Comparing a Web-Server Energy Benchmarking Framework with the Green Metrics Tool across Static, Dynamic, and WebSocket Workloads

Youssef Gharbi[1,*], István Bozó[1] and Melinda Tóth[1]

[1]ELTE Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary

Abstract

Energy use is now treated as part of software quality, and the tool we pick to measure it shapes what we conclude. We compare two measurement tools on the same web servers and the same workloads. The first is the Web Server Energy Benchmark (WSEB), a dedicated benchmarking framework for web servers and containers. The second is the Green Metrics Tool (GMT), a general platform for energy and carbon. The servers are behaviourally identical Erlang and Elixir programs, so any differences come from the runtime or the tool, not from the server code. We run static, dynamic, and WebSocket workloads. Both tools rank the workloads in the same order, but GMT reports more than 10 times WSEB's energy for the same workload. GMT measures the entire machine over a padded phase, while WSEB measures a single container over the load window. The energy GMT reports also depend on how runs are grouped, and running each load individually costs about a quarter more than a single combined sweep. GMT reports energy first and does not show how many requests actually succeeded. A public GMT result made Elixir look far greener than Erlang, yet that Elixir run had completed only about three quarters of its requests. With fair servers and full success, the two languages land within about one percent. We read WSEB as a dedicated benchmarking framework for servers and containers and GMT as a general whole-machine tool, and we show where each fits.

Keywords

energy efficiency, green software, web servers, benchmarking, Erlang, Elixir, WebSocket, software carbon intensity, measurement methodology

1. Introduction

Energy use has become a central concern in software engineering. Data centres continue to grow, and network services run without interruption [1, 2, 3]. Web servers are a clear example. They run all the time and serve large volumes of requests. Therefore, their energy use adds directly to the running cost and the carbon footprint of online services [4, 5]. A growing body of work treats energy efficiency as a software quality attribute and not only a hardware concern [6, 7, 8].

Measuring the energy of software is hard. The result depends on the measurement setup and on the workload. Researchers and practitioners follow one of two routes. In the first route, they build a measurement harness around interfaces such as the Running Average Power Limit (RAPL) [9, 10]. They pair it with a monitoring agent such as Scaphandre [11]. In the second route, they adopt a general-purpose platform such as the Green Metrics Tool (GMT) [12]. The two routes differ in what they measure. They also differ in how they assign energy to a workload and in how they report the result. The numbers from these tools are often compared. The tools themselves are rarely compared on equal terms.

This paper compares two such tools. We apply them to the same web servers under the same workloads. The first tool is the Web Server Energy Benchmark, which we call WSEB. It is a purpose-built tool. It drives containerised servers and assigns per-container energy through Scaphandre [13, 14]. The second tool is GMT, a general-purpose platform that measures the energy and carbon of any

SQAMIA 2026: 13th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 16–18, 2026, Niš, Serbia

*Corresponding author.

ikycue@inf.elte.hu (Y. Gharbi); bozoistvan@elte.hu (I. Bozó); tothmelinda@elte.hu (M. Tóth)

0009-0005-3183-0343 (Y. Gharbi); 0000-0001-5145-9688 (I. Bozó); 0000-0001-6300-7945 (M. Tóth)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

CEUR-WS.org/Vol-4336/paper14.pdf

containerised application through a set of metric providers and a usage-scenario description [12]. We run both tools across three workload families. These families cover much of modern web traffic. They are static Hypertext Transfer Protocol (HTTP) responses, dynamically generated HTTP responses, and persistent WebSocket connections [15, 16].

To keep the comparison fair, the servers under test follow a single specification. They differ only in their language runtime. We use servers built on the BEAM, the virtual machine of the Erlang ecosystem [17, 18]. One server is written in Erlang, and one in Elixir [19]. This isolates the behaviour of the runtime from incidental implementation choices. A difference reported by a tool can then be assigned to the runtime or to the tool rather than to the server code.

The study addresses three questions. The first asks how the two tools compare across the benchmarking lifecycle. This lifecycle runs from defining a workload to running it, measuring it, and reporting the outcome. The second asks whether the two tools lead an observer to the same conclusion about the systems under test. It also asks whether the default reporting of either tool can mislead. The third question asks when each tool is the appropriate choice.

This paper makes three contributions. First, it presents a structured comparison of WSEB and a general-purpose measurement platform across three workload families. Second, it shows that the choice of tool can change the conclusion an observer draws from the same experiment. The choice of which quantities a tool treats as primary matters most. Third, it positions WSEB as an extensible measurement foundation and identifies the capabilities that set it apart from the general-purpose platform.

