The Energy Impact of Batch Testing in Continuous Integration

An Empirical Study of Static and Dynamic Batching Strategies

Máté Oszkó[1]

Supervisor(s): Carolin Brandt[1]**, Xutong Liu**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

This paper was submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2026

Name of the student: Máté Oszkó
Final project course: CSE3000 Research Project
Thesis committee: Carolin Brandt, Xutong Liu, Benedikt Ahrens

Abstract

Continuous integration pipelines execute automated tests on every commit, consuming substantial energy. Batch testing, which groups multiple commits into a single test run, has been shown to reduce test executions in simulation studies, but no prior work has measured whether these reductions translate into actual energy savings.

This study measures CPU package energy consumption of CI builds under a baseline run-all approach and two batching strategies (BatchStop4 and linear-4 lwd) across eight open-source Java projects. BatchStop4 achieves energy savings between 57% and 88% (mean: 80.3%), while linear-4 lwd achieves savings between 59% and 94% (mean: 85.2%). Energy savings correlate almost perfectly with time savings (r >0.99), and baseline failure rate strongly predicts achievable savings, while CPU utilisation shows no significant relationship. These findings provide empirical evidence that batch testing effectively reduces the environmental footprint of continuous integration.

1 Introduction

Continuous Integration (CI) is now a standard part of modern software development workflows: every commit triggers automated builds and tests that provide rapid feedback to developers. While this practice improves software quality and developer productivity, it also results in large volumes of test executions that consume non trivial computational resources and energy. Recent work has started to quantify the environmental impact of testing, showing that automated tests can contribute a significant share of a project's overall energy footprint [1]. At the same time, CI providers and organizations are under pressure to reduce both operational costs and environmental impact. Data centers, which underpin CI/CD infrastructure, account for approximately 1% of global electricity use [2], while the ICT sector overall is responsible for 2% of global carbon emissions [3]. This has driven recent research into carbon-aware CI/CD services that align workflow execution with low-carbon energy availability [4]. This creates a growing need for CI strategies that are not only fast and reliable, but also energy efficient.

One promising direction to reduce CI costs is batch testing, where multiple commits are grouped into a single build and test run. Instead of executing the full test suite on every commit, the CI system accumulates commits into batches and only invokes the test suite once per batch. Prior work has demonstrated that batching strategies can reduce the number of test executions by approximately 50% on average across projects [5], with simple fixed-batch approaches achieving 48% reductions [5] and dynamic batching algorithms offering comparable savings while adapting to project characteristics [6]. These studies evaluate batch testing algorithms analytically on historical CI data, simulating test execution savings without measuring actual resource or energy consumption [5][6]. Conversely, research on the energy impact of software testing measures real CI executions but focuses exclusively on standard run-all configurations [1]. This gap means that the energy implications of batch testing strategies remain unquantified, despite their demonstrated potential to reduce test executions.

Bridging this gap is important because simulated reductions in test executions or build time do not automatically translate into proportional energy savings when the same strategies are deployed on real hardware. The energy usage of a build is influenced by hardware utilisation, idle and waiting periods, and the mix of CPU, memory, disk and network activity. A shorter build that keeps the CPU near peak utilisation can consume similar or more energy than a longer build that spends much of its time waiting for I/O.

This research addresses that gap by measuring the energy consumption of CI builds under different batch testing strategies and by relating observed energy profiles to project and test suite characteristics. The main research question of this work is:

How do different batch testing approaches affect energy profiles?

To answer this question, the study focuses on the following subquestions:

- To what extent do batch testing strategies reduce total energy per build compared to baseline?
- How do test suite CPU utilisation and baseline failure rate influence the energy savings of batch testing?

The project tests slightly modified versions of BatchStop4 [5] and linear-4 lwd [6], one static and one dynamic batch testing algorithm. Energy usage is recorded with EnergiBridge [7], an open source tool that records power usage during each CI build and produces per build energy measurements. For a set of open source projects, the study replays sequences of commits and runs a baseline configuration that always executes the full test suite on every commit, alongside the two algorithms. For each build, the study logs energy, build duration, hardware resource usage and test outcomes.

2 Background and Related Work

This section reviews prior work on batch testing in continuous integration and on measuring energy consumption in software testing, establishing the research gap that motivates our empirical study

2.1 Continuous Integration and Batch Testing

Continuous Integration (CI) is a software development practice where developers frequently integrate code changes into a shared repository, with each integration verified by automated builds and tests [5]. In a standard run-all CI workflow, every commit triggers a complete build and execution of the entire test suite. This provides rapid feedback to developers, helping identify faults quickly, but leads to the maximum number of test executions. For projects with frequent commits or expensive test infrastructure, this approach becomes costly in terms of both computational resources and time.

Batch testing addresses this cost by grouping multiple commits together and testing them as a single unit. If a batch passes, all commits in the batch are considered passing with only one test execution, saving substantial resources. For example, if eight commits are batched together and the batch passes, seven test executions are saved. However, when a batch fails, it is not known which commit caused the failure, requiring additional executions to find the source.

