Enhancing EnergiBridge: A Service-Based Approach to Energy Profiling

Sofia Konovalova (6174019), Kaijen Lee (5100887), Violeta Macsim (5498031), Jakub Patałuch (5514274)
Delft University of Technology, Delft, The Netherlands

***Abstract*—Energy efficiency is an increasingly important concern in software development, yet existing measurement tools are often difficult to integrate due to permission issues and limited compatibility with modern environments. This paper presents two service-based implementations of EnergiBridge, an open-source energy profiling tool, designed to improve accessibility and support fine-grained energy monitoring.**

We implement the services in Rust and C++, exposing EnergiBridge's functionality via JSON-RPC. These services allow developers to measure energy consumption at the function level using language-agnostic RPC clients, such as a Python decorator. The Rust service performs in-process measurements using native libraries, while the C++ version wraps the original CLI tool.

We compare both services with the classic EnergiBridge across idle and compute-intensive scenarios. Results show minimal overhead and consistent energy readings, validating the reliability of the service-based approach.

Our findings showcase the potential of service-based tools to provide accurate, automatable, and flexible energy measurements—suitable for integration in development pipelines and containerized environments.

Our work can be found at https://github.com/reglayass/EnergiBridge_RPC

***Index Terms*—energy, consumption, RPC, service, measurement tool**

I. Introduction

As the world transitions to unplugged systems, the demand for battery-powered devices grows, making energy consumption efficiency more critical than ever. Battery production is resource-intensive and pollutant [1], emphasizing the importance of maximizing energy efficiency wherever possible. All development plans [2]–[4] frequently mention clean energy and efficient utilization, which are essential for our daily interactions with technology. Traditionally, a software developer's primary focus has been on developing innovative products and services, but nowadays, energy efficiency should also be listed as a top priority.

The pursuit of energy efficiency can start at its foundation by promoting energy-conscious software development among developers, as recommended by the Sustainable Software Engineering principles [5]. However, integrating energy measurement tools can be challenging, as they often require elevated system permissions or are incompatible with containerized environments commonly used to simplify development and deployment. These limitations may discourage industry professionals from engaging with this emerging field. By improving the accessibility of such tools, developers can gain more precise insights into the energy impact of their software, allowing them to make informed optimizations and contribute to more sustainable computing practices.

Automating energy measurements through a service-based approach eliminates the need for manually running a tool for each measurement, allowing continuous monitoring in the background [6]. However, while this architecture offers practical advantages, its reliability compared to traditional command-line execution remains uncertain. This paper investigates whether a service-based implementation of an energy measurement tool can provide accurate and consistent data, looking to answer the question "*Can a service-based version of a measurement tool provide accurate data about energy consumption?*". We evaluate this approach using EnergiBridge[1], a tool known for its cross-platform adaptability and language-agnostic nature, making it an ideal candidate for this study.

This paper first presents relevant background knowledge on application energy measurement solutions and service implementations. It then details the development and implementation of two variations of EnergiBridge services. Following this, the services' measurements are compared to those obtained through the classic usage of *EnergiBridge*. Finally, the challenges encountered during development are discussed, along with potential improvements for future work.

II. Background

A. Energy Profilers

In a computer, energy is consumed by the hardware, but it is the software that drives this consumption by issuing operations and managing workloads. Energy profilers (EP) are tools that monitor and analyze the power consumption of applications and system components during execution. Energy profilers (EPs) are tools designed to monitor and analyze the power usage of software during execution by collecting metrics such as CPU and GPU utilization, memory access patterns, and disk activity. Depending on the detected hardware and operating system, these tools interface with various low-level frameworks to access real-time energy data. For example, Intel CPUs provide energy readings through RAPL (Running Average Power Limit), a hardware feature that exposes consumption data via model-specific registers (MSRs). NVIDIA GPUs are monitored using NVML (NVIDIA Management Library), a proprietary API that supports real-time power tracking. AMD CPUs also expose MSRs for energy metrics, while AMD GPUs can be monitored via ROCm SMI (System

[1]Official link of the documentation: https://github.com/tdurieux/EnergiBridge

Management Interface). On macOS, energy usage for both CPUs and GPUs is accessed through the System Management Controller (SMC).

When direct hardware interfaces are unavailable, energy profilers rely on empirical energy models, which estimate power consumption based on activity counters and pre-calibrated benchmarks. However, such estimates are often inaccurate, and have been shown to produce results that can vary significantly depending on workload and platforms [7].

