GradlEnergi: Raising awareness about local build pipeline energy consumption

Gijs Margadant
Delft University of Technology
Delft, Netherlands
g.e.margadant@student.tudelft.nl

Jamila Seyidova
Delft University of Technology
Delft, Netherlands
j.c.q.seyidova@student.tudelft.nl

Michael Chan
Delft University of Technology
Delft, Netherlands
j.m.chan@student.tudelft.nl

Roberto Negro
Delft University of Technology
Delft, Netherlands
r.negro@student.tudelft.nl

***Abstract*—The growing emphasis on sustainable software engineering has largely overlooked the energy consumption associated with developers' local build processes. This paper introduces GradlEnergy, a cross-platform tool designed to measure and visualize the energy usage of Gradle build pipelines. By combining hardware-level energy measurements with an intuitive graphical user interface, the tool enables developers to design repeatable experiments, analyze energy consumption at the level of individual build tasks, and compare results across systems and configurations. A case study using the JUnit5 project demonstrates the tool's effectiveness and highlights practical challenges such as task redundancy and measurement accuracy. GradlEnergy is publicly available and extensible, contributing to the broader field of Green ICT by addressing critical gaps in energy transparency and developer empowerment.**

I. Introduction

The past decade has witnessed a growing interest in green software engineering. While much of the focus has been on optimizing data centers, mobile devices, and AI model training [1], the day-to-day activities of software developers remain largely overlooked in sustainability discussions [2]. To fully realize energy efficiency across the software life cycle, a more holistic approach is required — one that extends beyond hardware to include software development practices.

Automated build pipelines, a fundamental component of modern software development [3], represent a significant yet often hidden source of energy consumption [4]. In modern development environments, builds are triggered frequently, such as with every commit, pull request, or merge. These processes are typically executed in continuous integration (CI) environments like GitHub Actions, GitLab CI/CD, or Jenkins, often hosted in the cloud. As the frequency of these builds increases, their energy demands grow, yet the environmental impact remains largely obscured from developers and organizations. This abstraction makes it difficult to connect development practices with their real-world energy costs.

The lack of visibility into energy consumption in build processes is a major barrier to improvement. Most developers are unaware of the energy costs associated with their workflows and lack the tools to measure and optimize these impacts. Studies show that a significant portion of developers have little to no understanding of energy consumption in software applications, with only a small fraction taking power consumption into account during development [5]. This gap in knowledge highlights the importance of promoting greater awareness and providing accessible tools to empower developers to make informed decisions on build configuration, frequency, and resource management. This issue is also reflected in the broader Green ICT research field, where "people awareness"—despite being frequently discussed—has not been sufficiently integrated into existing Green ICT frameworks [6]. Our tool aims to bridge this gap by making energy consumption more visible and actionable for developers, fostering a shift toward more sustainable practices in software development.

The high frequency of builds across industries means that even small inefficiencies can translate into considerable environmental costs. Without proper tools to monitor and analyze the energy consumption of build pipelines, developers lack the necessary insights to minimize their carbon footprint. This is further compounded by the fact that even when energy concerns are recognized, programmers are often unsure of how to effectively reduce consumption. Research shows that despite acknowledging energy efficiency as a priority, most developers lack the detailed knowledge or metrics to make meaningful adjustments [5]. This recurring issue in Green ICT research has been identified as a lack of metrics and standards for measuring energy efficiency [6]—a gap that our tool directly addresses by providing developers with a practical way to measure and compare the energy consumption of their builds.

To address this gap, we propose a platform-independent tool that measures and visualizes the energy consumption of Gradle builds. Through a Graphical User Interface (GUI), developers can design and conduct experiments to measure energy usage at various stages of the build process. Additionally, the tool enables visualization and comparison of energy consumption across different configurations and devices, fostering better decision-making and encouraging the adoption of energy-

efficient practices.

By providing insights into the energy cost of builds, our approach aims to raise awareness among developers, enabling them to make more sustainable choices and ultimately contributing to a more energy-efficient software development ecosystem. In doing so, we contribute to the growing field of Green ICT research and help fill the critical gaps in awareness and measurement that hinder the industry's shift toward more sustainable software engineering practices[6].

The remainder of this paper is structured as follows. Section II introduces our tool and outlines its key features. Section III presents a case study that illustrates the practical utility of the tool, while section IV discusses the implications and limitations of the project. We then review the related work in Section V, before concluding in Section VI with a summary of our findings and directions for future work.

II. The Tool

The motivation behind our tool is to raise developer awareness about the energy consumption of build pipelines. To that end, our solution is designed to support structured experimentation, enabling developers to measure energy usage at each stage of the build process.

We identify three key goals: i) Enable developers to design and set up easily repeatable experiments; ii) Provide a way to execute these experiments reliably; iii) Offer intuitive visualizations to compare and analyze results across different systems and build configurations.