The rest of the paper is organised as follows. Section 2 reviews related work on green software measurement, on energy measurement tooling, and on web server benchmarking. It also examines the Green Software Foundation and its Impact Framework. Section 3 presents the scope of the study and describes the two tools. Section 4 sets out the comparison methodology. Section 5 reports the results, discusses the findings, and states the threats to validity. Section 6 concludes.

2. Related Work

Green software and energy as a quality attribute The study of software energy efficiency has matured. It moved from a general sustainability discussion into concrete measurement and quality frameworks. Early work established that energy efficiency is a software concern and not only a hardware one [4, 6]. Later surveys confirmed this direction. They related it to software quality models and to the broader sustainability agenda [3, 7, 20, 5]. Pinto and Castor showed that application developers shape energy outcomes through their design choices [2]. A recurring theme is that abstraction layers and development environments can change energy behaviour even when the hardware is unchanged [6]. The Green Software Measurement Model consolidates the methods of several research groups into a reference model for assessing the resource and energy use of software [8]. This signals the need for comparable and reproducible measurement practices.

Energy measurement methodologies and tooling Energy studies in software systems combine system metrics with energy telemetry. RAPL exposes processor and memory energy counters through model-specific registers. It is widely used for software energy attribution. Its readings can be affected by power capping and need careful interpretation [9, 10]. In containerised settings, Scaphandre samples power and assigns it to processes and containers [11]. At the level of whole machines, the SPECpower methodology rates energy efficiency across load levels using a physical power meter. It underpins later server efficiency benchmarks [21, 22]. These approaches differ in where they read energy, in how they assign it, and in the granularity they expose. This motivates explicit reporting of the sensor stack and the attribution model behind any reported figure [8].

Web server and container benchmarking Web benchmarking has long studied server behaviour under load. It has also studied the influence of the operating system and the protocol stack [23, 24]. Practical servers span traditional implementations such as Apache and Nginx [25, 26]. They also include

servers built on the BEAM, such as Cowboy and Yaws [27, 28, 29]. Containerisation improved portability and repeatability. A body of work evaluates its performance and overhead [30, 31, 32, 33, 34, 35]. Persistent communication adds a further dimension. The WebSocket protocol provides bidirectional channels over a single connection [15]. Studies report that protocol choice and implementation affect throughput and latency [16, 36, 37]. Our benchmark exercises static HTTP, dynamic HTTP, and WebSocket workloads on the same runtime. This lets us observe how a measurement tool behaves across communication styles rather than within a single synthetic case.

The Green Software Foundation and the Impact Framework The Green Software Foundation (GSF) is a consortium. It develops standards, tooling, and practices for reducing the greenhouse gas emissions of software [38]. Two of its outputs are directly relevant to this work.

The first is the Software Carbon Intensity (SCI) specification. It defines a rate of emissions per unit of useful work. The specification expresses the score as $\mathrm{SCI} = (O + M)/R$. Here O denotes the operational emissions. They are obtained by multiplying the energy consumed by the carbon intensity of the electricity supply. M denotes the embodied emissions assigned to the hardware over its lifetime. R denotes a functional unit that describes how the application scales, such as a request or a user [39]. GMT reports the SCI of a measured run. This links the platform examined in this paper to the wider standardisation effort [12].

The second is the Impact Framework (IF), an open-source project that models the environmental impact of software [40]. The Impact Framework is a calculation framework rather than a measurement instrument. It does not read hardware energy counters. Instead, it takes observations such as processor utilisation or request counts. It transforms them into impact metrics such as energy and carbon through a pipeline of plugins. A manifest file records the observations, the chosen plugins, their configuration, and the computed results. A third party can then rerun the manifest and audit the assumptions behind a reported value. This design places the emphasis on transparency and reproducibility. It complements direct measurement rather than replacing it. A framework of this kind can take the energy figures produced by a measurement tool and convert them into a standardised carbon score. This makes it a natural target for future work that builds on the measurements reported here.

This positioning clarifies the scope of the present study. WSEB and the GMT platform we compare both produce energy measurements. The Impact Framework operates one level above measurement. It turns measured or observed quantities into standardised impact metrics. We therefore treat it as related infrastructure rather than as a directly comparable measurement tool.

Prior work by the authors This paper continues an established line of work on the energy behaviour of the BEAM. Earlier studies measured Erlang energy use and runtime effects [41, 42]. Other work examined the influence of just-in-time compilation on energy and execution time [43]. Further work developed WSEB, a benchmarking tool for web servers that was later extended to a BEAM-focused comparison [44]. The present paper builds on WSEB. It shifts the focus from measuring servers to comparing the measurement tools themselves.