B. Services

Services are background processes that perform monitoring, automation, and resource management without requiring direct user interaction. They can be implemented across multiple levels, including **operating system services** (e.g., Linux daemons, Windows services) and **application-level services** (e.g., microservices, API backends). Depending on the application, services can run continuously or be event-driven, triggering actions in response to system events or external requests. Communication between a service and the system is typically accomplished via system calls, inter-process communication (IPC) mechanisms, or remote procedure calls (RPC), with technologies such as **gRPC**[2], **JSON-RPC**[3], and **message queues** facilitating efficient data exchange.

III. Related Work

Numerous tools have been created to measure and analyze software energy consumption. These tools vary in granularity, platform support, and methodological approach, making the choice of a profiler highly dependent on specific requirements such as accuracy, overhead, environment, and integration options.

A. System-Level Energy Profilers

System-level profilers typically monitor the overall power draw of hardware components (CPU, GPU, RAM, etc.) and attribute consumption at a coarse process level. These profilers often rely on hardware registers like RAPL (Intel) or the NVIDIA Management Library.

Commercial services such as Kontram's *OverSight*[4] and EnergyCAP's *UtilityManagement*[5] provide total energy tracking that is integrated with other proprietary products.

Additionally, there exists a wide range of open-source alternatives such as EMaaS [8], a peer-to-peer solution with inexpensive setup that combines hardware-based energy measurements with predictive energy models, while EACOF [9] centralizes multiple simultaneous measurements, aggregates results, and communicates data via API calls.

PowerAPI [10] provides an open-source library to measure per-process energy consumption on Linux. It can combine direct hardware readings (where available) with model-based estimations to generate real-time measurements. **IgProf** (Ignominous Profiler), originally developed at CERN for scientific workloads, extended its sampling approach to record energy usage via Intel RAPL support [11]. All of those tools do really well in terms of identifying global or system-wide energy hotspots, yet offer limited granularity for dissecting energy consumption within specific code segments.

B. Application-Level Energy Profilers

A second group of tools aims to measure energy usage within a single application or runtime environment. They typically instrument code, intercept function calls, or rely on platform-specific libraries (e.g., Java, .NET, Python) to capture performance counters.

Jalen [12] and **Jolinar** [13] instrument Java or .NET applications to estimate function-level energy usage using model-based inference. **pyJoules** makes use of Python decorators and hardware interfaces (RAPL, NVML) to attribute energy consumption to annotated code segments. Proprietary vendor tools such as **Intel Power Gadget** and **AMD uProf** provide specialized power metrics for their CPUs, respectively, but are bound to specific hardware and closed-source APIs.

In the mobile realm, **E-MANAFA** [14] makes use of models and instrumentation to offer device-agnostic energy tracking of Android apps, while **Trepn Profiler** relies on Qualcomm hardware interfaces to measure CPU, GPU, and network usage on Snapdragon devices. These approaches are especially useful for guiding energy-aware development in mobile ecosystems, but they introduce constraints regarding the devices to which they can be applied.

C. Multi-Grained Energy Profilers

While application-level solutions may capture method-level or thread-level usage, certain tools are explicitly designed to operate at multiple layers of granularity—from coarse process measurements down to specific function calls or even instruction blocks. For instance, **jRAPL** integrates deeply with Intel RAPL counters to measure energy usage in Java applications, providing a flexible API that can be used for high-level or low-level instrumentation. Similarly, **JoularJX** extends RAPL-based monitoring to gather multi-grained statistics on JVM-based systems, and can be used alongside **PowerJoular** for broader system-level context [15].

Also, with the constantly developing area of machine learning, there has been developed tools like *Smaragdine* [16], which enables detailed measurement of energy consumption in TensorFlow workloads. A novelty in this field, it allows for measurements from high-level structures up until the smallest division and individual tensor operations, generating results plotted into energy distribution diagrams.

Despite the constraints caused by multithreading in deep learning models, *Smaragdine* periodically logs timestamps for each tensor operation, as well as the related device power consumption. These records are then integrated with empirical energy models to assign energy consumption to specific hardware components responsible for carrying out the

[2]Official link of the documentation: https://grpc.io/
[3]Official link of the documentation: https://www.jsonrpc.org/
[4]https://kontram.fi/en/digital-solutions/oversight-energy-total-energy-measurement-service/
[5]https://www.energycap.com/utility-bill-energy-management-software/

tasks. Smaragdine uses TensorFlow's computation graph to divide model execution into fine-grained operations, allowing for precise energy monitoring. This enables developers to identify which specific processes or layers are the most energy-intensive, compared to how the other tools would only provide overall model-wide consumption.

To summarize, these multi-grained profilers allow developers to switch between lightweight system-wide measurements and targeted function-level analysis. However, many of these tools are language-specific (e.g., Java) or rely on specialized APIs that do not uniformly span across various OS. They may also prove challenging to integrate into container-based or microservices environments due to the permissions required for direct hardware counter access.

