Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

From Code-Level Change to CI/CD Feedback: Empirical Validation of an Energy-Aware Software Framework

Muhammad Asif Khan
asif.khan@lut.fi
Lappeenranta-Lahti University of Technology
Mbanyick Joof
Lappeenranta-Lahti University of Technology
Shola Oyedeji
Lappeenranta-Lahti University of Technology
Jari Porras
Lappeenranta-Lahti University of Technology

Research Article

Keywords: Green software engineering, Energy-efficient software, Continuous integration and deployment, Energy-aware testing, Software energy measurement, Empirical validation

Posted Date: November 12th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10739499/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

From Code-Level Change to CI/CD Feedback: Empirical Validation of an Energy-Aware Software Framework

Muhammad Asif Khan[1*], Mbanyick Joof[1], Shola Oyedeji[1], Jari Porras[1]

[1*]LUT University, Lappeenranta, Finland.

*Corresponding author(s). E-mail(s): asif.khan@lut.fi;
Contributing authors: Mbanyick.Joof@lut.fi; shola.oyedeji@lut.fi; jari.porras@lut.fi;

Abstract

Purpose: Energy stays invisible in the development workflow: continuous integration pipelines report build status, tests, and latency, but not that a change raised energy use. This study provides an initial empirical operational validation of a previously proposed energy-aware feedback loop that connects code-level intervention, CI/CD energy measurement, and energy-regression detection, testing whether these three technical components operate together on real software rather than proposing a new framework.

Methods: The framework was applied to two open-source Spring Boot applications and a GitHub Actions pipeline, using controlled before-and-after measurements at ten runs per condition and injected energy regressions. Differences were reported as deltas, with effect sizes and average power used to separate runtime from power effects. The pipeline classified the per-build and per-endpoint energies against a runner-calibrated baseline.

Results: The investigated code-level interventions produced application-specific differences in measured energy and execution duration. PetClinic showed an approximately 5–8% reduction in total energy following the removal of an identifiable redundant repository lookup, whereas the RealWorld post-refactoring condition showed 21.1% lower measured energy and 17.3% shorter execution time. In both cases, the energy differences were primarily driven by the runtime rather than statistically established reductions in the average power. Within CI/CD evaluation, endpoint-level measurements showed low run-to-run variation in the evaluated baseline, and the pipeline detected and localized the injected regressions while distinguishing them from the observed runner variability.

Conclusion: The results provide an initial operational validation that code-level energy assessment, CI/CD energy feedback, and regression detection can function as an integrated feedback loop in the evaluated software systems. The results are bounded by the investigated interventions, applications, runner configuration, and injected regression scenarios; practitioner adoption remains to be evaluated separately.

Keywords: Green software engineering, Energy-efficient software, Continuous integration and deployment, Energy-aware testing, Software energy measurement, Empirical validation

1 Introduction

The energy consumed by a computing system depends heavily on the software it executes (Georgiou et al. 2019). Two implementations of the same functionality can differ significantly in the energy they draw, and at the scale of cloud and data center workloads, these differences accumulate into operating costs and carbon emissions (Katal et al. 2022; Kaur et al. 2020). Energy efficiency has become a quality attribute of software in its own right, governed by routine engineering decisions rather than hardware alone (Andringa et al. 2025; Ournani et al. 2020).

Acting on this attribute is difficult because energy remains invisible throughout the development process (Jagroep et al. 2017). A continuous integration pipeline reports the build status, test results, coverage, and latency but does not indicate that a change has increased the energy consumption (Kruglov et al. 2021). Developers receive immediate automated feedback on the correctness and performance of their code but not on the energy cost. Where energy is measured at all, it is usually assessed in separate studies conducted after release, once a regression has already reached production. Energy efficiency can become a routine quality concern only when feedback is available within the tools that developers already use (Pereira 2018).

A further point follows from the definition of energy as the power integrated over time. A code change can reduce energy consumption by either lowering the power consumption or reducing the execution time at a similar power level. These mechanisms are distinct, and this study treats the distinction as a result to report rather than an issue to discuss later. As shown below, both code changes reduce energy primarily by shortening the execution time rather than by reducing the power consumption.

Four strands of research address this issue. Software energy measurements provide tools and models for quantifying consumption (Gordillo et al. 2024; Guldner et al. 2024); work on green coding practices catalogs and organizes energy-relevant techniques (Koçak et al. 2015; Oyedeji et al. 2025); research on energy-aware continuous integration brings measurement into the pipeline (Perez et al. 2024; Aneggi et al. 2026); and studies on software testing quantify the energy consumed by test execution (Zaidman 2024; Danglot et al. 2024). Section 2 provides a detailed examination of these four strands. Each is well developed on its own and stops short of the same point. Measurement tools are validated as standalone instruments rather than continuous pipeline signals. Practice frameworks prescribe what to do without being

implemented in operations. Energy-aware CI work addresses the footprint or carbon timing of the pipeline, not the regressions in the code under change. Testing research shows that energy is measurable but does not gate a build. No existing study has validated a code-level change, in-pipeline energy measurement, and energy-as-test-failure together, end-to-end, on real software.

This study provides an initial empirical operational validation of the technical feedback loop proposed in our earlier framework (Khan et al. 2025). We did not introduce any new frameworks. We tested whether its three technical parts, code-level intervention, CI/CD energy measurement, and energy-regression detection, work together on real software and produce signals that a team could act on. This validation is limited to the interventions, applications, and pipelines we tested; it is not a claim that the framework holds in general. We apply it to two open-source Spring Boot applications, Spring PetClinic[1] and the Spring Boot RealWorld example[2], and to a GitHub Actions pipeline, VerdeFlow (Joof et al. 2026). The scope is deliberately narrow: two services, controlled before-and-after measurements, and injected regressions on a single pipeline. We read the results as preliminary and bounded, not as population-level estimates.

This study is organized around one main research question, broken into three subquestions:

Main RQ. How can an energy-aware software engineering framework be operationalized to connect code-level energy measurements, CI/CD feedback, and energy regression detection in real software systems?

RQ1. What changes in the measured energy consumption, execution time, and average power are observed following targeted code-level interventions under controlled conditions?

RQ2. What measurement stability is observed when build-level and endpoint-level energy feedback are integrated into a CI/CD pipeline running on a shared infrastructure?

RQ3. How can runner-calibrated build- and endpoint-level measurements detect and localize the injected energy regressions while distinguishing them from the observed shared-runner variability?

The main contribution of this study is the initial empirical operational validation of an integrated energy-feedback loop: code-level change assessment, CI/CD energy measurement, and energy-regression detection, run together on real software. Across the evaluated cases, we show that the energy effect of a code change can be measured against an application-specific baseline, that this feedback can be surfaced at the build and endpoint levels inside a standard CI/CD workflow, and that runner-calibrated measurements can tell the injected regressions apart from ordinary infrastructure variability. This is evidence that the three technical parts of the framework can operate

[1] https://github.com/spring-projects/spring-petclinic
[2] https://github.com/gothinkster/spring-boot-realworld-example-app

together, but it does not validate the framework as a whole. The experiments also surface baseline maintenance and infrastructure variability as technical constraints that any adoption effort would need to handle.

Section 2 presents the background and identifies the research gaps. Section 3 describes the proposed framework, and Section 4 explains the study design and the methodology. Section 5 presents the results of the study. Section 6 discusses the adoption and implications, Section 7 addresses threats to validity, and Section 8 discusses future work and concludes the paper.

2 Background

This section establishes the study's conceptual foundations. It encompasses four interconnected areas: software energy measurement, green coding practices, CI/CD energy integration, and energy-aware testing. The following subsection defines the concept and identifies the specific gap addressed by this study.

2.1 Software energy measurement

Measuring the energy consumed by a program is a precondition for improving it, and the available methods trade accuracy for portability (Fahad et al. 2019). Hardware-based approaches read external power meters or on-chip counters, such as Intel RAPL, to obtain ground truth values (Guldner et al. 2024). They are accurate but tied to specific processors and privileged interfaces, which limits their use on shared, ephemeral machines that run continuous integration (Schubert et al. 2012). Software-based estimators, such as PowerAPI (Noureddine et al. 2013), Jolinar (Noureddine et al. 2016), and EnergiBridge (Sallou et al. 2023), instrument a process at runtime with lower hardware dependency but higher measurement uncertainty. EnergiBridge runs on Linux, Windows, and macOS on Intel, AMD, and Apple silicon, making it portable across heterogeneous developer and CI environments.

Recent tools have narrowed this accuracy gap. CodeGreen[3] separates instrumentation from measurement and reports $R^2 = 0.99$ against RAPL across Python, C++, C, and Java (Rajput and Sharma 2026), and METRION attributes energy at thread level while controlling the background activity and thermal drift that inflate variance in long-running measurements (Weigell et al. 2025). Cloud- and container-oriented estimators such as CodeCarbon [4] and Scaphandre [5] make energy tracking usable in ordinary build and deployment environments without on-chip instrumentation (Bouza et al. 2023), which is why this study builds on CodeCarbon rather than a RAPL-only tool.

