Github Actions to report the energy consumption of CI/CD

Kevin Ji Shan Emre Cebi Cosmin Anton Mohammed Nassiri Alexandru

***Abstract*—Continuous Integration and Continuous Deployment (CI/CD) pipelines are central to modern software development, enabling frequent and automated building, testing, and deployment of applications. While these practices improve productivity and reliability, their energy consumption is non-negligible due to repeated execution of compute-intensive tasks. However, this energy usage is typically not visible to developers and is rarely considered in engineering decisions. This paper presents a lightweight approach for measuring and reporting energy consumption within CI/CD pipelines using GitHub Actions. Our solution integrates into existing workflows and estimates energy usage using Code Carbon, providing fine-grained insights at the level of individual pipeline stages. By exposing this information, developers can better understand the environmental impact of their processes, compare energy consumption of different versions and identify opportunities for optimization. We evaluate our approach by integrating the tool with well-known repositories such as Flask. The results show that observable energy usage requires minimal effort and slight performance overhead , while supporting better informed and more sustainable software engineering practices.**

I Introduction

Continuous Integration / Continuous Deployment (CI/CD) is a fundamental part of modern software engineering, allowing the integration of code changes from multiple contributors to be automated. As organizations increasingly rely on cloud-based infrastructure to execute these pipelines, the scale and frequency of automated builds and deployments continue to grow.

Despite these benefits, the energy consumption associated with CI/CD processes remains largely invisible to developers. Pipeline executions involve computationally intensive tasks such as compilation, testing, and containerization, all of which contribute to the overall resource utilization. However, unlike metrics such as execution time or code quality, energy usage is rarely exposed or considered during development. The research question becomes: How can we contribute to sustainable software by making the energy consumption of CI/CD observable?

In this paper, we present an approach to measure and report energy consumption of CI/CD pipelines using Github Actions. Our solution can be directly integrated into existing workflows and provide insights into the energy usage of individual pipeline stages. By making energy consumption observable, we aim to support more informed and sustainable engineering choices. The paper will be presented in the following order: the background and motivation of the problem, followed by the design and implementation, results, conclusion and discussion.

II Background

Continuous Integration and Continuous Deployment are widely used practices in modern software development. The main idea is simple: developers push code frequently, and an automated system builds, tests, and sometimes deploys it without manual intervention. Tools like GitHub Actions and GitLab CI/CD make this very easy to set up. You define workflows, and every time something changes in the repository, a pipeline runs automatically. This is great for productivity and for catching bugs early. However, there is a hidden cost: these pipelines run very often. Every push, every pull request, and every retry triggers compute-heavy tasks such as builds, tests, and sometimes container creation.

A. Energy Consumption in Software

When running code, energy consumption is usually not considered. However, every operation uses hardware resources such as CPU, memory, disk, and network, all of which consume power. In CI/CD pipelines, this usage accumulates fast. Even a simple pipeline might install dependencies, compile code, run tests, and build artifacts. In active projects, these processes can run many times per hour. The problem is that developers do not see this energy usage. They typically monitor execution time, logs, and pass/fail results, but not energy. As a result, energy is not considered during optimization. As CI/CD pipelines scale across large projects and organizations, their cumulative energy consumption becomes significant, contributing to the overall environmental impact of software systems.

B. Related Work

Some tools attempt to estimate energy usage. One widely used example is CodeCarbon, which tracks resource usage during execution and estimates both energy consumption and carbon emissions. CodeCarbon is mainly used in machine learning experiments, where workloads can be very resource-intensive. Its main advantage is that it is lightweight and does not require specialized hardware.

Other approaches rely on direct measurements using physical sensors. Although these methods can provide accurate results, they are difficult to implement in cloud environments such as CI/CD pipelines.