D. Rationale for EnergiBridge

Despite the diversity of existing profilers, *EnergiBridge* sets itself apart through:

- **Cross-Platform, Language-Neutral Design:** Implemented in Rust, *EnergiBridge* supports major OS, using direct hardware counters when possible, and falling back otherwise.
- **Portability:** Its Rust-based architecture compiles into a lightweight, standalone binary with minimal runtime dependencies, making it highly portable.

EnergiBridge addresses gaps in cross-platform compatibility and overall design simplicity. The service-based approach presented in this paper further expands EnergiBridge's applicability to scenarios requiring method-level profiling or distributed deployment.

IV. Study Motivation

EnergiBridge is useful in aiding developers towards sustainable code. However, at its current state as of this report, it operates by passing execution commands of the application being measured to it. An example terminal command depicting this is as follows:

```
./EnergiBridge python3 ./test_app.py
```

In the context of *EnergiBridge*, the main purpose of passing execution commands is to indicate when the energy measurements should start being collected, and subsequently when the energy measurements should stop being collected. The former is right before the commands are executed, and the latter is after the commands have been successfully executed and have terminated with some exit code.

The benefits of running *EnergiBridge* as a service are as follows:

1) **Fine-Grained Potential:** It allows fine-grained measurements for sequential code executions without requiring the application under measurement to terminate before collecting results. Lightweight RPC calls could facilitate this by determining the time window in which *EnergiBridge* should collect readings. This enables energy measurements to be easily integrated with large code bases.
2) **Container Support:** The application execution process is decoupled from the energy measurement process by having *EnergiBridge* as a service. *EnergiBridge* at its current state cannot be run within a containerized environment. Tools such as Docker and Kubernetes isolate containers from host hardware for security, reproducibility, and easy setup. However, energy measurement requires system-level function calls on the host machine to collect relevant CPU and GPU data, which is impossible to do directly from a containerized environment. Having *EnergiBridge* as a service can circumvent this limitation by listening to signals or requests from a containerized application to start or stop measurements.
3) **CI/CD friendliness:** As a CLI-based tool combined with its portability, *EnergiBridge* as a service can be easily integrated into automated workflows. A workflow runner machine could have *EnergiBridge* as a service running in the background and pulling images to execute some form of automated code test in the resulting container. The energy measurements collected on the machine could then be processed and analyzed.

In an example scenario, if we want to compare the energy efficiency of different frameworks for data stream aggregation in a backend server implementation, we can annotate or decorate two equivalent functions—each using a different framework but producing the same aggregation result—and directly measure their respective energy consumption either via live testing or unit testing with the *EnergiBridge* service running in the background. In contrast, to do so with the original *EnergiBridge* would require extracting the relevant code as isolated scripts to be tested, making it less convenient.

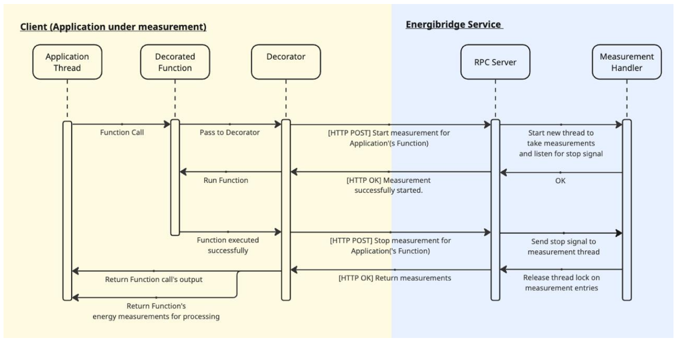
V. Architecture

This section presents the architectures of two service implementations for *EnergiBridge*: one in Rust and another in C++. These programming languages were chosen as they are compiled to machine code. Hence, we conjecture they would introduce minimal energy consumption overhead while running as a service compared to other programming languages. The services have a similar core design: a client asks the server to begin a measurement session, which could be immediately before the execution of a particular process or a function call. Once the execution is completed, the client signals the server to conclude the measurement session, to which the server will respond with the measurements collected corresponding to the session.

Our implementations in the repository included safeguards, such as allowing only one measurement session to be in progress at a time. However, we will only elaborate on their core functionality in this section.

A. Rust-based service

The Rust-based service is a fork of *EnergiBridge*. A server is launched and listens on a specified port, awaiting RPC requests over TCP. The client (or software under measurement) interacts with the server via the JSON-RPC protocol to initiate measurements. It sends a `start_measurements` request to indicate that the server should start collecting energy measurement data from the system. This prompts the server to spawn a worker thread to perform the measurement collection. When stopping, the client sends a `stop_measurements` request to the server, which signals the worker thread to conclude its measurement taking, and the server responds with the recorded measurements in JSON. A simplified communication diagram is depicted in Figure 1.