2.2 Green coding practices

Green coding is a set of development practices that reduce a program's energy use without sacrificing correctness or functionality (Oyedeji et al. 2025; Jünger et al. 2024). Koçak et al. (2015) showed that sustainability criteria can be integrated into standard

[3] https://smart-dal.github.io/codegreen/
[4] https://codecarbon.io/
[5] https://hubblo-org.github.io/scaphandre-documentation/

software quality models, which lets energy be treated as a quality attribute alongside correctness and performance rather than a hardware concern. Documented practices operate at several levels: algorithmic choice, data structure selection, input/output (I/O) batching, caching, and removal of redundant computation. Catalogues of life-cycle techniques and tools now span most development stages (Georgiou et al. 2019). The practice this study exercises sits at the code level: redundant repository access inside a request is a known anti-pattern in ORM-backed systems (Chen et al. 2014), and code-level changes of this kind measurably alter energy use (Sahin et al. 2014; Poy et al. 2025).

Adoption lags behind awareness. Pang et al. (2016) found that developers hold limited and often inaccurate knowledge of where software spends energy. Practitioner studies attribute this to missing tool support rather than indifference; developers treat energy as relevant but receive no feedback in their normal workflow (Manotas et al. 2016). Our own state-of-practice survey (Oyedeji et al. 2025) found the same pattern, with approximately 17% of respondents reporting green coding practice in their projects. A cross-organization study in Finland and internationally Moktan et al. (2025) identified the absence of automated measurements in the development loop as the primary barrier to its adoption. One barrier is workflow integration: developers know which changes reduce energy, but receive no signal indicating whether a given change helped or hurt.

2.3 CI/CD energy integration

Continuous Integration and Continuous Deployment (CI/CD) pipelines automate the build, test, and deployment processes and run on every change, making them the natural place to surface energy feedback. Drangmeister et al. (2013) first proposed measuring energy efficiency as a build-quality signal in the CI loop, but early work stayed at the level of proposals. Perez et al. (2024) quantified the problem across 396 Java repositories on GitHub Actions: a single run costs on the order of 10 Wh, but repeated invocation raises the aggregate to roughly 22 kWh and 10.5 kg CO_2 per project. The per-run energy is low, but the cumulative cost is not.

Recent CI/CD energy work mostly pulls a different lever, lowering the carbon of a fixed workload by shifting when or where it runs, rather than reducing the energy of the code. Classen et al. (2024) defer jobs to periods of lower grid-carbon intensity, Laskar (2025) add geographic migration and workload consolidation, and Dhawan et al. (2026) treat carbon as a first-class input to DevOps automation. None of these tools inform a developer that a commit has caused the code to draw more energy. In a related study, Aneggi et al. (2026) measured the energy of an application under test in CI using PPTAM-η, whereas our earlier work (Khan et al. 2025) proposed the framework validated in this study. Both studies show that in-pipeline energy measurement is feasible, but neither validates the full loop end-to-end on real applications nor attributes an energy regression to the code that caused it at the granularity of a changed end-point.

2.4 Energy-aware testing

Energy regression testing is the energy analog of performance regression testing, which has an established body of work (dos Santos et al. 2025). One line of research measures the energy costs associated with testing. Zaidman (2024) quantified the energy of continuous integration and testing across ten open-source Java projects and showed that automated suites carry a non-trivial, largely invisible cost, and Kifetew et al. (2025) found that automated test generation is itself energy-intensive and that the technique chosen changes the cost meaningfully. This line treats the test suite as a consumer of energy, not as an instrument for catching energy regressions in the code that it executes.

The second line treats energy as an attribute that can be tested. Early frameworks, such as JalenUnit (Noureddine et al. 2014) and JouleUnit (Wilke et al. 2013), demonstrated that energy can be profiled at the granularity of the unit tests. Building on this, Danglot et al. (2024) showed that developers' own tests can reveal energy regressions in a CI context; they injected regressions through automated program mutations, localized them to source lines, and applied statistical filtering to separate real differences from measurement noise. Their measurements were noisy enough that CPU cycles were adopted as a more stable proxy for energy, which leaves open whether an energy signal can be read directly on infrastructure that the developer does not control. This is the closest work to the present study, and the difference is one of deployment: their oracles classify offline and remain configurable, evaluated on controlled hardware, rather than a fixed policy that gates a live pipeline against a calibrated baseline at the level of the changed end-point.

2.5 Research gap

The four threads come together around a single unfulfilled requirement: measurement tools have been validated against hardware ground truth on dedicated machines, but their performance as a continuous signal on shared build infrastructure, which is stable enough from run to run to distinguish a genuine regression from noise, has not been confirmed yet. Green coding practices are documented, and their value is understood; however, developers receive no feedback on them in the workflow where changes are made. CI energy work measures the pipeline's own footprint or shifts its carbon timing but does not attribute energy regression to the code that caused it. Testing research shows that energy is measurable through tests without gating a build on the results.

What is missing is a single mechanism, shown to work on real software, that measures energy at the level of a changed endpoint, surfaces regressions automatically within CI/CD, and gates builds on energy thresholds as conventional pipelines gate on failing tests. To the best of our knowledge, prior studies have not demonstrated this combination end-to-end on real applications in a live CI/CD pipeline.Tables 1 and 2 summarize this comparison: the first covers the measurement scope and experimental infrastructure, while the second details the capabilities required by an operational energy gate.

Table 1: Comparison of related work based on measurement scope and experimental infrastructure

Study	Primary lever	Measurement approach	Infrastructure	Subject systems	Energy granularity
Drangmeister et al. (2013)	Energy as a build signal	Conceptual proposal	Not specified	None	Not defined
Perez et al. (2024)	Pipeline energy footprint	Workflow-level estimation	GitHub Actions	396 Java repositories	Pipeline run
Classen et al. (2024)	Carbon-aware pipeline timing	Grid carbon-intensity estimation	GitHub Actions	7,392 workflow runs	Workflow job
Zaidman (2024)	Energy cost of testing	Direct energy measurement	Local execution environment	10 open-source Java projects	Test level
Danglot et al. (2024)	Energy regression through developer tests	RAPL, TLPC-sensor, Linux perf, developer tests, and program mutations	△ Controlled RAPL-compatible host; hardware configuration not reported	Six open-source Java projects	Test level
Aneggi et al. (2026)	Energy consumption of the application under test	RAPL-based probes using PowerJoular and Linux perf	Dedicated bare-metal testbed	One self-designed JWT API evaluated across four commits	Host and container level
Our study	**Energy impact of code changes**	**CodeCarbon measure-ments at machine and CI-runner levels**	**Unmodified shared CI runners**	**Two Spring Boot applications and one REST API**	**Build and endpoint levels**

3 Proposed Framework

This framework builds on the conceptual model introduced in our previous research by Khan et al. (2025), as an operational framework for integrating energy awareness into the software-development workflow. As shown in Figure 1, it connects three technical practices into a single feedback loop: code-level intervention (called API optimization in the earlier framework paper), CI/CD energy integration and energy-aware testing. The design principle is that each part targets a distinct source of energy waste, but their value increases when they are operated in combination. The measurement in the pipeline makes energy consumption visible. Energy-aware testing converts energy regression into a build-level signal. The code-level intervention component then supplies the corrective action to which the signal points. Therefore, the framework is not defined by the three parts in isolation but by the loop that connects them within the development process itself.

These three parts were chosen to close the path from detection to correction. Code-level intervention addresses code-level waste, where redundant or unnecessary calls,

Table 2: Comparison of related work based on regression detection, statistical analysis, and operational validation

Study	Calibrated baseline	Code-level regression detection	Operational CI gate	Statistical analysis	Runtime and energy distinction	End-to-end validation
Drangmeister et al. (2013)	— Proposal only	— Proposal only	△ Build-level signal proposed	— Proposal only	—	✗ Not evaluated
Perez et al. (2024)	—	—	✗ No	△ Descriptive	—	△ Evaluated on repository workflows
Classen et al. (2024)	—	—	✗ No	△ Descriptive	—	△ Evaluated on workflow runs
Zaidman (2024)	—	—	✗ No	△ Descriptive	—	△ Evaluated on open-source projects
Danglot et al. (2024)	✗ No persistent calibrated baseline	✓ Mutation and commit-level comparison	△ Configurable pass or break classification; no deployed CI gate	✓ Student's t-test and Cohen's d, $n = 100$ per version	△ Conceptual distinction; CPU cycles used as an energy proxy	△ Evaluated on open-source projects in a controlled environment
Aneggi et al. (2026)	✗ No	△ Version comparison with manual inspection	✗ Dashboard only	✗ Statistical testing named as future work	✗ No	✗ Evaluated on a synthetic application
Our study	**✓ Runner-calibrated baseline**	**✓ Automated endpoint-level localization**	**✓ PASS, WARN, and FAIL policy executed in the CI pipeline**	**✓ Welch's t-test, Cohen's d, and Mann–Whitney U, $n = 10$ per condition**	**✓ Execution time and average power reported separately**	**✓ Evaluated end to end on the two applications and one pipeline tested**

✓ Reported or implemented; △ partially reported, proposed, or evaluated with limitations; ✗ in scope but not reported or implemented; — outside the aim of this study.

particularly database access, draw energy without adding functions (Chen et al. 2014; Sahin et al. 2014). CI/CD integration addresses visibility: pipelines automate build, test, and deploy but do not report energy; thus, a regression ships unseen. Energy-aware testing addresses timing by treating an energy increase as a test outcome; thus, a regression is caught at the point of change rather than in a later study. Each part

supplies what the next needs, which is why the proposal combines them rather than offering three separate tools.

The framework focuses on the iterative loop and workflow integration that connects the components. By design, it excludes the choice of measurement tool, hardware platform, and fixed energy threshold, as these depend on the project context and the runner's environment. This separates the framework from the tools reviewed in Section 2, which typically address only one aspect of metrics, API analysis, or CI/CD, and are often tied to a specific tool, platform, or configuration. The evaluation reported here covers only the technical loop, exercised through the three parts described in Section 4.