Two main approaches exist for finding failing commits in batches. Static batching uses a fixed batch size and applies bisection when a batch fails: the batch is split in half, each half is tested separately, and the process repeats until individual failing commits are identified [5]. The BatchStop4 algorithm improves on pure bisection by stopping the bisection process when the batch size reaches four and testing remaining commits individually, as bisection becomes inefficient at small batch sizes. Dynamic batching algorithms adjust batch sizes based on observed failure rates [6]. When failures are frequent, batch sizes shrink to provide faster feedback; when the codebase is stable, batch sizes grow to maximize execution savings. Prior work has shown that both approaches can reduce test executions by around 50% on average [5][6], making batch testing a promising strategy for reducing CI costs.

2.2 Energy Measurement in Software Testing

While batch testing demonstrably reduces the number of test executions, the relationship between execution count and actual energy consumption is not straightforward. Energy measurement in software systems has traditionally relied on execution time as a proxy, assuming that shorter execution times correspond to lower energy consumption. However, this assumption breaks down in practice because energy consumption depends on hardware utilization patterns, not just duration.

Modern processors support hardware-based energy measurement through Running Average Power Limit (RAPL) interfaces, which expose energy counters via model-specific registers (MSRs). Tools such as EnergiBridge leverage these interfaces to measure the actual energy consumed by CPU packages during program execution [1]. These measurements capture the cumulative energy draw over time, accounting for variations in CPU utilization, frequency scaling, and power states.

3 Research Design

This section describes the empirical study design used to measure the energy consumption of batch testing strategies in CI. Furthermore, I go over the study setup and research questions, subject systems, experimental environment, energy measurement instrumentation, analysis methods, and the approach used to adapt batching algorithms to public repository histories.

3.1 Study Design and Research Questions

This study addresses the following research questions:

RQ1: To what extent do batch testing strategies reduce total energy per build compared to baseline?

RQ2: How do test suite CPU utilisation and baseline failure rate influence the energy savings of batch testing?

To answer these questions, this study presents an empirical evaluation of batch testing strategies that executes real continuous integration (CI) builds on open source Java projects and measures their energy usage on a dedicated machine. For each project, the experiment replays a fixed sequence of 160 commits from the main development branch and runs the CI pipeline under three configurations: a baseline configuration that executes the full test suite on every commit, a static batching strategy (BatchStop4), and a dynamic batching strategy (linear-4 lwd). All builds are executed on bare metal on the same host in order to avoid interference from virtualisation or multi-tenant scheduling.

The unit of analysis in this study is a single commit and its associated CI build. For each project and each configuration, the experiment walks through the 160 selected commits in chronological order from oldest to newest and triggers one CI build per commit slice, or one build per batch in the case of batching strategies. The baseline configuration invokes the full test suite on every commit. The BatchStop4 configuration groups commits into fixed-size batches of 8 commits, and bisects on failure until the batch size reaches 4, then goes commit by commit. The linear-4 lwd batching configuration adjusts batch sizes based on the observed failure rate, while falling back to BatchStop4 behaviour when a batch fails.

For each build, the experiment records the build exit code, wall-clock duration, test outcome, hardware utilisation metrics, and CPU package energy usage measured by EnergiBridge. Energy measurements are collected at 100 millisecond intervals over the duration of each build, starting when the build command is invoked. The failure status of a commit or batch is determined from the exit code of the Maven or Gradle command. To reduce the impact of transient effects such as dependency downloads and just-in-time compilation, each project and configuration is preceded by a warm-up build that is not included in the analysis. Between successive builds, a fixed idle period of 10 seconds is enforced to allow the system to return to a stable state.

To account for measurement noise and potential flakiness, each project–configuration combination is executed three times. For every project, the failure rate observed under the baseline configuration is compared across repeats. If the baseline failure rates differ between repeats, the project is classified as flaky and excluded from further analysis, since in that case it is not possible to attribute differences between batching strategies to the batching behaviour rather than to unstable tests. This filtering excluded 10 projects from the original candidate set. All other runs are retained, and their per-build measurements form the dataset used in the subsequent analysis.

3.2 Energy measurement setup

Energy consumption is measured with EnergiBridge version 0.0.7 [7]. In this study, the measurement scope is limited to CPU package energy. Other components such as DRAM, GPU, IO and storage devices are not included.

For each build, a dedicated EnergiBridge process is started immediately before the build command is issued and stopped as soon as the build completes. EnergiBridge samples the MSR counters at a fixed interval of 100 ms and writes the raw samples for that build into a plain-text log file, with one log file per build. Each sample record contains a timestamp, the instantaneous power estimate for the CPU package, and basic system utilisation information.

