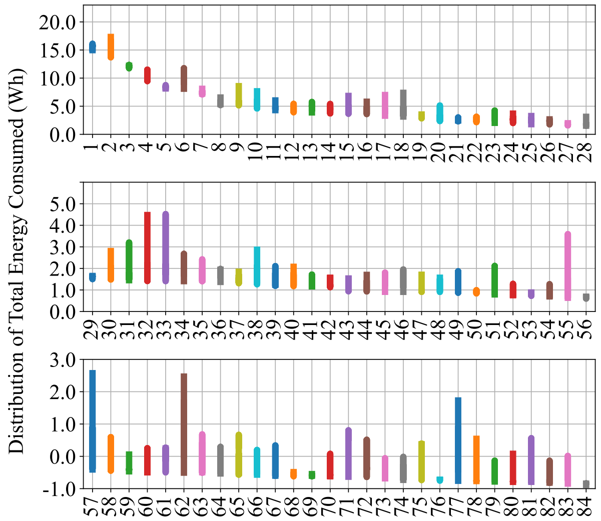


Figure 5: Total energy consumption from Maven projects ordered by energy intensity.

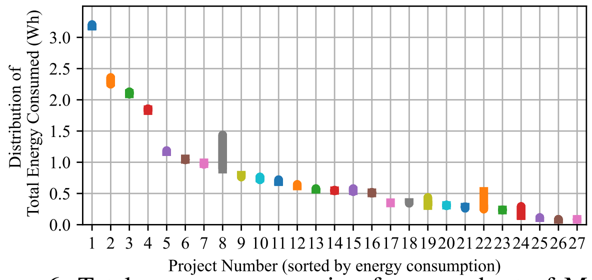


Figure 6: Total energy consumption from a subset of Maven projects with caching.

After enabling caching, measurement inconsistencies in Maven largely disappeared, with most projects yielding nearly identical results, with **Shopizer** as a notable exception due to an unexplained multi-minute compilation delay at near-idle power levels; similar Maven freezes have been reported previously [5].

In summary, Maven projects exhibit lower measurement consistency than Gradle projects, largely due to dependency retrieval during CI builds. This highlights the critical role of build system characteristics and dependency management in energy consumption, and calls for additional caution when interpreting CI-related energy results for Java projects.

C. Yearly estimates

After establishing the average energy consumption per CI run, we estimate yearly CI energy usage for the twenty projects with the highest per-run energy costs. This estimation relies on commit counts to the project's `HEAD` in 2023 and the assumption that each commit triggers a single build-and-test cycle. The resulting yearly estimates are reported in Table IV.

Table IV shows that yearly CI energy consumption varies widely across projects and is driven by both per-run energy cost and commit frequency. For example, *Apache Iceberg* has the highest yearly estimate (31.61 kWh), due to its high per-run cost (21.47 Wh) and 1472 commits. In contrast, *Hibernate ORM* ranks third with 17.13 kWh, driven by frequent commits (2483) despite a low per-run cost (6.90 Wh). Similarly, OpenRewrite and Apache Dubbo accrue substantial yearly consumption because of high development activity, whereas Picard and Shopizer remain comparatively low due to infrequent commits despite per-run costs near 9 Wh.

It is important to note that these estimates represent conservative lower bounds. We assume a single build per commit, whereas in practice CI pipelines often trigger multiple builds across different Java versions or operating systems [13]. Even under this assumption, several projects already reach tens of kilowatt-hours per year. For example, accounting for multiple Spark CI runs increases the estimate for *Apache Iceberg* to 442.54kWh annually, corresponding to about 26% of the average yearly household electricity consumption per person in the EU [19], or roughly 213kg of CO_2 emissions

[5]*Maven hangs for ∼20 mins during the project build (used to work fine)*, last accessed: August 30th, 2024. See: https://stackoverflow.com/questions/43792427/maven-hangs-for-20-mins-during-the-project-build-used-to-work-fine

Table IV: Energy and commits of 20 projects with the highest energy consumption during CI.