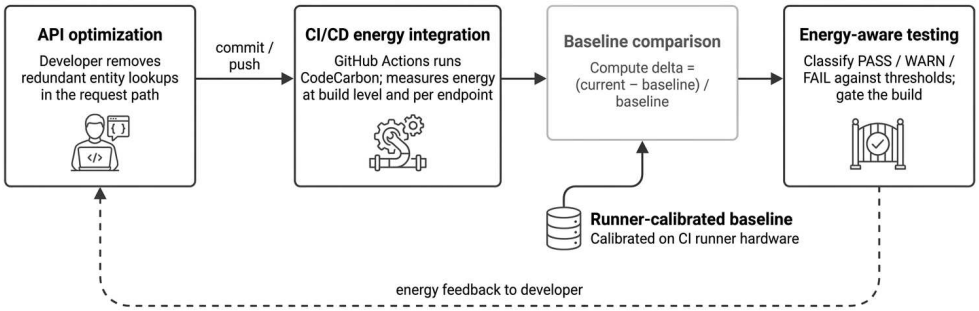


Fig. 1: The green coding framework. "API optimization" here corresponds to the code-level intervention evaluated in this study.

4 Study design

4.1 Overview

This study is an empirical operational validation of the framework described in Section 3. The design combines two forms of evidence: controlled before-and-after experiments for the code-level intervention part, and a pipeline case study with fault injection for the CI/CD and energy-aware testing parts.

The three framework components were evaluated using different forms of evidence. Code-level intervention is assessed through statistical comparison, CI/CD integration through implementation feasibility, and energy-aware testing through fault injection. This distinction prevents the study from overstating its findings. Each component is evaluated only at the level supported by its evidence and mapped to the corresponding research question.

- **Code-level intervention (RQ1).** The data were evaluated experimentally. For each of the two applications, the energy was measured before and after a targeted code change under fixed conditions, and the difference was statistically tested. PetClinic is treated as an optimization with an identified mechanism; RealWorld is

treated as a refactoring for which a before-and-after difference is established, but the mechanism is not.

- **CI/CD energy integration (RQ2).** This was evaluated as an implementation and feasibility issue. Energy measurement was built into a GitHub Actions pipeline to establish whether per-build and per-endpoint feedback could be produced on a standard, shared infrastructure.
- **Energy-aware testing (RQ3).** A fault injection method was used for the evaluation. Energy regressions of known types and magnitudes were introduced, and the pipeline was checked to determine whether it could detect and separate them from infrastructure noise.

The parts differ in evidential strength: one is a statistical comparison, one is a feasibility demonstration, and one is a detection test for the injected faults. The claims are based on the strength of each design and are not generalized beyond these claims.

Two decisions bound this research. First, the framework was applied to two open-source Spring Boot applications rather than one, so the intervention results do not depend on a single build system or application shape (see Figure 2). Spring PetClinic (Maven, Spring Data JPA over a relational schema) and Spring Boot RealWorld example (Gradle, H2, content, and commenting API). Both are established, non-synthetic systems; therefore, the measured changes reflect the code patterns that developers write in practice. Second, the energy was measured using a single machine under fixed system conditions. This holds the measurement environment constant across the baseline and post-change conditions, which the before-and-after comparison requires, but fixes the absolute energy figures to one hardware configuration; the results are read as relative reductions, not portable absolute values. The subject applications, code changes, instrumentation, and measurement protocols are described in Sections 4.2-4.6.

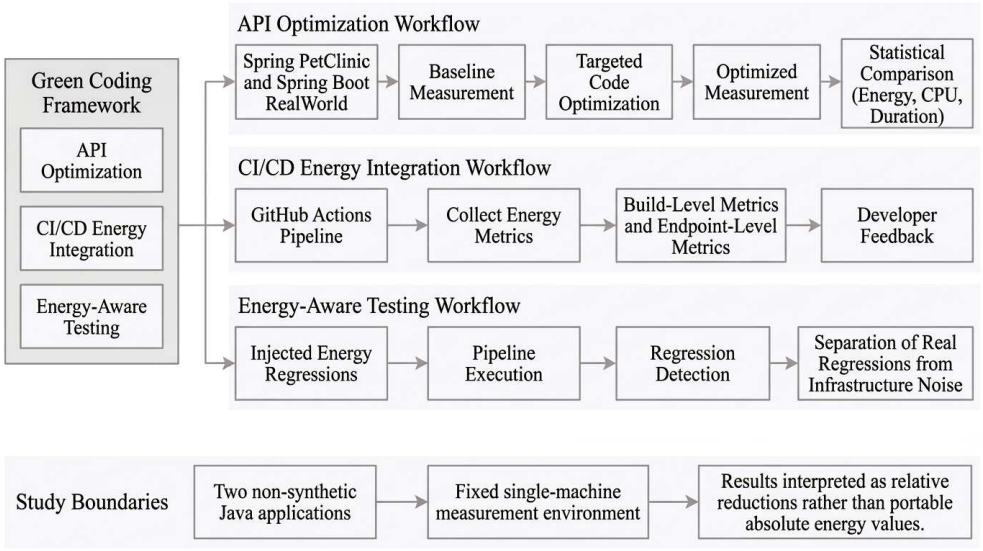


Fig. 2: Study design: code-level intervention (RQ1), CI/CD integration (RQ2), regression detection (RQ3).

4.2 Subject applications

Two established open-source Spring Boot applications were used rather than synthetic microbenchmarks, allowing the interventions to be evaluated on realistic application structures and development workloads.

Spring PetClinic is the canonical Spring reference application. It was built using Maven and used Spring Data JPA over a relational schema for owners, pets, visits, and veterinarians. The measured workload is the project's own test suite, run with `mvnw clean test`, which exercises the web controllers, JPA repositories, and service layer against an in-memory database. The intervention target is the owner-pet request path in `PetController`.

Spring Boot RealWorld implements the RealWorld (Conduit) API specification: a content backend with articles, comments, users, and favorites. It was built using Gradle 7.4 (Java 17) and used H2 as the datastore. The measured workload was the suite run with `gradlew cleanTest test`, which drives the article and comment endpoints through their respective service and repository layers. The intervention targets the article lookup paths in `ArticleApi` and `CommentsApi`.

For both applications, the unit of measurement was one full execution of the test suite from a fresh JVM under the protocol described in Section 4.5. Test-suite execution is a workload rather than a synthetic load generator; thus, each run drives the same code paths as those triggered by a developer when validating a change locally or in CI. The mean run duration was 66.1s for PetClinic and 45.1s for RealWorld at baseline, which set the timescale of each energy measurement.

4.3 Code-level interventions under test

The two investigated code changes were associated with lower measured energy consumption and shorter execution time under the controlled before-and-after protocol described in Section 4.5, although the mechanism and magnitude differed between cases. Both changes leave the public API, persistence layer, build configuration, and test suite unchanged, preserve observable behavior, and pass the unmodified test suites. Therefore, the localized code change was the only variable manipulated in each application.

In PetClinic, a request resolves the same owner twice: `findOwner` loads it for the model, and `findPet` loads it again before returning the pet. The change reuses the already resolved instance and drops the second lookup (Figure 3). This is described as an optimization because the mechanism is explicit: a specific and identifiable redundant repository call is removed from the request path.

In RealWorld, the same article lookup is duplicated inline across five request-handling methods in `ArticleApi` and `CommentsApi`; the change extracts it into one helper that each method reuses (Figure 4). One method, `updateArticle`, may assign a new slug when the title changes; therefore, its response is built from the updated slug to preserve baseline behavior. This is described as refactoring rather than optimization. It consolidates the duplicated code, but the lookup is still executed once per method call in both the baseline and changed versions. The only difference is where the call is written, not on the available evidence, how often it runs. The current implementation evidence does not establish a reduction in repository access per request; therefore, the RealWorld experiment demonstrates a measured before-and-after difference without establishing that removing redundant database operations caused it.

These two changes differ in scope and what they establish. PetClinic removes one redundant lookup from a single controller path using a stated mechanism. RealWorld consolidates a repeated lookup from five methods across two controllers, and the larger measured difference is consistent with the larger amount of duplicated code touched; however, this consistency is not evidence of the mechanism: the applications also differ in build system and workload and are not directly comparable in magnitude.

4.4 Energy measurement and instrumentation

Energy was measured with CodeCarbon 3.2.6, using the `OfflineEmissionsTracker` with `country_iso_code = "FIN"` and `tracking_mode = "machine"`. A Python wrapper started the tracker, executed the build command through a subprocess, stopped the tracker, and appended one summary row for each run. It records the wall clock duration, total energy, CPU energy, RAM energy, and estimated emissions. Only the runs with a return code of 0 were retained.

All runs used one dedicated machine: an Intel Core i7-7600U (two cores, four threads) with 31.9 GB RAM, Windows 11, plugged in, with no other user workload active. The machine had no discrete GPU; therefore, the `gpu_energy` was zero for all 40 runs. The total energy was the sum of the CPU and RAM component energies.