In addition, we established several requirements. First, the system must be compatible with a wide range of hardware and operating systems - including macOS, Linux and Windows running on Intel, AMD or Apple Silicon (M1) processors. Second, it must feature a GUI that allows developers to explore and interpret energy usage data. Lastly, while our current implementation focuses on Gradle, the design should be openly accessible and extensible in order to support other build tools in the future.

To realize these goals, we built a tool that combines a framework to setup experiments with an interface for analysis. In the remainder of this section, we first describe the methodology for conducting energy measurements, followed by an overview of the GUI.

A. Experiment Methodology

Accurately measuring of energy consumption is inherently challenging, as computers often run multiple background processes, making it difficult to attribute energy usage to a specific software task. While software-based estimators[1] exist, they typically rely on assumptions about the underlying hardware, limiting their applicability across platforms. Moreover, tracking process scheduling at a fine-grained level introduces overhead — which itself consumes energy.

To ensure portability and reduce measurement bias, our approach relies on hardware-level energy measurements. Although these readings do not isolate the energy usage of the build process alone, they offer a consistent baseline for comparing experiments across configurations and systems. The accuracy of such measurements can be improved by following a set of best practices, which our tool supports and encourages.

a) Minimizing background interference: Reducing background activity is crucial. We provide guidance on how to limit external processes — such as unplugging peripherals, disabling wireless connections, and closing unnecessary applications — to ensure more stable and representative measurements.

b) Repetition and averaging: Experiments should be repeated multiple times (e.g., 30 times) to reduce variance, allow for statistical analysis, and mitigate outliers. Averaging results across runs helps smooth out random fluctuations in background energy usage. To ensure consistency, cached build results are cleared after each run so that all tasks are re-executed from scratch.

c) Tail energy and resting phases: Another challenge is the influence of tail energy: energy consumed after a task finishes but still attributable to it. To prevent tail consumption of one task from influencing the measurements of another, there should be a resting phase (e.g., 60 seconds) in between tasks.

d) Time-based bias and external factors: Environment conditions, such as ambient temperature, may influence energy readings over time. Ideally, tasks across repetitions would be executed in a random order to mitigate this effect. However, due to the stateful nature of build pipelines - with intermediate cashes and dependencies - shuffling is not feasible in most cases. That said, we argue that if external factors significantly impact energy consumption, it is beneficial for developers to become aware of them. To mitigate temperature-related variance, we do provide a warm-up procedure to bring hardware up to temperature before experiments begin.

We recognize that these best practices may limit developer's ability to use their machine during experimentation. To reduce this burden, or tool automates the full experimental process, allowing runs to be scheduled during idle times. Additionally, none of the recommendations above are enforced. Developers remain free to run a single build or experiment under less controlled conditions if desired.

B. Measurement Methodology

Energy measurements are collected using EnergiBridge [7], a cross-platform energy measurement tool that supports macOS, Linux, and Windows across Intel, AMD, and Apple Silicon processors. It is designed to collect energy consumption data on the hardware level and output it to a CSV file. Currently, the only measurement consistently available across platforms is CPU energy consumption, which we use as the basis for our comparisons.

To enable fine-grained energy profiling of individual build tasks, our tool accepts a Gradle command through the GUI. Internally, the command is first executed with the `--dry-run` flag to retrieve the list of tasks that would be triggered. This

[1] https://github.com/marmelab/greenframe-cli

list is then presented to the user, who can selectively enable or disable specific tasks for the experiment.

Once the task selection is finalized, the tool sequentially executes each task in isolation by wrapping it in a separate EnergiBridge measurement command. This approach guarantees that energy consumption is measured independently for each task, reducing interference and ensuring clearer attribution of energy costs to individual build phases.

We provide the user with the option to conduct an experiment measuring idle energy consumption. It serves as a baseline of the energy consumed by the system and can be used to obtain more reliable estimates of the actual consumption of the build process. Using this correction, experiment results can more easily be compared across systems. The system's energy consumption is determined by first computing the systems power consumption during the idle phase and integrating this over the time it took for the task to complete. For example, if the system consumed 10 Watts during the idle phase and the task executed in 2.5 seconds, then 25 Joules of energy ar attributable to the system and will be subtracted from the total amount of energy consumed.

C. Graphical User Interface

Our tool provides a GUI that supports developers in setting up, running, and analyzing experiments. It is implemented in Python using the tkinter library[2], which ensures compatibility with all supported platforms. The GUI is structured into three views:

a) Experiment setup: As shown in Figure 1, this view allows users to configure and launch experiments. All the parameters mentioned in Section II-A can be specified here. Each setting includes a tooltip with recommended best practices, accessible by clicking on the adjacent question mark icons. Additionally, users can connect to EnergiBridge and select the target Gradle project folder from this view.

b) Single experiment analysis: Figure 2 illustrates the analysis page for individual experiments. Here, the energy consumption of each task within the build process is visualized using a pie chart. Users can choose which metric to visualize (e.g., CPU energy or RAM energy), enabling a quick and intuitive understanding of where energy is being consumed during a specific build.

c) Multiple experiment analysis: Figure 3 displays the comparison view for multiple experiments. Energy usage for tasks across different experimental runs is presented in a bar chart. Users can select a subset of experiments to include in the visualization and choose which metric to compare. This functionality is particularly useful for evaluating the impact of configuration changes, hardware differences, or caching strategies on energy consumption.

