# A Linter and Guideline Framework for Sustainable GitHub Actions Workflows

Rebecca Andrei, Boris Annink, Paul Anton, Radu Serban, Kasper van Maasdam
Delft University of Technology

## 1 INTRODUCTION

Software is eating the world [2], and so is its energy footprint. Data centers already consume roughly 1-2% of global electricity [23], and that share continues to grow as software systems scale in size and complexity [8, 22]. Yet much of this consumption is avoidable: inefficient software practices waste compute cycles, and, by extension, energy [10, 17].

Continuous Integration and Continuous Delivery (CI/CD) pipelines sit at an underexplored intersection of software development and energy consumption. Every code change can trigger multiple automated jobs that consume real power, with GitHub Actions [13] alone processing up to 71 million workflow runs per day [29], a number which has grown exponentially from previous years [16]. In practice, many of these jobs perform redundant work, such as rebuilding the same artifacts across jobs or reinstalling dependencies. Although individually, these inefficiencies may seem minor, their cumulative effect is substantial given how frequently CI/CD pipelines are executed [35].

While CI/CD platforms provide mechanisms to reduce redundant computation, such as caching and trigger scoping, these features are not consistently applied in practice. As a result, inefficient patterns remain common in real-world workflows, even though the platform provides the means to avoid them. This creates a disconnect between what platforms enable and what developers actually implement.

This paper addresses that gap by treating inefficient CI/CD workflows as a concrete and measurable sustainability problem. We propose **`suslint`**, a static *linter* for GitHub Actions workflows that detects wasteful patterns and directs developers toward sustainable alternatives. The tool is based on a *rulebook* of ten evidence-based guidelines for sustainable CI/CD pipelines, which are documented on the accompanying *website*. We empirically evaluate **`suslint`** by applying it to a curated set of real-world open-source repositories and measuring the energy impact of the detected inefficiencies using EcoCI [18]. Our results show that these patterns are widespread and that addressing them leads to meaningful reductions in CI-related energy consumption.

The remainder of this paper is structured as follows. Section 2 presents the background and reviews related work. Section 3 defines the problem and our research questions. Sections 4 and 5 describe the proposed solution and its implementation. Section 6 outlines the experimental methodology, Section 7 presents the results, Section 8 discusses their implications, and Section 9 concludes the report.

## 2 RELATED WORKS

Sustainable software engineering has started to increasingly focus on reducing the energy consumption of software systems, especially in large-scale computing environments. Previous work has emphasized that the design and execution of software have a direct impact on the energy consumption of the system [10, 17]. Apart from this, recent research has also introduced methods to measure the emissions of software systems to support more sustainable decision-making.

Recent studies have begun to explore the environmental effect of CI/CD pipelines. For example, Saavedra et al. [35] is one of the first large-scale studies of GitHub Actions, and has demonstrated that CI workflows have a significant carbon and water footprint. This study further highlighted the need to reduce computations and optimize the execution of CI workflows. In parallel, CI/CD pipelines do offer some built-in mechanisms, such as dependency caching and scoped triggers, to reduce redundant computations [13]. However, these optimizations are not consistently adopted in practice. Ghaleb et al. [12] show that CI caching is used by only a minority of projects, with adoption limited by a lack of awareness and the complexity of maintaining caches. This suggests that, despite available platform support, inefficiencies remain common in real-world CI workflows.

Research on the evolution of CI/CD workflows also underscores the difficulties associated with maintaining efficient configurations. For instance, CI/CD workflows are normally represented using a YAML configuration file, and even minor changes to this file may have a substantial impact on the execution of the workflow. According to Rostami et al. [33], CI/CD workflows are updated frequently, but usually only incrementally, by modifying a single element of the configuration. These changes are largely focused on maintaining the functionality of the pipeline rather than improving its structure. The existing tools in this domain primarily target correctness, reliability, and security [15], with limited support for identifying sustainability issues in the design of the workflow.

Several tools have been proposed for more sustainable software development, each of which operates at a particular level of abstraction. EcoCode [25] is a SonarQube plugin that detects energy-related code smells in Android applications, helping developers identify inefficient programming patterns that impact battery consumption. At a higher level, the Software Carbon Intensity (SCI) specification [19] provides a standardized methodology for quantifying the carbon emissions of software systems, encouraging choices that reduce emissions. Other solutions measure the energy consumption at the execution level. CodeCarbon [36] estimates the carbon footprint of a program based on hardware utilization, and EcoCI [18] offers a similar solution for CI/CD pipelines.

Although these approaches improve sustainability at the code, system, and execution levels, they do not directly address any inefficiencies introduced at the CI/CD configuration level. In particular, there is a lack of support for statically analyzing CI/CD workflows to detect inefficient design patterns and provide feedback to the developer before execution. This gap motivates the need for tools that translate sustainability guidelines into automated checks applied directly to CI/CD workflows.

# 3 PROBLEM DEFINITION AND RESEARCH QUESTIONS

CI/CD workflows are executed frequently and at scale, yet they are typically designed to be functional rather than efficient. As a result, workflows often perform unnecessary computations. While CI/CD platforms provide mechanisms to reduce this overhead, their effective use depends on how workflows are configured and is not enforced in practice.

In this work, we treat inefficient CI/CD workflow design as a measurable sustainability problem. Specifically, we focus on identifying workflow patterns that lead to unnecessary computations and, consequently, increased energy consumption. These inefficiencies are formalized into a rulebook, which forms the basis of a static linter that detects them automatically.

Based on this problem, we define the following research questions:

(1) **To what extent do these inefficiencies impact energy consumption in CI/CD pipelines?** This evaluates whether the detected patterns correspond to measurable differences in energy usage when workflows are executed.
(2) **Can automated static analysis effectively identify and guide the removal of these inefficiencies?** This examines whether a rule-based linter can reliably detect problematic patterns and provide actionable feedback to developers.

# 4 PROPOSED SOLUTION

## 4.1 Sustainability rulebook for CI/CD Pipelines

### 4.1.1 Description

To improve the sustainability of GitHub Actions, we introduce a set of rules that, when followed, reduce avoidable computations, and, consequently, the energy consumption. This rulebook provides developers with clear guidance for identifying and improving unsustainable patterns in CI/CD workflows. For instance, a rule could suggest adding caching to an action to prevent re-execution.

Other alternatives were considered. One such example is a guidebook that explains important optimization concepts and their application. Guidebooks offer flexible recommendations and rely on the user's interpretation, leading to variability in their application [1]. In contrast, rule-based systems provide explicit, actionable constraints that reduce ambiguity and improve consistency across users [11]. In this context, we favor broad adoption and enforceability in order to maximize the impact of our findings across workflows; therefore, a rulebook is preferred, as it enables both consistent application and integration into automated analysis tools.