Baseline

```
@ModelAttribute("pet")
public Pet findPet(@PathVariable("
ownerId") int ownerId,
       @PathVariable(name = "petId"
,
       required = false) Integer
petId) {
    if (petId == null) {
        return new Pet();
    }
    Optional<Owner> optionalOwner =
           this.owners.findById(
ownerId).
    Owner owner = optionalOwner.
orElseThrow(
           () -> new
IllegalArgumentException(
                   "Owner not found:
 " + ownerId));
    return owner.getPet(petId);
}
```

Optimized

```
@ModelAttribute("pet")
public Pet findPet(@ModelAttribute(
"owner") Owner owner,
       @PathVariable(name = "petId"
,
       required = false) Integer
petId) {
    if (petId == null) {
        return new Pet();
    }
    return owner.getPet(petId);
}
```

Fig. 3: PetClinic optimization. The baseline version reloads the owner inside `findPet`, even though `findOwner` has already resolved it; the optimized version reuses the `@ModelAttribute("owner")` binding and removes the second lookup.

Baseline

```
// repeated inline in updateArticle
, deleteArticle,
// createComment, getComments,
deleteComment
Article article = articleRepository
       .findBySlug(slug)
       .orElseThrow(
ResourceNotFoundException::new);
```

Refactored

```
private Article getArticle(String
slug) {
    return articleRepository
           .findBySlug(slug)
           .orElseThrow(
ResourceNotFoundException::new);
}
// each method now calls:
Article article = getArticle(slug);
```

Fig. 4: RealWorld refactoring. The baseline version repeats the same article lookup inline in five methods across `ArticleApi` and `CommentsApi`. The refactored version extracts it into a single `getArticle` helper that each method reuses.

Two properties of this setup govern the interpretation of these results. First, machine-level tracking attributes the entire machine energy over the measured interval rather than just the energy of the test process. Therefore, the total energy includes background operating system activity, which the fixed conditions hold constant across the baseline and post-change runs but is not removed. Second, the absolute values are hardware-specific. The measurement supports the relative comparison within each application, not a portable absolute energy figure; results are reported as differences,

not as reductions attributed to a single cause. CPU energy is the component closest to the code change; however, because the measurement is machine-level, a CPU energy change supports a mechanism rather than proving it in isolation from other factors. These limitations are discussed in Section 7.

4.5 Measurement protocol

For each application, 20 runs were collected: 10 baseline runs on the unmodified code and 10 post-change runs on the modified one. Runs were blocked by condition within each session for every application. A warm-up build preceded each block and was not included in the final count. Each run was an independent execution of the test suite from a fresh JVM using the same JDK, build configuration, and CodeCarbon settings under both conditions. No run was discarded, except for a non-zero return code, of which there were none.

Energy measurements vary between otherwise identical runs, and controlling this variation requires a deliberate experimental design (Ournani et al. 2020). Blocking the conditions in time introduces a further risk: if the machine drifts over the session, the drift confounds the code changes. The design does not prevent this; therefore, it was tested rather than assumed. Within each block, energy is regressed against the run index using Spearman's ρ; a trend indicates drift within that block. This check is reported per metric in Section 5, and one baseline block does show a trend, which is why the results are read with it rather than without it.

Total energy per run (kWh) was the primary metric; CPU energy, RAM energy, and duration were secondary. Run-to-run stability was reported as the coefficient of variation per condition, and per-run consistency was defined as the count of post-change runs below the baseline mean.

4.6 Statistical analysis

Each application is assessed separately rather than collectively because they differ in their build systems, persistence layers, and workloads. We compared the baseline and post-change scenarios for the total energy, CPU energy, RAM energy, and duration, with each reported as a percentage change; therefore, the outcome was a delta rather than a specific absolute energy for the hardware. Because energy is power integrated over time, and each intervention also shortens the runtime, we additionally report the average power (energy divided by duration per run), which separates a reduction in the duration of the code execution from a reduction in power consumption.

Means were compared using Welch's two-sided t-test at $\alpha = 0.05$, which does not assume equal variance, and effect size with Cohen's d (0.2 small, 0.5 medium, 0.8 large). Both are reported because they answer different questions: a change is worth acting on only when it is significant, and the effect is at least large enough. At $n = 10$ normality is not assumed, so we report skewness and kurtosis per condition and add the Mann-Whitney U test as a distribution-free check; where it agrees with the t-test, the result does not depend on the distribution, and where they diverge, the Mann-Whitney result is weighted more. The ten runs per condition provide evidence for the observed within-application differences, but the sample size limits the precision of

the estimates. We therefore report effect sizes and distribution-free checks alongside significance tests and restrict interpretation to the evaluated cases.

4.7 CI/CD pipeline and baseline design

RQ2 and RQ3 were evaluated using a second application, Spring PetClinic REST, which was integrated with a GitHub Actions pipeline (VerdeFlow). This is a different system from the PetClinic used for RQ1: a REST API exposing 14 endpoints over six domain entities, backed by an H2 in-memory database with fixed seed data, whose endpoints provide a stable, repeatable set of request paths to measure and inject faults.

Instrumentation. VerdeFlow measures energy at two levels of granularity. Build-level measurement wraps the test suite (`mvnw test`) and application startup. Per-endpoint measurement warms the JVM and H2 caches, then profiles each endpoint separately, wrapping 200 sequential GET requests in a dedicated CodeCarbon tracker at 1-second sampling to record energy per request.

Baseline. Each measurement was compared against a baseline of mean and standard deviation over N runs, stored per mode and per endpoint, and computed only from the runs on the CI runner (`ubuntu-latest`). This is deliberate because baselines seeded from an Apple Silicon laptop differed from the runner measurements by an order of magnitude. Therefore, all RQ2 and RQ3 figures are relative deltas against a runner-calibrated baseline, not absolute values comparable to RQ1.

Classification. For a measurement E_{current} against a baseline mean $\overline{E}_{\text{baseline}}$, the fractional delta is

$$\Delta_f = \frac{E_{\text{current}} - \overline{E}_{\text{baseline}}}{\overline{E}_{\text{baseline}}}, \tag{1}$$

classified against two thresholds as

$$\text{Class} = \begin{cases} \text{PASS}, & \Delta_f < 0.50, \\ \text{WARN}, & 0.50 \leq \Delta_f < 1.00, \\ \text{FAIL}, & \Delta_f \geq 1.00. \end{cases} \tag{2}$$

The thresholds are wide because shared runners show 20-50% inter-run variance from co-tenancy, frequency scaling, and thermal throttling. The injected regressions in Section 5.6 sit well above +100%, so they remain clearly separable from the noise.

Two-phase workflow. The pipeline separates the evaluation from the calibration across the two tracks (Figure 5). Phase 1 runs on every push and pull request: it measures both modes, classifies each by Equations (1) and (2) against the committed baseline, posts the report as a pull-request comment, and blocks the pull request on a FAIL. Phase 2 runs on the master branch only, and runs unconditionally: over N further runs it recomputes the baseline as

$$\overline{E}_{\text{baseline}} = \frac{1}{N} \sum_{i=1}^{N} E_i, \tag{3}$$

and commits the result by updating the reference for all subsequent Phase 1 checks. The unconditional design was deliberate. An earlier version gated recalibration on Phase 1 passing, which deadlocked when the runner hardware rotated: a stale baseline made every measurement look like a regression, blocking the recalibration that would have fixed it. Running Phase 2 unconditionally keeps the baseline tracking the current runner while the pull requests stay gated.

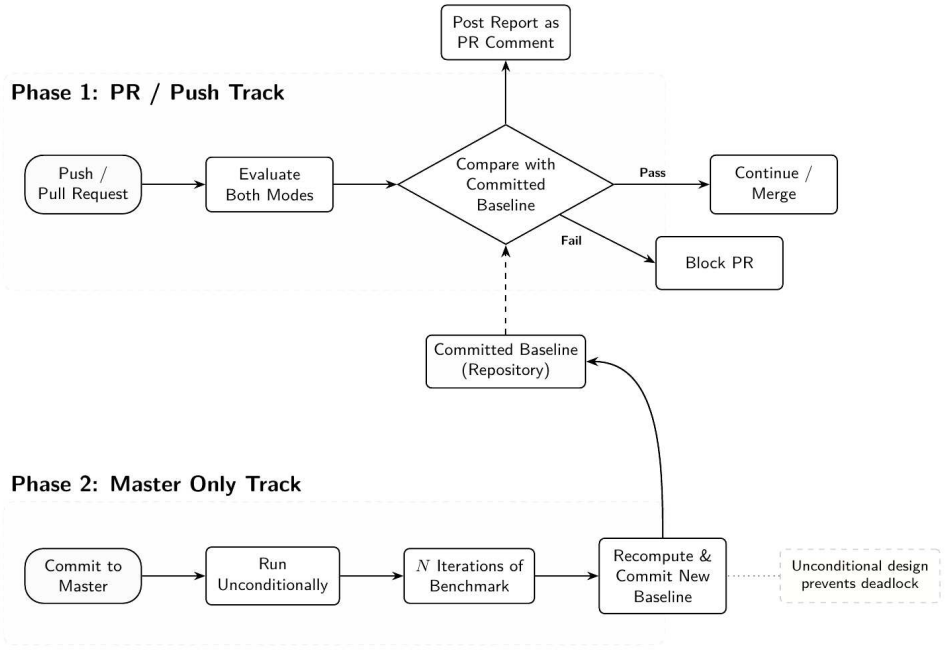


Fig. 5: The two-phase VerdeFlow pipeline. Phase 1 checks every push and pull request against the baseline and gates the merge process. Phase 2 is unconditionally recalibrated on the master. Decoupling calibration from per-change checking prevents the runner-rotation deadlock.