Project	Avg. Energy (Wh)	Commits (2023)	Yearly Energy (kWh)
Apache Iceberg	21.47	1472	**31.61**
Apache Groovy	17.51	671	**11.75**
Apache IoTDB	15.56	1742	**27.11**
Spring Cloud Stream	15.47	287	**4.44**
Pravega	14.10	103	**1.45**
Apache DolphinScheduler	12.10	571	**6.91**
Armeria	11.66	423	**4.93**
Fabric8 Kubernetes Client	10.57	723	**7.64**
Apache ShenYu	9.82	524	**5.15**
Typetools Checker Framework	9.20	887	**8.16**
Picard	9.09	38	**0.35**
Teku	9.07	754	**6.84**
SmallRye Reactive Messaging	8.42	616	**5.19**
OpenRewrite	7.66	1499	**11.48**
Apache Dubbo	7.53	1282	**9.65**
Egeria	7.48	1080	**8.08**
Hibernate ORM	6.90	2483	**17.13**
Enonic XP	6.18	281	**1.74**
Spring Cloud GCP	6.00	493	**2.96**
Shopizer	5.94	25	**0.15**

based on 2023 carbon intensity figures [37]. Moreover, our measurements were performed on a MinisForum device[6] with a Thermal Design Power (TDP) of 28 W, i.e., the maximum heat a processor can generate under normal workload, which roughly reflects its power consumption. This is much lower than typical server CPUs such as the Intel Xeon Gold 6338[7] (205 W), suggesting that real-world CI energy consumption is likely higher. These findings highlight that the CI energy usage should not be overlooked.

VI. Related Work

Research on the energy consumption of software systems has approached the problem from multiple perspectives, ranging from the evolution of energy usage over time to the impact of development practices and testing strategies.

Measurement and evolution of software energy consumption focus on how energy usage changes as software evolves and how it can be quantified in practice. Hagen analyzed longitudinal energy trends using regression testing as a proxy, focusing on relative rather than absolute values due to hardware abstraction [38]. Hindle proposed a general methodology linking software changes and object-oriented metrics to energy consumption, with case studies on large systems and library evolution [29]. At a lower level, Marini et al. showed that compiled and semi-compiled languages tend to be more energy-efficient than interpreted ones, although algorithmic choices may dominate [39].

Energy impact of development and testing practices examine how engineering processes affect energy usage. Perez et al. showed that in DevOps pipelines, cumulative energy consumption is mainly driven by the frequency of build executions [40]. Focusing on testing, Verdecchia et al. demonstrated that energy-aware test prioritization can reduce energy consumption without harming effectiveness [41], while Kifetew et al. identified test generation strategies and cyclomatic complexity as key drivers of energy cost [42], [43]. From a socio-technical angle, Blokland et al. reported that limited awareness and tooling hinder the adoption of energy as a first-class development metric [44].

[6]https://minisforumpc.eu/products/minisforum-em680-em780-refurbisched?_pos=1&_psq=MinisForum+EM680&_ss=e&_v=1.0&variant=51834741653870

[7]https://www.intel.com/content/www/us/en/products/sku/212285/intel-xeon-gold-6338-processor-48m-cache-2-00-ghz/specifications.html

VII. Threats to Validity

Internal validity. Although we employed a hardware power monitor to avoid the limitations of profilers [28], [29], measurement noise and background processes could still affect results. To mitigate this, each experiment was repeated five times with waiting intervals, following established guidelines for energy measurement [31].

External validity. Our dataset consists of buildable Java repositories selected from a curated benchmark [34], excluding projects requiring extensive configuration. This may bias the sample toward smaller or less complex systems, limiting generalization to other languages, industrial-scale projects, or evolving CI practices. Moreover, our exploratory study simulates CI builds on a Linux mini PC, which enables controlled and reproducible measurements but does not reflect large-scale cloud infrastructure. As a result, reported values should be interpreted as conservative approximations rather than representative industrial figures. Nevertheless, this work aims to raise awareness of CI-related energy consumption, in line with prior calls for improved energy insights [24]–[26].