Another alternative was to use an Artificial Intelligence (AI) agent to optimize the workflow on the user's behalf. This would offer flexibility and adaptability but introduce challenges related to reproducibility and explainability. In the context of this work, it is important that developers understand why a recommendation is made. AI solutions, while powerful, often function as black boxes, making it difficult to justify or validate decisions [34]. In contrast, rule-based systems allow each recommendation to be traced back to a clearly defined principle, which improves user trust and makes it more likely for them to adopt the suggestion.

The rulebook defines the patterns that are considered inefficient and therefore directly supports the research questions presented in Section 3. It provides the basis for linking patterns to energy consumption (RQ1) and enables their detection through static analysis (RQ2).

### 4.1.2 Rules

To construct a robust and meaningful rule set, we defined a set of selection criteria that each candidate rule must satisfy. Only rules conforming to all criteria were included. This ensures that the resulting rulebook is a coherent and enforceable system.

The criteria are as follows: (1) *Effective* - the rule must lead to a measurable optimization of workflow execution; (2) *Atomic* - the rule addresses a single, indivisible concern; (3) *Tool Independent* - the rule captures general workflow patterns, regardless of specific technologies (e.g., `npm`, `pip`); (4) *Non-intrusive* - the rule does not alter the intended functional outcome of the workflow; (5) *Actionable* - violations can be resolved through clear and concrete remediation steps; and (6) *Detectable* - violations can be identified through static analysis with reasonable accuracy.

Using explicit criteria to guide our selection of the rules is in line with established practices in engineering design, where structured criteria improve consistency and traceability in the design process [11]. These constraints also help to avoid vague or context-dependent recommendations that would difficult to apply or enforce in practice.

Similarly, the rules themselves were derived through a combination of peer-reviewed research sources [7, 48], and existing industry best practices [9, 28, 40, 41]. The following rules were selected, with `SUS000` representing their unique ID:

(1) **Cache Dependencies (`SUS001`)** and **Reuse Build Artifacts (`SUS005`)** reduce redundant computation and repeated downloads.
(2) **Set Job Timeouts (`SUS002`)** and **Prevent Duplicate Concurrent Runs (`SUS010`)** prevent computations being wasted on stalled or overlapping executions.
(3) **Avoid Triggering on All Branches (`SUS003`)** targets unnecessary workflow triggers.
(4) **Prevent Duplicate Jobs (`SUS004`)** eliminates redundant jobs within a single workflow.
(5) **Avoid Unnecessary Job Serialization (`SUS006`)** and **Merge Small Sequential Jobs (`SUS008`)** reduce the overhead resulting from job orchestration.
(6) **Use Shallow Clones (`SUS007`)** minimizes the data transfer and checkout time.
(7) **Limit Matrix Size (`SUS009`)** controls the growth of job combinations in matrix-based test configurations.

Importantly, each rule satisfies all defined criteria. For example, caching dependencies is both *effective* (reducing runtime and network usage), *actionable* (through configuration changes), and *detectable* (via inspection of workflow steps).

Although the rules are intended to be universally applicable, it is possible that some of them might be context-specific and need to be applied with appropriate care to avoid any adverse trade-offs. For instance, limiting the size of the matrices needs to be appropriately balanced with the requirement for adequate test coverage. As such,

the purpose of this rulebook is to facilitate decision-making and not to impose rigid constraints in all possible scenarios.

By filtering candidate rules through these criteria, the final set provides a practical foundation for detecting inefficiencies. This structured selection process ensures that each rule contributes to improving the sustainability of GitHub Actions workflows in a consistent and measurable way.

### 4.2 Workflow Linter

The guidelines presented as rules in the rulebook are valuable insights into how to improve GitHub CI/CD workflows. However, it may be difficult for developers to apply these rules consistently in their code-bases if they only have this list as a reference. Nevertheless, it is essential for developers to be mindful of the energy footprint of their code and applying the rules inconsistently or incompletely can result in an increase of energy consumption. For this reason, developers should receive help to get a better understanding of the rules and how applying them would reduce energy inefficiencies. To investigate whether automated static analysis can effectively identify and guide the removal of these inefficiencies, a linting tool has been developed. The linter operationalizes the rulebook by translating its guidelines into automated checks, allowing inefficiencies to be systematically identified and their impact on energy consumption to be empirically evaluated.

Linters have been used for educational purposes before [26]. However, normally, a linter outputs warnings that are often non-actionable [47] and lack comprehension [46]. The linting tool in this work enforces the rules in the rulebook by giving descriptive warnings at specific points in CI/CD workflow files. By giving warnings at specific locations, developers can identify where in their code improvements can be made to make their GitHub CI/CD pipeline more sustainable, increasing their actionability.

To further increase actionability, remediation techniques are provided alongside the warnings. If developers want to learn more about the warnings they encounter, they can reference the website[1] on which the rules and remediations are explained in more detail. The purpose of the website is to educate developers on when warnings are truly false-positives and when they are not such that fewer warnings are needlessly ignored, that would otherwise result in an increased energy footprint.

The linter is a static analysis tool. It parses the workflows to perform checks that identify patterns in workflows that match to broken rules. Static analysis was chosen over dynamic analysis because of the nature of GitHub workflows. GitHub workflows are run by default on runners provided by GitHub [15], not on the devices of the developers, making dynamic analysis more complex to deploy and reproduce. Using it would allow for energy measurements and other techniques that could provide more specific suggestions as to how to improve the code's sustainability [31], but it would also add complexity, both to the developer that needs to execute this and to the tool itself [43]. At the cost of more specific and precise suggestions, static analysis circumvents these issues.

Another advantage that stems out of the simplicity of a static analysis tool is the extensibility of the set of rules in the linter, meaning that more rules can be added to the rulebook in the future. As the functionality of GitHub workflows continues to develop and more sustainable practices are discovered, the rule set can be extended accordingly. The extensibility of the linter ensures that new rules can be enforced without much effort.

## 5 IMPLEMENTATION

### 5.1 Linter

This subsection describes the implementation of the workflow linter[2] introduced in Section 4.2 and shows how the rulebook from Section 4.1.2 is applied in practice. The linter is written in Python [32], chosen primarily because it provides access to `ruamel.yaml` [42], a YAML parser that preserves both source coordinates and YAML comments. These two properties are essential for this project, since source coordinates are needed to report violations at precise locations in workflow files and preserved comments make it possible to support inline suppression of warnings. Alternative languages, such as Haskell [20], were considered, but they did not offer the same combination of YAML parsing, comment preservation, and convenience required for this tool.

The linter is implemented as a command-line application exposed through the **`suslint`** entry point. Users can pass individual files, directories, or glob patterns, where glob expansion is used to resolve multiple workflow files [39].

When a directory is given, the tool recursively scans for `.yml` and `.yaml` files. Each resolved file is parsed into a YAML mapping structure and then checked against the active rule set. If the parsing fails, the tool reports this as a clean linting error instead of crashing. If the parsing succeeds, every matching rule violation is returned as a warning containing the violated rule identifier, a human-readable message, the location trail inside the workflow, and the exact line and column number. The text output is designed for direct use by developers, while a JSON output mode is also available [5]. The JSON format includes the same warning data together with severity, category, remediation text, and a summary, making the linter easier to integrate into external tools and automated pipelines.