These raw per-build logs are not used directly in the analysis. Instead, a Python script post-processes them into a structured execution log, `exec_log.csv`. For each build, the script integrates the power samples over time to obtain the total energy consumption in joules, computes the wall-clock time of the build from the first and last samples, and derives the average CPU utilisation during the build interval. The resulting execution-level dataset contains one row per build, with fields such as:

- `repo`: the project under test,
- `strategy`: the batch testing strategy or the baseline configuration,
- `tested_commit_sha`: the head commit of the (possibly batched) commit slice,
- `wall_time_seconds`: the end-to-end build duration in seconds,
- `cpu_package_energy_joules`: the total CPU package energy consumed by the build, expressed in joules,
- `avg_cpu_usage_percent`: the average CPU utilisation of the processor during the build, expressed as a percentage.

This `exec_log.csv` file serves as the main input for subsequent aggregation into a higher-level summary dataset, which is then used for the statistical analysis of energy usage and performance across projects and batching strategies.

3.3 Subject systems and CI configuration

The experiment is conducted on eight open source Java projects obtained from public GitHub repositories. Seven of these projects are Apache Commons libraries (`commons-lang`, `commons-compress`, `commons-io`, `commons-jxpath`, `commons-logging`, `commons-scxml`, and `commons-text`), built with Maven. The eighth project is Apache Lucene, built with Gradle. These projects are selected because they are actively maintained, provide non trivial test suites, and collectively cover a range of test failure rates and CPU utilisation profiles. For each project, the master branch is used and the experiment replays the 160 most recent commits at the time of the study. The same 160 commit slices are used for all configurations in order to enable a direct comparison between batch testing strategies and the baseline.

For the Maven based projects, each CI build is invoked with the command `mvn clean test`, which performs a clean build followed by the full test suite. For Lucene, builds are invoked with `./gradlew cleanTest test`, which clears previous test results and then runs all tests. In both cases, the default project specific build and test configurations are used without modification. Build tool level behaviours such as compiler and dependency caching are left at their defaults, but all build caches are cleared once at the start of each experiment to ensure that the first warm up build populates them consistently.

For each project and configuration, the experiment replays the selected commits in chronological order from oldest to newest. A lightweight Python driver script checks out the appropriate commit between builds and triggers the Maven or Gradle command while simultaneously starting the energy measurement tool. The exit code of the build command is used to classify each commit slice or batch as passing or failing. In addition to energy and duration, the experiment records average CPU utilisation per build, which is later used as an empirical proxy for how computationally heavy it is. The failure rate observed under the baseline configuration serves as the reference failure rate for each project and is used to characterise differences in how projects respond to batching.

3.4 Experimental environment

All experiments are executed on a dedicated desktop machine to avoid interference from other workloads and from virtualisation layers. The host is equipped with an AMD Ryzen 7 9800X3D processor with 8 physical cores and 16 hardware threads, all of which are available to the operating system and may be used by the builds. The system has 48 GB of

DDR5 memory (Patriot Viper Xtreme 5, 5600 MHz) and uses a Samsung 850 EVO 500 GB solid state drive as the operating system and build volume.

The operating system is Kubuntu 22.04.3 LTS with Linux kernel version 6.14.0-36-generic. The machine runs a fresh installation of the operating system, with only the build tools and dependencies required for the subject systems installed in addition to the default packages. The machine is not used for any other tasks while the experiments are running, and no user facing applications are launched during data collection. Standard power management and CPU frequency scaling settings are left at their distribution defaults, so that the results reflect a realistic developer workstation configuration rather than a laboratory tuned system.

All automatic software update services and scheduled maintenance tasks are disabled for the duration of the experiments. The energy measurement tool, build tools and measurement script are the only long lived user processes. A replication package containing the measurement script, configuration files and instructions for reproducing the environment have been published [8].

3.5 Adapting batching to dependent commits in GitHub histories

Commit batching is typically described under an independence assumption: commits are treated as distinct changes that can be accumulated into a batch before any test feedback is available. Public GitHub repositories violate this assumption in two ways. First, commits in the master branch are dependent in the sense that a newer commit already contains all earlier commits in the sequence. Second, the published commit history reflects development decisions that may have been influenced by CI feedback that would have been delayed under true batching.

To account for commit dependence, I adapt the batching algorithms as follows. A batch is evaluated by checking out and executing only the *head* (most recent) commit of that batch. If the head commit passes, all commits in the batch are treated as passing. If the head commit fails, the batch is split and the same head-only evaluation rule is applied recursively to the sub-batches, using the culprit-finding logic of the active strategy.

This adaptation enables real build execution on public histories, but it introduces two limitations. First, failing intermediate commits may be masked by subsequent fixing commits, which can lead to underestimation of failure rates compared to a setting with independent commits. Second, the feedback-delay aspect of batching cannot be realistically simulated, because the evaluated commit sequence already exists with knowledge of historical outcomes, whereas in an online batching setting the outcomes would only be known after the entire batch has executed.

3.6 Analysis methods

The execution log produced by the experiment contains one row per *executed* build, including baseline per-commit builds and the builds triggered by batching (batch-head executions, bisection sub-batches, and commit-by-commit fallback executions).