Client (Application under measurement)
Energibridge Service
Application Thread
Decorated Function
Decorator
RPC Server
Measurement Handler
Function Call
Pass to Decorator
[HTTP POST] Start measurement for Application'(s Function)
Start new thread to take measurements and listen for stop signal
Run Function
[HTTP OK] Measurement successfully started.
OK
Function executed successfully
[HTTP POST] Stop measurement for Application('s Function)
Send stop signal to measurement thread
Return Function call's output
[HTTP OK] Return measurements
Release thread lock on measurement entries
Return Function's energy measurements for processing

Fig. 1: Diagram of communication steps to signal a measurement start and obtain data back in Rust

B. C++ service

The C++ service is similar to the Rust service, but still uses *EnergiBridge* under the hood. The service calls upon *EnergiBridge* rather than having its energy measurement functionality natively baked into the code. When the service runs, a server is launched and listens on a port for the `start_measurements` request. Upon receiving the request, *EnergiBridge* is executed with the `sleep infinite` command, to ensure that it does not terminate prematurely. When a `stop_measurements` request is received, *EnergiBridge* is terminated with the `SIGTERM` signal, and the collected results are returned in JSON format. The C++ server converts the data collected by the underlying *EnergiBridge* from CSV to JSON, which is returned to the client as a server response.

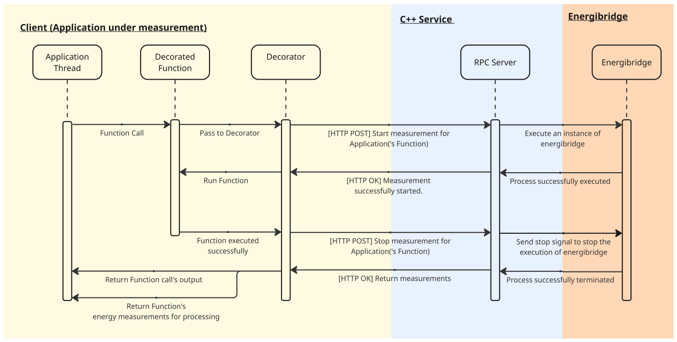
This implementation is "simpler" than the Rust-based service as it does not require managing a separate thread to collect energy measurements, and it uses the output of *EnergiBridge* to create the output for the RPC server. The C++ service implementation is a wrapper, unlike the Rust service implementation, which is a dedicated service.

C. Usage

As each service is an RPC server, these RPC calls can be made in many ways. For our study, we created a decorator in Python that sends the RPC requests right before a function call and then sends a request to stop measurements once the function is completed. An example of its usage can be seen in Listing 1.

When the decorated function is called, the decorator first obtains the current process ID and sends a `start_measurements` request to a specified port. This request includes the PID and function name, allowing the server to determine which function is being measured and the process from which the function originates. To avoid duplicate `start_measurements` calls for recursive functions, the decorator keeps track of which functions are currently being measured. When the function is within the tracking set, the decorator does not intercept it, and it behaves normally. After the function finishes or crashes, the decorator sends a `stop_measurements` request to stop the measurement, removes the function from the tracking list, and prints the measurement results in a formatted JSON output of the columns that the original *EnergiBridge* implementation provided.

Since both services use the RPC-JSON protocol, it allows for language-agnostic implementations of measurements through decorators, annotations, attributes, or manual RPC calls before a method is executed – the point being that the services provide for granular, method-specific or even code block-specific energy measurements rather than requiring an entire application to be run like with classic *EnergiBridge*.



Client (Application under measurement)
C++ Service
Energibridge
Application Thread
Decorated Function
Decorator
RPC Server
Energibridge
Function Call
Pass to Decorator
[HTTP POST] Start measurement for Application('s Function)
Execute an instance of energibridge
Run Function
[HTTP OK] Measurement successfully started.
Process successfully executed
Function executed successfully
[HTTP POST] Stop measurement for Application('s Function)
Send stop signal to stop the execution of energibridge
Return Function call's output
[HTTP OK] Return measurements
Process successfully terminated
Return Function's energy measurements for processing

Fig. 2: Diagram of communication steps to signal a measurement start and obtain data back in C++

```
    @energibridge_rpc(port, exp)
    def fib(n: int = 1) -> int:
        return 1 if n == 0 or n == 1 else fib(n - 1)
             + fib(n - 2)
```

Listing 1: Example implementation in Python

VI. Ablation Study

To validate the viability of having *EnergiBridge* as a service, we perform an ablation study to compare the differences in energy usage between the service-based versions of *EnergiBridge* and the 'classic' *EnergiBridge*. This study consists of two parts. In the first part, we compare the difference in mean energy consumption between classic *EnergiBridge*, the C++ service, and the Rust services with a sleep function of 10 (`sleep(10)`) and 20 seconds (`sleep(20)`). The second part functions the same, except the services and classic *EnergiBridge* are tested with a recursive function that computes the Fibonacci sequence for $n = 10$, $n = 35$, and $n = 40$.