Internally, the linter is organized around a small rule interface. Each rule provides an identifier, description, metadata, and a `check` function that inspects a parsed workflow and yields zero or more issues. This design keeps the rule implementations independent from the command-line front end and the output renderer.

The current implementation includes ten rules, each stored in its own module under the rules package. Before and during runtime, the linter dynamically imports all rule modules and sorts them by their identifier before execution. This is the element that makes the architecture extensible, as adding a new rule does not require modifying the main linting loop, only implementing a new rule module that conforms to the expected interface.

The implemented rules rely on lightweight static analysis heuristics over the YAML structure. The linter traverses jobs, triggers, steps, and dependencies, directly from the parsed document. For

[1] The website may be found here: https://kaspervanm.github.io/sustainable-gh-workflow-linter/

[2] The replication package for the linter can be found here: https://github.com/KaspervanM/sustainable-gh-workflow-linter

example, it detects missing dependency caching by matching installation commands and cache-related actions, unnecessary serialization by inspecting `needs` relations and checking whether downstream jobs actually consume upstream outputs or artifacts.

For flexible usage, the linter supports two warning suppression mechanisms. First, rule IDs can be enabled or disabled globally from the command line, allowing users to select only relevant checks or ignore rules that do not fit their context. Second, individual warnings can be suppressed inline by adding a YAML comment of the form `# suslint ignore: SUS001, SUS002` to the relevant line. Since comments are preserved by the parser, these overrides can be matched back to the exact warning locations during linting.

### 5.2 Supporting Website

A website was created to support the linter, serving as an entry point where users can understand the purpose of the tool and explore the rulebook, allowing them to manually assess its usefulness before fully adopting it. In addition, it also provides documentation on how to install and start using the tool.

The website was developed using basic HTML, CSS, and JavaScript, allowing for a lightweight and fully customizable implementation. This approach was chosen to maintain full control over the structure and content while keeping development overhead low. Alternatives such as adapting generic templates would require additional effort to learn and customize, and full-scale development using more complex frameworks was deemed unnecessary, as development effort is better allocated to expanding and refining the linter and rule set.

The website is hosted using *GitHub Pages* [14]. The platform enables free hosting of static websites and integrates directly with the repository. Compared to other hosting solutions, it offers a low-complexity and cost-effective deployment option, whereas alternatives may introduce additional setup effort or cost.

The content of the website is centered around the rules. Each rule is presented with an identifier, a name, a short motivation, and a *before/after* example. The rules are intentionally kept brief so that users can quickly scan them, while the examples help the users to rapidly identify issues and understand how to fix them. In this way, the website complements the linter by providing the context and explanations needed to interpret its warnings.

## 6 METHODOLOGY AND EXPERIMENTAL SETUP

### 6.1 Study Design

To address RQ1 (Section 3), we conduct a controlled experiment[3] in which we measure the energy consumption of GitHub Actions workflows under two conditions: a baseline condition reflecting the un-optimized pattern flagged by rule `SUS001` (Section 4.1.2), and a treatment condition in which the `SUS001` fix has been applied. The independent variable is the presence or absence of dependency caching, while the dependent variable is the total energy consumed per workflow run in Joules, as measured by EcoCI [18].

[3]The replication package for the experiment can be found here: https://github.com/PaulAnton03/suslint-experiments-SE/.

We focus specifically on rule `SUS001` because it is both *effective* and *non-intrusive* (Section 4.1.2). Dependency caching can be introduced without modifying the functional behavior or structure of the workflow, which allows us to vary a single factor while holding all others constant. This makes it possible to attribute any observed energy differences, which we expect to be significant, directly to the presence of caching.

For each subject repository, we execute 30 runs of the baseline workflow and 30 runs of the treatment workflow, for a total of 60 runs per repository, as recommended by L. Cruz (2021) [6]. To control for time-of-day effects on shared GitHub Actions runners, whose energy behavior can vary with background system load and network conditions [35], runs are interleaved: each baseline run is immediately followed by a treatment run before the next baseline is triggered (i.e., $B_1$, $T_1$, $B_2$, $T_2$, …, $B_{30}$, $T_{30}$). Interleaving is chosen over batching (e.g., all baseline runs followed by all treatment runs) to prevent systematic bias caused by temporal variance [45], and over randomization because it provides a simple and deterministic pairing between conditions. This ensures that both conditions are exposed to comparable environmental noise throughout the experiment [6].

Both workflow variants are committed to an `experiment` branch on each forked repository. Runs are triggered via the GitHub Actions `workflow_dispatch` API and polled until completion. The sole difference between the baseline and treatment workflow for each repository is the caching configuration. All other steps: checkout, runtime version, install command, are identical. This strict control is necessary to ensure that any measured difference in energy is causally attributable solely to the presence or absence of dependency caching.

### 6.2 Repository Selection

Subject repositories are selected from real-world open-source projects hosted on GitHub.

Repositories were selected according to the following criteria, derived from industry-standard approaches.

(1) **`SUS001` violation.** The repository must contain a GitHub Actions workflow that **`suslint`** flags for rule `SUS001`, i.e., a workflow that installs dependencies without caching. This is necessary to ensure that both baseline and treatment conditions are meaningful and comparable, as repositories without such violations would not allow for a controlled before-and-after comparison. The query that has been used to search for appropriate GitHub repositories can be seen in Figure 1.
(2) **Popularity.** The repository must have a substantial number of GitHub stars, used here as a proxy for community adoption and real-world relevance. Stars are widely used in empirical software engineering research as a selection criterion for this purpose [3, 21, 27].
(3) **Active maintenance.** The repository must be actively maintained. This is measured by checking if it has recent commits (within the last year) and open issues or pull requests, and if it has a number of authors and committers larger than 2, indicators described by Kalliamvakou et al. [24]. Inactive

```
(path:.github/workflows/*.yml OR path:.github/workflows/*yaml)("actions/setup-node" OR "npm install" OR "yarn
    install" OR "pnpm install" OR "pip install" OR "docker/build-push-action") NOT content:"cache-" NOT
        content:"cache:" NOT content:"actions/cache" NOT content:"actions/cache/restore" NOT
            content:"actions/cache/save" NOT content:"secrets" NOT content:"npm ci"
```

**Figure 1:** Search query used to identify GitHub Actions workflows without caching.

repositories may have workflows that no longer reflect current development activity, reducing the ecological validity of the experiment.

(4) **Ecosystem diversity.** The selected repositories should cover different programming languages and dependency ecosystems, to demonstrate that the energy impact of `SUS001` is not specific to a single tool-chain.