D. Public Availability and Extensibility

The tool is publicly available on GitHub[3], along with detailed setup instructions in the README file. The repository also includes a dedicated section titled *How to Use the Tool*, which explains how to properly configure and run experiments through the graphical interface. For those interested in contributing or extending the tool, the execution logic of the experiment is located in `logic/experiment_setup.py`, while the user interface logic is split between `GUI/views/settings_view.py` and `GUI/views/statistics/statistics_view.py`.

III. Case Study

To evaluate the effectiveness and reliability of our tool in a real-world setting, we conduct a case study. Unfortunately, many of the repositories that integrated the Gradle build tool could not be build locally on our systems, due to failures in the build process. We took, among others, the repositories mentioned in [4] and checked out the commits they mentioned. For example, the Junit5[4] repository fails because statical analysis tools detect the usage of http addresses where https should be used. As a result, we are unfortunately not able to compare our measurements with the ones they obtained.

Instead, we will verify our measurement methodology of manually splitting up tasks using Gradle's CLI and running them to obtain detailed energy measurements. First, we will discuss our experiment setup and, subsequently, we present the results.

A. Experiment setup

We build the junit-jupiter-engine[5] sub-project from Junit5 in two ways. First, we build it using the high-level `gradle build` command by selecting the :junit-jupiter-engine:build task in the experiment setup screen (Figure 1). Second, we use the Task Tree[6] plugin to extract all the sub-tasks on which the build command depends and select all sub-tasks in the first 3 layers of the tree. Below, the partial task tree is listed, from which :junit-jupiter-engine:build, :junit-jupiter-engine:assemble, and :junit-jupiter-engine:jar are included in the task selection as they belong to the first 3 layers. The sub-tasks lower in the tree are not selected. Using this procedure, we selected 13 sub-tasks in total.

```
:junit-jupiter-engine:build
+--- :junit-jupiter-engine:assemble
|    +--- :junit-jupiter-engine:jar
|    |    +--- :junit-jupiter-api:jar
|    |    |    +--- :junit-jupiter-api:allM...
|    |    |    |    \--- :junit-jupiter-api...
|    |    |    |         +--- :junit-jupite...
|    |    |    |         |    +--- :junit-j...
|    |    |    |         |    |    +--- :ju...
|    |    |    |         |    |    +--- :ju...
|    |    |    |         |    |    \--- :ju...
```

We set the experiments up as follows:

- **Idle energy baseline:** A 60-second idle period is recorded before an experimental run to estimate background energy consumption. This idle energy is later

