# Energy-Aware Code Analyzer: Detecting Energy-Inefficient Code Patterns for Sustainable Software Engineering

**Maciej Wierzbicki, Mikołaj Magiera, Figen Ulusal,**
**Radu Chiriac, Ibrahim Badr**
CS4575 Sustainable Software Engineering 2026
Delft University of Technology (TU Delft)
Delft, The Netherlands

*Abstract*—**Software design choices influence how efficiently hardware resources are used and therefore affect energy consumption. While prior work has compared programming languages in terms of energy efficiency, developers often work within fixed ecosystems such as Python. A more practical question is which coding patterns are unnecessarily inefficient and how they can be improved. Existing tools either estimate overall energy use without identifying code-level causes or provide static analysis without focusing on sustainability. In this paper, we present GREENLINT, a lightweight Python command-line tool that detects energy-relevant code patterns using static AST analysis. The tool targets seven common inefficiencies and provides linter-style warnings together with benchmark-informed feedback and a simple scoring mechanism. Runtime differences between functionally equivalent inefficient and improved implementations are used as a reproducible proxy for relative energy impact. We evaluate GreenLint using a curated benchmark suite and a set of open-source Python repositories. The results show that the targeted patterns are consistently associated with higher runtime in controlled settings, although their impact varies across rules. The repository analysis further shows that while such patterns occur in practice, detection accuracy is limited by the lack of contextual information.These results show that sustainability-oriented feedback can be integrated into development workflows in a lightweight way, but improving contextual precision and strengthening the empirical basis of the feedback remain important challenges.**

## I. INTRODUCTION

Software is part of the environmental footprint of digital systems because it affects how intensively hardware resources are used and how long computations take to complete. Estimates of the ICT sector's contribution to global greenhouse gas emissions vary, but recent work places it in the range of roughly 1.4% to 3% of global emissions [1]. This makes software design one relevant part of sustainable software engineering. Even when hardware and infrastructure remain fixed, implementation choices can still affect runtime behavior and therefore operational resource use [2].

Python is a relevant case for studying this issue. It is widely used in web development, automation, data analysis and machine learning, but benchmark studies have also shown that Python is relatively energy-intensive compared with more efficient compiled languages [3], [4]. In practice, developers often cannot simply switch to another language. They work within existing ecosystems, libraries and team conventions. For that reason, the more practical question is not whether Python should be replaced, but which coding patterns within Python are relatively more wasteful and how developers can be supported in improving them.

Existing tools do not fully address this problem. Runtime-oriented tools can estimate overall energy use, but they do not indicate which parts of the code are responsible. Static analysis tools can detect code smells and anti-patterns, but they are generally aimed at maintainability, correctness or style rather than energy efficiency. Moreover, existing energy-oriented static tools are often focused on other ecosystems or require heavier infrastructure [5]. As a result, developers working in Python still lack a lightweight tool that provides pattern-level feedback grounded in empirical comparison.

This creates a gap at the intersection of four requirements. A useful tool should (1) target Python specifically, (2) work at the level of concrete code patterns, (3) rely on empirical comparison rather than heuristics alone and (4) remain lightweight enough to fit ordinary development workflows. We did not identify a lightweight Python tool that combines these four properties.

This paper presents GREENLINT, a lightweight command-line linter for Python that detects seven energy-relevant code patterns through static AST analysis. The tool reports warnings in a linter-style format and links each detection to benchmark-informed feedback. By comparing functionally equivalent inefficient and improved implementations, GreenLint estimates the relative impact of each pattern and reflects this in its scoring model.

A practical challenge in this work is how to evaluate energy impact reliably. We initially aimed to use direct energy measurement. However, in practice, measurements proved difficult to standardize across devices, operating systems and benchmark runs. At our benchmark scale, the readings were noisy and inconsistent enough that they were not suitable as the main basis for comparing patterns. For this reason, we use runtime as a reproducible proxy for energy consumption. More specifically, we compare functionally equivalent inefficient and

improved implementations and interpret runtime differences as relative indicators of likely energy impact. This choice is supported by earlier work showing that runtime and energy use are often positively correlated, although not identical in all settings [6], [7]. Derived joule and $CO_2$ values are therefore used only for relative comparison and ranking, not as precise physical measurements. This approach allows us to evaluate patterns consistently while remaining explicit about the limits of the method.

To evaluate GreenLint, we address the following research questions:

- **RQ1:** To what extent are the targeted code patterns associated with measurable energy waste?
- **RQ2:** To what extent does faster code correspond to greener behavior in our evaluation setting?
- **RQ3:** How frequently do these patterns occur in real-world Python repositories?
- **RQ4:** How accurately does GreenLint detect these patterns in practice?

This paper makes three main contributions. First, we introduce GREENLINT, a lightweight Python command-line tool that detects energy-relevant code patterns using AST-based static analysis and integrates directly into existing development workflows. Second, we provide a benchmark-informed rule set by comparing functionally equivalent inefficient and improved implementations and using these results to guide feedback and scoring. Third, we evaluate the tool on both curated benchmarks and open-source repositories to assess its impact, prevalence and detection quality.

## II. BACKGROUND & RELATED WORK

This section positions GreenLint in relation to earlier work on software energy efficiency, energy-oriented code smells, runtime measurement tools and Python static analysis. It also explains why this study uses runtime as a proxy for energy impact instead of relying on direct hardware-level measurements.

### A. Software Energy Efficiency

Software affects energy use because implementation choices influence how long hardware remains active and how intensively system resources are used. Prior work has shown that software design decisions, algorithms and programming language choice can all affect energy consumption [2], [6].