Aggregation. Analysis is performed at two levels:

- **Per-build level:** used to compute descriptive quantities such as build duration, CPU package energy, and average system CPU utilization for each executed build.
- **Per-repository level:** used to compare overall energy and time consumption between strategies. For each repository r, strategy s, and repeat k, total energy and total time are computed by summing over all executed builds in that run:

$$E_{r,s,k} = \sum_{i \in \mathcal{B}(r,s,k)} E_i, \qquad T_{r,s,k} = \sum_{i \in \mathcal{B}(r,s,k)} T_i. \quad (1)$$

In the equations above, $\mathcal{B}(r, s, k)$ is the set of executed builds observed for repository r under strategy s in repeat k, E_i is the CPU package energy of build i in joules, and T_i is the wall-clock duration of build i in seconds. Energy is also reported in watt-hours (Wh) using:

$$E_{\mathrm{Wh}} = \frac{E_{\mathrm{J}}}{3600}, \quad (2)$$

where E_{Wh} is energy in watt-hours and E_{J} is energy in joules.

Relative comparisons to baseline. For each repository, relative energy and time savings are computed by comparing a strategy against the baseline totals of the same repository and repeat. Energy savings S_E and time savings S_T (in percent) are defined as:

$$S_E(r,s,k) = 100 \cdot \left(1 - \frac{E_{r,s,k}}{E_{r,\mathrm{baseline},k}}\right), \quad (3)$$

$$S_T(r,s,k) = 100 \cdot \left(1 - \frac{T_{r,s,k}}{T_{r,\mathrm{baseline},k}}\right). \quad (4)$$

Here, $E_{r,\mathrm{baseline},k}$ and $T_{r,\mathrm{baseline},k}$ denote the baseline totals for repository r in repeat k.

Repeats and stability. Each repository–configuration pair is executed three times. Repositories whose baseline failure rate differs across repeats are excluded as flaky (as described earlier in Section 3.1).

Project characteristics used for interpretation. Two derived project characteristics are computed and later compared to relative energy savings:

- **Baseline average CPU utilization** $C_{\text{baseline}}(r)$, defined as the mean of the per-build average system CPU utilization over baseline builds of repository r (averaged across repeats).
- **Baseline failure rate** $F_{\text{baseline}}(r)$, defined as:

$$F_{\text{baseline}}(r) = \frac{N_{\text{failed}}(r)}{N_{\text{total}}(r)}, \quad (5)$$

where $N_{\text{failed}}(r)$ is the number of failing baseline builds and $N_{\text{total}}(r)$ is the total number of baseline builds for the 160-commit replay of repository r.

Visualization and descriptive association. Results are presented using (i) grouped bar charts of absolute and normalized total energy per repository, and (ii) scatter plots comparing time savings to energy savings and relating baseline characteristics (C_{baseline} and F_{baseline}) to relative energy savings.

4 Results

This section presents the empirical findings on energy consumption under batch testing strategies. The analysis is organized to directly address the two research questions posed in Section 3, reporting both relative savings percentages and absolute energy consumption in watt-hours (Wh) per repository.

4.1 RQ1: Energy Impact of Batch Testing

Research Question 1: *To what extent do batch testing strategies reduce total energy per build compared to baseline?*

To quantify energy savings, the notion of "savings" is defined explicitly. Energy savings S_E and time savings S_T (both in percent) are calculated as:

$$S_E = 100 \cdot \left(1 - \frac{E_{\text{strategy}}}{E_{\text{baseline}}}\right), \quad S_T = 100 \cdot \left(1 - \frac{T_{\text{strategy}}}{T_{\text{baseline}}}\right). \quad (6)$$

Here, E_{baseline} is the total baseline energy consumption (in Wh) for a repository, E_{strategy} is the total energy consumption (in Wh) under a batching strategy, T_{baseline} is the total baseline execution time, and T_{strategy} is the total execution time under a batching strategy.

Energy and time savings relative to baseline

Figure 1 compares time savings and energy savings across repositories for both strategies. The points lie close to a diagonal trend, indicating a strong positive relationship between time reduction and energy reduction: repositories that save more execution time also tend to save more energy. Pearson correlation analysis between time savings and energy savings yields $r = 0.9999$ with $p = 1.16 \times 10^{-12}$ for BatchStop4, and $r = 1.0000$ with $p = 2.77 \times 10^{-14}$ for linear-4 lwd, confirming a statistically significant linear relationship. This is consistent with the expectation that reducing CI wall-clock duration reduces total energy consumption when average power draw remains within a comparable range.