Some platforms also offer general monitoring dashboards. For example, GitHub Actions provides usage metrics such as execution time and total runtime minutes, while GitLab CI/CD includes pipeline analytics such as duration and success

rates. Cloud providers such as AWS, Google Cloud, and Microsoft Azure also offer dashboards that estimate resource usage or carbon impact at the infrastructure level. However, these metrics are often too high-level. They are not tied to specific pipeline steps, such as individual jobs or tasks, and are not directly integrated into the developer workflow. As a result, developers cannot easily identify which parts of a CI/CD pipeline are responsible for higher energy consumption or optimize them accordingly.

C. Limitations of Current Approaches

Despite existing work, several limitations remain:

- Lack of integration: most tools are not designed to work directly within CI/CD pipelines
- Limited details: it is difficult to identify which pipeline steps consume the most energy
- Hardware dependency: accurate methods often require physical sensors, which might not be available in cloud environments
- Low visibility: energy consumption is not presented alongside standard CI/CD metrics

Due to these limitations, energy usage is rarely considered during development.

These limitations highlight the need for a solution that integrates energy awareness directly into developer workflows.

D. Motivation

Our work is motivated by the idea that making energy consumption visible can influence developer behavior. Our project tries to make energy usage visible, easy to measure and integrate into any existing CI/CD workflows. By doing so, the developers will have more insight on how much energy each pipeline or individual step consumes. This allows them to compare different versions of their workflows, identify inefficient steps, and make more informed decisions. Although our implementation focuses on GitHub Actions, the same approach might be extended to other platforms such as GitLab in the future. Overall, the goal is not to complicate CI/CD pipelines but to add an important missing metric that can support more sustainable software development.

III Design and Implementation

This tool is designed to monitor the energy consumption and environmental impact of CI/CD pipelines. This section outlines the key functionality, the design choices and the validation procedure.

A. Architecture Overview

The primary design principle was to make integrating this tool as easy as possible with existing CI/CD pipelines. To achieve this, our tool is packaged as a GitHub Action, since GitHub Actions workflows can be added directly to any existing GitHub repositories with minimal setup. Each workflow operates through two composite steps: *start* and *stop* that can be layered onto existing YAML configurations alongside standard build and test jobs. An overview of the approach is presented in Figure 1. To minimize external dependencies and ensure data privacy, all configurations are handled internally, and output metrics are stored using GitHub artifacts and repository files.

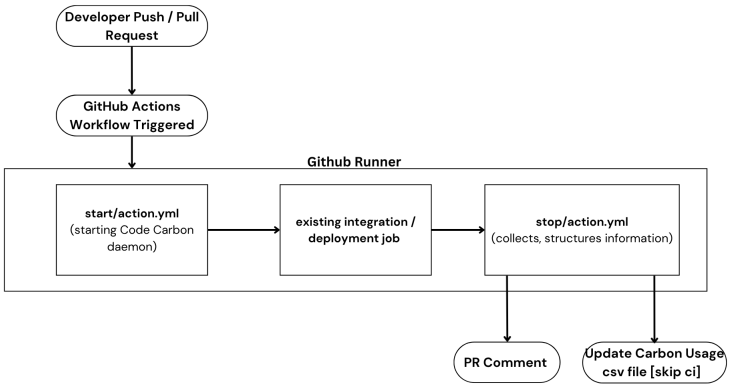

Fig. 1. Overview of the architecture

B. Energy Measurement

The core measurement engine is powered by Python scripts utilizing *CodeCarbon* [1] to estimate power consumption and carbon emissions. By default, CodeCarbon estimates carbon emission based on static grid averages. As the system was designed with extensibility in mind, it can utilize the Electricity Maps API [2] to obtain real-time grid carbon intensity. Making this API optional was a deliberate design choice: ensuring simple integration was preferred over slight accuracy boosts. While CodeCarbon provides universal robust baseline tracking, custom logic had to be implemented to adapt its power tracking capabilities specifically for ARM macOS GitHub runners. Furthermore, the measuring accuracy can be improved by passing specific parameters to CodeCarbon. An example includes the Power Usage Effectiveness (PUE) parameter, which is set to 1.18 to simulate a common datacenter like Azure [3]. More details about the possible configurations can be found in Subsection D.