5 Results

5.1 Overview

This section reports the code-level intervention experiment (RQ1) conducted in the study. A total of 40 runs were collected, comprising 10 baseline and 10 post-change runs for each application. Four metrics were recorded for each run: total energy, CPU energy, RAM energy, all measured using CodeCarbon, and wall-clock duration. The two applications were analyzed separately rather than pooled. Each metric is reported as a percentage change from baseline to post-change, with Welch's t-test, two-sided

p-values, and Cohen's d effect sizes. The percentage change is the primary quantity because the absolute values are hardware-specific. Where a distribution check changes how a result should be interpreted, the Mann-Whitney U test and within-block Spearman trend are reported alongside it. Table 3 presents the descriptive statistics, and Table 4 presents the comparison and effect sizes of the two groups.

The two investigated code changes were associated with lower measured energy consumption and shorter execution time under the controlled before-and-after protocol, although the mechanism and magnitude differed between the cases (Figure 6). The PetClinic change removed an identifiable redundant repository lookup and was associated with an approximately 5-8% reduction in the measured energy, primarily through a shorter execution time. The RealWorld refactoring was associated with a 21.1% reduction in the measured energy and a 17.3% reduction in the execution time. Because the current implementation evidence does not establish a reduction in repository access per request, the experiment demonstrates a measured before-and-after difference but does not establish that the removal of redundant database operations caused that difference. Section 5.2 reports the PetClinic case in detail, Section 5.3 reports RealWorld, and Section 5.4 compares the two.

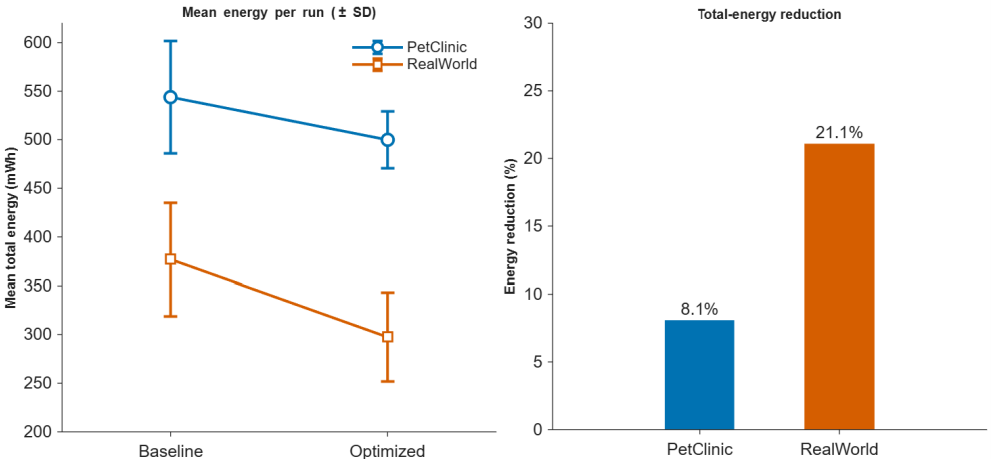


Fig. 6: Energy comparison between the PetClinic and RealWorld applications under baseline and post-change conditions.

The two cases are discussed in Sections 5.2 and 5.3, and the differences between them are discussed in Section 5.4.

5.2 PetClinic

In PetClinic, removing an identifiable redundant owner lookup was associated with a lower total energy and shorter runtime. This change reduced the total energy by 8.1% (Table 4). The reduction is significant with a large effect, but p sits on the 0.05 threshold, and the 95% confidence interval reaches close to zero. The RAM energy

Table 3: Descriptive statistics per application, metric, and condition (10 runs each). CV is the coefficient of variation (SD/mean). Energy in mWh, duration in s

Metric	Condition	Mean	SD	CV	Min	Median	Max
PetClinic							
Total energy	Baseline	543.9	57.7	10.6%	506.1	516.9	688.2
	Post-change	499.9	28.8	5.8%	471.3	487.8	545.6
CPU energy	Baseline	190.7	34.6	18.1%	168.0	175.9	279.3
	Post-change	172.3	14.3	8.3%	158.2	167.2	196.3
RAM energy	Baseline	353.3	23.3	6.6%	337.7	340.2	409.0
	Post-change	327.6	14.9	4.5%	312.2	320.0	349.2
Duration	Baseline	66.1	4.2	6.4%	63.3	63.8	76.2
	Post-change	61.2	2.9	4.8%	58.3	59.6	65.5
RealWorld							
Total energy	Baseline	377.0	58.1	15.4%	288.5	362.1	480.6
	Post-change	297.5	45.7	15.4%	267.4	276.0	413.6
CPU energy	Baseline	127.7	21.6	16.9%	94.1	126.7	174.2
	Post-change	99.0	25.2	25.4%	83.1	86.5	163.9
RAM energy	Baseline	249.3	40.5	16.2%	194.3	238.7	331.5
	Post-change	198.5	20.7	10.4%	184.3	189.4	249.7
Duration	Baseline	45.1	5.3	11.8%	36.5	44.5	57.3
	Post-change	37.3	3.8	10.1%	34.7	35.6	46.6

and duration decreased by similar amounts and were both significant; the CPU energy moved in the same direction but did not reach significance, so it was directional evidence rather than a confirmed component-level effect.

The baseline block was not stationary either. The total energy rises with the run index within the baseline (Spearman $\rho = 0.87$, $p = 0.001$): the first six runs cluster near 510 mWh, then runs 7 to 9 rise to 562-688 mWh, while the post-change block shows no such trend. Because runs were blocked by condition, this drift raised the baseline mean and inflated the measured reduction, leaving the baseline distribution right-skewed, whereas the post-change distribution was not (Table 3).

The effect survived the drift. The Mann-Whitney U test, which down-weights the high baseline values, still finds the difference significant ($p = 0.031$), and 9 of 10 post-change runs fall below the baseline mean. However, the mean-based figure overstates it: comparing medians, which the drift does not move, gives a 5.6% reduction. The reduction is real, and its size is better read as roughly 5 to 8% than as a single-point estimate, consistent with removing one redundant lookup from a single controller path, where duration, RAM, and total energy fall together, and CPU moves the same way without reaching significance.

5.3 RealWorld

In RealWorld, the post-refactoring condition showed a 21.1% lower mean total energy estimate than the baseline condition ($p = .0034$, $d = 1.52$), along with reductions in CPU energy, RAM energy, and execution duration. The average power decreased by 4.72%, but this difference did not reach statistical significance ($p = .0919$). These

results establish a measurable before-and-after difference for the investigated refactoring under experimental conditions. However, because the current code-level evidence does not establish that the refactoring reduced the number of repository accesses executed per request, the observed difference should not be interpreted as evidence that eliminating redundant database lookups caused this reduction.

The baseline block remained stationary. Total energy shows no trend with run index (Spearman $\rho = 0.03$, $p = 0.93$), so the drift that affects PetClinic is absent, and the mean is not inflated by a late excursion. The Mann-Whitney U test confirms the difference ($p = 0.003$), and nine of ten post-change runs fall below the baseline mean; the result therefore depends on neither a single run nor the choice of test.

CPU energy is significant here, unlike in PetClinic. The refactoring consolidated the repeated article lookups across the five methods into one call site. The CPU reduction observed alongside this change is consistent with less computational work; however, because the per-request lookup count is not independently verified and the measurement is machine-level, this reduction is read as part of a coherent before-and-after pattern across CPU, RAM, and total energy, not as confirmation of the underlying mechanism.

Table 4: Baseline vs post-change comparison, per application. Energy and duration were measured in mWh and s, respectively. Reduction is the percentage decrease from baseline to post-change mean. p from Welch's two-sided t-test

Metric	Baseline	Post-change	Reduction	t	p	d
PetClinic						
Total energy (mWh)	543.9	499.9	8.10%	2.16	0.0496*	0.97
CPU energy (mWh)	190.7	172.3	9.65%	1.56	0.1458	0.70
RAM energy (mWh)	353.3	327.6	7.26%	2.93	0.0102*	1.31
Duration (s)	66.14	61.23	7.43%	3.02	0.0080*	1.35
Avg. power (W)	29.5	29.4	0.54%	0.41	0.6869	0.19
RealWorld						
Total energy (mWh)	377.0	297.5	21.10%	3.40	0.0034*	1.52
CPU energy (mWh)	127.7	99.0	22.50%	2.74	0.0136*	1.23
RAM energy (mWh)	249.3	198.5	20.38%	3.53	0.0035*	1.58
Duration (s)	45.06	37.25	17.33%	3.80	0.0016*	1.70
Avg. power (W)	30.0	28.6	4.72%	1.80	0.0919	0.80

* $p < 0.05$

5.4 Cross-application comparison

The two applications agreed on the direction but disagreed on the size and the results. Both showed lower measured energy and shorter runtime after the code-level change, but RealWorld's reductions were approximately 2.5 times PetClinic's (21.1% versus 8.1% on total energy, with a similar gap on CPU, RAM, and duration). RealWorld is significant on all four metrics with a stationary baseline, while PetClinic is significant

on three, borderline on total energy, not significant on CPU, and carries a baseline drift that puts its reduction closer to 5-8%.

The size difference tracks the amount of duplicated code touched: PetClinic removes one duplicate lookup from a single controller path, and RealWorld consolidates a repeated lookup from five methods across two controllers. This is consistent with the larger reduction but does not prove a shared mechanism, because the applications also differ in build system, persistence layer, and workload, and because RealWorld's per-request repository-access count was not independently verified to have decreased. RQ1 is therefore answered at the level of the evidence: the evaluated code-level interventions were associated with application-specific reductions in measured energy and runtime, with an explicit mechanism established for PetClinic and a measured, mechanism-agnostic difference established for RealWorld.