3. The Analysed Tools

In this paper, we evaluate two frameworks for web servers implemented in BEAM languages. We chose these two tools because they represent the two routes described in the Introduction. WSEB is a purpose-built harness around RAPL and Scaphandre, representing the first route. GMT is a general-purpose platform, representing the second route. Comparing one tool from each route shows what a team gains or gives up by choosing one measurement route over the other.

The BEAM and the languages under test The servers under test run on the BEAM. The BEAM is the virtual machine that executes Erlang and related languages. It provides lightweight processes, preemptive scheduling, and per-process isolation [17, 18]. We compare an Erlang implementation with

an Elixir implementation [19]. Both compile to BEAM bytecode and share the same runtime services. This makes them suitable for a controlled comparison.

Web-server workload families We distinguish three workload families. A static workload returns a fixed resource. A dynamic workload generates the response at request time. A WebSocket workload keeps a persistent bidirectional connection and exchanges messages over it [15]. These families exercise different parts of the server and of the measurement pipeline.

How the tools read energy The evaluated tools read energy from the same kind of hardware interface. RAPL exposes the energy used by the processor package and the memory through model-specific registers. A reader samples these registers over time and reports the energy for an interval. Scaphandre reads this interface and assigns power to processes and containers [11]. GMT reads the same interface through its own metric providers. The hosted cluster adds a hardware meter on the power supply. This meter records the physical wall power of the whole machine. It serves as a reference that does not depend on RAPL.

The Web Server Energy Benchmark (WSEB) WSEB is a benchmarking framework for the energy of web servers and containers [13, 14]. It has been used to compare languages, runtimes, and servers. For a run, it finds the containers that are already running, drives the chosen one with a built-in load generator, and reads that container's energy from Scaphandre on the host. It has a module for each workload family, so the static, dynamic, and WebSocket loads are each driven by code written for them. A run produces one row of data. That row carries the number of requests or messages sent, how many succeeded and how many failed, the time taken, the energy in joules, and the processor and memory use. Because the effort and the energy sit in the same row, a reader sees at once whether a low energy figure came with a high drop rate.

The Green Metrics Tool (GMT) GMT measures the energy and carbon of a whole containerised application [12]. The application is described in a usage scenario, which lists the services, how to build them, and the commands that make up the workload. GMT runs the scenario through a fixed set of phases and reads the machine with a set of metric providers that cover processor and memory energy, processor use, and more. It can run on a laptop or on a hosted cluster that also carries a hardware power meter. From the energy, it computes a Software Carbon Intensity score, and it stores the run on a dashboard so others can see it. The design favours a complete, repeatable, and shareable record of one whole run.

Where they differ On our laptop, both tools read the same RAPL energy counters, so they start from the same source. They part company in three ways that drive the rest of the paper.

The first is scope. WSEB reports the energy of a single container during the load. GMT reports the energy of the whole machine across a phase, and that phase includes the setup and idle time around the load. The two numbers answer different questions, so they are not meant to be equal.

The second is granularity. WSEB measures one configuration per run and times only the window when the load runs. GMT wraps each scenario in a baseline, an installation, a boot, an idle, the runtime, and a removal phase. The fixed phases make a run reproducible, but they also add a cost that depends on how the runs are grouped.

The third, and the one that changes conclusions, is the choice of primary number. WSEB keeps the success of the work next to the energy. GMT treats energy, carbon, and the carbon-intensity score as the primary results, and it leaves the request success in the load generator log rather than in the stored metrics. A server that drops part of its load, therefore, looks cheap in GMT unless the reader goes digging.

This contrast is the heart of the comparison. WSEB is a dedicated benchmarking framework that measures each workload in depth and reports the delivered work with the energy. It is built for web

servers and containers and works across many servers and runtimes. GMT is a general platform that surveys the whole machine and turns the result into a shareable energy and carbon record. We test both views in Section 5.

Table 1 brings these differences together and sets out, side by side, what each tool measures and what it reports.

Table 1
What WSEB and GMT each measure, side by side.