[2]https://docs.python.org/3/library/tkinter.html

[3]https://github.com/JamilaSeyidova/sse-g22-p2/tree/main

[4]https://github.com/junit-team/junit5

[5]Commit: 96daa92

[6]https://github.com/dorongold/gradle-task-tree

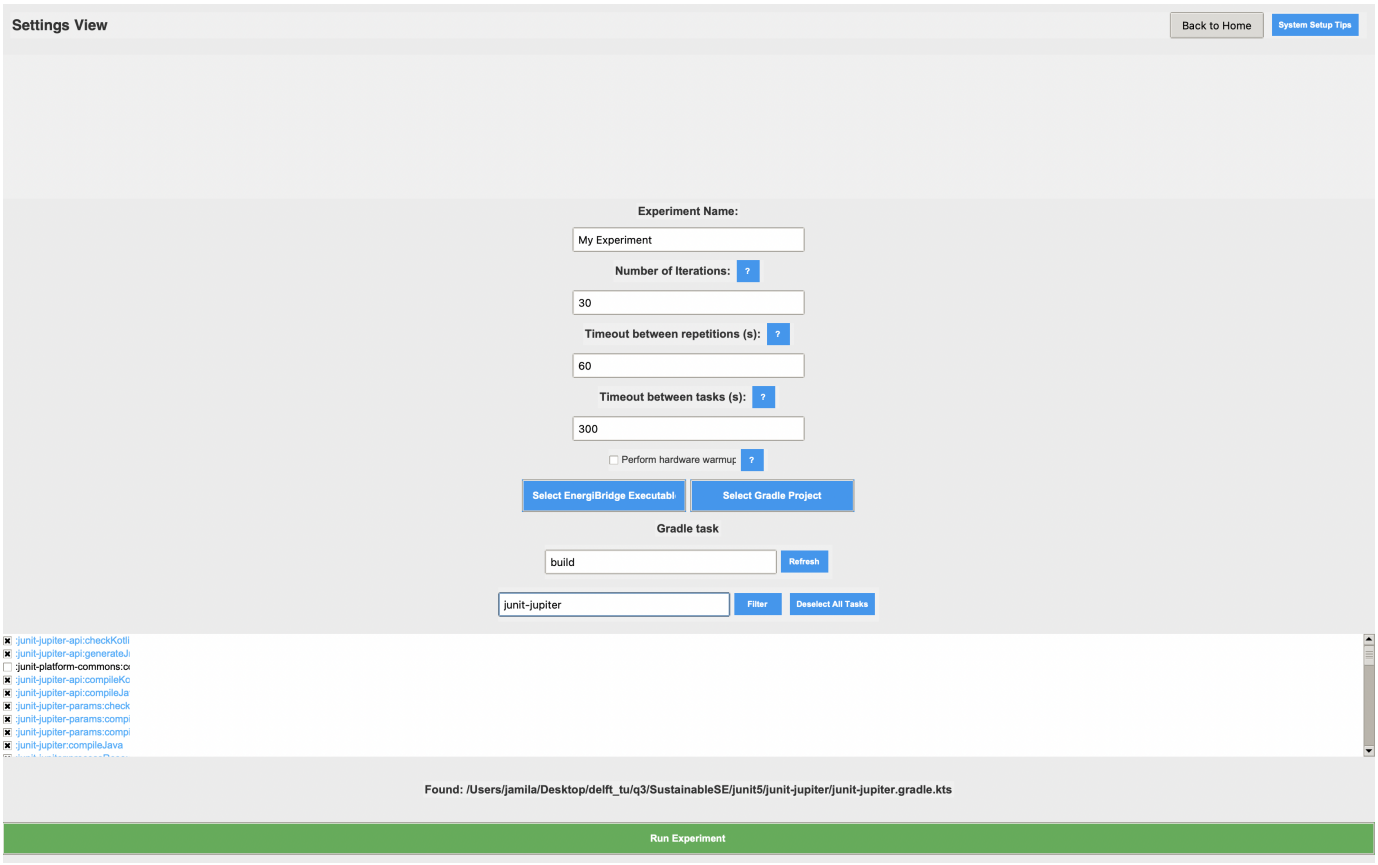

Fig. 1: Experiment setup page

subtracted from the total to isolate the energy cost of the build task.

- **Warm-up phase:** Prior to the start of an experimental session, a 5-minute warm-up is conducted by executing a CPU-intensive task (computing a large Fibonacci sequence) to bring the system to a stable temperature and power state.
- **Repetitions:** Each experiment is repeated 30 times to enable averaging and statistical analysis. Build caches are cleared between runs to ensure that tasks are re-executed consistently.
- **Timeouts between tasks:** A 30-second cooldown phase is inserted between each task to allow the system to prevent tail energy effects from skewing results.
- **Timeouts between build repetitions:** When measuring multiple iterations sequentially, a 30-second timeout is introduced between build executions to isolate each repetition in a consistent way.
- **Zen mode:** to reduce background interference, we disconnected all peripheral devices, closes unnecessary applications, connected the laptop to the wall outlet, and put it in airplane mode. In order to run the experiment without internet connection, we first did a full build of the repository in order to have all external dependencies downloaded and cashed. We would then run `gradle clean` to clean the build output.
- **System specifications:** the experiments were conducted on an Acer laptop with the following hardware specifications:
  - Series: Aspire 5 (A515-54G-59MW)
  - CPU: Intel i5-10210U @ 1.60 GHz
  - RAM: 16GB DDR4 @ 2400 MHz
  - OS: Windows 11, v 24H2
  - Screen resolution: 1920 x 1080

A limitation emerged from the use of EnergiBridge as the measurement backend. During testing, we encountered a bug related to *duration subtraction overflow*, which occurred under certain timing conditions and could compromise measurement accuracy. This issue is documented in EnergiBridge issue #20. We resolved it by patching the library in a forked version used throughout our experiments. While this fix ensured stability, it introduces a dependency on a modified version, which may affect reproducibility or portability unless the patch is integrated upstream.

B. Results

Figure 4 illustrates the results of our experiment. The blue bars correspond to the sub-tasks that are executed, while the orange bar depicts the energy consumed by the single build