Applying these criteria, we selected four repositories, summarized in Table 1. Each repository was forked under the experimenter's GitHub account[4]. An experiment branch was created on each fork containing the two workflow files; the rest of the repository is an unmodified mirror of upstream.

| Repository | Description | Stars |
|---|---|---|
| `jpanther/congo` | Hugo theme (Tailwind CSS) | ~1.6k |
| `rq/rq` | Job queues for Python | ~10k |
| `JustArchiNET/ArchiSteamFarm` | Steam automation | ~12k |
| `django-compressor/django-compressor` | JS/CSS compression | ~2.8k |

**Table 1:** Subject repositories used in the `SUS001` experiment.

For each repository, the experiment workflow is scoped to isolate the dependency installation phase targeted by `SUS001`. Steps unrelated to installation, such as test execution or coverage reporting, are stripped from both the baseline and treatment workflows. Where the original workflow uses a build matrix (e.g., multiple Python or Django versions), a single representative configuration is pinned. This serves two purposes. First, it preserves internal validity by removing confounding variables: CPU-intensive test execution would introduce energy variance unrelated to the caching signal being measured. Second, it keeps individual runs short enough to allow for 30 repetitions per condition within a reasonable time interval. The specific scoping decisions for each repository are documented in the accompanying experiment branch.

## 6.3 Evaluation Metrics

The primary metric is the total energy consumed per workflow run, measured in Joules. This is collected for each run via the EcoCI GitHub Action [18], which instruments the workflow runner by sampling CPU use at regular intervals and multiplying by a power model calibrated for the specific runner hardware. All runs in this experiment execute on GitHub-hosted `ubuntu-latest` runners, which use AMD EPYC 7763 CPUs. EcoCI's power model is calibrated for this hardware.

Direct measurement of hardware-level energy consumption is not possible in this environment, as the runners hosted on GitHub do not provide any power sensors. EcoCI offers a solution to this by approximating the energy consumption. Although there are some estimation errors involved, it allows for a consistent and repeatable measurement across runs, which is necessary for our analysis.

[4] https://github.com/PaulAnton03

Energy in Joules is preferred over proxy metrics such as wall-clock duration or CPU time because it captures both the duration and the power draw of a run in a single physically meaningful quantity. Carbon emissions are not used as the primary metric because grid carbon intensity varies by the location of the associated data center and the time of day, which introduces variance that is unrelated to the workflow configuration under study. Energy is the stable, hardware-level quantity that carbon estimates are derived from, and is therefore the more appropriate basis for comparison.

EcoCI reports energy at the step level, labeled by the `get-measurement` calls placed in the workflow. For the purposes of this study, step-level measurements are summed to yield the total energy per run. This aggregation is correct even for workflows with multiple jobs, since all steps across all jobs carry the same GitHub Actions run identifier and are summed accordingly.

The EcoCI action transmits measurement data to the Green Metrics Tool API [18] after each run. Measurements are then retrieved via the `/v1/ci/measurements` endpoint, matched to individual runs by their GitHub run ID, and stored for analysis.

## 6.4 Analysis Procedure

To determine whether the application of `SUS001` leads to statistically significant and practically meaningful changes in energy consumption (RQ1), we define the statistical analysis procedure before inspecting the collected data. Specifying the analysis beforehand mitigates the risks of making decisions based on data, rather than methodology, and improves the credibility and reproducibility of the results [38].

The analysis is conducted in two stages: a per-repository analysis and a cross-repository aggregation of results. First, each repository is analyzed independently to assess the impact of `SUS001`. For each repository, because the experiment uses a paired design (interleaved baseline and treatment runs), the analysis begins by computing the paired differences between the baseline and treatment runs. The distribution of differences is then tested for normality using the Shapiro-Wilk test [37], which determines whether a parametric or non-parametric statistical procedure is applied for that repository. Testing for normality allows us to select a statistical procedure whose assumptions align with the observed data, rather than applying a single method indiscriminately.

If the paired differences are approximately normally distributed, a parametric approach is used. Specifically, we apply a paired t-test to assess whether the mean difference is statistically significant. In addition, we compute Cohen's d as a measure of effect size and calculate a 95% confidence interval for the mean difference.

If the normality assumption is not satisfied, a non-parametric approach is used instead. In this case, we apply the Wilcoxon signed-rank test [44], which does not assume normality and is robust to skewed distributions, to assess statistical significance. We also compute the rank-biserial correlation as a measure of effect size, as it is specifically suited to paired non-parametric data and provides an

interpretable estimate of the strength of the observed effect. Finally, we estimate a 95% confidence interval for the median difference using bootstrapping. This per-repository split ensures that the statistical method is appropriate for the distribution of the observed data in each individual repository.

In the second stage, all trial-level measurements are analyzed jointly using a linear mixed-effects model, with the presence or absence of `SUS001` as a fixed effect and repository as a random effect [30]. We use it because it allows us to estimate the overall effect of `SUS001` across repositories while accounting for variability between repositories. This model is fitted on the raw data and does not depend on the repository specific summary statistics, so as to avoid the loss of accuracy that occurs when averaging. We report the estimated overall mean difference, its 95% confidence interval, and the p-value for the fixed effect. In addition, we report the estimated variance of the random effect to quantify the variation between repositories.

Together, these two stages provide both detailed results per repository and a combined estimate of the effect across all repositories.

# 7 RESULTS

## 7.1 Normality of Paired Differences

The Shapiro-Wilk test was applied to the paired differences of each repository to assess the normality of their distribution. All four repositories produced non-significant results, indicating that the differences were approximately normally distributed in each case. This justifies the use of paired t-tests, which are more statistically powerful than non-parametric alternatives when their assumptions hold. A paired $t$-test was therefore applied to all four repositories.

Figure 2 shows the Q-Q plots of the paired differences for each repository, which further support this conclusion, as the observed distributions closely follow the theoretical normal distribution.

## 7.2 Energy Distributions

The energy distributions provide an initial, qualitative view of the effect of `SUS001`. Figure 3 shows the distribution of energy consumption per condition for each repository, while Figure 4 shows the distribution of paired differences. Both of the figures indicate that only the `ArchiSteamFarm-SE` repository shows a visible reduction of energy usage when caching is applied, compared to without it. The remaining three show visually less distinctive results. These visualizations are useful because they reveal distributional characteristics, such as spread and overlap, that are not captured by summary statistics alone.

## 7.3 Per-Repository Statistical Tests

Table 2 summarizes the results of the paired $t$-tests per repository. These results confirm our previous observation. For the `ArchiSteamFarm-SE` repository, the decrease in energy consumption is both statistically and practically significant. The very low p-value implies that the result is highly unlikely to occur by chance, while the narrow confidence interval implies a precise estimate.

In contrast, the remaining three repositories fail to demonstrate statistically significant differences at the $\alpha = 0.05$ level and have confidence intervals that contain zero. This implies that there is not enough evidence to conclude that caching had a significant effect on energy consumption in these repositories. Moreover, the effect sizes are small, which implies that even if a significant effect exists, it would likely be negligible in the real world.