At the programming-language level, Pereira et al. [3] compared energy consumption across 27 programming languages using Intel RAPL measurements on benchmarks from the Computer Language Benchmark Game. They found that compiled languages such as C, Rust and C++ were substantially more energy-efficient than interpreted languages, with Python ranking near the bottom. Their later journal extension [4] added memory analysis and a validation study using Rosetta Code benchmarks, confirming the earlier results. These studies are useful at the language level, but they do not identify which patterns within Python are relatively wasteful. For developers who need to use Python because of team choices, dependencies or deployment constraints, the fact that another language is more efficient does not provide direct guidance on what to improve in existing Python code.

### B. Energy Code Smells and Green Static Analysis

The idea that certain code patterns are unnecessarily wasteful has mainly been studied under the label of *energy code smells*. Early work in this area focused strongly on mobile software. Cruz and Abreu [8], [9] identified recurring energy-related patterns in Android and iOS applications based on commits, issues and pull requests from more than 1,000 mobile apps. These patterns included issues related to wake locks, GPS use, display behavior, network activity and caching. Le Goaer and Hertout [5] translated part of this work into ecoCode, a SonarQube plugin for detecting energy smells in Android Java.

More recently, energy-oriented rule sets have expanded beyond mobile development. The broader creedengo initiative includes Python-related rules, for example for repeated string concatenation, wildcard imports, SQL inside loops and some PyTorch-specific patterns. This is relevant because it shows that energy-oriented static analysis is possible in Python as well. However, such tools generally rely on predefined heuristics and heavier infrastructure, while the empirical basis of individual rules often remains limited.

Other work has examined energy implications of specific code-level choices. Hasan et al. [10] profiled the energy consumption of Java collection classes and showed that choosing a suboptimal collection type can lead to substantially higher energy costs than using a better alternative. Oliveira et al. [11] extended this line of work with a recommendation system for selecting more energy-efficient Java collections. Gurung et al. [12] focused specifically on inefficient loop constructs in Java and developed a SonarQube plugin called GreenForLoops.

These studies support the broader idea that code-level patterns can have measurable energy consequences, but they are not Python-specific. Python differs from Java in runtime behavior and idiomatic usage: it is interpreted, dynamically typed, constrained by the GIL and commonly relies on constructs such as comprehensions, generators and dataframe operations. As a result, a Python-specific rule set is needed rather than a direct transfer of Java- or Android-oriented findings.

### C. Runtime Energy Measurement Tools

A separate line of work focuses on measuring the energy use of software during execution. CodeCarbon [13] is a widely used open-source Python library that estimates CPU and RAM energy use through hardware counters where available, with fallback methods on systems where those counters are not accessible. Green Algorithms [14] offers an online calculator for estimating the carbon footprint of computations.

These tools are useful for estimating overall energy consumption, but they typically do not indicate which specific code pattern should be changed. In other words, they answer

how much energy a program used, but not where in the code that use is likely to originate. GreenLint focuses on that second question by linking static detections to benchmark-informed feedback.

### D. Why We Use Runtime as a Proxy

One challenge in this project was how to evaluate energy impact in a way that is reproducible across our benchmark suite. We initially aimed to rely on direct energy measurements. However, in practice, hardware-level readings proved difficult to standardize across devices, operating systems and measurement setups. Even though cross-platform energy tools have improved, obtaining stable and comparable readings remains technically demanding and sensitive to machine state, background activity and platform-specific conditions.

For our benchmark scale, direct measurements were noisy and inconsistent enough that they were not suitable as the main basis for comparing rules. We therefore chose to use runtime as a reproducible proxy. For each ECO rule, we compare a functionally equivalent inefficient implementation and improved implementation and interpret the runtime difference as a relative indicator of likely sustainability impact. Any joule and $CO_2$ values derived from this process are therefore approximate and intended for comparison rather than exact hardware accounting.

This choice is supported by prior empirical work showing that execution time and energy usage are often positively correlated, even though they are not identical. For example, a recent empirical study on Pandas and Polars found a strong positive relationship between performance and energy use across the evaluated data analysis tasks [7]. At the same time, earlier software-energy literature has shown that the relationship is not universal and can be affected by factors such as memory behavior, I/O characteristics and CPU state changes [15]. For that reason, GreenLint uses runtime as a practical proxy for relative ranking, not as a claim of exact physical energy savings.

### E. Python Linters and Developer Workflow

Python already has a mature ecosystem of static analysis tools. Pylint [16] focuses on code quality, style and potential errors, while Flake8 [17] mainly enforces PEP 8 style conventions. These tools are widely used in IDEs, CI pipelines and pre-commit hooks and developers are accustomed to acting on their output.

However, their rules are not designed specifically around energy efficiency. Some rules may overlap indirectly with resource use, such as warnings about unused imports, but they do so for code quality reasons rather than energy reasons. They also do not target a benchmark-informed set of Python-specific energy-relevant patterns. GreenLint builds on the familiar linter model, but uses it to provide sustainability-oriented feedback that remains lightweight enough for ordinary development workflows.

### F. Summary of the Positioning

Existing work shows three things clearly. First, software design choices can influence energy use. Second, code-level patterns can matter, but most energy-smell research and tooling has focused on mobile or Java ecosystems. Third, energy measurement tools and static analysis tools solve different parts of the problem: measurement tools provide overall estimates, while linters provide localized feedback.

GreenLint is positioned at that intersection. It targets Python specifically, works at the level of concrete code patterns, stays lightweight enough for normal development workflows and grounds its feedback in benchmark-based comparison. Its purpose is not to provide exact hardware-level energy accounting, but to give developers practical and empirically informed guidance on patterns that are likely to be unnecessarily wasteful.

## III. Energy-Aware Code Analyzer

GreenLint analyzes Python source files using AST traversal. It detects a set of predefined patterns that are associated with potentially inefficient execution. Examples include:

- Loop-based list construction vs. list comprehensions
- Membership checks in lists vs. sets
- Iterative DataFrame processing vs. vectorized operations

For each detected pattern, the tool provides:

- A description of the issue
- A suggested alternative
- An indicative estimate of relative energy impact (based on benchmarks)

The tool outputs results in a format similar to existing linters, enabling integration into CI/CD pipelines.