Dimension	WSEB	GMT
Scope	Single container, during the load	Whole machine, across a full phase
Granularity	One configuration per run, load window only	Fixed phases: baseline, install, boot, idle, runtime, remove
Energy metrics	Total Energy (J), Avg Power (W), from Scaphandre	Package and DRAM RAPL energy; wall power on the metered cluster
System metrics	CPU %, memory (MB), and for WebSocket also throughput and latency	CPU utilisation, memory used, network I/O, disk I/O
Request success	Stored in the same row as the energy	Printed only in the load generator log, not in the stored metrics
Output	One CSV row per run, plus a bundled plotting tool	One record per metric per phase in a database, browsable on a web dashboard

What each tool reports WSEB writes a comma-separated row for every run. The row holds the energy in joules, the average power, the request or message counts, how many succeeded and how many failed, the runtime, the throughput and for WebSocket, the latency, along with processor and memory use. Success and energy are both first-class here. They sit in the same row, so a reader never has to guess whether a low-energy figure came with dropped work. WSEB also ships a graph tool (a visualiser) for this output. It can plot one run or many CSV files together, let the user choose which metrics to show, and draw bar charts, line charts, and heat maps. The plots can be exported one by one or in a batch. This suits the repeated, side-by-side analysis that research on servers and runtimes needs.

WSEB is also meant to be extended. The workloads are predefined, but a user can define new ones. A new kind of measurement, such as a gRPC workload, can be added, and it runs through the same pipeline. The command-line interface is developed enough to drive all of this, and the run attributes can be overridden and extended without changing the core.

GMT reports a different result shape. It stores each run in a database and shows it on a web dashboard, with the package and memory energy, processor use, network and disk traffic and on the cluster, the wall power from the meter. From the energy, it derives carbon and a Software Carbon Intensity score, and it offers a side-by-side compare view for two runs, which is how we first noticed the Elixir and Erlang difference in Section 5. Its central number is energy, taken over a whole run, and the request success is not stored beside it.

Most of GMT's strengths over WSEB are features rather than big differences, and WSEB already has many of them or can grow them. WSEB has its own comparison and plots. A carbon score is a formula that combines energy and a carbon-intensity figure. A hardware meter can be attached to any machine. So the choice is less about a missing feature and more about fit. WSEB is built for the deep, repeatable analysis of web servers and containers, with success and energy reported together. GMT is built to survey a whole machine and to publish a shareable energy and carbon record.

4. Methodological Considerations

Fair comparison through identical implementations A fair comparison requires that the servers perform equal work. Each server follows a single specification and differs only in its language runtime. A reported difference then reflects the runtime or the measurement tool rather than the server code.

We use behaviourally identical Erlang and Elixir servers. Each server is a small application built on raw transport sockets and runs as a release. The static server returns a cached page. The dynamic server builds the body for each request. The WebSocket server performs the standard handshake and echoes frames. The servers, the workload scenarios, and the run scripts are released as a separate artefact [45].

Workloads and parameters We use three workload families. The static and dynamic HTTP workloads run across a sweep of request counts. The counts range from 100 to 80 000 in 13 steps. The client uses 100 concurrent workers in every run. The WebSocket workload runs across a sweep of client counts of 5, 50, and 100. It uses two patterns. In the burst pattern, each client sends a fixed number of echo round trips as fast as it can. In the stream pattern, each client sends at a fixed rate for a fixed time. The message size is 64 kilobytes. Both tools use the same client concurrency and the same workload parameters, so the offered work matches.

Measurement environments We use three environments. The first is a hosted measurement cluster with a hardware power meter on the power supply. It records physical wall power and serves as a reference. The cluster machine is a Fujitsu Esprimo P956 desktop with an Intel Core i5-6500 processor, running Ubuntu. We used the free tier of GMT on this cluster. Its daily allowance limited how many runs we could submit, so the cluster numbers are single runs.

The second environment is a local GMT installation, and the third is WSEB on the same local machine. The local machine is a Lenovo ThinkPad laptop with an Intel Core i7-10610U processor with four cores and eight threads and 32 GB of memory. It runs Debian 13. During measurement, the screen was dimmed, time synchronisation was off, and no other interactive work ran. The two local tools were run one after the other and never at the same time. The head-to-head tool comparison uses the two local tools on this single machine, and the hosted cluster provides a physical reference point.

The two tools also drive the load in different ways. WSEB runs its load generator on the host and reaches the server through a published port. GMT runs its load generator in a container and reaches the server over the internal Docker network. We keep the client count and the workload parameters the same, but this plumbing differs, and we return to its effect in Section 5.

Packaging of runs GMT can measure a workload in two ways. It can run one sweep that covers all request counts in a single measurement. It can also run one measurement for each request count. By design, WSEB measures one configuration per run. We use the per-request-count form for the head-to-head comparison because it matches the WSEB structure. We also run the single sweep, so we can show how packaging changes the reported energy.