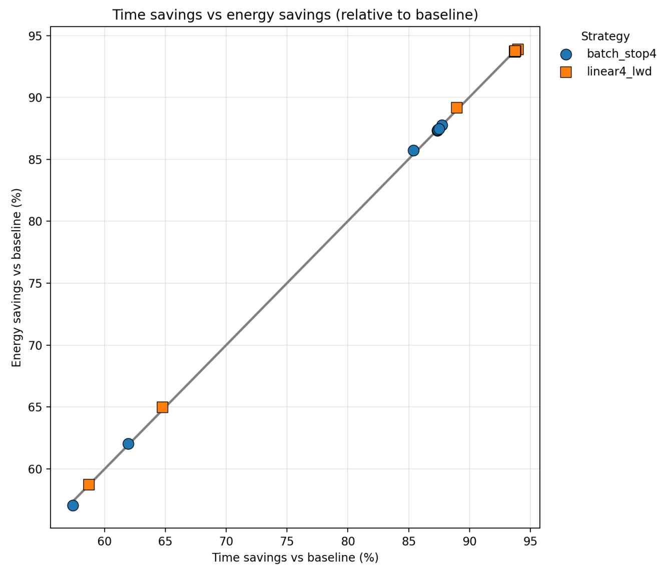

Figure 1: Time savings versus energy savings relative to baseline. Each point corresponds to one repository and one strategy. The near-diagonal clustering indicates that larger time reductions generally coincide with larger energy reductions.

Relative energy consumption by repository

Figure 2 shows the relative total energy consumption per repository, expressed as the ratio $\frac{E_{\text{strategy}}}{E_{\text{baseline}}}$ (baseline is normalised to 1.0). Both strategies reduce total energy well below baseline across all repositories. The magnitude of the reduction varies by project, ranging from approximately 57% to 94% energy savings, indicating that the effectiveness of batching is project dependent.

Absolute Energy Consumption

To complement normalized values, Table 1 reports absolute total energy in Wh, highlighting the spread in baseline energy consumption between repositories. Even when two repositories achieve similar relative savings, the absolute Wh reduction can differ substantially due to different baseline magnitudes. This distinction matters for practical impact, since improving a high-consumption repository yields larger absolute savings. Commons-lang is a notable outlier in resource consumption: its test suite runs approximately ten times longer than other projects in the sample, maintaining over 80% CPU utilisation throughout execution.

Table 1: Total energy consumption per repository in watt-hours (Wh) for each strategy, with relative savings compared to baseline.

Repository	**Baseline** (Wh)	**BatchStop4** (Wh)	**Linear-4 LWD** (Wh)	**BS4 Savings** (%)	**L4 Savings** (%)
commons-lang	3029.00	1150.46	1061.13	62.0	65.0
lucene	375.86	45.92	22.93	87.8	93.9
commons-io	233.87	29.28	14.62	87.5	93.7
commons-compress	105.76	13.39	6.60	87.3	93.8
commons-text	58.67	7.42	3.70	87.4	93.7
commons-scxml	29.89	12.84	12.34	57.0	58.7
commons-jxpath	16.82	2.12	1.06	87.4	93.7
commons-logging	9.10	1.30	0.99	85.7	89.2

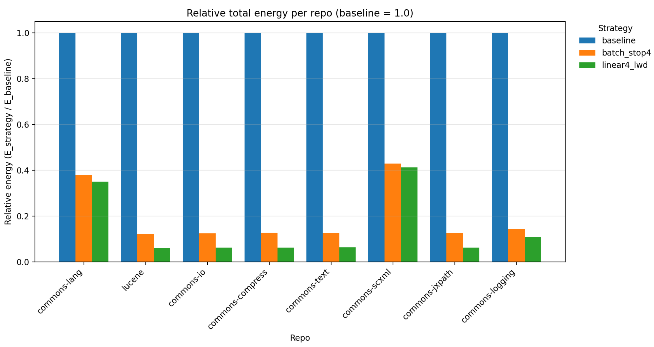

Figure 2: Relative total energy per repository, normalized by baseline ($E_{\text{baseline}} = 1.0$). Lower bars indicate better energy reduction. Both strategies reduce energy consumption, with `linear-4_lwd` typically slightly lower than `BatchStop4`.

Answer to RQ1

Both batch testing strategies substantially reduce total energy consumption compared to the baseline run-all approach. Across the eight evaluated repositories, BatchStop4 achieves energy savings ranging from 57.0% to 87.8% (mean: 80.3%, median: 87.3%), while linear-4 lwd achieves savings ranging from 58.7% to 93.9% (mean: 85.2%, median: 93.7%). The strong correlation between time and energy savings (BatchStop4: $r = 0.9999$, $p = 1.16 \times 10^{-12}$; linear-4 lwd: $r = 1.0000$, $p = 2.77 \times 10^{-14}$) indicates that batching strategies reduce energy consumption primarily by reducing total execution time. However, the magnitude of savings varies substantially by project, suggesting that project-specific characteristics influence the effectiveness of batch testing.

4.2 RQ2: Influence of Test Suite CPU utilisation

Research Question 2: *How do test suite CPU utilisation and baseline failure rate influence the energy savings of batch testing?*

To investigate whether CPU utilisation affects the energy efficiency of batch testing, baseline average CPU utilization is compared against achieved energy savings. Additionally, the influence of baseline failure rate is examined as an alternative project characteristic.