Both parts will result in energy consumption values. However, the objectives of both parts are different. In the sleep experiment, the aim is to compare the overhead that the three different versions of *EnergiBridge* introduce to energy consumption testing, as the experiments involved are to test an idle machine with no code running. This is important as we want to mitigate as much of it as possible in order not to give skewed results to much bigger tests. The Fibonacci experiment aims to mirror a real-life application of these tools by measuring the energy consumption of a specific function. From the results of the experiments in this part, a conclusion can be made on the practicality of using one service over the other.

A. *Methodology*

The tools were tested with a Python script running a sleep function and the Fibonacci sequence function using the decorator outlined in Section V-C. For the classical version, the executable is run with the Python script containing the Fibonacci or sleep function as the argument.

We performed our experiments per the `/py/experiment.py` script in our repository. We test the two types of functions on different occasions, i.e., one session for `sleep(n)` and one for `fib(n)`. However, the setup is the same; we repeat each experiment for 30 iterations with randomized ordering to ensure statistical validity. During the start of a session, we perform a warm-up phase by running a recursive Fibonacci function with n = 30. In between each experiment, we perform a cool-down phase of 30 seconds. When an experiment involves a service, the service is launched, followed by a 5-second resting period before executing the function, to account for any energy usage due to the initialization of the service.

After the energy consumption is obtained for each experiment, we pre-process them by only retaining readings with a Z-score of less than 2, a conventional threshold, to eliminate possible outliers.

B. *Results*

1) Sleep: We identified that the readings obtained follow a normal distribution for the experiments involving `sleep(n)`. This was indicated by performing the Shapiro-Wilk test on them, which yielded p-values above 0.05, as shown in Table I.

Subsequently, we generated a violin plot for each experiment, as shown in Figure 3, and tabulated their corresponding mean energy consumption values in Table II.

	Rust	**C++**	**Classic**
Sleep(10)	0.301927	0.539135	0.863428
Sleep(20)	0.583192	0.122816	0.686482

TABLE I: P-values after applying the Shapiro-Wilk test on the samples of each `sleep(n)` experiment and their corresponding measurement tools.

To identify the statistical significance between the difference in energy consumption measurements between the services and classic *EnergiBridge*, we performed Welch's t-test for the classic *EnergiBridge* against each of the services for

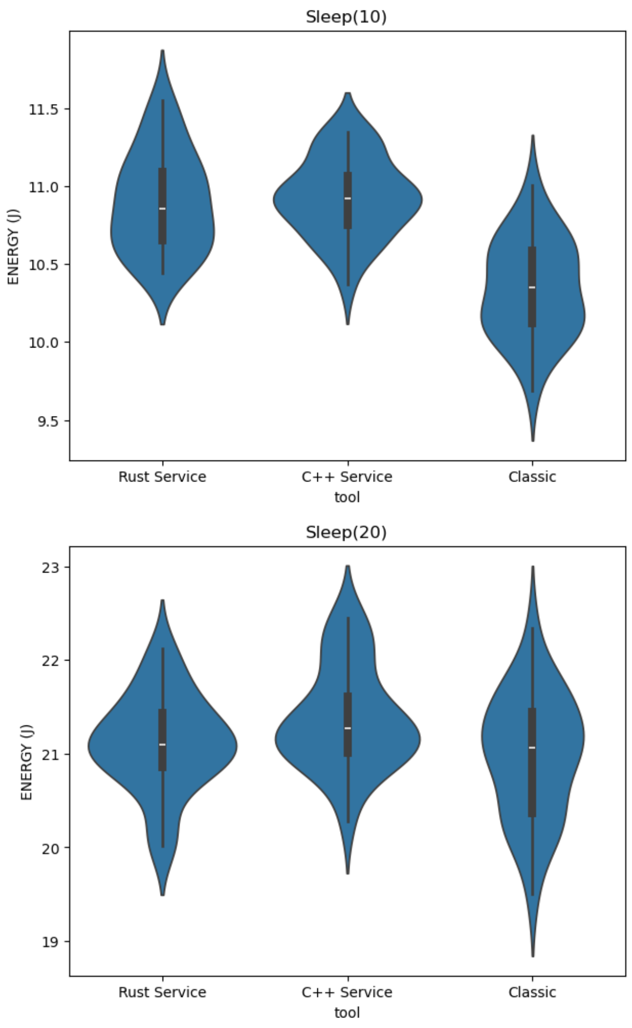


Fig. 3: Violin plots of the energy consumption on `sleep(n)` when measured with different measurement tools.