Metrics and normalisation We report energy in joules, execution time, and the success of the offered work. For HTTP, this is the count of successful requests. For WebSocket, this is the count of completed messages. We also report energy per successful unit of work. We compare only the workload window. For GMT, this is the runtime phase, which holds the load step alone and excludes the baseline, install, boot, idle, and remove phases. For WSEB, this is the load window during which the client drives the server. The two tools assign energy at different scopes. WSEB reports per-container energy. GMT reports the package energy for the whole machine. The absolute joule values are therefore not expected to match. We report one run per configuration for the local campaign. Repeated runs with confidence intervals are left for future work. The raw and joined measurement data are released as a separate artefact [46].

5. Results and Discussion

We ran both tools on the same machine, one after the other, over the same servers and the same workloads. The servers, the scenarios, and the scripts are released so the runs can be repeated [45], and the raw and joined data are released as well [46]. Every run finished with full success, and the load

generators reported no failed requests. So the numbers below compare the energy each tool assigns to identical work that both tools agree was done.

Two terms come up often. WSEB energy is the energy of the server container over the load window, read from Scaphandre [11]. GMT energy is the package energy of the whole machine over the runtime phase. The two are measured at different scopes, so we do not expect the joule values to match. What we look for is whether the two tools point in the same direction and whether their default reports lead to the same conclusion.

Static and dynamic HTTP Table 2 shows the HTTP sweep, averaged over the four servers, because here we compare tools and not languages. The four servers track each other closely at every load. Both tools agree on the order, and energy grows with the request count in both. A reader who only wants to rank the loads gets the same answer from either tool.

The size of the numbers is another matter. GMT reports much more energy than WSEB for the same requests, between about 15 and 34 times across the sweep. The ratio is not fixed. It is about 34 at 5000 requests and falls to about 15 at 80000. This is what we expect from the fixed cost of a GMT run. GMT pads each run with baseline, boot, and idle phases and counts the whole package, so a constant overhead is spread over more work as the load grows. The energy per request follows the same trend, and the two tools move closer together, from 0.0196 down to 0.0147 joules per request for GMT and around 0.0010 joules per request for WSEB. We drop the smallest load of 100 requests from the table. That workload finishes in under a tenth of a second, below the resolution of the WSEB sampler, so its per-container window rounds to zero, and the ratio means nothing. That floor is itself worth noting for very small workloads.

Table 2
HTTP sweep, averaged over the four servers. Both tools agree on the order. GMT reports more energy because it counts the whole package over a padded phase, and the gap shrinks as the fixed cost of a run is spread over more work.

Requests	GMT (J)	WSEB (J)	GMT J/req	WSEB J/req	GMT/WSEB
1000	34.2	0.10	0.0342	0.0001	396.1
5000	97.9	2.96	0.0196	0.0006	34.0
8000	142.9	5.74	0.0179	0.0007	25.4
10 000	172.3	7.66	0.0172	0.0008	23.0
15 000	247.0	12.80	0.0165	0.0009	20.5
20 000	317.6	17.19	0.0159	0.0009	19.0
30 000	448.9	26.76	0.0150	0.0009	17.1
40 000	643.6	40.30	0.0161	0.0010	16.5
50 000	791.3	48.74	0.0158	0.0010	16.9
60 000	929.7	65.45	0.0155	0.0011	14.4
70 000	1051.3	60.19	0.0150	0.0009	17.5
80 000	1172.3	79.90	0.0147	0.0010	14.7

Grouping the runs changes the GMT number Because a GMT run carries a fixed cost, the way an experiment is grouped changes the energy it reports. We measured the HTTP loads under GMT in two ways. One way runs all loads inside a single sweep. The other runs each load as its own measurement, which matches how WSEB works. Table 3 compares them. The separate runs report about a quarter more energy than the single sweep for the same work, a mean ratio of 0.78. The extra energy is the sum of the repeated baseline, boot, and idle phases in every run. This is a property of the protocol, not of the servers. It means a single energy figure from GMT only makes sense with a note on how the runs were grouped.

Figures 1 and 2 show the same local index comparison in each tool, so the two tools can be seen in use on one workload. WSEB's own graph generator (Figure 1) plots the energy of each request



Figure 1: WSEB's own graph generator: energy against request count for the fair Erlang and Elixir index servers. The two lines track each other closely across the whole sweep.