Construct validity. Our experiments were conducted over several months, during which temperature and network conditions were not constant. Such environmental variation may have influenced build durations and thus energy usage. Moreover, our setup simulates CI on a local mini-PC with an energy-efficient AMD Ryzen processor (TDP 28W), whereas server-grade processors typically operate at much higher TDP values (e.g., 85W [23]). The reported values should therefore be interpreted as conservative lower bounds.

VIII. Conclusion

In this work, we conducted a large-scale empirical study of the energy consumption of open-source Java projects in CI. Our analysis of Maven- and Gradle-based projects reveals three key insights. CI energy use is highly skewed, with a few integration-heavy projects dominating overall consumption while the choice of build tool has minimal effect (RQ1). Compilation is the primary energy hotspot, but its relative weight depends on project domain, being highest in game-related projects and lowest in data- and tool-oriented ones (RQ2). Dependency caching consistently improves efficiency, reducing energy by about 30% in Maven projects and in some Gradle cases by over 90% (RQ3).

In our discussion, we extrapolate to yearly consumption estimates, highlighting that some projects reach tens to hundreds of kilowatt-hours, comparable to a non-trivial share of household electricity usage. These findings demonstrate that CI energy consumption is not negligible and should be treated as a sustainability concern. Future work will expand to other ecosystems, develop predictive models linking project features to energy profiles, and explore optimization strategies for greener software pipelines.

Data Availability

The datasets and research artifacts related to this paper are available at: https://zenodo.org/records/17242523.

References

[1] R. Kazman and L. Pasquale, "Guest editors introduction: IEEE theme issue on software engineering in society," *IEEE Software*, vol. 37, no. 1, pp. 7–9, 2020.

[2] H. Pham, *Software Reliability*. Springer, 2000.

[3] S. Matteson, 2018. [Online]. Available: https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/

[4] A. J. Ko, B. Dosono, and N. Duriseti, "Thirty years of software problems in the news," in *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2014, pp. 32–39.

[5] M. Aniche, *Effective Software Testing: A Developer's Guide*. Manning Publications, 2022.

[6] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, 2013.

[7] G. Balogh, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Are my unit tests in the right package?" in *Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 137–146.

[8] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.

[9] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[10] M. Beller, A. Bacchelli, A. Zaidman, and E. Jürgens, "Modern code reviews in open-source projects: which problems do they fix?" in *Working Conf. on Mining Software Repositories (MSR)*. ACM, 2014, pp. 202–211.

[11] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Int'l Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 470–481.

[12] D. Han, C. Ragkhitwetsagul, J. Krinke, M. Paixao, and G. Rosa, "Does code review really remove coding convention violations?" in *Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 43–53.

[13] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub," in *Int'l Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[14] A. Rahman, A. Partho, D. Meder, and L. Williams, "Which factors influence practitioners' usage of build automation tools?" in *Int'l Workshop on Rapid Continuous Software Engineering (RCoSE)*, 2017, pp. 20–26.

[15] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 426–437.

[16] O. Elazhary, C. M. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M. D. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Trans. Software Eng.*, vol. 48, no. 7, pp. 2570–2583, 2022.

[17] E. Gelenbe and Y. Caseau, "The impact of information technology on energy consumption and carbon emissions," *Ubiquity*, pp. 1–15, 2015.

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

[19] N. Scarlat, M. Prussi, and M. Padella, "Quantification of the carbon intensity of electricity produced and used in europe," *Applied Energy*, vol. 305, p. 117901, 2022.

[20] E. Gelenbe, "Electricity consumption by ict: Facts, trends, and measurements," *Ubiquity*, no. 8, pp. 1–15, 2023.