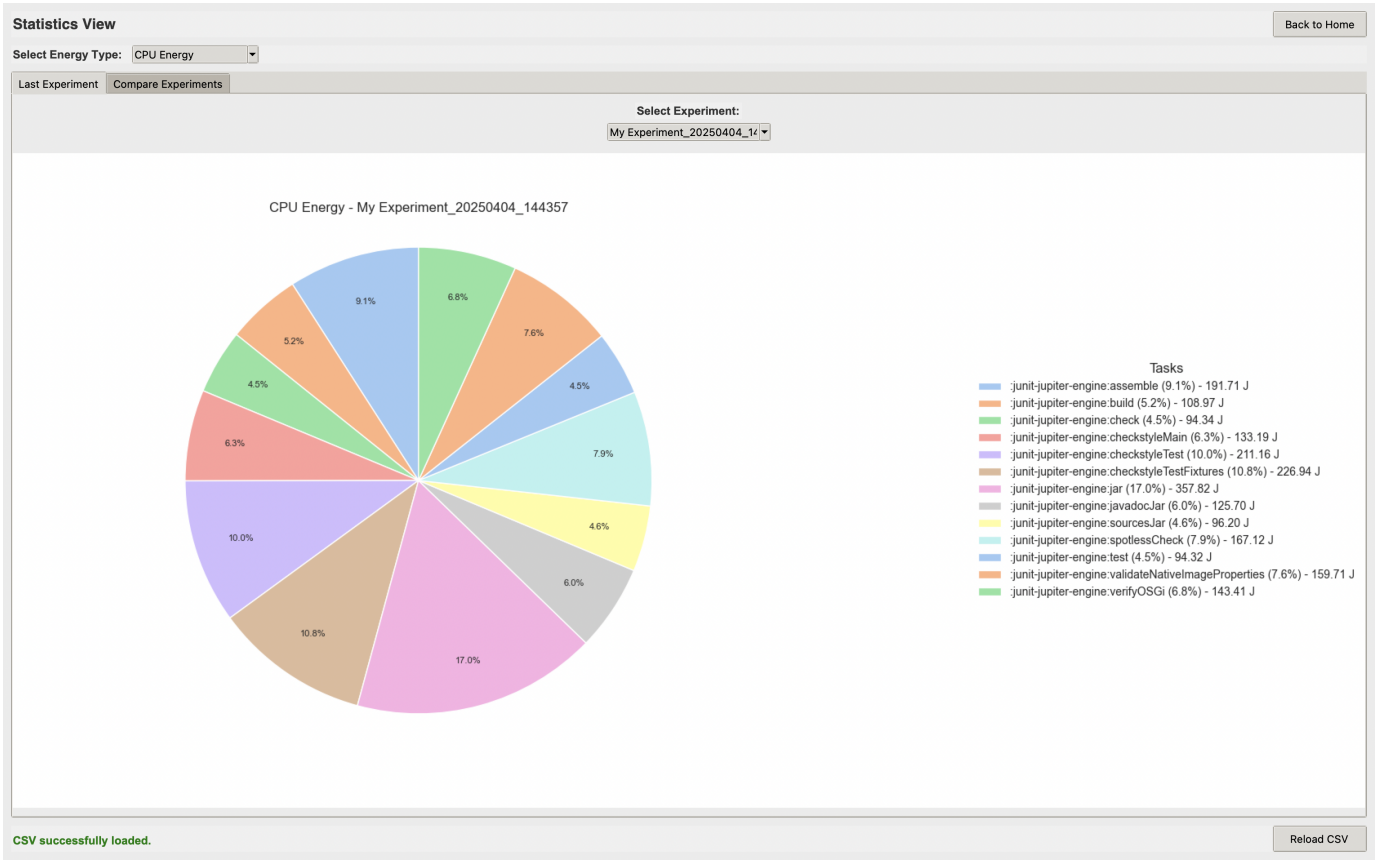

Fig. 2: Single experiment analysis page

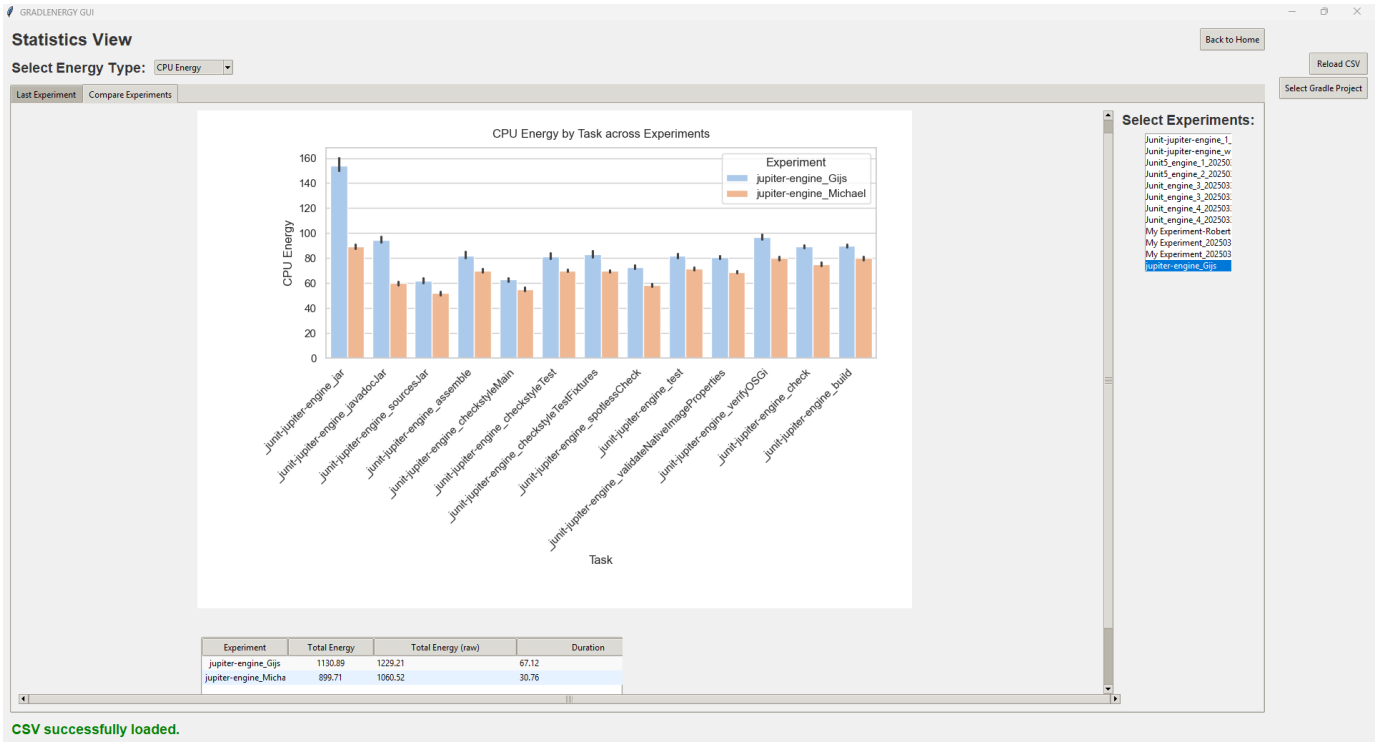

Fig. 3: Multiple experiment analysis page

command. The tasks are executed in the order shown at the horizontal axes from left to right.

The total amount of energy consumed during both experiments differs substantially. Without any sub-tasks selected, the build consumed 145.25 Joules, whereas 1130.89 Joules are consumed with sub-tasks selected. On average, the sub-task experiment took 67.12 seconds to complete, whereas the other experiment completed in 7.04 seconds. The jar task consumes almost twice as much energy as the other sub-tasks that are part of the pipeline. This is the result of shared dependencies, which can also be observed from the full task tree, that are cashed once successfully executed during the jar task. Hence, it performs some of the work of the other tasks as well.

The energy consumed by the experiment without sub-tasks is slightly less than the :junit-jupiter-engine:jar task. This is remarkable, since that task is executed by :junit-jupiter-engine:build implicitly. The gray bars represent the 95% confidence intervals around the measurements and are shown in Table I for the aforementioned tasks. According to the Shapiro-Wilk test, our data was not normally distributed because of heavy right-tails. This remained the case, even after removing outliers more than 3 standard deviations away from the mean. Hence, a bootstrap approach using 1000 samples was used to generate the confidence intervals in a non-parametric way.

Task	CI Lower Bound	CI Upper Bound
`:junit-jupiter-engine:jar`	147.29	157.64
`:junit-jupiter-engine:build`	137.85	150.94

TABLE I: 95% confidence intervals for CPU energy consumption across different tasks.

As shown in Table I, the difference in energy consumption between the jar and build tasks is not significant. Since both experiments were conducted under the same conditions (e.g., no cache cleaning between tasks, only after the entire pipeline finished), we believe the difference is unlikely to be the result of different setups. Table II shows how many tasks are executed for eacy