## IV. Implementation

### A. Implementation overview

We implemented GreenLint as a lightweight, installable Python package (declared in `pyproject.toml`) that exposes a `greenlint` console command. The CLI accepts either a single `.py` file or a directory. In the latter case, it recursively collects all Python files, analyzes them one by one, prints diagnostics in a linter-style single-line format, and exits with a non-zero status code if at least one warning is found. This allows the tool to be easily integrated into scripts or CI pipelines.

The codebase is organized by responsibility. `greenlint/rules.py` defines the seven pattern checks, `greenlint/analyzer.py` parses source files using Python's `ast` module and applies the rules, `greenlint/diagnostics.py` defines the warning structure and formatting (including estimated savings), and `greenlint/cli.py` handles file discovery, execution flow, and aggregation of results across multiple files.

### B. How the analysis works

GreenLint performs *static* analysis by parsing source code into an abstract syntax tree using `ast.parse`. The code is not executed during linting, which keeps the tool fast and avoids any side effects.

For each file, GreenLint:

1) parses the source into an AST,
2) applies all enabled rules,
3) emits diagnostics including line number, rule code (`ECO1`–`ECO7`), message, and suggestion,
4) attaches benchmark-derived proxy estimates (Joules and $CO_2$e) and computes a file-level Green Score.

### C. Implemented rule set

The linter implements seven rules aligned with benchmark pairs under `benchmarks/eco1`–`benchmarks/eco7`:

- **ECO1**: index-based loops using `range(len(...))` instead of direct iteration,
- **ECO2**: repeated `list.append` in loops instead of comprehensions or batching,
- **ECO3**: repeated string concatenation inside loops,
- **ECO4**: inefficient membership checks in loops,
- **ECO5**: unused heavyweight imports,
- **ECO6**: row-wise DataFrame iteration via `iterrows()`,
- **ECO7**: use of `apply()` instead of vectorized operations.

Each diagnostic is designed to be actionable, providing both a short explanation and a concrete suggestion.

### D. Benchmark pipeline, `bad.py`, and `good.py`

To connect static warnings with sustainability-related signals, we implemented an offline benchmarking pipeline under `experiments/`. Each rule is associated with a pair of scripts: `bad.py`, which intentionally contains the anti-pattern, and `good.py`, which provides a refactored version solving the same task.

These scripts act as controlled micro-benchmarks. Their purpose is not to model realistic applications, but to isolate a single pattern and measure the relative difference between two implementations under repeatable conditions. Importantly, GreenLint itself never executes user code—these scripts are only used during benchmarking.

The benchmark runner executes each script multiple times, records results in `raw_runs.csv`, and aggregates them into `summary.json`. For each rule, the summary stores median runtime, proxy energy, proxy emissions, and the resulting difference between bad and good implementations. A `weight` is derived from the relative size of this difference and later used in scoring.

### E. From wall-clock time to proxy Joules and $CO_2$

Each benchmark run is timed using a high-resolution monotonic clock. Instead of relying on hardware energy measurements, runtime $t$ is converted into proxy energy and emissions using fixed scaling factors:

$$E_{\text{proxy}} = t \cdot (\text{J/s}), \quad m_{\text{CO}_2,\text{proxy}} = t \cdot (\text{kg/s}).$$

These factors are configurable via environment variables, allowing the model to remain simple but adjustable. For each rule, the pipeline computes median values across runs and reports the non-negative difference between the bad and good variants. This difference is used as the estimated saving.

### F. Why we use runtime-based proxies instead of direct energy measurement

Our approach relies on wall-clock runtime as a proxy for energy consumption rather than direct measurement. While this does not produce precise physical energy values, it enables consistent and reproducible comparisons across different machines.

Prior work in green software engineering shows that energy consumption is closely related to execution time, as energy can be approximated as power integrated over time. Empirical studies demonstrate that, for functionally equivalent implementations, faster programs tend to consume less energy [3], [18]. At the same time, direct energy measurements are highly dependent on hardware, operating systems, and tooling, making them difficult to standardize and reproduce across environments [19].

Because of this, many studies rely on relative comparisons rather than absolute energy values [18]. We follow the same principle: the goal is not to estimate exact energy usage, but to determine which implementation is more efficient and by how much.

In our setting, this choice is motivated by:

- **Accessibility:** benchmarks can be run on standard machines without specialized tools,
- **Reproducibility:** results are consistent across environments,
- **Simplicity:** the pipeline remains lightweight and easy to maintain,
- **Interpretability:** results directly reflect relative efficiency (longer runtime implies higher proxy cost).

The reported Joule and $CO_2$ values should therefore be interpreted as relative indicators rather than exact measurements.

### G. Output and scoring

GreenLint outputs diagnostics in a standard linter format: file path, line number, rule code, message, and suggestion, followed by an estimated saving derived from `summary.json`.

Each file is also assigned a **Green Score** between 0 and 10. The score starts at 10, and each triggered rule subtracts its corresponding weight. The result is fixed to the valid range and rounded to an integer. This ties the score directly to the measured impact of each pattern, rather than treating all warnings equally.

### H. Validation

We validated the implementation using automated tests under `tests/`. These ensure that each `bad.py` example triggers the expected rule, that `good.py` remains clean, and that CLI behavior, summary loading, and scoring work correctly. Together with the benchmark regeneration process

described in the repository, this provides a reproducible way to verify that detection, measurement, and output remain aligned.

## V. Evaluation

After presenting the design and implementation of GreenLint, we now assess how well the tool performs in practice. The goal of this evaluation is not only to show that the tool works technically, but also to examine whether its warnings are meaningful from a sustainability perspective. Therefore, we evaluate GreenLint from both a controlled and a practical point of view.