## 7.4 Linear Mixed-Effects Model

A linear mixed-effects model was fitted on the full trial-level dataset ($N = 220$ observations across four repositories), with `SUS001` as a fixed effect and repository as a random intercept. The model accounted for the hierarchical structure of the data by separating the global effect of `SUS001` from repository-specific variation, and converged successfully. Figure 5 shows the per-repository effects alongside the combined estimate, and Figure 6 shows the overall effect plot in more detail.

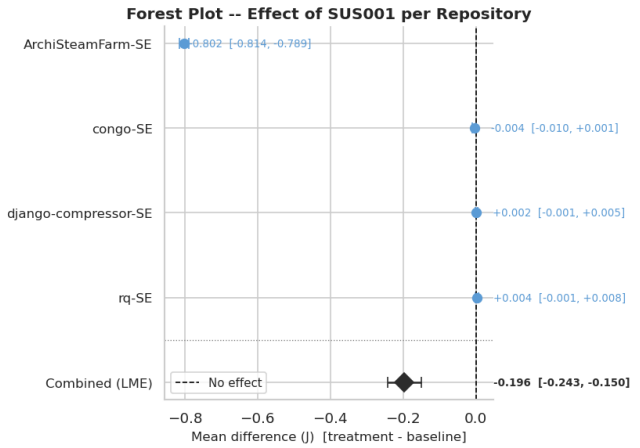


**Figure 5:** Forest plot showing the mean paired difference per repository with 95% CI, and the combined estimate from the linear mixed-effects model (diamond).

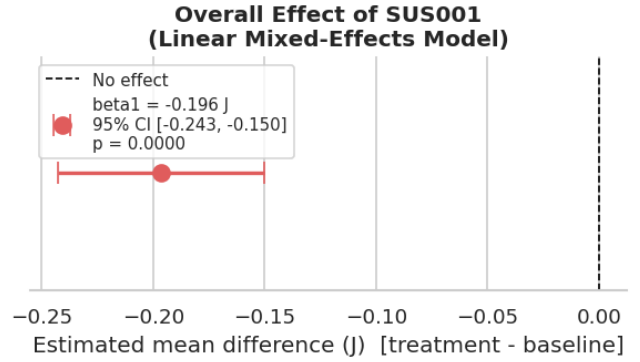


**Figure 6:** Overall effect of `SUS001` on energy consumption estimated by the linear mixed-effects model, with 95% CI.

The estimated fixed effect is negative ($\hat{\beta}_1 = -0.196$ J) and statistically significant ($p < 0.001$), suggesting that, overall, applying `SUS001` leads to a reduction in energy consumption. Nevertheless, the variance of the random effect is large, and the individual variations for each repository indicate that `ArchiSteamFarm-SE` makes a disproportionate contribution to the overall result, which means that the observed effect is not uniform across repositories. This