C. Pipeline Implementation

The tool consists of two modular steps that are added to the existing pipelines:

- The Start Step: Initializes the CodeCarbon tracking instance in the background based on the JSON configuration and environment variables, safeguarding any secrets like API keys.
- Targeted Workflows: Integration workflow, Image Building workflow or any other workflow.
- The Stop and Aggregation Step: The measured data is collected and structured into readable text.

To ensure measurement accuracy across concurrently running jobs, the stop logic also includes a safety check. If multiple workflows are running, the script waits; only the workflow with the lowest PID is allowed to terminate the

tracking first. The system then uses the generated artifacts from previous instances and aggregates the data for the report. Additionally, the new history data are preserved by committing a carbon-history.csv file back to the repository using a bot account with a [skip ci] flag to prevent infinite runner loops.

D. Additional Configuration

In order to customize the behavior of the tool and finetune the accuracy of the measurements, we introduced an additional configuration file at the root of the repository. This JSON file acts as a single source of truth for the product and contains three blocks: *thresholds*, *notifications* and *CodeCarbon*, also shown in Figure 2.

The first block *thresholds* allows you to configure a custom threshold to trigger notifications. The metrics we have defined include energy consumption (wh) , CO2 emission (g) and runtime (s). The user can use one or more of these values to set a customized limit for their CI pipelines, avoiding notifications when not necessary.

The second block *notifications* defines which notifications to send when one of the thresholds is reached. The user can toggle to receive notifications in the form of a PR comment, GitHub issue and customize the corresponding label. The report will always be generated at the specific run as a comment despite the settings in this section.

The final block *CodeCarbon* exposes specific energy measurement parameters: *Power Usage Effectiveness* and *force cpu power*. The PUE indicates how efficiently a computer data center uses energy. *Force cpu power* overrides the maximum power draw (in watts) of the CPU; this manual adjustment is only needed when the CI pipeline operates on custom, self-hosted runners. For the default GitHub provided ones this is not necessary.

```
{
  "thresholds": {
    "energy_wh": 0.15,
    "co2_g": null,
    "duration_s": null
  },
  "notifications": {
    "create_issue": true,
    "issue_label": "carbon-alert",
    "pr_comment": true
  },
  "codecarbon": {
    "pue": 1.18,
    "force_cpu_power": null
  }
}
```

Fig. 2. Additional Configuration

E. Reporting the result

After the stop/action.yml finishes, the raw consumption data is parsed and formatted into a comprehensive markdown report. The report is divided into five levels of granularity:

- run-specific: How much energy the current run has consumed.
- workflow specific: How much energy this workflow has consumed in total.
- per-workflow breakdown: How much energy each workflow has consumed in total.
- PR-specific breakdown: How much energy this PR has consumed.
- repository total: How much energy this entire repository has consumed.

This report (excluding the PR specific breakdown) is by default available as a comment on each GitHub Action workflow page, on each instance of the run. If the user enables it in the configuration, it will also be posted as a new comment in the corresponding PR, replacing any previous ones. A slightly modified version of this report containing extra info will be posted as an issue when a workflow breaches the threshold, as mentioned before in subsection D. The main reasoning behind choosing GitHub Issues as the main notification backend was due to its simplicity. GitHub Issues is a feature available by default to all GitHub repositories and the user does not need to install any extra dependencies. A visualization of the report can be viewed in Figure 3.