More specifically, we study four research questions. RQ1 asks whether the targeted code patterns are associated with measurable energy waste. Next, RQ2 examines whether lower runtime always corresponds to greener behavior in our evaluation setting. Then, RQ3 investigates how common these patterns are in real-world Python repositories. Finally, RQ4 evaluates how accurate GreenLint's detections are. These questions allow us to assess GreenLint along four important dimensions: impact, interpretation, prevalence and precision.

### A. Experimental Setup

We evaluated GreenLint in two complementary settings, a curated benchmark suite and a collection of open-source Python repositories. The benchmark suite contains seven pairs of Python programs, one pair for each implemented GreenLint rule. In each pair, the `bad.py` file contains the targeted inefficient pattern, whereas the corresponding `good.py` file contains a more efficient alternative. This setup was used to study controlled differences between inefficient and improved code.

The benchmark experiments were executed on a single machine running macOS Tahoe 26.4 with Python 3.11.5 inside a virtual environment. Each benchmark pair was run 30 times. We then used the observed runtime differences to derive proxy estimates for joules and $CO_2$ emissions. Therefore, the reported energy values should be interpreted as runtime-based proxy estimates rather than direct hardware-level power measurements.

In addition, we applied GreenLint to ten open-source Python repositories drawn from different usage domains in order to complement the curated benchmark suite with a lightweight real-world evaluation. The selected repositories were **pallets/flask**, **pandas-dev/pandas**, **pallets/click**, **psf/requests**, **pydantic/pydantic**, **tox-dev/platformdirs**, **pallets/itsdangerous**, **pallets-eco/blinker**, **pallets/markupsafe**, and **jpadilla/pyjwt** [20]–[29]. We ran GreenLint on the main Python source directory of each project and recorded the resulting warnings per repository and per rule.

### B. Metrics

We used different metrics for the benchmark-based and repository-based parts of the evaluation. For the curated benchmark suite, the main metric was the median runtime difference between the inefficient and improved variants. We chose this metric because each benchmark was executed 30 times and repeated runtime measurements can vary slightly due to background activity and normal system noise. Therefore, the median provides a more robust summary than the mean, since it is less affected by occasional outliers. In addition, the runtime difference between the `bad.py` and `good.py` variants captures the effect of the targeted pattern more directly than reporting raw runtimes in isolation.

Next, we report the corresponding proxy estimates for joules and $CO_2$ emissions. We include these metrics because the goal of GreenLint is not only to highlight slower code, but also to approximate its sustainability impact. At the same time, these values are derived from runtime using fixed conversion factors. As a result, runtime difference remains the primary measured quantity, whereas joules and $CO_2$ are reported as derived proxy metrics. We also report the rule weight assigned to each pattern, because this weight reflects the relative severity inferred from the benchmark results and explains how GreenLint translates benchmark evidence into its scoring model.

For the repository scan, we manually inspected a sample of warnings produced by GreenLint and labeled each one as a true positive, false positive or borderline case. A warning was labeled as a true positive when GreenLint correctly identified the intended inefficient pattern and the suggested improvement was reasonable in context. A warning was labeled as a false positive when the warning was misleading or not applicable. Borderline cases were syntactically correct matches whose usefulness depended on contextual factors.

To assess detection accuracy, we manually inspected a sample of warnings from the repository scan and labeled each one as a true positive, false positive or borderline case. We used this manual assessment because prevalence alone does not indicate whether GreenLint's warnings are actually correct or useful in realistic code. This allowed us to estimate how reliably the implemented rules transfer from the curated benchmark suite to real-world repositories. In addition, for the curated benchmark suite, we verified whether each `bad.py` variant triggered the intended rule and whether each corresponding `good.py` variant remained free of warnings. We included this check because it provides a direct way to confirm that each implemented rule behaves as intended on controlled examples before moving to more realistic code.

### C. RQ1: Energy Impact

To answer RQ1, we compared the inefficient and improved variants in the curated benchmark suite. More specifically, for each of the seven rule categories, we measured the median runtime of the `bad.py` and `good.py` programs over 30 repetitions. Next, we converted the observed runtime differences into proxy estimates for joules and $CO_2$ emissions using the fixed conversion factors defined in the benchmark runner. The full benchmark results are reported in Appendix A, Table IV.

Figure 1 shows the median runtime difference between the inefficient and improved variants for each rule. Generally, all seven inefficient variants exhibited higher median runtime than their corresponding improved versions. Therefore, all seven patterns were also associated with higher runtime-based proxy

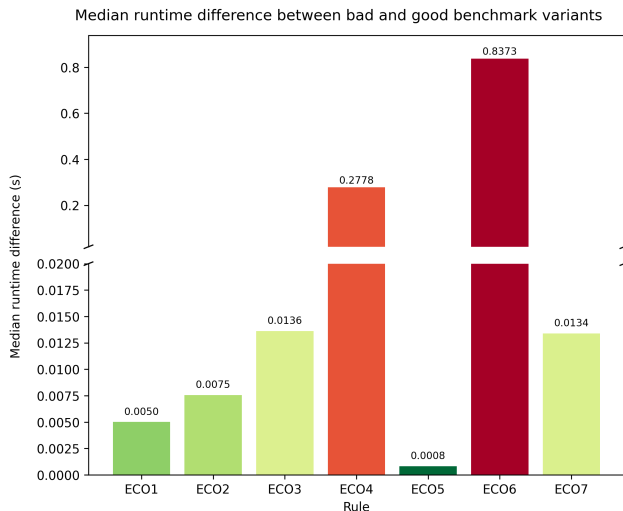


Fig. 1. Median runtime difference between the inefficient and improved benchmark variants for each GreenLint rule.

energy consumption. However, the magnitude of this effect varies substantially across rules.