The energy and duration are not independent of each other. Energy is power integrated over time, and both interventions shorten the runtime; therefore, average power (energy divided by duration) separates a shorter run from a lower-power run. For PetClinic, average power was essentially unchanged (29.5 vs 29.4 W, -0.5%, $p = 0.69$), so its 8.1% energy reduction was almost entirely its 7.4% runtime reduction. For RealWorld, average power also fell (30.0 vs 28.6 W, -4.7%, large but not significant at $n = 10$, $p = 0.09$), so its 21.1% reduction is a 17.3% runtime reduction plus a smaller, not-yet-significant power reduction. In both cases, the post-change workload consumed less measured energy primarily because it was completed faster, not because it drew less power. This is a result of the study, not a caveat, and it holds independently of how the RealWorld mechanism question above is eventually resolved.

5.5 CI/CD integration

RQ2 asks what measurement stability is observed when build-level and endpoint-level energy feedback are integrated into a CI/CD pipeline running on a shared infrastructure. VerdeFlow performs this integration on an unmodified GitHub Actions infrastructure, measuring energy at both granularities on every push and pull request and returning a pull-request comment that classifies each build and endpoint as PASS, WARN, or FAIL, with no self-hosted runner, hardware probe, or privileged interface required. Integration itself is not the difficult part; producing a stable signal on a shared infrastructure is.

The observed signal was sufficiently stable for the comparisons performed in this study. Across the 14 endpoints, the baseline standard deviation was 0.07% to 1.6% of the endpoint mean; for the busiest endpoint, list_owners, the coefficient of variation was 0.5% (2.694×10^{-8} kWh/req, SD 1.29×10^{-10}). This holds despite short measurement windows: 200 requests of roughly 1 ms give a window of a few hundred milliseconds against CodeCarbon's 1-second sampling. Per-endpoint variation remained low, with baseline variation ranging from 0.07% to 1.6% across the evaluated endpoints, enabling threshold-based detection on hardware the developer does not control.

The two granularities are not interchangeable; therefore, the pipeline reports both: build-level covers the aggregate test suite and startup, per-endpoint attributes energy to a single request path, and the two detect different regressions, as Section 5.6 shows.

Table 5: Detection outcomes for the two injected regressions. Deltas are the percentage increase over the runner-calibrated baseline.

Measure	Experiment 1 (N+1 query)	Experiment 2 (CPU load)
Regression type	Query pattern	Injected computation
Scope	All 14 endpoints	3 endpoints
Tests (build)	+689.7% FAIL	+23.2% PASS
Startup (build)	+607.3% FAIL	+36.7% PASS
Endpoint deltas	+152% to +175%	+292% to +954% (modified) -17% to +30% (unmodified)
Detected at	Build and endpoint	Endpoint only

5.6 Regression detection

RQ3 asks how runner-calibrated measurements detect and localize injected energy regressions while distinguishing them from the observed shared-runner variability. Two faults were injected into separate branches, each opened as a pull request and classified against the runner-calibrated baseline (Table 5).

Experiment 1: N+1 query. The owner listing was changed from a single join-fetch to a lazy loading pattern issuing one query per owner. The build-level energy and all 14 endpoints increased sharply and were classified as FAIL at both levels (Table 5). The uniform rise across the 13 untouched endpoints shows that an $N + 1$ query propagates system-wide through shared JVM effects rather than remaining local.

Experiment 2: graduated CPU load. Fixed-size `Math.sqrt` loops, guarded against JIT elimination, were injected into three service methods (2,000,000, 800,000, and 600,000 iterations). The build level missed the regression, and both build metrics stayed within PASS, whereas the per-endpoint measurement caught it: the three modified endpoints failed, and the 11 unmodified endpoints passed (Table 5). The three endpoint deltas order with the injected work but scale sublinearly, consistent with a fixed per-request cost diluting the added computation.

Together, the two experiments separate the signal from the noise and show why both granularities are measured. The runner noise floor reached +37% against the smallest detected regression of +152%, a threefold gap. The systemic $N + 1$ regression is visible at the build level, while the localized CPU regression is not; a build-only pipeline would have passed Experiment 2 despite three endpoints drawing three to ten times their baseline energy. Granularity, therefore, affects detection: which regressions are caught depends on whether energy is read at the build level, endpoint level, or both.

A hardware change is distinguished from a regression based on its shape. When the runner class rotated, every endpoint shifted by roughly +230%, within ±3% of each other, whereas a code regression was localized or spread unevenly, not a flat shift across all 14. Unconditional recalibration on the master resets the reference, after which the unchanged code returns to within 1-2% of the baseline. Detection remains effective against both injected regressions and runner noise in the two tested scenarios;

however, it depends on the baseline remaining representative of the hardware on which the code runs.

6 Discussion

6.1 What the validation establishes

The three technical parts of the framework operate together on real software, in the sense that code-level changes are empirically measured, the resulting energy information is surfaced as developer feedback inside a CI/CD workflow, and potential energy regressions are detected within that same workflow. The evaluated code-level interventions were associated with lower measured energy in two Java services, with an explicit mechanism established for PetClinic and a measured, mechanism-agnostic difference established for RealWorld (RQ1); the same measurement was integrated into a standard GitHub Actions pipeline at the build and endpoint levels, with per-endpoint variation remaining low across evaluated endpoints (0.07%-1.6%) on the tested runners (RQ2); and the pipeline detected the two injected regressions, isolated the localized one to the modified endpoints, and separated both from runner noise by at least threefold (RQ3). Run together on real applications and a live pipeline, this is an initial empirical operational validation of the loop described in Section 3; it is not a claim that the loop holds beyond the cases evaluated here.

The two code-level cases also show why the measurement step is necessary in the first place, not merely useful. PetClinic and RealWorld share the same surface pattern consolidating a duplicated repository call yet produce different effect sizes (8.1% versus 21.1%), and only one of the two supports a stated causal mechanism. The different magnitudes, together with the inability to infer RealWorld's energy behavior from source-level reasoning alone, reinforce the need for measurement rather than assuming that a refactoring is energy efficient because of its shape.

The two code-level cases illustrate why green coding should be measurement-guided, rather than inferred solely from the form of a refactoring. PetClinic provides a clear mechanism through the elimination of an additional repository lookup, whereas the RealWorld intervention produced a substantial measured before-and-after difference without establishing an equivalent reduction in repository operations. Accordingly, the latter is read as evidence of an observed energy and runtime difference following the investigated refactoring, not as proof of a specific database access mechanism. This distinction is part of what the study establishes: it demonstrates the role of empirical measurement in evaluating code-level changes rather than assuming it.

The claims match the evidence, which differs across the three sections of the paper. The code-level intervention track is a controlled comparison of two applications: the direction is consistent, and the effect is large, but the magnitude is application-specific (8.1% for PetClinic, 21.1% for RealWorld), and only the PetClinic case supports a stated mechanism. The integration and detection results are a demonstration of one pipeline and one application, showing that the mechanism works and returns actionable output, not that thresholds transfer to other runners or codebases.

Our prior empirical studies of practitioners, a survey and an interview-based study (Oyedeji et al. 2025; Moktan et al. 2025), together with our earlier framework proposal (Khan et al. 2025), identify workflow integration as an important adoption requirement. The present study does not evaluate adoption. Instead, it evaluates whether a technical mechanism for providing workflow-integrated feedback can be operationalized. Baseline ownership, recalibration cadence, and build-blocking rules are technical and operational constraints that this study can assess directly. Whether addressing these constraints is sufficient to support adoption is a separate question, examined through capability, constraints, and context in the remainder of this section.

6.2 Capability and constraints

Baseline ownership and setup cost. VerdeFlow can be added to an existing GitHub Actions setup without new infrastructure: it consists of two Python scripts, a workflow file, and a JSON baseline running on the runners a project already uses. The effort is not in running it but in owning the baseline, which, as a committed artifact, has the same owner as any repository file, whoever reviews CI changes. The technical constraint is that a committed baseline is only meaningful for the hardware that produced it.

Recalibration cadence. The measurements set the cadence for the experiment. A runner-class rotation shifted every endpoint by roughly +230% at once, and the RQ1 baseline drifted within a single session ($\rho = 0.87$, Section 5.2): energy is stationary neither across the hardware nor over time. Therefore, the baseline must be recalibrated on the hardware class with which it is compared, and often enough to track the rotation. VerdeFlow recalibrates after each merge with the master, aligning the calibration with code integration. A uniform shift across all endpoints was treated as a recalibration signal rather than an energy regression.

Block or warn. A localized regression is caught only per endpoint and passes at the build level; blocking on build-level energy alone would pass a change that tripled the energy of three endpoints. Therefore, the useful gate is a per-endpoint FAIL. However, runner noise reached +37%, and a stale baseline could produce a false FAIL, which argues against blocking a single run. The rule that fits the evidence warns on a single run, blocking only on a FAIL that survives recalibration is what VerdeFlow's split between pull-request gating and master recalibration implements.

The experiments identified baseline maintenance and infrastructure variability as technical-operational constraints that must be considered for practical adoption. Their relative importance to practitioners requires separate user or field evaluations.

6.3 Context

The pipeline fits the existing practice rather than replacing it. Energy measurement runs at the same points as performance testing, on the same CI runs, but measures a different quantity: a change can pass a latency check and still raise energy (Section 5.6, where a build passed on time, while three endpoints tripled their energy). Energy testing is a second gate beside performance testing, not one above it.