command, as printed out by gradle. We see that for the sub-tasks, a total of 58 tasks are executed, while only 22 were for the full build command. Moreover, even tasks that performed (almost) nothing (e.g., checkstyleMain), still took 3.72 second to execute. Hence, we think the overhead of running individual Gradle tasks causes this the difference in total energy consumed between both experiments.

IV. Limitations and Threats to Validity

While our tool provides insights into the energy consumption of Gradle build processes, there are several limitations and potential threats to the validity. These issues primarily stem from the inherent complexity of build systems and the challenges of precise energy measurement on commodity hardware. In this section, we discuss the main limitations encountered during the development and evaluation of our tool.

Task	Executed	Duration (s)
`jar`	10	8.18
`javadocJar`	1	6.20
`sourcesJar`	1	3.72
`assemble`	4	4.93
`checkstyleMain`	0	3.72
`checkstyleTest`	4	4.95
`checkstyleTestFixtures`	4	4.96
`spotlessCheck`	5	4.50
`test`	5	4.99
`validateNativeImageProperties`	5	4.88
`verifyOSGi`	7	5.68
`check`	6	5.17
`build`	6	5.25
`:junit-jupiter-engine:build`	22	7.04

TABLE II: Number of tasks executed per command and execution time.

a) Gradle Task Redundancy in Dry-Run Output: When using `gradle build --dry-run` to identify tasks for measurement, a limitation arises from *task composition and nesting*. Many Gradle tasks invoke others internally—for example, a higher-level task may implicitly trigger multiple subtasks. As a result, naïvely executing all tasks listed in a dry-run can lead to *redundant executions*, inflating the total energy consumption attributed to the build. This challenges efforts to isolate and compare the energy cost of individual tasks accurately and suggests a need for improved task resolution logic in future versions of the tool. To overcome the issue, we have integrated an additional Gradle plugin in the monitored repositories that exposed the full dependency tree of the tasks. By analyzing this tree, the user can easily identify a subset of high-level tasks that minimizes redundancy and overlap.