In particular, ECO6 shows by far the largest separation, followed by ECO4. In contrast, ECO5 shows only a very small difference. The same ordering is reflected in the proxy joule and proxy $CO_2$ estimates reported in Appendix A, Table IV. Thus, while all patterns are directionally consistent, they are clearly not equally costly. These results indicate that the targeted code patterns are consistently associated with higher runtime-based proxy energy consumption in the curated benchmark suite. At the same time, the observed effect sizes differ markedly. As a result, the findings support GreenLint's decision to assign different rule weights rather than treating all warnings as equally severe.

### D. RQ2: Energy $\neq$ Performance?

To answer RQ2, we examined whether the benchmark results separate runtime from greener behavior. We compared the ordering of the inefficient and improved variants under both runtime and proxy-energy estimates. As reported in the full benchmark results in the Appendix A, Table IV, all seven inefficient variants showed both higher median runtime and higher proxy-energy values than their corresponding improved variants.

At first sight, this suggests that faster code is always greener. However, this interpretation would be too strong. In our current evaluation setup, proxy energy is derived directly from runtime using fixed conversion factors. Therefore, once an inefficient variant takes longer to execute, it also receives a higher proxy-energy estimate by construction. In other words, runtime and proxy energy are fully aligned in our benchmark model.

On the other hand, this alignment should not be generalized beyond the current setup. In real software systems, the relationship between performance and energy can be more complex. For instance, memory access patterns, I/O behavior, hardware characteristics and CPU frequency scaling may all influence energy use in ways that are not captured by runtime alone. RQ2 can be answered in two ways. Within our benchmark setting, faster code is always greener under the adopted proxy model. However, this result reflects the design of the evaluation method rather than a universal property of software systems.

TABLE I
REPOSITORY-BASED PREVALENCE SCAN (RQ3)

| Repository | Files | Warnings | Warn./File |
|---|---|---|---|
| Flask | 24 | 6 | 0.25 |
| pandas | 1415 | 1581 | 1.12 |
| Click | 17 | 0 | 0.00 |
| Requests | 18 | 19 | 1.06 |
| Pydantic | 105 | 6 | 0.06 |
| platformdirs | 8 | 0 | 0.00 |
| itsdangerous | 8 | 0 | 0.00 |
| blinker | 3 | 0 | 0.00 |
| markupsafe | 2 | 0 | 0.00 |
| pyjwt | 12 | 0 | 0.00 |

### E. RQ3: Prevalence

To answer RQ3, we conducted an exploratory scan of ten open-source Python repositories drawn from different usage domains, including web frameworks, data science, CLI/developer tooling, utility libraries and application infrastructure. For each repository, we ran GreenLint on the main Python source directory and recorded the number of detections per rule.

Table I summarizes the repository-based prevalence scan using repository size and normalized warning density. We report warnings per file in order to reduce the influence of repository size on the comparison.

Across the ten repositories, GreenLint detected 1612 warnings in total. In raw counts, pandas dominated the results, accounting for most detected warnings. However, this is partly explained by the fact that the scanned pandas source tree was substantially larger than the other repositories. When normalized by the number of Python files, pandas still showed the highest warning density (1.12 warnings per file), but Requests also exhibited a similarly high density (1.06 warnings per file), whereas Flask and Pydantic were much lower.

The rule distribution was also uneven. The most frequently occurring rules were ECO7 and ECO5, whereas ECO3 did not appear at all and ECO2 appeared only once. In the repositories with non-zero detections, Flask and Pydantic were dominated by ECO4 warnings, Requests was dominated by ECO5 warnings and pandas was dominated by ECO7 and ECO5.

At the same time, several smaller repositories produced no warnings at all, including Click, platformdirs, itsdangerous, blinker, markupsafe and pyjwt. This is useful negative evidence: it suggests that GreenLint is not simply flagging every repository indiscriminately, but instead reacts selectively depending on repository structure and coding patterns.

Overall, these findings suggest that some of GreenLint's targeted anti-patterns do occur in real-world Python code, but that their observed prevalence depends strongly on both repository type and repository size. Therefore, the repository-based results should be interpreted as an exploratory prevalence signal rather than a representative estimate for Python repositories in general.

### F. RQ4: Detection accuracy

To answer RQ4, we assessed how accurately GreenLint detects the targeted patterns. We approached this question in two steps. First, we verified detection correctness on the curated benchmark suite. Second, we manually inspected a sample of warnings from the open-source repository scan in order to estimate precision on realistic code.

For the benchmark suite, the results were fully consistent with the intended rule design. As shown in Table II, each `bad.py` variant triggered exactly one warning corresponding to its target rule, whereas each corresponding `good.py` variant produced no warnings. In addition, GreenLint assigns a Green score to summarize the relative severity of the detected patterns. Higher scores indicate greener code, whereas lower scores indicate that the analyzed file contains patterns associated with greater estimated inefficiency. Therefore, the `good.py` variants all received the maximum score of 10, while the `bad.py` variants received lower scores, with ECO6 receiving the strongest penalty.

TABLE II
GREENLINT OUTPUT ON THE CURATED BENCHMARK SUITE

| Rule | Bad warnings | Bad score | Good warnings | Good score |
|---|---|---|---|---|
| ECO1 | 1 | 9 | 0 | 10 |
| ECO2 | 1 | 9 | 0 | 10 |
| ECO3 | 1 | 9 | 0 | 10 |
| ECO4 | 1 | 9 | 0 | 10 |
| ECO5 | 1 | 9 | 0 | 10 |
| ECO6 | 1 | 8 | 0 | 10 |
| ECO7 | 1 | 9 | 0 | 10 |

For the repository scan, we manually inspected a sample of distinct warning locations and labeled each one as a true positive, false positive or borderline case. We focused on distinct warning locations rather than every repeated occurrence, since some rules, especially ECO5, produced duplicate warnings on the same file and line. We also inspected warnings from both smaller repositories and selected pandas files, so that the precision analysis would not be dominated only by the largest repository.