The per-endpoint delta is a useful unit for sustainability requirements. An organization with an energy or carbon target must attribute consumption to a code path to act on it, which build-level totals do not support. VerdeFlow reports energy per endpoint against a baseline, turning a sustainability goal into a per-change check that a developer sees on a pull request.

The fit with existing DevOps practices comes from the warn-then-block rule described in Section 6.2. Teams already use CI gates that block failing tests and warn on coverage drops. An energy check follows the same logic: it warns on a single run and blocks only when the regression persists after recalibration.

Whether this reduces the effort of adoption in practice, as opposed to only fitting an existing pattern, is a question that this study does not answer.

6.4 Relation to prior work

The findings return to the four threads in Section 2. Each supplied part of a working energy-feedback loop; this study adds an initial demonstration that the parts operate together.

Measurement work established accuracy against hardware ground truth in standalone settings (Rajput and Sharma 2026; Weigell et al. 2025). Pipeline integration requires a property that work does not test: run-to-run stability. The low observed per-endpoint variation in the evaluated shared-runner baseline suggests that CodeCarbon estimates were sufficiently stable for the relative comparisons performed in this pipeline, a property the standalone accuracy studies above do not test.

CI energy work measures the pipeline's own footprint or shifts its carbon timing, not the energy of the code under change (Perez et al. 2024; Classen et al. 2024). VerdeFlow attributes a regression to a single endpoint (Section 5.6). The closest systems, PPTAM-η (Aneggi et al. 2026) and our earlier proposed framework (Khan et al. 2025), showed that in-pipeline measurement is feasible; the results here show that it detects and localizes the two injected regressions evaluated in this study. Testing work established that energy regressions can be detected through developers' tests, with statistical filtering to separate real differences from noise, although on controlled hardware and with CPU cycles standing in for energy where direct readings proved unstable (Danglot et al. 2024). The results show that a direct energy signal is usable on shared runners without a proxy in the evaluated setup: per-endpoint variation remained low across evaluated endpoints (0.07%-1.6%), and thresholds set above the observed noise band separate the two injected regressions from runner variance by at least threefold. The two experiments also turn detection into an operational gate that fails a build at the changed endpoint and add a distinction that the thread did not draw: a systemic $N{+}1$ regression is visible at the build level, while a localized one is caught only per endpoint; therefore, energy-aware testing needs both granularities.

Adoption work found that developers lack energy feedback in their workflow, not knowledge of what to change (Moktan et al. 2025; Oyedeji et al. 2025). This study does not test that finding directly; it evaluates one technical mechanism capable of supplying such feedback and identifies the conditions under which the signal it produces is a trustworthy baseline for tracking the correct hardware, recalibration cadence, and

a warn/block policy. Whether supplying feedback is sufficient for adoption remains unclear.

Therefore, the contribution is integration, not a new measurement method, scheduler, or test technique: initial evidence that three technical practices can run as one loop on real software at the scale evaluated here.

7 Threats to validity

7.1 Construct validity

Energy is estimated but not measured. CodeCarbon derives energy from CPU TDP models and RAPL interfaces rather than a physical power meter, and both experiments are measured at the whole-unit granularity: machine-level for RQ1 and runner-level for RQ2 and RQ3. Therefore, the total energy includes a background draw that no code change can remove. The energy and wall-clock duration are coupled; CodeCarbon integrates power over the run, so a shorter run registers lower energy at equal power. We report the average power in Section 5.4 to separate the runtime effect from the power effect. In both applications, the post-change workload consumed less measured energy primarily because it was completed faster, not because it drew statistically less power; this is reported as a finding, not only as a limitation. Because the same estimator runs under both conditions on the same hardware, the relative deltas remain even if the absolute values carry a systematic bias. This is why the CPU energy result is read as supporting evidence rather than proof of mechanism: for PetClinic, the CPU result did not reach significance and was not treated as a component-level claim; for RealWorld, CPU energy did reach significance, but because the measurement was machine-level and the per-request repository-access count was not independently verified, this was read as part of a coherent before-and-after pattern, not as confirmation that fewer database operations were executed. For RQ3, per-endpoint windows are short against the 1-second sampling interval, so each figure rests on few power samples. The sub-1% coefficient of variation indicates that the estimate is stable enough for a relative comparison, not that it is a ground-truth value.

7.2 Internal validity

The RQ1 runs were blocked by condition, which confounded the code change with session time. This risk materialized: the PetClinic baseline block drifted upward (Spearman $\rho = 0.87$, Section 5.2), inflating its mean reduction. The effect survives a distribution-free test and a median comparison; therefore, it is real, but its magnitude is reported as a range rather than as a point estimate. RealWorld showed no such drifts. In both applications, the code change was the only manipulated variable, and the post-change versions passed the unchanged test suites, ruling out the altered test behavior as the cause. For RealWorld, the per-request repository-access count was not independently instrumented, which is why the causal claim for that case is limited to a measured difference rather than a confirmed mechanism. For RQ3, the injected

computation used a dead-code guard to prevent JIT elimination, and the shared-runner co-tenancy noise was handled using a mean-and-standard-deviation baseline with thresholds set above the observed noise band.

7.3 External validity

The evidence comprises two applications for RQ1 and one application on one pipeline for RQ2 and RQ3. The reduction magnitudes are application-specific, ranging from approximately 5-8% to 21%, and cannot be generalized to a fixed figure. RQ1 was run on a single machine, and RQ2 and RQ3 on a single CI runner class; detection thresholds were calibrated to `ubuntu-latest` and required recalibration on other runners. These are case studies, not samples from which population estimates are derived. The discussion of adoption conditions in Section 6 draws on prior adoption studies (Moktan et al. 2025) and the operational constraints observed here, not on data collected from developers in this study, which bounds the extent of the practical claims made.

7.4 Conclusion validity

The sample sizes were small: ten runs per condition for RQ1 and three baseline runs for RQ2 and RQ3. Ten runs were adequate for the large within-application effects observed but limited the precision of the estimate, visible in PetClinic's borderline result ($p = 0.0496$). Effect sizes and the Mann-Whitney check are reported, so significance is not read in isolation. RQ3 detection in Phase 1 rests on a single measurement per mode against a multi-run baseline; the threefold margin between the noise floor and the weakest detected regression justifies acting on one run. Runner-hardware rotation can still produce a false FAIL against a stale baseline, which can be mitigated by unconditional recalibration without removal.

7.5 Limitations

Two limitations bound the study beyond the validity threats mentioned above. First, it did not include expert or practitioner evaluation. The discussion of adoption conditions in Section 6 draws on the measurements reported here and on prior adoption findings, but does not measure developer effort, trust in the signal, or workflow fit directly; the practical statements made there are analytical, not observational. Second, RQ1 measured the test-suite workload rather than production traffic, and VerdeFlow's sustained-load mode was implemented but not evaluated; the results describe energy under test execution and controlled endpoint profiling, not the sustained production load.

8 Conclusion and future work

This study provides an initial empirical operational validation of a previously proposed framework by connecting code-level intervention, CI/CD energy measurement, and energy-aware testing by running its three parts together on real software and a live pipeline. The two evaluated code-level changes were associated with lower measured energy in two Java services, with application-specific magnitudes: an explicit

mechanism for PetClinic and a demonstrated before-and-after difference without a confirmed causal mechanism for RealWorld (RQ1). The measurement was integrated into a standard GitHub Actions pipeline at the build and endpoint levels, with per-endpoint variation remaining low (0.07%-1.6% across evaluated endpoints) on the tested shared runners (RQ2). The pipeline detected the two injected regressions, isolated the localized one to the modified endpoints, and separated both from runner noise by at least threefold, with build-level and per-endpoint measurements capturing different regression types (RQ3).

This is not a new measurement method or test technique but initial bounded evidence that three technical parts of a previously proposed framework can operate as one loop in real applications at the scale evaluated here. The study is bounded by two code-level cases (one with an established mechanism and one without), one pipeline, and the absence of developer-facing evaluation. This shows that energy can be measured, surfaced as feedback on a code change, and checked within a CI/CD workflow of the kind developers already use. The experiments identified baseline maintenance and infrastructure variability as technical and operational constraints that must be considered for practical adoption. Their relative importance to practitioners requires separate user or field evaluations, which this study does not provide.

Future research should extend this work in several directions. The current study evaluates two Spring Boot applications and a limited set of code-level interventions. Replication across different languages, frameworks, architectures, databases, and kinds of redundant or wasteful code would determine whether the observed energy reductions and regression-detection behaviour extend beyond the studied setting. Direct measurement of database calls would also help verify the causal mechanism behind the 21.1% reduction reported in the RealWorld experiment. Second, energy measurement requires stronger infrastructure-aware calibration. The experiments showed substantial variability on shared CI runners, including approximately 37% measurement noise and a much larger shift following a runner-class change. Future work should investigate adaptive baselines, hardware-aware calibration, and statistical methods for distinguishing infrastructure variability from genuine software regressions. Comparisons with hardware-based measurements would further establish the accuracy of software energy estimators. Third, future studies should evaluate the framework under realistic and sustained workloads. The current experiments primarily use controlled test and API workloads, while the implemented sustained-load capability remains unevaluated. Production-like traffic, varying concurrency, and long-duration workloads could determine whether measured energy reductions persist under realistic operating conditions and whether they translate into meaningful operational and carbon savings. Finally, future work should investigate developer adoption and threshold sensitivity. The current 50% WARN threshold provides robustness against measurement noise but would not detect smaller improvements such as the 8.1% reduction observed in PetClinic. Future research should therefore evaluate adaptive or infrastructure-specific thresholds and potentially separate policies for detecting regressions and improvements. Developer studies should additionally examine whether engineers understand, trust, and act on these energy signals in real development workflows. Together, these

directions would move the framework from experimental validation toward broader generalization, reliable measurement, and practical adoption in software engineering.