	Rust	**C++**	**Classic**
Sleep(10)	10.900	10.927	10.343
Sleep(20)	21.117	21.373	20.980

TABLE II: Mean energy consumption in Joules per service and method for each `sleep(n)`.

	Rust	**C++**
Sleep(10)	1.28217e-08	2.09002e-10
Sleep(20)	5.18794e-64	0.0167908

TABLE III: Welch's t-test results comparing classic EnergiBridge with services for each `sleep(n)`

each `sleep(n)`. The resulting P-values obtained are shown in Table III.

2) Fibonacci: We identified that the readings obtained largely demonstrate non-normal behavior for the experiments involving `fib(n)`. This was indicated through the Shapiro-Wilk test, which yielded mostly p-values under 0.05, as shown in Table IV.

	Rust	C++	Classic
Fib(10)	1.27111e-05	0.000463756	0.686482
Fib(35)	0.0883172	0.0264623	0.00040684
Fib(40)	0.0014801	0.00282737	9.96085e-05

TABLE IV: P-values after applying the Shapiro-Wilk test on the samples of each `fib(n)` experiment and their corresponding measurement tools.

Similarly, we generated a violin plot for each experiment, as shown in Figure 4, and tabulated their corresponding median energy consumption values in Table V. We observe the median here as we have identified that most experiments presented non-normal readings.

	Rust	C++	Classic
Fib(10)	1.31677	1.27066	6.28159
Fib(35)	92.46469	94.08493	102.78302
Fib(40)	975.96390	958.66568	1091.70334

TABLE V: Median energy consumption in Joules per service and method for each `sleep(n)`

To identify the statistical significance between the difference in energy consumption measurements, we performed the Mann-Whitney U test for each pair of measurement tools used for each `fib(n)`. The resulting P-values obtained for each `fib (n)` can be found in Table VI.

		Rust	C++	Classic
Fib(10)	**Rust**	-	0.368	2.077e-10
	C++	0.368	-	1.404e-10
	Classic	2.077e-10	1.404e-10	-
Fib(35)	**Rust**	-	0.784	4.573e-09
	C++	0.784	-	3.497e-09
	Classic	4.573e-09	3.497e-09	-
Fib(40)	**Rust**	-	0.340	0.0003
	C++	0.340	-	5.607e-05
	Classic	0.0003	5.607e-05	-

TABLE VI: Mann-Whitney U-test p-values for each `fib(n)` per service

VII. Discussion

In this section, we will elaborate upon our results from the previous section. We divide this section into two parts; first, we will elaborate on the results relating to the experiments with `sleep(n)`, next, we will discuss the results relating to the experiments with `fib(n)`.

A. Sleep

From Figure 3, we observe that for `sleep(10)`, running classical *EnergiBridge* consumes less energy than the services. The difference in energy consumption can be explained by the overhead introduced by the service to run an RPC server in the background, listening for requests. However, this difference