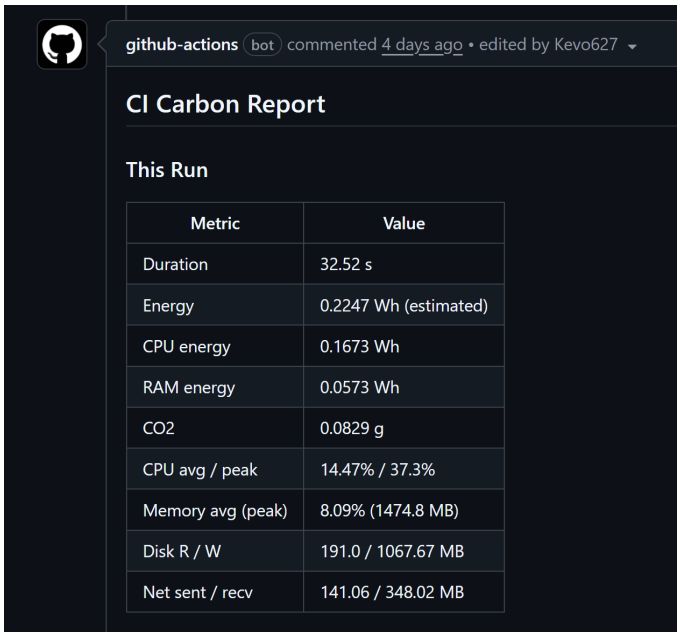

Fig. 3. PR Comment

F. Experiment Design

To validate the system, the Python framework *Flask* [4] was utilized as a testbed. While the repository contains numerous CI pipelines, this experiment focused on two specific workflows: code testing and pre-commit actions. This approach was chosen to compare the tool against the unmodified Flask library in two different CI types: a simpler, routine pipeline,

and a more complex, computationally heavy process. The code testing workflow utilizes a matrix of runners across multiple versions of Python on various operating systems and machine architectures. To accurately measure the energy consumption of each individual runner, we opted to execute the matrix jobs in sequential order instead of concurrently. The optional Electricity Map API is not used. An overview of all jobs of the testing workflow can be found in Table 1.

TABLE I
ALL JOBS FOR THE WORKFLOW TESTS

Architecture / Version
3.14
3.14t
Windows
Mac
3.13
3.12
3.11
3.10
PyPy
Minimum Versions
Development Versions

IV Results

As mentioned previously in Section 3, to validate the performance of our tracking tool, we integrated it into the Python library Flask. We targeted the test suite and pre-commit workflows; the running time of both workflows using CodeCarbon is compared against the original Flask repository, see Figure 4 and 5.

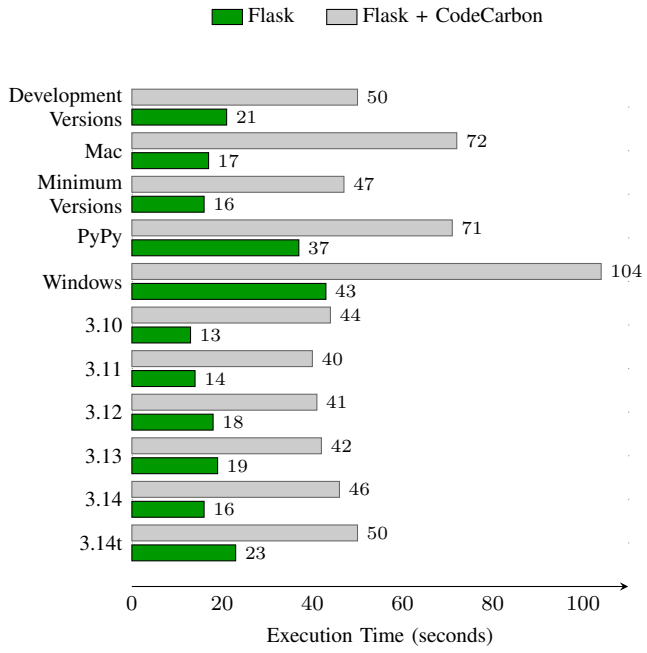

Fig. 4. Performance comparison of the workflow Tests

The collected results indicate that integrating the CodeCarbon introduces a latency of approximately 30 seconds to