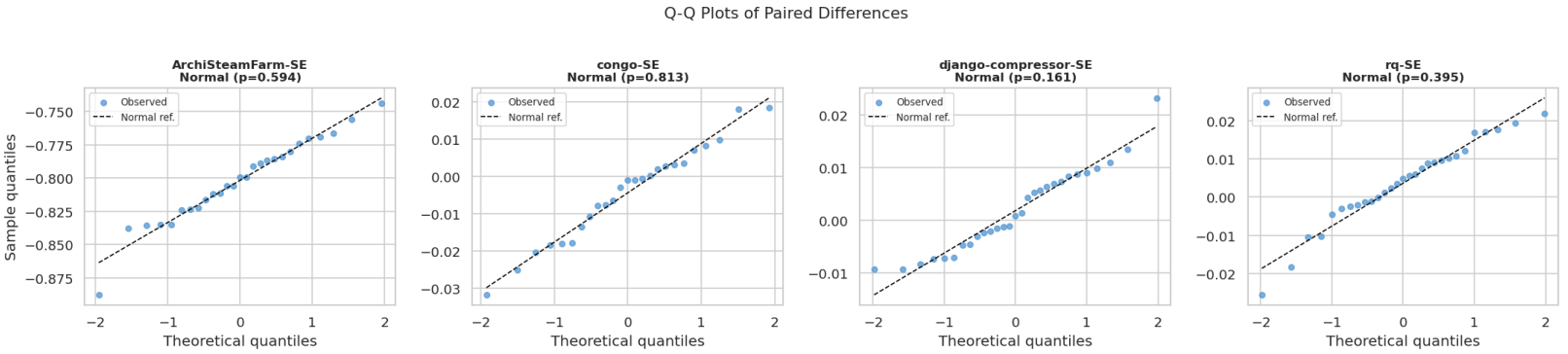


**Figure 2:** Q-Q plot of paired differences per repository

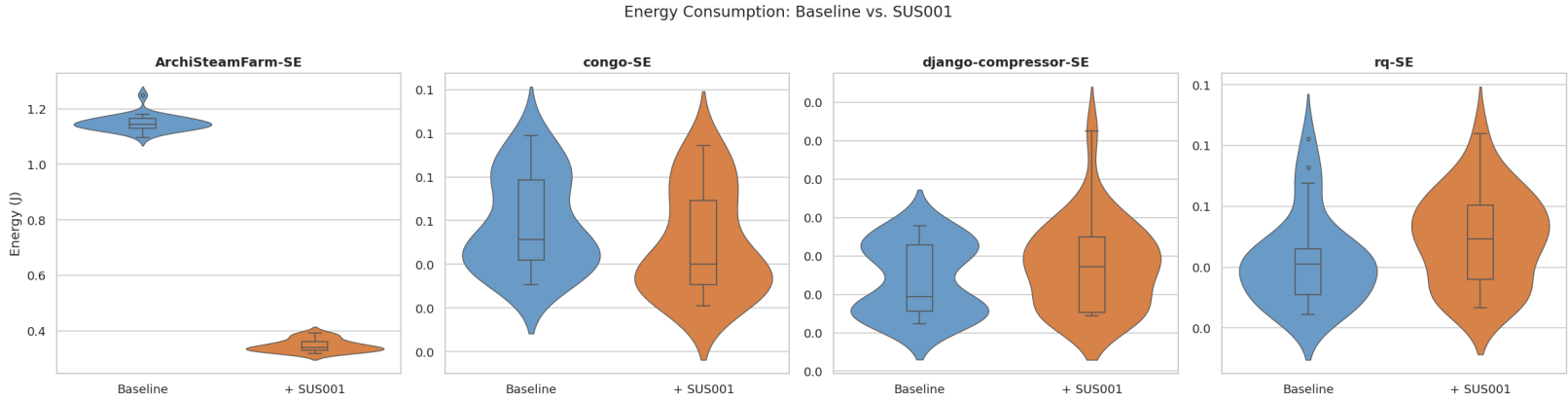


**Figure 3:** Violin-boxplot of energy measurements per repository made with EcoCI.

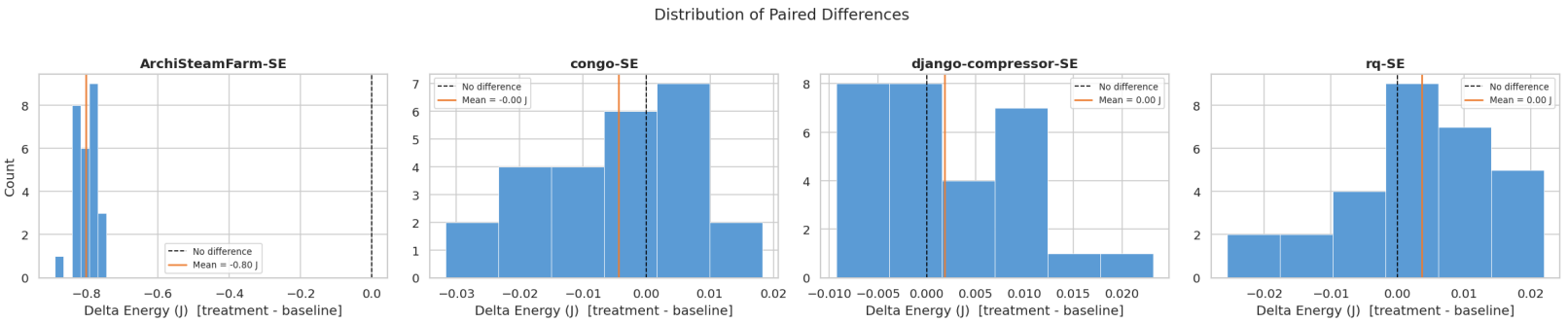


**Figure 4:** Histogram of paired differences per repository

| Repository | n | Mean diff (J) | SD (J) | t | p | Cohen's d | 95% CI (J) |
|---|---|---|---|---|---|---|---|
| ArchiSteamFarm-SE | 27 | −0.802 | 0.031 | −134.20 | < 0.001 | −25.83 | [−0.814, −0.789] |
| congo-SE | 25 | −0.004 | 0.013 | −1.73 | 0.097 | −0.35 | [−0.010, +0.001] |
| django-compressor-SE | 29 | +0.002 | 0.008 | +1.23 | 0.229 | +0.23 | [−0.001, +0.005] |
| rq-SE | 29 | +0.004 | 0.011 | +1.79 | 0.085 | +0.33 | [−0.001, +0.008] |

**Table 2:** Per-repository paired $t$-test results. Mean diff = treatment − baseline. Significance threshold $\alpha = 0.05$.

is where using a mixed-effects model is justified, as otherwise, a simple pooled analysis would mask this information and possibly overgeneralize the findings.

# 8 DISCUSSION

## 8.1 Interpretation of Results

The analysis per repository showed very diverse results. In the `ArchiSteamFarm-SE` repository, applying `SUS001` produced a reduction of approximately 0.80 J per pipeline run, a difference that

was statistically and practically significant ($d = -25.83$). The baseline mean of 1.149 J dropped to 0.347 J with the fix applied, representing a reduction of roughly 70%. The narrow confidence interval and extremely low $p$-value indicate that this effect was measured with high precision and is very unlikely to be a chance finding. Importantly, the magnitude of the reduction indicates that dependency installation constitutes a substantial portion of the total energy cost in this pipeline, making it highly sensitive to optimization.

In contrast, no statistically significant effect was observed in the three remaining repositories. The confidence intervals in all three cases were around zero and the effect sizes were small, which suggest that any potential effect is either negligible or below the detection threshold of the experiment. This suggests that applying `SUS001` to these repositories did not produce a detectable change in energy consumption at the $\alpha = 0.05$ level. Therefore, the results support a causal explanation: the effectiveness of `SUS001` depends on how dominant the affected code is within the overall workflow.

The mixed-effects model further reinforces this interpretation. When interpreting the combined estimate from the linear mixed-effects model ($\hat{\beta}_1 = -0.196$ J, $p < 0.001$), we must take the high variance between repositories ($\hat{\sigma}_u^2 = 0.126$ J$^2$) into account. The `ArchiSteamFarm-SE` repository has a much higher absolute energy level than the other three, which means its effect dominates the combined estimate. The overall significance of the fixed effect therefore primarily reflects the results of `ArchiSteamFarm-SE` instead of a consistent effect across all repositories.

Based on the large difference in results per repository, we can argue that the effect of applying rule `SUS001` on GitHub workflows depends on the workflow design: It depends on how much of the total workflow energy can be attributed to the code path affected by the pattern. In `ArchiSteamFarm-SE`, the affected path appears to account for a large proportion of total pipeline energy. In the other repositories, its contribution is negligible.

Notably, no repository showed an increase in energy consumption upon applying `SUS001`. This is consistent with what the rule measures: the trials capture only the cache-hit scenario, in which a valid cache is found, and the cached step is skipped. The overhead of caching, i.e., populating the cache on a miss, was not measured, as cache misses occur only when the relevant part of the workflow changes. Caching favors use-cases where the ratio of cache-hits to cache-misses is large. GitHub CI/CD pipelines fit this use-case perfectly, because workflows often repeatedly run on code that does not modify the cached data.

## 8.2 Practical Implications

This study addresses RQ1 (Section 3) by evaluating whether the inefficiency patterns formalized in our rulebook correspond to measurable differences in energy consumption. The results confirm that they do: in `ArchiSteamFarm-SE`, remediating a single instance of `SUS001` reduced pipeline energy by roughly 70%, a practically meaningful saving for any project executing CI/CD workflows at scale. When combining this with the fact that only 32.9% of paid-tier repositories adopted caching [4], this demonstrates that the patterns targeted by the rulebook represent genuine sources of unnecessary energy consumption, not merely theoretical inefficiencies.

For the three remaining repositories, the measured energy impact of `SUS001` was negligible. Crucially, however, no repository showed a statistically significant increase in energy consumption as a result of applying the rule. This means that developers can adopt the linter's recommendations without risk of increasing their pipeline's energy footprint, regardless of whether their workflow is structured in a way that yields a large benefit. This is an important property for our tool, because the environmental impact of CI/CD runs is rarely exposed [35].

The results also provide evidence relevant to RQ2 (Section 3). The linter successfully identified a case where a large and measurable inefficiency exists, which supports its usefulness as a diagnostic tool. However, the variability in outcomes highlights a limitation of static analysis: it can detect the presence of a pattern, but not necessarily its runtime impact. This suggests that, in practice, developers may benefit from prioritizing warnings based on expected impact rather than treating all violations equally.

Together, these results show that the inefficiencies found by the rulebook can indeed lead to a measurable energy difference, but the extent to which this is true varies considerably across repositories. Moreover, `suslint`'s ability to identify an instance where a large and measurable inefficiency exists speaks to its potential as a diagnostic tool (RQ2), although the variability in results suggests that static detection alone may not fully capture the runtime impact of all inefficiencies. The fact that only some repositories show strong effects suggests that these inefficiencies are not uniformly impactful, and that simply identifying them is not sufficient to also determine their practical significance. Rather, the value of automated detection lies in enabling developers to reason about them in context, a process supported by both the linter, through identification, and the rulebook, through justification.