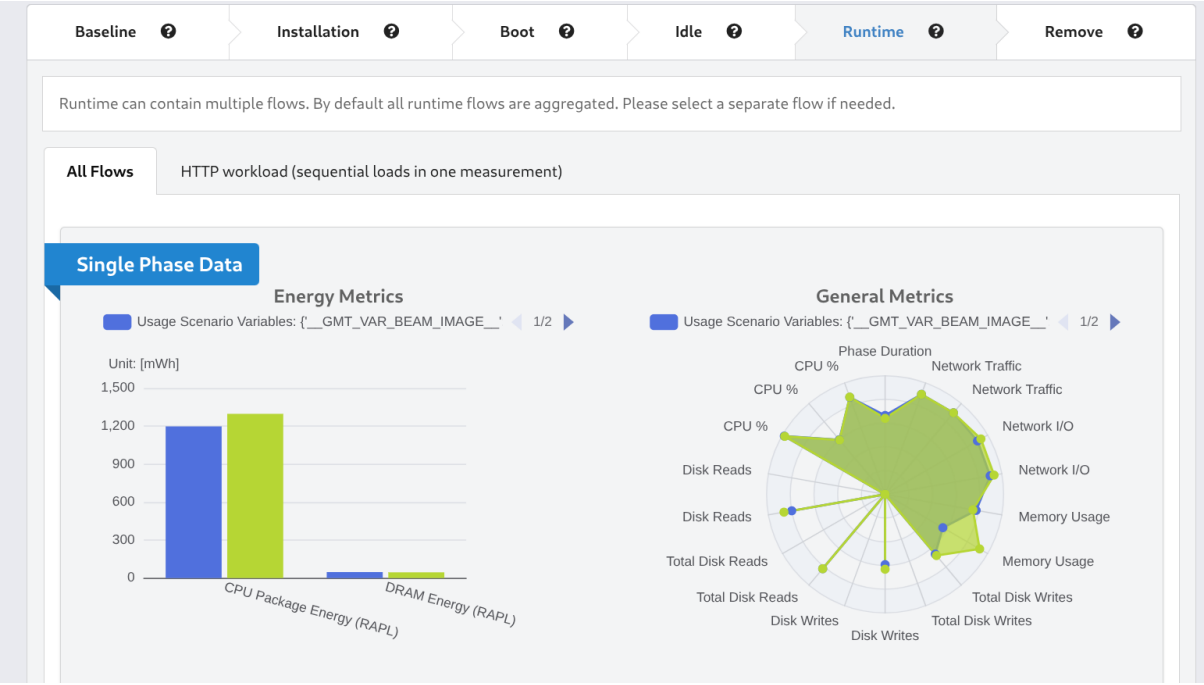


Figure 2: GMT's compare view of the two index-sweep runs, showing the Runtime-phase energy (CPU package and DRAM, from RAPL). Run 1 (blue) is Erlang, Run 2 (green) is Elixir. The two languages use close to the same energy, with Elixir marginally higher.

count for the fair Erlang and Elixir index servers, and the two lines track each other closely across the whole sweep, which is the equal result the tables report. GMT's compare view (Figure 2) shows the two index-sweep runs. GMT stores a sweep as one run, so it reports a single energy value per server for the Runtime phase, the CPU package and DRAM energy from RAPL, and again the two languages are close, with Elixir marginally higher, matching Table 3. The figures also show a real difference between the tools. WSEB reports energy per load, so it gives a curve, while GMT reports one aggregate per run.

Table 3
GMT energy for the same HTTP loads, measured as one bulk sweep and as the sum of separate per-load runs. The separate runs repeat the fixed phases and report about a quarter more energy for the same work.

Server	Bulk (J)	Per-load sum (J)	Bulk/Per-load
fair-elixir-dynamic	4833.0	6145.5	0.786
fair-elixir-index	4833.5	6014.8	0.804
fair-erlang-dynamic	4833.5	6060.2	0.798
fair-erlang-index	4482.6	6060.6	0.740
Mean	4746	6070	0.782

WebSocket The WebSocket runs are a cautionary case rather than a clean comparison, because the two tools use different WebSocket load generators with different defaults (Table 4). In the burst pattern, WSEB pauses one second between bursts, so a run lasts about 2000 seconds and the server is near idle for almost all of it, while GMT's burst has no pause and finishes in seconds. In the stream pattern, WSEB runs for 20 seconds, and GMT for 25, and the two generators complete different message totals, about 114000 against 78000 at 100 clients. The WebSocket energy is therefore not comparable across the tools the way the HTTP energy is.

The wider lesson is that matching a workload across two separate tools is harder than it looks. A one-second pause, or a five-second difference in duration, is enough to break the comparison. The HTTP workloads matched easily because both tools send a fixed number of requests at the same worker count. The WebSocket ones did not, and the runtime column is what exposed it. A clean WebSocket comparison would need the two load generators to share the same timing, which we leave for a follow-up.