b) Gradle Task Overhead: We observed considerable overhead when executing individual Gradle tasks separately. This overhead affects the accuracy of the energy estimates, as the energy consumed by Gradle's task orchestration is non-negligible when tasks are run in isolation. Circumventing this issue remains an open challenge. One potential solution could involve interacting with Gradle through its API, allowing finer control over task execution with reduced overhead. Moreover, this phenomenon may not be unique to Gradle. Other build systems might exhibit similar behavior, making this an open challenge in general.

c) Energy Measurement Frequency: Finally, EnergiBridge, the energy measurement backend used in our tool, is limited to a maximum sampling frequency of 5 measurements per second. Consequently, tasks lasting less than a half are typically captured by only a few measurements. If resource competition delays sampling, the number of measurements can be even lower. This limitation reduces the granularity at which the energy consumption of short-lived tasks can be profiled, potentially obscuring finer variations in energy usage.

V. Background and Related Work

There are several tools and studies that focus on measuring the energy consumption of build pipelines.

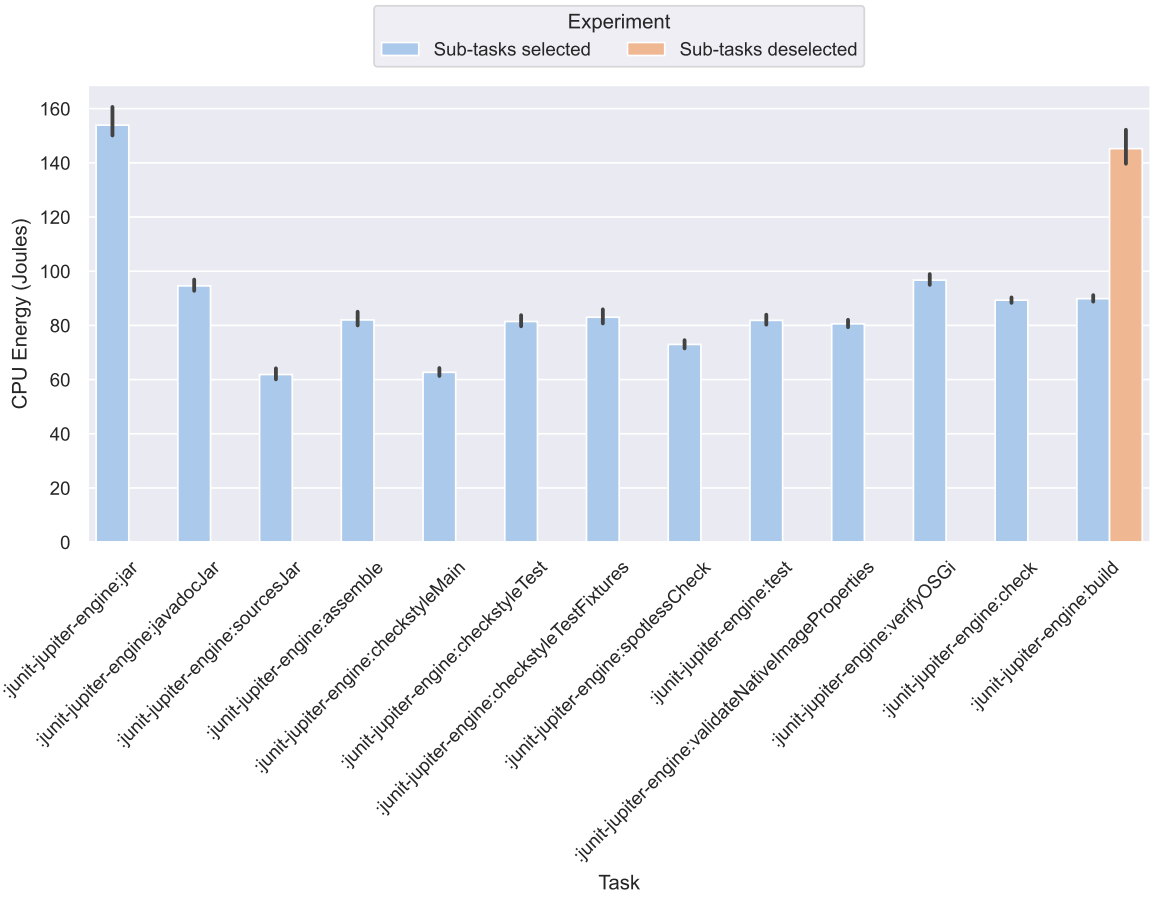

Fig. 4: Energy consumed per (sub-)task, system energy subtracted.

Recently, Limbrunner [8] proposed a framework for measuring the energy consumption of CI/CD pipelines in a cloud environment. Since cloud environments often run builds on virtual machines, they usually do not provide access to hardware-level metrics. Because of this, hardware-based methods like RAPL cannot be used. Instead, the framework uses software metrics that estimate energy consumption based on resource allocation. There are two downsides to this approach. First, these estimates rely on assumptions about the actual energy use of the hardware, which might not always be correct. Second, the framework is designed for cloud systems where resources are shared, making it difficult to run repeatable experiments.