On the manually inspected repository sample, GreenLint produced several context-dependent detections and multiple clear false positives. The most problematic rules were ECO5 and ECO7. In particular, ECO5 flagged package-level and compatibility-related imports that were not ordinary unused imports, while ECO7 matched a non-pandas method named `apply`, showing that the rule currently over-approximates pandas-specific usage. By contrast, ECO1 and ECO4 warnings were usually syntactically plausible, but whether they were worth fixing depended strongly on context, so these cases were labeled as borderline rather than clearly correct or incorrect. Finally, ECO6 matched a real iterrows() use, but the surrounding code performs stateful dictionary construction and row-dependent key assignment, so the suggested vectorized replacement is not obviously appropriate. As shown by the manually inspected sample, we did not identify any clear true positives. In all inspected cases, the warning was either misleading in context or too dependent on surrounding implementation details to be labeled definitively as a true positive.

TABLE III
PRECISION RESULTS ON SAMPLED REPOSITORY WARNINGS

| Rule | TP | FP | Borderline | Total |
|---|---|---|---|---|
| ECO1 | 0 | 0 | 3 | 3 |
| ECO2 | 0 | 0 | 0 | 0 |
| ECO3 | 0 | 0 | 0 | 0 |
| ECO4 | 0 | 0 | 12 | 12 |
| ECO5 | 0 | 7 | 0 | 7 |
| ECO6 | 0 | 0 | 1 | 1 |
| ECO7 | 0 | 1 | 0 | 1 |
| Total | 0 | 8 | 16 | 24 |

## VI. DISCUSSION

The evaluation results show a clear contrast between GreenLint's behavior on curated benchmarks and on realistic repository code. On the one hand, the results of RQ1 support the main motivation of the tool. All seven implemented patterns were associated with higher runtime-based proxy energy consumption in the controlled benchmark setting. Moreover, as shown in Figure 1, the effect sizes were not uniform. In particular, ECO6 and ECO4 showed much stronger separation than the other rules, whereas ECO5 showed only a very small difference.

At the same time, the results of RQ2 clarify how these benchmark findings should be interpreted. In the current setup, runtime and proxy energy are fully aligned because the proxy-energy estimates are derived directly from runtime. As a result, the benchmark comparison provides evidence that the inefficient variants are consistently worse under the adopted model, but it does not show that runtime and real-world energy always behave in the same way. This distinction is important. It means that GreenLint's benchmark results should be read primarily as controlled evidence that the selected patterns are plausibly energy-relevant, rather than as precise measurements of real electricity consumption.

The results of RQ3 and RQ4 add a second perspective. The repository scan shows that some of the targeted patterns do appear in real-world Python projects. As reported in Table I, the warnings were not distributed evenly across repositories. In particular, pandas and Requests exhibited much higher warning densities than the other scanned projects, whereas several smaller repositories produced no warnings at all. This is useful because it shows that GreenLint is not simply flagging every repository indiscriminately. Instead, it appears to react selectively depending on project structure, coding style and library usage. In other words, the repository-based results suggest that the targeted patterns are relevant beyond the

synthetic benchmark suite, but that their prevalence depends strongly on context.

However, prevalence alone is not enough. The results of RQ4 show that detection quality is much less consistent on realistic code than on curated examples. On the benchmark suite, GreenLint behaved exactly as intended. Each `bad.py` variant triggered one warning, each corresponding `good.py` variant remained clean and the Green score consistently distinguished the inefficient and improved variants, as shown in Table II. This is strong evidence that the implemented rules work correctly in isolated settings. Yet the manually inspected repository sample gives a more cautious picture. As reported in Table III, many warnings were borderline and some were clear false positives. In particular, ECO5 and ECO7 were the weakest rules in realistic code. This suggests that syntactic matching alone is often not sufficient to determine whether a warning is truly meaningful in practice.

These findings suggest that GreenLint currently works best as a lightweight awareness tool. Its main strength lies in showing that sustainability-oriented feedback can be integrated into ordinary development workflows through static analysis. This is already valuable from a sustainable software engineering perspective, because developers rarely receive any feedback at all about how implementation choices may affect resource usage. At the same time, the evaluation shows that moving from controlled benchmark evidence to reliable repository-scale feedback is difficult. Therefore, the project highlights both the promise and the current challenge of bringing sustainability-oriented analysis into everyday software development.

More broadly, the results suggest that sustainability feedback at code level should be interpreted as guidance rather than as ground truth. GreenLint is most convincing when it identifies patterns that are both technically recognizable and strongly supported by the benchmark results, such as ECO6 and ECO4. By contrast, rules that depend heavily on project context are harder to translate into reliable recommendations. This contrast helps explain why the tool performs strongly on the curated benchmarks but more unevenly on realistic repositories. Consequently, the main contribution of GreenLint is not that it already offers precise sustainability assessment, but that it demonstrates a practical and reproducible way to surface energy-relevant coding patterns during development.

## VII. Limitations and Future Work

Our study has several limitations. Many of them follow directly from deliberate design and implementation choices in GreenLint rather than from abstract concerns alone, which also clarifies what would need to change in the code or experiments to address them.

*a) Proxy energy model (implementation mechanism).:* In our benchmark pipeline (`experiments/codecarbon_utils.py`, `experiments/benchmark_runner.py`), we record wall-clock time for each run and, in the default configuration, map duration to Joules and $CO_2$ using fixed linear factors (`GREENLINT_PROXY_J_PER_S`, `GREENLINT_PROXY_CO2_KG_PER_S`). As a result, the values reported by GreenLint are not direct hardware measurements but *runtime-derived proxies*. This choice improves reproducibility and keeps the workflow lightweight, but it ignores factors such as memory access, I/O, dynamic frequency scaling, and device-specific power behavior. Although the pipeline can optionally use CodeCarbon, it may still fall back to duration-based estimation when measurements are unavailable. Therefore, in our study, joule and $CO_2$ values should be interpreted as *relative indicators* within controlled benchmarks rather than precise real-world measurements.