CPU Utilization and Energy Savings

Figure 3 relates baseline average CPU utilization to achieved energy savings for both strategies. Across repositories, baseline CPU utilization varies from approximately 10% to 82%, however similarly high relative energy savings are observed for both low and high utilization projects, as well as lower energy savings at various utilization rates.

Pearson correlation analysis confirms no significant relationship between baseline CPU utilization and energy savings: BatchStop4 has $r = -0.138$ with $p = 0.745$, and linear-4 lwd has $r = -0.137$ with $p = 0.746$. In other words, no correlation is evident between baseline CPU utilization and relative energy savings in this dataset.

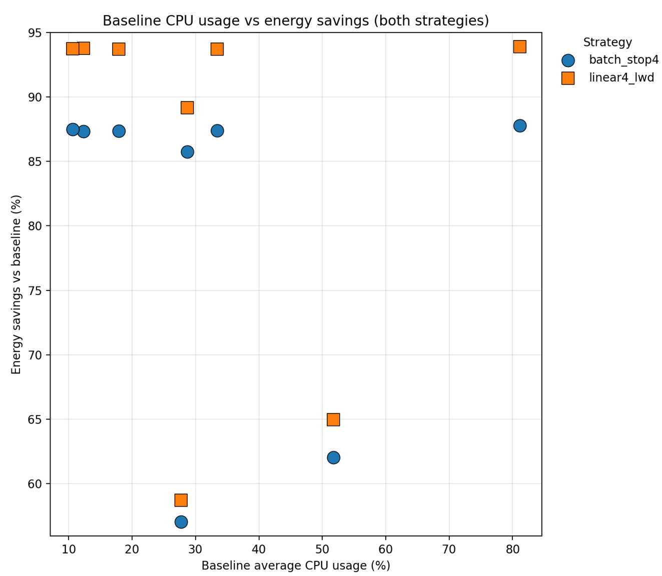

Figure 3: Baseline average CPU usage C_{baseline} (in percent) versus energy savings S_E (in percent) for `BatchStop4` and `linear-4_lwd`. No clear correlation is visible in this dataset.

In comparison, Figure 4 shows that baseline commit failure rate is strongly associated with relative energy savings:

repositories with higher baseline failure rates tend to exhibit lower energy savings. Pearson correlation coefficients are $r = -0.997$ with $p = 4.93 \times 10^{-8}$ for BatchStop4 and $r = -0.994$ with $p = 5.04 \times 10^{-7}$ for linear-4 lwd, indicating a statistically significant negative relationship. This pattern is clear and consistent across both strategies, suggesting that the frequency of failures is a stronger predictor of achievable energy reduction than CPU utilization.

Baseline average CPU usage is denoted by C_{baseline} (in percent), where C_{baseline} is the mean CPU utilization observed during baseline executions. Baseline failure rate is denoted by F_{baseline} and defined as:

$$F_{\text{baseline}} = \frac{N_{\text{failed}}}{N_{\text{total}}}, \quad (7)$$

where N_{failed} is the number of failing baseline commits and N_{total} is the total number of baseline commits evaluated for the repository.

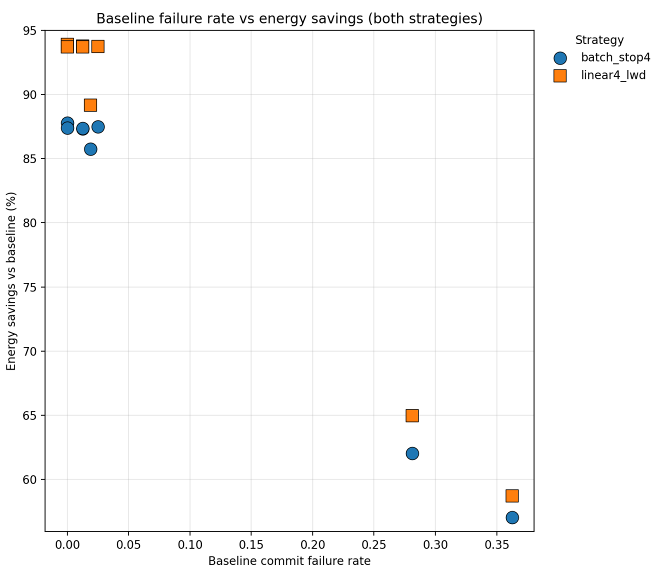

Figure 4: Baseline failure rate F_{baseline} versus energy savings S_E for `BatchStop4` and `linear-4_lwd`. Higher baseline failure rates are associated with lower savings.

Answer to RQ2

Test suite CPU utilization, does not significantly influence the energy savings achieved by batch testing strategies (BatchStop4: $r = -0.138$, $p = 0.745$; linear-4 lwd: $r = -0.137$, $p = 0.746$). This suggests that the energy benefits of batching are independent of whether test suites are CPU-intensive or I/O-bound. In contrast, baseline commit failure rate strongly predicts energy savings, with a significant negative correlation (BatchStop4: $r = -0.997$, $p = 4.93 \times 10^{-8}$; linear-4 lwd: $r = -0.994$, $p = 5.04 \times 10^{-7}$). Projects with higher failure rates achieve lower energy savings because batching must frequently bisect or test commits individually to identify failures, reducing the opportunity for execution savings. This finding indicates that batch testing is most energy-efficient for projects with stable codebases and infrequent test failures.