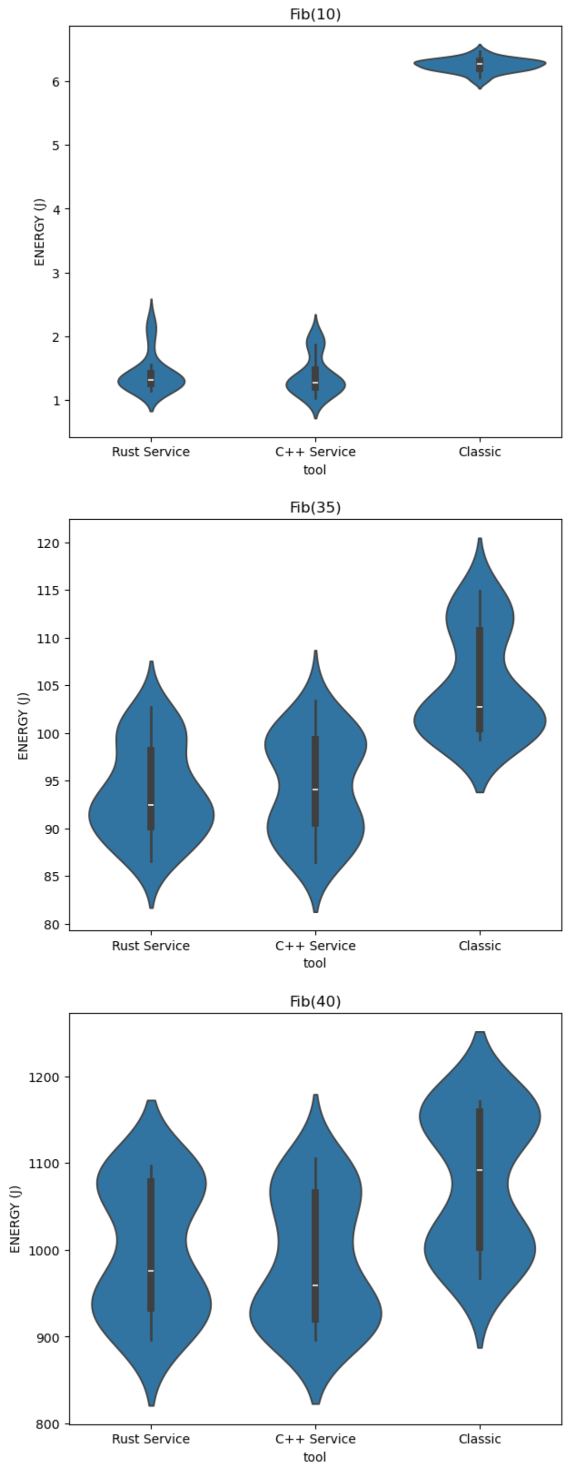


Fig. 4: Violin plots on the energy Consumption on `fib(n)` when measured with different measurement tools.

in energy consumption becomes smaller for `sleep(20)`. We illustrate this in Table VII.

Although the p-values from running the t-tests shown in Table III suggest that these differences are statistically significant, it should be noted that an overhead of less than 0.6 Joules

	Rust	C++	Classic
Sleep(10)	10.900 (+0.557)	10.927 (+0.584)	10.343
Sleep(20)	21.117 (+0.137)	21.373 (+0.393)	20.980

TABLE VII: Mean energy consumption in Joules per service and method for each `sleep(n)`, with the difference to classic *EnergiBridge* annotated.

in practice will not impact real-life energy measurements of applications much more than measurement noise, environmental factors, or system-level fluctuations. Additionally, as demonstrated by the reduced overhead percentage in the `sleep (20)` measurements, this energy difference becomes proportionally smaller as the duration of operations increases, making it even less relevant for typical application workloads, which often involve more complex and longer-running operations than simple sleep calls.

B. Fibonacci

The non-normal behavior of the majority of our experiment data involving `fib(n)` could be addressed by the nature of running deep-recursive functions, such as our Fibonacci implementation in Listing 1. Such functions leave a large memory footprint. The memory management, garbage collection, or even internal caching in Python could introduce irregular energy consumption patterns, yielding non-normal energy measurement readings.

From Figure 4, we observed that classic *EnergiBridge* produced larger energy consumption readings than the services. Additionally, the Mann-Whitney U test on all three instances of `fib(n)` in Table IV suggests that these differences were statistically significant.

We hypothesize that this difference exhibited by using classic *EnergiBridge* compared to the services was due to the initialization of the Python interpreter and the possible cleanup it performed upon termination. Although the experiments all run on scripts containing the corresponding `fib(n)` function, the time window when measurements are taken differs. The services take measurements from the start of the function call and stop right after the function is executed. Classic *EnergiBridge*, however, requires the script to be passed to it as an argument, which means that it takes measurements from the start of the script being executed until the script terminates. Thus, with our hypothesis explaining the difference in energy consumption measured, having *EnergiBridge* as a service demonstrates its capability to produce finer-grained function-level energy measurements.

Lastly, comparing our C++ and Rust implementations of *EnergiBridge* as a service, Table VI indicates no statistical significance in the difference between their collected readings. This suggests that utilizing either tool would produce similar readings. However, in practice, we recommend the Rust implementation as it compiles into a standalone binary. Conversely, the C++ implementation still relies on an accompanying *EnergiBridge* binary, making the Rust implementation more portable.

VIII. Limitations & Future Work

While the study provides promising and valuable insights into measuring software energy consumption, several limitations remain that open avenues for future improvement. First, the C++ service relies on Unix-specific system commands to interface with *EnergiBridge* for starting and stopping measurements. Although Windows-compatible alternatives for those commands exist, adapting the current codebase for cross-platform support would require additional work. A potential solution would involve detecting the operating system at runtime and adjusting command execution accordingly.

Another limitation comes from using service-based tools, where we added a one-second delay before starting the method and another after finishing the measurement. This gives the server enough time to initialize *EnergiBridge* and record the final values properly. In the C++ implementation, we forcefully stop the measurements of *EnergiBridge* by providing the `SIGTERM` signal. Hence, a slight delay must be introduced to account for any remaining energy consumption. While this two-second delay is small, it should still be considered when analyzing the performance and energy results.