*b) Single micro-benchmark pair per rule (metadata shape).:* In our current design, per-rule savings and weights are derived from a single curated `bad.py`/`good.py` pair per rule, aggregated into `experiments/results/summary.json`. During analysis, GreenLint loads this file and attaches the corresponding median $\Delta$ values to each warning. This means that the reported "estimated saving" and rule weight reflect one specific benchmark setup, input size, and environment. Consequently, in our study, these values provide useful relative comparisons but may not generalize to different workloads or execution contexts without extending the benchmark suite.

*c) Syntactic rules without whole-program context (`greenlint/rules.py`).:* In our implementation, rules operate by traversing a single file's AST using syntactic pattern matching. We do not perform type inference, import graph construction, interprocedural analysis, or control-flow reasoning. This design keeps GreenLint lightweight and fast, but it also limits its ability to interpret context and explains the borderline cases and false positives observed in the repository evaluation.

For example, ECO5 checks whether a top-level import appears elsewhere in the same file's AST. It does not account for re-exports, framework conventions, or imports required elsewhere in the package. Similarly, ECO6 and ECO7 flag calls to `.iterrows()` and `.apply()` based only on attribute names, without verifying that the object is a pandas DataFrame. In our evaluation, such limitations led to warnings that were syntactically correct but not always meaningful in context.

*d) File-local analysis (`greenlint/cli.py`, `greenlint/analyzer.py`).:* Our analysis operates on a per-file basis: the CLI iterates over files and analyzes each one independently. We do not construct a project-wide representation of symbols or dependencies. As a result, cross-file relationships cannot be captured, which further contributes to false positives, especially for imports and usage patterns that only make sense at the package level.

*e) Output and usability.:* In the current prototype, each detected AST match produces a separate diagnostic, and the CLI outputs all warnings without deduplication or grouping. In larger files, this can lead to repeated or overlapping warnings and may affect the readability of the output as well as the

Green Score. This limitation is mainly related to usability rather than detection logic.

*f) Repository evaluation and tests.:* Our repository-based evaluation is exploratory and based on a limited number of projects and a manually inspected sample of warnings. Therefore, it does not support general claims about prevalence across the Python ecosystem. Similarly, our test suite focuses primarily on curated benchmark examples, ensuring that each `bad.py` triggers the intended rule and each `good.py` remains clean. While this provides strong validation in controlled settings, it does not yet establish large-scale precision and recall on real-world code.

### *Future work*

These limitations suggest several directions for future work:

- **Context-aware rules.** Extending our rules with lightweight semantic information, such as type hints or import resolution, could reduce false positives. For example, we could restrict ECO7 to confirmed pandas objects or refine ECO5 to account for package-level usage.
- **Richer benchmarks.** Expanding the benchmark suite to include multiple variants per rule, different input sizes, and runs on multiple machines would make the estimated savings more robust and less dependent on a single micro-benchmark.
- **Validation of proxies.** Comparing our runtime-based estimates with hardware-supported measurements in controlled environments would help quantify when the proxy model is accurate and where it diverges from real energy consumption.
- **Product improvements.** Adding IDE integration, autofix suggestions, rule configuration, and output deduplication would improve usability and make GreenLint more suitable for everyday development workflows. A larger evaluation with more repositories and annotators would also strengthen empirical validation.

Overall, these limitations highlight that GreenLint currently operates best as a lightweight awareness tool rather than a precise energy-analysis system. While our implementation demonstrates that sustainability-oriented static feedback for Python is feasible, the gap between controlled benchmark evidence and real-world code remains significant. Addressing this gap requires improving contextual understanding, expanding the empirical basis of the benchmarks, and validating proxy estimates against more direct measurements.

## VIII. CONCLUSION

This paper presented GREENLINT, a lightweight Python command-line linter that detects seven energy-relevant coding patterns using static AST analysis and links them to benchmark-informed feedback. The motivation behind GreenLint is practical: Python developers often lack localized sustainability-oriented guidance in their normal workflow, even though implementation choices can influence runtime behavior and, by extension, likely resource use.

Our evaluation produced a mixed but useful picture. In the curated benchmark suite, all seven targeted patterns were consistently associated with higher runtime and therefore higher runtime-derived proxy energy estimates. This supports the idea that the selected patterns are meaningfully related to inefficient execution and justifies using different rule weights rather than treating all warnings equally. The benchmark-based evaluation also confirmed that the implemented rules behave as intended in controlled settings: each `bad.py` benchmark triggered the expected warning, while each corresponding `good.py` benchmark remained clean.

The repository-based evaluation showed that these patterns do appear in real-world Python code, but very unevenly. Some repositories, especially pandas and Requests, produced relatively high warning densities, while several smaller repositories produced no warnings at all. This suggests that GreenLint is selective rather than indiscriminate, but also that prevalence depends strongly on repository size, domain, and coding style. At the same time, the manual inspection of repository warnings showed that detection quality is less reliable in realistic code than in curated examples. In particular, broader syntactic rules such as ECO5 and ECO7 were more prone to false positives, while other rules were often plausible but too context-dependent to classify as clear true positives.

Taken together, these findings suggest that GreenLint is currently best understood as a lightweight awareness and guidance tool rather than a precise energy-analysis system. Its main contribution is to show that sustainability-oriented static feedback for Python can be integrated into ordinary developer workflows in a practical and reproducible way. At the same time, the results make clear that moving from benchmark-grounded pattern detection to robust project-scale recommendations remains difficult. Future work should therefore focus on improving contextual understanding, expanding the benchmark basis, and validating runtime-based proxy estimates against more direct measurements.

Finally, a key takeaway is that recommended coding patterns consistently reduce runtime and are therefore likely to reduce energy usage in practice, while exact absolute savings would require controlled hardware and dedicated instrumentation.

## REFERENCES


[1] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, “The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations,” *Patterns*, vol. 2, no. 9, p. 100340, 2021.