Another tool is the Gradle Energy Consumption Plugin[7], which measures the energy consumption of the Gradle build process. It uses RAPL for its measurements and requires minimal setup to get started. However, this limits the use of the plugin to systems with processors that support RAPL, such as most modern Intel and AMD CPUs. Another limitation is that the plugin only reports energy consumption for the entire build process, which makes it less useful for detailed analysis of individual build steps.

EnergiBridge by Sallou et al. [9] is a cross-platform energy measurement tool that supports Linux, Windows, and MacOS, as well as a wide range of CPU architectures, including Intel, AMD, and Apple M1. EnergiBridge aims to make energy measurements more accessible by providing a simple and unified interface that works across different hardware and operating systems. It collects energy data by using low-level system calls, such as reading CPU registers or using the System Management Controller (SMC) on Mac systems. One of its key advantages is that it offers broader compatibility. EnergiBridge outputs detailed CSV files with energy measurements over time, which makes it suitable for more detailed analysis of energy consumption patterns. The tool has been designed with controlled experiments in mind, making it valuable for researchers, practitioners, and educational purposes. EnergiBridge is available as an open-source project on GitHub[8].

In addition to tools, there is also research that looks at the bigger picture of energy use in software development. Zaidman et al. [4] carried out an exploratory study on the energy consumption of continuous integration and testing in open-source Java projects. They used direct hardware-level

[7]https://github.com/eskatos/gradle-energy-consumption-plugin

[8]https://github.com/tdurieux/EnergiBridge

measurements with USB-C power meters on two platforms: a Raspberry Pi 4 with a Broadcom Cortex-A72 CPU, and a Minisforum MiniPC with an AMD Ryzen 7 CPU. Their results show clear differences in energy consumption between projects and illustrate how the frequency of builds contributes to the total yearly energy usage. For example, the Elasticsearch project alone used approximately 161 kWh in 2022, which is almost 10% of the annual electricity use of an average EU household. This study shows that there is growing interest in understanding the energy impact of common software development tasks and that there is a need to raise awareness among developers.

Together, these tools and studies show that there is increasing attention to the use of energy in software development. At the same time, they also show that there are still challenges in getting accurate, repeatable, and detailed energy measurements. This motivates our work, which aims to improve measurement accuracy and provide more detailed insights into the energy use of build pipelines.

VI. Conclusion

This project introduced GradlEnergi, a cross-platform tool designed to bring visibility into the energy consumption of local software build processes. By integrating EnergiBridge with the Gradle build system commands and providing a graphical interface for designing and analyzing experiments, the tool empowers developers to make more informed, energy-conscious decisions. The case study discussed the application in practice, providing insight into the tool's functionality as well as the challenges associated with using the tool.

Several directions for improvement have also emerged. While our current implementation focuses on Gradle, the underlying design is flexible and can be extended to support other build systems in the future. First, reducing the overhead introduced when measuring individual tasks and better attributing the energy usage of subtasks to their parent tasks would make the results more precise. Second, adding more analytical power with richer statistics and metrics would open the door for more insightful patterns in energy use. Third, adding RAM consumption in addition to CPU would enable more inclusive profiling. And lastly, enhancing the usability and visual presentation of the interface would make the tool even easier to use.

References

[1] I. Manotas, C. Bird, R. Zhang, *et al.*, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 237–248.

[2] B. C. Mourão, L. Karita, and I. do Carmo Machado, "Green and sustainable software engineering-a systematic mapping study," in *Proceedings of the XVII Brazilian Symposium on Software Quality*, 2018, pp. 121–130.

[3] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International conference on mining software repositories (MSR)*, IEEE, 2017, pp. 356–367.

[4] A. Zaidman, "An inconvenient truth in software engineering? the environmental impact of testing open source java projects," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 2024, pp. 214–218.

[5] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software?," Jul. 2015. DOI: 10.7287/PEERJ.PREPRINTS.886.

[6] R. Verdecchia, F. Ricchiuti, A. Hankel, P. Lago, and G. Procaccianti, "Green ict research and challenges," English, in *Advances and New Trends in Environmental Informatics*, V. Wohlgemuth, F. Fuchs-Kittowski, and J. Wittmann, Eds., ser. Progress in IS, EnviroInfo 2016 ; Conference date: 14-09-2016 Through 16-09-2016, Springer, 2017, pp. 37–48, ISBN: 9783319447100. DOI: 10.1007/978-3-319-44711-7_4.

[7] J. Sallou, L. Cruz, and T. Durieux, *Energibridge: Empowering software sustainability through cross-platform energy measurement*, 2023. arXiv: 2312.13897 [cs.SE].

[8] N. Limbrunner, *Dynamic macro to micro scale calculation of energy consumption in ci/cd pipelines*, 2023.

[9] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," *arXiv preprint arXiv:2312.13897*, 2023.