5 Responsible Research

This study was conducted with attention to both environmental impact and scientific reproducibility.

5.1 Environmental Considerations

The experiments required substantial computational resources, with baseline runs alone consuming over 3,700 Wh across all projects and configurations. To minimise the environmental impact of this energy-intensive research, all experiments were executed on a machine powered entirely by renewable energy. Additionally, the heat generated by the hardware during extended test runs was used to supplement space heating, to make sure that thermal output was not wasted.

5.2 Reproducibility

To support replication and extension of this work, the scripts used to run the experiments were made publicly available [8]. The study uses EnergiBridge, an open-source energy measurement tool, and standard build tools (Maven, Gradle). The experimental hardware and software environment are fully described in Section 3.4.

5.3 Ethical Considerations

This study involved no human participants and required no ethical approval. No external funding was received; this work was conducted as part of the CSE3000 Research Project at TU Delft.

5.4 AI Use Disclosure

Generative AI tools were used to implement parts of the experiment automation script [8], the prompts used are included in Appendix A. The batching algorithm logic is human written; generative AI code was used for automating git, iterating through commits, parsing EnergiBridge output, compiling the results into CSV and preparing the script for publishing in a replication package. All generated code was thoroughly checked and tested before use in live experiments. The text in the paper is human written; no AI tools were used to generate text or find sources.

6 Discussion

This section interprets the empirical findings, relates them to prior work, discusses practical implications, and acknowledges limitations of the study.

6.1 Comparison to Prior Work

Relating these results to prior batching studies is difficult due to methodological differences. Unlike traditional batching simulations that assume independent commits [5][6], this study evaluates batches by testing only the head commit. When the head passes, all commits in the batch are marked as passing, which means failing intermediate commits can be masked by succeeding fixes. This likely inflates savings estimates for low failure rate projects compared to scenarios with truly independent commits.

The results match these expectations. For projects with zero baseline failures (such as Lucene and commons-jxpath), this study observes 87–94% energy savings, substantially higher than the 60–70% execution savings reported by Beheshtian et al. [5] for their lowest failure rate projects (7–10%). This gap likely reflects the dependent commit methodology: when the head commit passes, intermediate failures are masked, inflating savings estimates. Meanwhile, commons-scxml, with a 36% failure rate, achieves only 57–59% savings. Prior work suggested that batching becomes ineffective above 40% failure rate [5], and this result shows diminishing but still meaningful returns as that threshold approaches.

A fundamental limitation of any experiment ran on historical commits is that batching cannot be evaluated in a way that reflects real deployment. In practice, batch sizes and feedback timing influence developer behaviour: the same project with and without batching would produce different commit sequences and different failure rates overall. This feedback loop cannot be captured when replaying an existing commit history.

6.2 CPU Utilisation and Energy Savings

The lack of correlation between baseline CPU utilisation and energy savings is particularly visible when comparing projects with identical failure rates but different CPU utilisation. Lucene and commons-jxpath both have zero test failures, but their average CPU usage differs substantially (82% versus 33%). Despite this, both achieve nearly identical relative savings (87.8% and 87.4% for batchstop4, 93.9% and 93.7% for linear-4 lwd respectively). This suggests that the relative benefit of batching does not depend on whether a test suite is CPU-intensive, at least when measuring CPU package energy.

However, this finding may partly reflect a measurement limitation. EnergiBridge captures only CPU package and does not account for DRAM, storage, or network energy consumption. Test suites with lower CPU utilisation may consume more energy in these unmeasured components. Future work with broader measurements could clarify whether batching benefits differ for I/O heavy versus computation-heavy workloads.

6.3 Practical Implications

The strong negative correlation between baseline failure rate and energy savings has a clear practical implication: batch testing delivers the largest energy benefits for projects with stable test suites. Teams considering batching should first address sources of test instability, as frequent failures force bisection or individual testing that lowers the efficiency gains. For projects already achieving low failure rates, even simple static batching strategies can reduce CI energy consumption by over 85%. However, even with higher failure rates, there are significant savings. Prior work using simulations suggested that the break even point for static batching algorithms is around 40–45% failure rate[5]. This study did not include any projects above this threshold, but commons-scxml, with a 36% failure rate, still achieved 57–59% energy savings. This suggests that batching remains worthwhile even for moderately unstable projects, though future work should evaluate projects closer to and beyond the theoretical break-even point to determine where energy savings become negligible or negative.

6.4 Threats to Validity

Several factors limit the generalisability of these findings. The sample of eight Java projects, while spanning different sizes and failure rates, may not represent other languages, build systems, or CI configurations. The use of a single desktop machine avoids virtualisation overhead but does not reflect cloud CI environments where resource sharing and scheduling introduce additional variability. Finally, restricting energy measurement to CPU packages means that projects with significant I/O or memory activity may have unmeasured energy costs that differ between batching strategies.