Table 4
WebSocket sweep, averaged over the Erlang and Elixir echo servers, with the runtime added. The two tools use different load generators. In the burst pattern, WSEB pauses one second between bursts, so its runtime is about 2000 seconds, while GMT finishes in seconds. In the stream pattern, WSEB runs for 20 seconds and GMT for 25. The energy is therefore not comparable across the two tools for this workload.

Pattern	Clients	Tool	Runtime (s)	Energy (J)	Messages
burst	5	GMT	2.6	76.6	10 000
burst	5	WSEB	2009	3.68	10 000
burst	50	GMT	17.3	380.5	100 000
burst	50	WSEB	2011	11.47	100 000
burst	100	GMT	36.6	759.0	200 000
burst	100	WSEB	2011	34.71	200 000
stream	5	GMT	25.1	47.2	5181
stream	5	WSEB	20.0	2.24	4203
stream	50	GMT	25.1	190.7	53 998
stream	50	WSEB	20.2	25.62	40 858
stream	100	GMT	25.2	527.0	114 124
stream	100	WSEB	20.2	44.70	78 035

Energy without success can mislead The clearest case for reporting success comes from our own earlier runs. GMT's dashboard holds the static index runs we measured on its metered cluster[1]. Two of them show the effect, an Elixir run[2] and an Erlang run[3], and the side-by-side view puts them together[4].

[1] https://metrics.green-coding.io/runs.html?&uri=joegharbi&show_archived=false&show_other_users=true
[2] https://metrics.green-coding.io/stats.html?id=eb27926c-7f74-4a1e-9185-dc84aa5f9b8a
[3] https://metrics.green-coding.io/stats.html?id=238bbef1-ff50-4af6-b956-4178528d7892
[4] https://metrics.green-coding.io/compare.html?ids=eb27926c-7f74-4a1e-9185-dc84aa5f9b8a,
238bbef1-ff50-4af6-b956-4178528d7892

On these runs, Elixir looked far greener than Erlang. The Elixir run reported about 23 kJ and the Erlang run about 63 kJ, so a reader would call Elixir nearly three times as efficient. The request success is not in GMT's stored metrics, but it sits in the load generator log of each run. There, the Erlang server completed about 99.6 percent of its requests, while the Elixir server completed only about 76 percent and dropped roughly 94000. GMT still records the energy the machine spent, but it does not place the request success next to that energy, so the dropped work does not show in the headline number. The rest of the gap was not the language either. The old Erlang server was slow because it did not send a Content-Length header, which is a bug in that server and not a trait of Erlang.

We saw the same pattern on the laptop with the local install. The Elixir static server completed only about 49000 to 51000 of 80000 requests, between a third and a half of the load shed, while Erlang stayed near 99.8 percent. So on both the cloud and the laptop, GMT gave Elixir a low energy figure for a run that had quietly dropped a large part of the work.

Once we removed both problems, the difference went away. We rewrote the two servers to a single specification so they behave the same, and we required complete success. The fair Erlang and Elixir static servers then used 13.93 and 14.03 kJ, a difference under one percent, and the dynamic servers used 13.96 and 13.81 kJ. The two languages are effectively equal in this work, and the earlier story that Elixir is greener was an artefact of hidden load shedding plus a server bug, not a real win.

Success accounting alone would not have settled it. Even when we divide the old energy by the requests that did succeed, the old Elixir server still looks better, about 79 against 163 millijoules per successful request, because the missing header inflates the Erlang figure. A fair comparison needs both pieces, the success of the work and a matched implementation. WSEB writes the success and failure counts in the same row as the energy, so the first piece is visible by default. GMT keeps energy and success apart, so a reader has to open the load generator log to notice that a server shed load.

What the two tools cost to run The tools also differ in how much machinery they bring to a measurement, and that machinery shows up both in the numbers and in daily use.

GMT wraps each measured workload in a fixed set of phases. It runs a baseline, an installation, a boot, a 60-second idle, the runtime, and a removal phase. It reads the workload energy during the runtime phase, but the surrounding phases still cost time and set up the run. This is why separate per-load runs cost about a quarter more energy than one sweep in Table 3. Each separate run pays for its own baseline, boot, and idle again.

GMT also needs the network on every run. It pulls its build image and the server images, and it installs the load generator packages inside a container each time, so a run cannot start offline. During our campaign, a dropped Wi-Fi connection failed every queued run at the image pull. WSEB has none of this. It talks to a server that is already running and reads energy from Scaphandre on the host, so it works offline and starts in seconds.