## 8.3 Threats to Validity

Several factors may affect the validity of the results. First, energy measurements were collected on shared GitHub Actions runners, where background noise from workloads cannot be fully controlled [35]. Although the paired experimental design mitigates systematic bias by comparing baseline and treatment runs under similar conditions, residual variance from the execution environment may have influenced measurements, particularly in the three repositories where the effect of `SUS001` was small.

Second, ten observations were dropped during pre-processing due to unpaired run indices, likely caused by runner failures or cache-misses. If such failures were non-random with respect to condition, a small selection bias may have been introduced.

Third, the approach relies on the energy estimation model provided by EcoCI, rather than any direct hardware measurement. Although this approach has the advantage of allowing consistent and reproducible data collection, it also runs the risk of suffering from approximation errors, given that there are certain assumptions made about the hardware power consumption. Therefore, the results should be taken with some degree of caution, given that there will be some estimation bias.

Finally, the total energy consumption in Joules, measured at the workflow level, is used as a proxy for the environmental impact of CI/CD execution. This measure does not account for the energy mix

of the data center hosting the runners, nor for embodied carbon in hardware. Furthermore, the measurement captures job-level energy and may not fully isolate the contribution of the specific code path targeted by `SUS001` from other concurrent steps in the workflow.

# 9 CONCLUSION

## 9.1 Summary of Findings

This study provides empirical evidence to support the idea that inefficiencies in CI/CD workflows can indeed result in measurable differences in terms of energy consumption, but that their impact is highly context-dependent. For instance, in one repository, applying `SUS001` resulted in a statistically and practically significant reduction of energy consumption, thus proving that caching can meaningfully improve sustainability when that step constitutes a large portion of the overall workflow. In contrast, the absence of significant effects in the remaining repositories shows that the same inefficiency does not universally lead to measurable savings, and may be negligible when the affected computation contributes only a small share of total energy consumption.

These findings answer RQ1 by showing that inefficiencies captured by the rulebook can correspond to real and practically relevant energy reductions, validating our rule-based approach. At the same time, they answer RQ2 by proving that our static analysis linting tool can successfully identify cases where large inefficiencies exist, even if it fails to determine the actual impact of such inefficiencies on the overall workflow. Taken together, the results establish that while the automated detection of inefficient patterns is both feasible and useful, their practical significance can only be interpreted in the context of the specific workflow in which they occur.

## 9.2 Limitations

This study has several limitations. Firstly, the evaluation is limited to rule `SUS001`, and so the measured energy readings cannot be generalized to the entire set of rules from the rulebook without further experimentation. In practice, we were unable to identify repositories that simultaneously exhibited all or most of the targeted inefficiencies. The reason for this is that some rules, such as `SUS010` and `SUS003` (Section 4.1.2), may require specific scenarios or development cycles before they provide benefit. As a result, the experiment focuses on a single, well-isolated rule, which limits our ability to study the effect of applying multiple rules. Secondly, the experiment is performed using only four repositories. Although the choice of these repositories is intended to provide some level of diversity, they do not represent the entire range of GitHub Actions workflows used in practice.

Thirdly, the workflows were simplified to isolate only the dependency installations, which is more relevant for the purposes of this study, but less representative of real-world CI pipelines. This improved the internal validity of our experiment by reducing confounding factors, but it also reduced our work's generalizability. Therefore, the effect size represents the practical significance of caching in a controlled setting rather than in production workflows, where additional steps may influence the energy usage. Lastly, all of our energy measurements depend on EcoCI's estimation model and on the GitHub runners, whose infrastructure is beyond our control. The use of interleaving helps to minimize this variance, but it cannot eliminate it entirely.

All in all, these limitations mean that our results should be interpreted as strong evidence for the usefulness of our rulebook and linting tool, but cannot be treated as conclusive with regards to sustainable CI/CD pipelines as a whole.

## 9.3 Future Work

Several directions for future work follow directly from the limitations identified in this study.

Firstly, the evaluation should be extended beyond only rule `SUS001`. As we were unable to find repositories that displayed most or all of the intended inefficiencies at the same time, the next logical step would be to create individual experiments for the other rules, and, where possible, for combinations of rules. This would allow us to evaluate the impact of each optimization, as well as to study how multiple rules interact with each other from this perspective.

Secondly, the scope of the evaluation should be expanded. Currently, the evaluation is based on four repositories and has simplified the workflows to isolate the dependency installation jobs. Future work should include a larger, and more varied, set of repositories, as well as more realistic end-to-end workflows. It would improve how generalizable the results are and help determine if the observed energy savings remain meaningful in production-style pipelines.

Thirdly, apart from the evaluation, the tool itself can be further developed. The linter currently uses static analysis, which is scalable and easy to apply. However, this approach is not able to guarantee that a detected pattern actually results in meaningful energy waste at runtime in a specific context. Using a hybrid approach that incorporates execution data within our static checks could provide more precise and context-aware suggestions. Furthermore, the tool could be integrated more tightly into the development process, so as to make it easier to apply in practice. This might be an important direction to pursue because it would allow future work to study how developers react to sustainability warnings over time.

Lastly, the tool currently focuses on identifying inefficiencies in the code, but it does not resolve them directly. Future research could extend the linter to automatically refactor the CI/CD workflows by applying the suggested fixes. Alternatively, the tool could semi-automatically refactor the pipelines by proposing concrete code changes to the user that would resolve the warnings it generated, before waiting for the developer to either approve, reject, or modify them. Regardless of which, the improvement would make it even easier to apply the tool in practice and, consequently, to increase the workflow's sustainability.

# REFERENCES


[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language.* Oxford University Press.

[2] Marc Andreessen. 2011. Why Software Is Eating the World. *The Wall Street Journal* (20 August 2011). https://www.wsj.com/articles/SB10001424053111903480904576512250915629460 Also available at https://a16z.com/why-software-is-eating-the-world/.

[3] Hudson Silva Borges, André C. Hora, and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[4] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24).* Association