7 Conclusion and Future Work

This study presents the first empirical measurement of energy consumption under batch testing strategies in continuous integration. By executing real CI builds on eight open-source Java projects and measuring CPU package energy with EnergiBridge, the results show that both static and dynamic batching strategies substantially reduce energy consumption compared to the standard run-all approach.

BatchStop4 achieves energy savings between 57% and 88% (mean: 80.3%), while Linear-4 LWD achieves savings between 59% and 94% (mean: 85.2%). The near-perfect correlation between time savings and energy savings ($r > 0.99$) indicates that batching reduces energy primarily by reducing total execution time. Baseline failure rate emerges as the dominant predictor of achievable savings: projects with stable test suites benefit most, while higher failure rates lower efficiency gains through repeated bisection. CPU utilisation, by contrast, shows no significant relationship with energy savings in this dataset.

These findings have practical implications for teams looking to reduce the environmental footprint of their CI pipelines. Even simple static batching can yield substantial energy reductions for projects with low failure rates, while dynamic strategies offer modest additional benefits.

7.1 Future Work

Several directions remain for future research. First, this study measures only CPU package energy; incorporating DRAM, storage, and network energy consumption could reveal whether batching benefits differ for I/O-heavy versus compute-heavy test suites. Second, the sample of eight Java projects limits generalisability; future work should evaluate batching across different languages, build systems, and CI configurations. Third, all experiments ran on a single desktop machine; cloud CI environments introduce resource sharing and scheduling variability that may affect energy profiles differently. Finally, historical commit history cannot capture how batching influences developer behaviour; a longitudinal study of batching deployed in active projects would provide insight into real-world energy impacts.

A Appendix: Generative AI prompts

The prompts used for code generation during the implementation of the scripts used to run the experiments for this study are listed below

- ”Included in the projects files are python scripts with logic for two batch testing algorithms. Your task is to implement functionality to tie these together, in addition to a baseline algorithm which runs all commits by default, into an experiment script to be used as a command line tool to automate runs of these algorithms and compare their results. The script must call EnergiBridge, which can be called system wide through the syntax provided in energibridge_documentation.md. A sample energibridge results file is provided, named energibridge-sample.txt. The script must iterate through commits from the head of the branch in the repositories cloned into folders under /projects. The script must invoke energibridge before each build run and save its output under /results/eb. Those outputs must be parsed after a successful run, and converted into CSV format under /results/raw, which must list every build ran as a separate row, and /results/summary, which must list every algorithm run for each project as a separate row. The results CSV formatting specifications are defined in outputformat.md.”
- ”Prepare the experimental scripts for use in a replication package. Replace all absolute paths with relative paths, clear the configuration files of project specific entries and replace them with placeholders explaining the syntax. Do not touch any of the run logic.”

References

[1] A. Zaidman, “An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects,” in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, ser. AST ’24, New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 214–218, ISBN: 979-8-4007-0588-5. doi:https://doi.org/10.1145/3644032.3644461.

[2] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, “Recalibrating global data center energy-use estimates,” *Science (New York, N.Y.)*, vol. 367, no. 6481, pp. 984–986, Feb. 2020, ISSN: 1095-9203. doi:https://doi.org/10.1126/science.aba3758.

[3] O. Danushi, S. Forti, and J. Soldani, “Carbon-Efficient Software Design and Development: A Systematic Literature Review,” *ACM Comput. Surv.*, vol. 57, no. 10, 267:1–267:35, May 2025, ISSN: 0360-0300. doi:https://doi.org/10.1145/3728638.

[4] H. Claßen, J. Thierfeldt, J. Tochman-Szewc, P. Wiesner, and O. Kao, “Carbon-Awareness in CI/CD,” in *Service-Oriented Computing – ICSOC 2023 Workshops*, F. Monti et al., Eds., Singapore: Springer Nature, 2024, pp. 213–224, ISBN: 978-981-97-0989-2. doi:https://doi.org/10.1007/978-981-97-0989-2_17.

[5] M. J. Beheshtian, A. H. Bavand, and P. C. Rigby, “Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2784–2801, Aug. 2022, ISSN: 1939-3520. doi:https://doi.org/10.1109/TSE.2021.3070269.

[6] D. M. Kamath, B. Adams, and A. E. Hassan, “Lightweight dynamic build batching algorithms for continuous integration,” *Empirical Software Engineering*, vol. 30, no. 2, p. 51, Jan. 2025, ISSN: 1573-7616. doi:https://doi.org/10.1007/s10664-024-10555-4.

[7] J. Sallou, L. Cruz, and T. Durieux, *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*, Dec. 2023. doi:https://doi.org/10.48550/arXiv.2312.13897.

[8] M. Oszkó, “Replication package of: The Energy Impact of Batch Testing in Continuous Integration,” Jan. 2026. doi:https://doi.org/10.5281/zenodo.18370532.