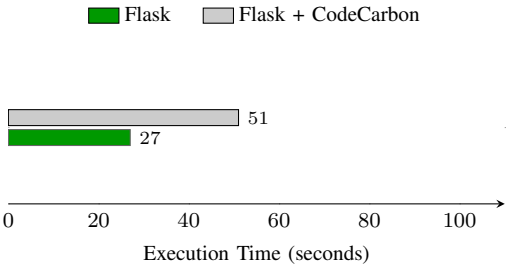

Fig. 5. Performance comparison of the workflow pre-commit (Only one version for all OS)

the execution time of most jobs. This overhead is primarily due to required I/O operations, specifically the downloading and aggregating of GitHub artifacts between jobs, as well as committing the history CSV file. While this represents a significant delay, it is a necessary cost to ensure accurate, aggregated reporting across distributed, concurrent jobs while maintaining the ease of use. In the broader context of standard CI/CD pipelines where builds and comprehensive test suites frequently take several minutes to execute—a 30-second footprint is almost negligible.

V Conclusion

In this work, we presented a lightweight tool to measure and visualize the environmental impact of GitHub CI/CD Pipelines. The ease of use of this tool allows us to integrate it into existing projects with minimal modification to the pipeline configuration. Our evaluation shows that the tool can be added to the CI/CD pipeline of the Flask library, running on different operating systems and hardware architectures with some limited performance cost due to I/O. This highlights the portability and flexibility of our approach in real world development environments. By making energy consumption visible at the level of individual workflows, our tool allows developers to better understand the environmental impact of their code and identify opportunities for further optimization. This contributes to a shifting trend in software engineering practices, where energy efficiency will be considered alongside metrics such as performance and cost.

Overall, this work demonstrates that integrating energy awareness into CI/CD pipelines is both feasible and valuable, representing a step toward more sustainable software engineering practices.

VI Limitations and Future Work

This project has some limitations; the reported energy values are only estimates and not exact physical measurements. The approach uses CodeCarbon, which estimates energy consumption from resource utilization instead of directly measuring power with hardware sensors. As a result, the reported values should be interpreted as approximations that primarily serve to raise awareness.

In addition, the current implementation focuses only on GitHub Actions. Although this choice can be justified by

its widespread adoption, it also means that the solution has only been tested in one CI/CD environment. Other platforms such as GitLab CI/CD or Jenkins may behave differently. Furthermore, the scope of the evaluation remains limited. While testing the tool on known repositories provides an initial proof of feasibility, it does not fully represent the variety and complexity of real software projects. CI/CD pipelines can vary significantly in size, programming languages, dependencies, and workflow structures. Because of this, further validation is needed before generalizing the approach.

For future work, the approach could be extended to other CI/CD platforms to make it more widely applicable. It would also be beneficial to test the tool on a larger number of repositories and more diverse workflows. Another possible improvement is to increase the accuracy of the measurements by combining software based estimation with more detailed hardware data. Finally, future developments could move beyond reporting by actively assisting developers, for example by identifying inefficient stages and recommending optimizations to reduce unnecessary energy consumption.

References

[1] B. Courty, V. Schmidt, Goyal-Kamal, MarionCoutarel, B. Feld, J. Lecourt, LiamConnell, SabAmine, inimaz, supatomic, M. Léval, L. Blanche, A. Cruveiller, ouminasara, F. Zhao, A. Joshi, A. Bogroff, A. Saboni, H. de Lavoreille, N. Laskaris, E. Abati, D. Blank, Z. Wang, A. Catovic, alencon, Michał Stechły, C. Bauer, Lucas-Otavio, JPW, and MinervaBooks, "mlco2/codecarbon: v2.4.1," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501

[2] "Electricity Maps | The world's most comprehensive electricity data platform." [Online]. Available: https://www.electricitymaps.com/

[3] N. Walsh-Elwell, "How Microsoft measures datacenter water and energy use to improve Azure Cloud sustainability," Apr. 2022. [Online]. Available: https://azure.microsoft.com/en-us/blog/how-microsoft-measures-datacenter-water-and-energy-use-to-improve-azure-cloud-sustainability/

[4] "Welcome to Flask — Flask Documentation (3.1.x)." [Online]. Available: https://flask.palletsprojects.com/en/stable/