for Computing Machinery, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/3597503.3623303
[5] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. (Dec. 2017). https://datatracker.ietf.org/doc/html/rfc8259
[6] Luis Cruz. 2021. Green Software Engineering Done Right: A Scientific Guide to Set Up Energy Efficiency Experiments. Blog post. (October 2021). https://doi.org/10.6084/m9.figshare.22067846.v1
[7] Paul M. Duvall. 2011. Continuous Delivery: Patterns and Antipatterns in the Software Life Cycle. DZone Refcard #145. (2011). https://dzone.com/refcardz/continuous-delivery-patterns
[8] European Commission, Directorate-General for Energy. 2025. In Focus: Data Centres – An Energy-Hungry Challenge. (17 November 2025). https://energy.ec.europa.eu/news/focus-data-centres-energy-hungry-challenge-2025-11-17_en
[9] Marcus Felling. 2025. Optimizing GitHub Actions Workflows for Speed. (2025). https://marcusfelling.com/blog/2025/optimizing-github-actions-workflows-for-speed
[10] Future Bridge NetZero. 2024. The Impact of Software Optimization on Data Center Energy Consumption. https://netzero-events.com/the-impact-of-software-optimization-on-data-center-energy-consumption/. (2024).
[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[12] Taher A. Ghaleb, Daniel Alencar da Costa, and Ying Zou. 2026. The Promise and Reality of Continuous Integration Caching: An Empirical Study of Travis CI Builds. (2026). arXiv:cs.SE/2601.19146 https://arxiv.org/abs/2601.19146
[13] GitHub. 2025. GitHub Actions Documentation. https://docs.github.com/actions. (2025).
[14] GitHub, Inc. 2026. GitHub Pages. https://pages.github.com/. (2026).
[15] GitHub, Inc. 2026. Workflows and Actions. https://docs.github.com/en/actions/concepts/workflows-and-actions. (2026).
[16] GitHub Staff. 2025. Octoverse: A New Developer Joins GitHub Every Second as AI Leads TypeScript to #1. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/. (28 October 2025).
[17] Google Cloud. 2026. Develop Energy-Efficient Software. https://docs.cloud.google.com/architecture/framework/sustainability/energy-efficient-software. (28 January 2026).
[18] Green Coding Solutions. 2026. Eco CI: Energy and Carbon Emissions for CI/CD Pipelines. https://www.green-coding.io/products/eco-ci/. (2026).
[19] Green Software Foundation. 2021. Software Carbon Intensity (SCI) Specification. https://sci.greensoftware.foundation. (2021).
[20] Haskell Community. 2024. The Haskell Programming Language. https://www.haskell.org. (2024). Accessed: 2024-03-25.
[21] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358
[22] International Energy Agency. 2024. What the Data Centre and AI Boom Could Mean for the Energy Sector. (18 October 2024). https://www.iea.org/commentaries/what-the-data-centre-and-ai-boom-could-mean-for-the-energy-sector
[23] International Energy Agency. 2025. Energy and AI. (2025). https://www.iea.org/reports/energy-and-ai
[24] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. An In-Depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071. https://doi.org/10.1007/s10664-015-9393-5
[25] Olivier Le Goaer and Julien Hertout. 2022. ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, 157:1–157:4. https://doi.org/10.1145/3551349.3559518
[26] Susan A. Mengel and Vinay Yerramilli. 1999. A case study of the static analysis of the quality of novice student programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99)*. Association for Computing Machinery, New York, NY, USA, 78–82. https://doi.org/10.1145/299649.299689
[27] Nuthan Munaiah, Shane Kroh, C. Cabrey, and Daniel M. German. 2017. Curating GitHub for Engineered Software Projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6
[28] Nizar. 2024. Optimizing GitHub Workflows for Efficiency and Sustainability. (2024). https://nizar.se/optimizing-github-workflows-for-efficiency-and-sustainability/
[29] Ben De St Paer-Gotch. 2025. Let's Talk About GitHub Actions. https://github.blog/news-insights/product-news/lets-talk-about-github-actions/. (11 December 2025).
[30] Jose C. Pinheiro and Douglas M. Bates. 2000. *Mixed-effects models in S and S-plus*. Springer.
[31] Carlos Pulido, Ignacio García, M Ángeles Moraga, Félix García, and Coral Calero. 2025. Pypen: Code instrumentation tool for dynamic analysis and energy efficiency evaluation. *Computer Standards Interfaces* 94 (2025), 104000. https://doi.org/10.1016/j.csi.2025.104000
[32] Python Core Team. 2019. *Python: A dynamic, open source programming language*. Python Software Foundation. https://www.python.org/
[33] Pooya Rostami Mazrae, Alexandre Decan, Tom Mens, and Mairieli Wessel. 2026. An empirical study of the evolution of GitHub actions workflows. *Journal of Systems and Software* 236 (June 2026), 112824. https://doi.org/10.1016/j.jss.2026.112824
[34] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1, 5 (2019), 206–215. https://doi.org/10.1038/s42256-019-0048-x
[35] Nuno Saavedra, Alexandra Mendes, and João F. Ferreira. 2025. Environmental Impact of CI/CD Pipelines. (2025). arXiv:cs.SE/2510.26413 https://arxiv.org/abs/2510.26413
[36] Victor Schmidt, Kamal Goyal, Aditya Joshi, Boris Feld, Liam Conell, Nikolas Laskaris, Doug Blank, Jonathan Wilson, Sorelle Friedler, and Sasha Luccioni. 2021. CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing. https://github.com/mlco2/codecarbon. (2021). https://doi.org/10.5281/zenodo.4658424
[37] Samuel S. Shapiro and Martin B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3-4 (1965), 591–611. https://doi.org/10.1093/biomet/52.3-4.591
[38] Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. 2011. False-Positive Psychology: Undisclosed Flexibility in Data Collection and Analysis Allows Presenting Anything as Significant. *Psychological Science* 22, 11 (2011), 1359–1366. https://doi.org/10.1177/0956797611417632 arXiv:https://doi.org/10.1177/0956797611417632 PMID: 22006061.
[39] Ken Thompson and Dennis M. Ritchie. 1971. *UNIX Programmer's Manual* (first ed.). Bell Telephone Laboratories, Murray Hill, NJ. https://web.archive.org/web/20000829224359/http://cm.bell-labs.com/cm/cs/who/dmr/man71.pdf Section: glob – global. Archived from the original: http://cm.bell-labs.com/cm/cs/who/dmr/man71.pdf.
[40] Turso Tech. 2024. A Simple Trick to Save Environment and Money When Using GitHub Actions. (2024). https://turso.tech/blog/simple-trick-to-save-environment-and-money-when-using-github-actions
[41] UCL Research Software Development Group. 2023. Using Continuous Integration Efficiently. (2023). https://blogs.ucl.ac.uk/research-software-development/using-continuous-integration-efficiently/
[42] Anthon van der Neut. 2025. ruamel.yaml. https://yaml.dev/doc/ruamel.yaml/. (2025).
[43] Radhika D. Venkatasubramanyam and Sowmya G. R. 2014. Why is dynamic analysis not used as extensively as static analysis: an industrial study. In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices (SERIPs 2014)*. Association for Computing Machinery, New York, NY, USA, 24–33. https://doi.org/10.1145/2593850.2593855
[44] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. https://doi.org/10.2307/3001968
[45] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2
[46] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. 2021. Understanding static code warnings: An incremental AI approach. *Expert Syst. Appl.* 167, C (April 2021), 12. https://doi.org/10.1016/j.eswa.2020.114134
[47] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. 2023. How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs. *IEEE Trans. Softw. Eng.* 49, 4 (April 2023), 2856–2872. https://doi.org/10.1109/TSE.2023.3234206
[48] Fiorella Zampetti, Claudio Vassallo, Sebastiano Panichella, Gerardo Canfora, and Massimiliano Di Penta. 2020. An Empirical Characterization of Bad Practices in Continuous Integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135. https://doi.org/10.1007/s10664-019-09785-8