Declarations

Funding

This work was supported by the Finnish Software Engineering Doctoral Research Network (FAST), funded by the Ministry of Education and Culture, Finland.

Competing interests

The authors declare no conflicts of interest.

Data availability

The data supporting the findings of this study, including the application forks, CodeCarbon output, analysis workbook, and VerdeFlow pipeline, are available in a public repository at (Khan 2026).

Author contributions

M.A.K. designed the study, implemented the code-level interventions and experiments, analyzed the data, and wrote the manuscript. M.J. developed the VerdeFlow pipeline. S.O. and J.P. supervised the study design and implementation, reviewed and edited the manuscript. All authors have read and approved the final manuscript.

References

Andringa L, Setz B, Andrikopoulos V (2025) Understanding the Energy Consumption of Cloud-native Software Systems. Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, https://doi.org/10.1145/3676151.3719371

Aneggi A, Li X, Janes A (2026) Pptamη: Energy aware ci/cd pipeline for container based applications. In: 2026 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C), IEEE, pp 218–221

Bouza L, Bugeau A, Lannelongue L (2023) How to estimate carbon footprint when training deep learning models? a guide and review. Environmental Research Communications 5(11):115014

Chen TH, Shang W, Jiang ZM, et al (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proc. 36th Int. Conf. on Software Engineering (ICSE), pp 1001–1012, https://doi.org/10.1145/2568225.2568259

Classen M, Weisenburger P, Salvaneschi G (2024) Carbon-awareness in ci/cd. In: Software Architecture, pp 291–308, https://doi.org/10.1007/978-981-97-0989-2_17

Danglot B, Falleri JR, Rouvoy R (2024) Can we spot energy regressions using developers tests? Empirical Software Engineering 29(5):121

Dhawan A, Kumar S, Patel R (2026) Carbon-aware ai control plane for devops automation: a reference architecture and next-generation sustainability framework. IEEE Access https://doi.org/10.1109/ACCESS.2026.3656467

dos Santos LBR, Érica Ferreira de Souza, Endo AT, et al (2025) Performance regression testing initiatives: a systematic mapping. Information and Software Technology 179:107641. https://doi.org/https://doi.org/10.1016/j.infsof.2024.107641, URL https://www.sciencedirect.com/science/article/pii/S0950584924002465

Drangmeister J, Kern E, Guldner A, et al (2013) Greening software with continuous energy efficiency measurement. In: INFORMATIK 2013, GI-Edition Lecture Notes in Informatics

Fahad M, Shahid A, Manumachu RR, et al (2019) A comparative study of methods for measurement of energy of computing. Energies 12(11):2204

Georgiou S, Rizou S, Spinellis D (2019) Software development lifecycle for energy efficiency: Techniques and tools. ACM Computing Surveys 52(4):1–33. https://doi.org/10.1145/3337773

Gordillo A, Calero C, Moraga MÁ, et al (2024) Programming languages ranking based on energy measurements. Software Quality Journal 32(4):1539–1580

Guldner A, Bender R, Calero C, et al (2024) Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—green software measurement model (gsmm). Future Generation Computer Systems 155:402–418

Jagroep E, Broekman J, Werf JVD, et al (2017) Awakening awareness on energy consumption in software engineering. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Society Track (ICSE-SEIS) pp 76–85. https://doi.org/10.1109/icse-seis.2017.10

Joof MB, Khan A, Oyedeji S, et al (2026) Integrating api energy profiling into developer workflows: The verdeflow prototype. In: Proceedings of the IEEE/ACM 10th International Workshop on Green and Sustainable Software. Association for Computing Machinery, New York, NY, USA, GREENS '26, p 144–152, https://doi.org/10.1145/3786148.3788637, URL https://doi.org/10.1145/3786148.3788637

Jünger DM, Westing M, Freitag CP, et al (2024) Potentials of green coding. arXiv preprint arXiv:240218227

Katal A, Dahiya S, Choudhury T (2022) Energy efficiency in cloud computing data centers: a survey on software technologies. Cluster Computing 26:1845 – 1875. https:

//doi.org/10.1007/s10586-022-03713-0

Kaur K, Garg S, Kaddoum G, et al (2020) A big data-enabled consolidated framework for energy efficient software defined data centers in iot setups. IEEE Transactions on Industrial Informatics 16:2687–2697. https://doi.org/10.1109/tii.2019.2939573

Khan MA (2026) Green-Coding-API-optimization-CI-CD-Experiment. GitHub repository, URL https://github.com/Asifkhan9697/Green-Coding-API-optimization-CI-CD-Experiment, accessed: 2026-07-14

Khan MA, Oyedeji S, Porras J (2025) A comprehensive framework for optimizing api calls, ci/cd pipelines, and energy testing. In: International Conference on Software Engineering of Emerging Technology, Springer, pp 593–605

Kifetew FM, Tiella R, Tonella P (2025) On the energy consumption of test generation. arXiv preprint arXiv:250109657 https://doi.org/10.48550/arXiv.2501.09657

Koçak SA, Alptekin GI, Bener AB (2015) Integrating environmental sustainability in software product quality. In: Proc. 4th Int. Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS), pp 17–24

Kruglov A, Succi G, Vasquez X (2021) Incorporating energy efficiency measurement into ci-cd pipeline. Proceedings of the 2021 European Symposium on Software Engineering https://doi.org/10.1145/3501774.3501777

Laskar R (2025) Carbon-aware continuous integration: reducing emissions without sacrificing performance. European Modern Studies Journal 9(5). https://doi.org/10.59573/emsj.9(5).2025.10

Manotas I, Bird C, Zhang R, et al (2016) An empirical study of practitioners' perspectives on green software engineering. In: Proc. 38th Int. Conf. on Software Engineering (ICSE), pp 237–248, https://doi.org/10.1145/2884781.2884810

Moktan I, Khan MA, Oyedeji S, et al (2025) Practical adoption of green coding: a comparative study across organizations in finland and globally. In: International Conference on Software Business, Springer, pp 361–378

Noureddine A, Rouvoy R, Seinturier L (2013) A review of energy measurement approaches. ACM SIGOPS Operating Systems Review 47(3):42–49

Noureddine A, Bourdon A, Rouvoy R, et al (2014) Unit testing of energy consumption of software libraries. In: Proc. 29th Annual ACM Symp. on Applied Computing (SAC), https://doi.org/10.1145/2554850.2554932

Noureddine A, Islam S, Bashroush R (2016) Jolinar: analysing the energy footprint of software applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp 445–448

Ournani Z, Rouvoy R, Rust P, et al (2020) On reducing the energy consumption of software: From hurdles to requirements. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–12

Oyedeji S, Khan MA, Puhtila P, et al (2025) Green coding: state of practice. In: 2025 11th International Conference on ICT for Sustainability (ICT4S), IEEE, pp 91–99

Pang C, Hindle A, Adams B, et al (2016) What do programmers know about software energy consumption? IEEE Software 33(3):83–89. https://doi.org/10.1109/MS.2015.83

Pereira RAA (2018) Energyware engineering: techniques and tools for green software development. PhD thesis, Universidade do Minho (Portugal)

Perez Q, Lefeuvre R, Degueule T, et al (2024) Software frugality in an accelerating world: the case of continuous integration. ArXiv abs/2410.15816. https://doi.org/10.48550/arxiv.2410.15816

Poy O, Ángeles Moraga M, García F, et al (2025) Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study. Journal of Systems and Software 222:112303. https://doi.org/https://doi.org/10.1016/j.jss.2024.112303, URL https://www.sciencedirect.com/science/article/pii/S0164121224003479

Rajput S, Sharma T (2026) Codegreen: Towards improving precision and portability in software energy measurement. arXiv preprint arXiv:260317924

Sahin C, Pollock L, Clause J (2014) How do code refactorings affect energy usage? In: Proc. 8th ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM), pp 1–10, https://doi.org/10.1145/2652524.2652538

Sallou J, Cruz L, Durieux T (2023) Energibridge: empowering software sustainability through cross-platform energy measurement. arXiv preprint arXiv:231213897

Schubert S, Kostic D, Zwaenepoel W, et al (2012) Profiling software for energy consumption. In: Proceedings of the 2012 IEEE International Conference on Green Computing and Communications. IEEE Computer Society, USA, GREENCOM '12, p 515–522, https://doi.org/10.1109/GreenCom.2012.86, URL https://doi.org/10.1109/GreenCom.2012.86

Weigell B, Hornung S, Bauer B (2025) Metrion: A framework for accurate software energy measurement. arXiv preprint arXiv:251206806

Wilke C, Götz S, Richly S, et al (2013) JouleUnit: a generic framework for software energy profiling and testing. In: Proc. Workshop on Green in/by Software Engineering (GIBSE), https://doi.org/10.1145/2451605.2451610

Zaidman A (2024) An inconvenient truth in software engineering? the environmental impact of testing open source java projects. 2024 IEEE/ACM International Conference on Automation of Software Test (AST) pp 214–218. https://doi.org/10.1145/3644032.3644461