Lastly, due to *EnergiBridge's* current usage through either terminal-based commands or service, one might consider wrapping the tool into a package manager to streamline setup, making it more accessible for developers to integrate into automated benchmarking or with other tools created to complement it.

IX. Conclusion

To conclude, this paper compared the accuracy of energy measurements between the classic usage of *EnergiBridge* and two service-based implementations (in Rust as a dedicated service and in C++ as a wrapper) that enable client-server communication via JSON-RPC. By evaluating all three approaches across two environments, idle running and measuring a simple recursive function, we found that a service-based design can offer a more effective alternative to the standard *EnergiBridge* usage, as it introduces finer-grained measurements with negligible additional overhead during the process. This opens the door for more practical and scalable use; looking ahead, *EnergiBridge* as a service could be integrated into CI/CD pipelines to audit an application's energy consumption and generate fine-grained reports for specific tasks within an executable.

References

[1] Greenly, "The harmful effects of our lithium batteries," 2025, accessed: March 10, 2025. [Online]. Available: https://greenly.earth/en-gb/blog/industries/the-harmful-effects-of-our-lithium-batteries

[2] European Commission, "Energy and the green deal," 2024, accessed: 2024-03-06. [Online]. Available: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal/energy-and-green-deal_en

[3] Google, "Google sustainability initiatives," 2024, accessed: 2024-03-06. [Online]. Available: https://sustainability.google/

[4] United Nations, "Sustainable development goals," 2024, accessed: 2024-03-06. [Online]. Available: https://www.un.org/sustainabledevelopment/sustainable-development-goals/

[5] Microsoft, "The principles of sustainable software engineering," 2024, accessed: 2024-03-06. [Online]. Available: https://learn.microsoft.com/en-us/training/modules/sustainable-software-engineering-overview/

[6] D. C. Schmidt, "A domain analysis of network daemon design dimensions," Washington University, Department of Computer Science, Tech. Rep. WUCS-94-33, 1994, accessed: March 10, 2025. [Online]. Available: https://www.cs.wm.edu/~dcschmidt/PDF/daemon-design-94.pdf

[7] E. Jagroep, J. M. E. M. van der Werf, S. Jansen, M. Ferreira, and J. Visser, "Profiling energy profilers," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 2198–2203. [Online]. Available: https://doi.org/10.1145/2695664.2695825

[8] L. Feenstra and P. H. J. Kelly, "Emaas: Energy measurements as a service," *arXiv preprint*, vol. arXiv:1902.02605, 2019, accessed: March 10, 2025. [Online]. Available: https://arxiv.org/abs/1902.02605

[9] D. Barreto, L. Barth, A. Sobe, and T. Fahringer, "Eacof: A framework for energy-aware computing," *arXiv preprint*, vol. arXiv:1406.0117, 2014, accessed: March 10, 2025. [Online]. Available: https://arxiv.org/abs/1406.0117

[10] L. Bourdon, L. Lefèvre, A.-C. Orgerie, and J.-M. Pierson, "Powerapi: A software library to monitor the energy consumed at the process-level," *ERCIM News*, no. 92, 2013, accessed: 2025-04-03. [Online]. Available: https://ercim-news.ercim.eu/en92/special/powerapi-a-software-library-to-monitor-the-energy-consumed-at-the-process-level

[11] CERN, "Igprof: The ignominous profiler," 2024, accessed: 2025-04-03. [Online]. Available: https://igprof.org

[12] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," in *Proceedings of the 2015 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems (SESoS)*. IEEE, 2015, pp. 62–68. [Online]. Available: https://doi.org/10.1109/SESoS.2015.15

[13] A. Noureddine, L. Seinturier, and R. Rouvoy, "Jolinar: A tool for multi-resource energy profiling of software," *Software: Practice and Experience*, vol. 46, no. 11, pp. 1501–1530, 2016. [Online]. Available: https://doi.org/10.1002/spe.2386

[14] R. Rua, M. Couto, and J. P. Cunha, "E-manafa: A device-independent model-based profiler for energy analysis of android apps," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3551349.3556922

[15] A. Noureddine *et al.*, "Powerjoular and joularjx: Multi-level energy profiling of java applications," *SoftwareX*, vol. 19, p. 101186, 2022. [Online]. Available: https://doi.org/10.1016/j.softx.2022.101186

[16] Y. Wang, Y. Guo, X. Li, Y. Bao, C. Xu, and W. Chen, "Tensor-aware energy accounting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2024, p. 662–677. [Online]. Available: https://doi.org/10.1145/3597503.3639156