[2] E. Kern, M. Dick, S. Naumann, and T. Hiller, “Sustainable software products—towards assessment criteria for resource and energy efficiency,” *Future Generation Computer Systems*, vol. 86, pp. 199–210, 2018.

[3] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, “Energy efficiency across programming languages: how do energy, time, and memory relate?” in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 2017, pp. 256–267.

[4] ——, “Ranking programming languages by energy efficiency,” *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[5] O. Le Goaer and J. Hertout, “ecoCode: a SonarQube plugin to remove energy smells from Android projects,” in *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–4.

[6] G. Pinto and F. Castor, “Energy efficiency: a new concern for application software developers,” *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, 2017.
[7] F. Nahrstedt, M. Karmouche, K. Bargieł, P. Banijamali, A. Nalini Pradeep Kumar, and I. Malavolta, “An empirical study on the energy usage and performance of pandas and polars data analysis python libraries,” in *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, 2024, pp. 58–68.
[8] L. Cruz and R. Abreu, “Performance-based guidelines for energy efficient mobile applications,” in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, 2017, pp. 46–57.
[9] ——, “Catalog of energy patterns for mobile applications,” *Empirical software engineering*, vol. 24, no. 4, pp. 2209–2235, 2019.
[10] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, “Energy profiles of Java collections classes,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 225–236, aCM SIGSOFT Distinguished Paper Award.
[11] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto, “Improving energy-efficiency by recommending Java collections,” *Empirical Software Engineering*, vol. 26, no. 3, p. 55, 2021.
[12] R. P. Gurung, J. Porras, and J. Koistinaho, “Static code analysis for reducing energy code smells in different loop types: A case study in Java,” in *2024 10th International Conference on ICT for Sustainability (ICT4S)*, Stockholm, Sweden, 2024, pp. 292–302.
[13] V. Schmidt, S. Goyal-Kamal, A. Joshi, B. Feld, S. Luccioni *et al.*, “CodeCarbon: Track and reduce CO2 emissions from compute,” https://github.com/mlco2/codecarbon, 2024.
[14] L. Lannelongue, J. Grealey, and M. Inouye, “Green algorithms: quantifying the carbon footprint of computation,” *Advanced Science*, vol. 8, no. 12, p. 2100707, 2021.
[15] R. Chattaraj, S. Chimalakonda, V. S. Sharma, and V. Kaulgud, “Who wins the race? (r vs python) - an exploratory study on energy consumption of machine learning algorithms,” 08 2025.
[16] Pylint Contributors, “Pylint: Static code analyser for Python,” https://pylint.readthedocs.io/, 2024.
[17] I. S. Cordasco, “Flake8: Your tool for style guide enforcement,” https://flake8.pycqa.org/, 2024.
[18] C. Sahin, F. Cayci, I. Gutiérrez, J. Clause, L. Pollock, and K. Winbladh, “How does code obfuscation impact energy usage?” in *IEEE International Conference on Software Maintenance and Evolution*, 2014.
[19] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Aßmann, “Comparing mobile applications energy consumption,” in *IEEE International Conference on Mobile Software Engineering and Systems*, 2013.
[20] Pallets, “pallets/flask,” https://github.com/pallets/flask, 2026, gitHub repository, accessed April 2026.
[21] pandas-dev, “pandas-dev/pandas,” https://github.com/pandas-dev/pandas, 2026, gitHub repository, accessed April 2026.
[22] Pallets, “pallets/click,” https://github.com/pallets/click, 2026, gitHub repository, accessed April 2026.
[23] Python Software Foundation, “psf/requests,” https://github.com/psf/requests, 2026, gitHub repository, accessed April 2026.
[24] Pydantic Contributors, “pydantic/pydantic,” https://github.com/pydantic/pydantic, 2026, gitHub repository, accessed April 2026.
[25] tox-dev, “tox-dev/platformdirs,” https://github.com/tox-dev/platformdirs, 2026, gitHub repository, accessed April 2026.
[26] Pallets, “pallets/itsdangerous,” https://github.com/pallets/itsdangerous, 2026, gitHub repository, accessed April 2026.
[27] Pallets-Eco, “pallets-eco/blinker,” https://github.com/pallets-eco/blinker, 2026, gitHub repository, accessed April 2026.
[28] Pallets, “pallets/markupsafe,” https://github.com/pallets/markupsafe, 2026, gitHub repository, accessed April 2026.
[29] PyJWT Contributors, “jpadilla/pyjwt,” https://github.com/jpadilla/pyjwt, 2026, gitHub repository, accessed April 2026.

## APPENDIX A
## FULL BENCHMARK RESULTS

Table IV shows the complete benchmark results for all seven GreenLint rules, including median runtime values, runtime differences, proxy joule estimates, proxy $CO_2$ estimates and rule weights.

TABLE IV
FULL BENCHMARK RESULTS FOR ECO1–ECO7

| Rule | Median bad runtime (s) | Median good runtime (s) | Median delta (s) | Delta joules | Delta $CO_2$ (g) | Weight |
|---|---|---|---|---|---|---|
| ECO1 | 0.03174 | 0.02673 | 0.00501 | 0.1252 | 0.0000025 | 0.260 |
| ECO2 | 0.01603 | 0.00848 | 0.00755 | 0.1887 | 0.0000038 | 0.266 |
| ECO3 | 0.01681 | 0.00320 | 0.01361 | 0.3403 | 0.0000068 | 0.278 |
| ECO4 | 0.27918 | 0.00138 | 0.27780 | 6.9450 | 0.0001389 | 0.831 |
| ECO5 | 0.06830 | 0.06749 | 0.00081 | 0.0203 | 0.0000004 | 0.252 |
| ECO6 | 0.83759 | 0.00025 | 0.83734 | 20.9335 | 0.0004187 | 2.000 |
| ECO7 | 0.01373 | 0.00034 | 0.01339 | 0.3348 | 0.0000067 | 0.278 |