[21] K. Kirkpatrick, "The carbon footprint of artificial intelligence," *Communications of the ACM*, vol. 66, no. 8, pp. 17–19, 2023.

[22] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations," *Patterns*, vol. 2, no. 9, 2021.

[23] A. Zaidman, "An inconvenient truth in software engineering? The environmental impact of testing open source Java projects," in *Int'l Conf. on Automation of Software Test (AST)*. ACM, 2024, pp. 214–218.

[24] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[25] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "Greenscaler: training software energy models with automatic test generation," *Empirical Software Engineering*, vol. 24, no. 4, p. 1649–1692, July 2018.

[26] R. Verdecchia, F. Ricchiuti, A. Hankel, P. Lago, and G. Procaccianti, "Green ICT research and challenges," in *International Conference on Environmental Informatics (EnviroInfo)*. Springer, 2016, pp. 37–48.

[27] E. A. Santos, C. McLean, C. Solinas, and A. Hindle, "How does docker affect energy consumption? evaluating workloads in and out of docker containers," *J. Syst. Softw.*, vol. 146, pp. 14–25, 2018.

[28] L. Cruz, "Tools to measure software energy consumption from your computer," July 2021, online; last accessed: July 3 2024. [Online]. Available: https://luiscruz.github.io/2021/07/20/measuring-energy.html

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

[30] E. Jagroep, J. M. E. M. van der Werf, S. Jansen, M. Ferreira, and J. Visser, "Profiling energy profilers," in *Proceedings ACM Symposium on Applied Computing*, ser. SAC '15. ACM, 2015, p. 2198–2203.

[31] L. Cruz, "Green software engineering done right: a scientific guide to set up energy efficiency experiments," October 2021, online; last accessed: August 31 2024. [Online]. Available: https://luiscruz.github.io/2021/10/10/scientific-guide.html

[32] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *Int'l Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 46–57.

[33] E. Barba Roque, L. Cruz, and T. Durieux, "Unveiling the energy vampires: A methodology for debugging software energy consumption," in *Int' Conf on Software Engineering (ICSE)*. IEEE, 2025, pp. 655–655.

[34] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in java-based open source software development," *Software: Practice and Experience*, vol. 54, no. 8, pp. 1408–1446, 2024.

[35] Anonymous, "Replication kit for "on the energy consumption of continuous integration in open-source java projects"," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17242523

[36] M. Viggiato, D. Paas, and C.-P. Bezemer, "Prioritizing natural language test cases based on highly-used game features," in *Proc. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2023, p. 1961–1972.

[37] H. Ritchie and P. Rosado, "Electricity mix," *Our World in Data*, 2020, last revised in January 2024. [Online]. Available: https://ourworldindata.org/electricity-mix

[38] K. Hagen, "E-compare: Automated energy regression testing for software applications," Master's thesis, TU Delft, 2024.

[39] N. Marini, L. Pampaloni, F. Di Martino, R. Verdecchia, and E. Vicario, "Green AI: Which programming language consumes the most?" in *Int'l Workshop Green and Sustainable Software (GREENS)*, 2025, pp. 12–19.

[40] Q. Perez, R. Lefeuvre, T. Degueule, O. Barais, and B. Combemale, "Software frugality in an accelerating world: the case of continuous integration," 2024. [Online]. Available: https://arxiv.org/abs/2410.15816

[41] R. Verdecchia, E. Cruciani, A. Bertolino, and B. Miranda, "Energy-aware software testing," in *Int'l Conf. on Software Engineering – New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2025, pp. 101–105.

[42] F. Kifetew, D. Prandi, and A. Susi, "On the energy consumption of test generation," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 360–370.

[43] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[44] E. Blokland, L. Cruz, and A. van Deursen, "Edata: Energy debugging and testing for android," in *Int'l Conf on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2025, pp. 94–104.