The footprint is also different. GMT pins the server to a fixed set of cores and a memory limit, starts its own database and metric providers, and runs the load generator in a second container that reaches the server over the internal Docker network. WSEB runs the load generator on the host, reaches the server through a published port, and starts Scaphandre only after the server passes a health check, so the idle time before the load does not dilute the reading. The heavier GMT stack runs on the same machine as the server under test, so part of what GMT measures is GMT itself.

A few smaller points came up while we worked with GMT, and they matter for planning a campaign. On our laptop, the processor filters the RAPL energy counters, and GMT refuses to run until that check is turned off, so the local RAPL numbers carry a caveat that the cloud meter does not. A short WebSocket burst, which does only about a second of real work, sometimes failed on the cloud because the measured window was mostly idle and fell below GMT's validity threshold. A long sweep could pass the per-run time limit and leave no record at all. When we interrupted a run, GMT left its measurement processes running and then refused to start the next run until we cleared them by hand. None of this is a reason to avoid GMT. It is the cost of a general platform that aims at reproducible, shareable, whole-machine measurement, and it is the friction that WSEB removes.

GMT on the cloud and on the laptop We also ran GMT in two places, and the same tool gave very different numbers. On the metered cluster, the fair static servers drew about 14 kJ each, read from the hardware meter on the power supply. On the laptop, the same servers read a few kilojoules from RAPL for the matching sweep. The two numbers are not meant to agree. The cluster meter sees the whole machine at the wall, including memory, disk, fans, and power-supply losses, while RAPL sees only the processor package and memory. The two machines also differ: a metered desktop against a laptop. So even with one tool, the place of measurement sets the size of the number, and a figure from the cluster cannot be compared with a figure from the laptop. This is why we keep the cluster as a physical reference and draw the tool comparison from the two tools on one laptop. The laptop also filters its RAPL counters, so the local GMT energy carries a caveat that the metered cluster does not.

A dedicated framework and a general platform These results point to two jobs rather than a winner. WSEB is a dedicated benchmarking framework. It measures each workload in depth, attributes energy to the container under test, times only the load window, and reports the delivered work with the energy. It works across many servers, languages, and containers, and it is built to be extended to new workloads and new kinds of measurement. This fits research that compares servers or runtimes under a chosen load, which cares how much work is completed, and that reaches into short or bursty workloads. GMT is a general platform. It measures the whole machine, follows a described scenario through fixed phases, and turns the result into energy, carbon, and a Software Carbon Intensity score that can be shared and compared across projects. This fits whole-machine accounting, carbon reporting, and keeping a public record. The outputs that GMT adds over WSEB, such as the carbon score, are mostly formulas over energy that WSEB can adopt as well, and the energy WSEB produces can feed a standardised carbon score through the Impact Framework [40]. The lasting difference is one of aim. WSEB studies a workload in depth, and GMT surveys the whole machine.

Threats to validity The three environments do not share hardware. The cloud cluster is a metered desktop, and the local runs are on a laptop, so joule values do not carry across machines. We treat the cloud meter as a reference and draw the tool comparison from the two local tools on one machine. The two tools also use different load generators. WSEB drives the server from the host through a published port, while GMT drives it from a container over the internal network. In the WebSocket stream pattern, the two generators delivered different message totals in the same time, so we report the message counts and compare energy per message. GMT runs a heavier stack on the measurement machine, which can disturb the server it measures, and this footprint is part of what we observe rather than a separate error. We report one run per configuration for the local campaign and leave repeated runs with confidence intervals for future work. On the laptop, the RAPL counters are filtered, and GMT runs only with that check disabled, so the local energy should be read with care. Finally, we wrote one of the two tools. We reduce this bias by using a single fair specification for the servers, by running GMT as released without tuning, by keeping the workloads and the client counts identical, and by releasing the servers, the scenarios, and the data [45, 46].

6. Conclusion and Future Work

In this paper, we compared WSEB and GMT on the same servers under the same workloads. On a fair workload, the differences that matter are not in the raw joules. Both tools rank the workloads the same way and only disagree on magnitude, because they measure at different scopes. What changes a conclusion is what each tool puts first. GMT reports energy and leaves the request success in a log, so a server that drops part of its load can look green. We saw this in our own earlier runs, where GMT made Elixir look far more efficient than Erlang while the Elixir server had quietly dropped about a quarter of its requests. With fair servers and full success, the two languages came out within about one percentage point. WSEB keeps the delivered work next to the energy, so that the trap is visible by default. The energy GMT reports also depend on how the runs are grouped, which is easy to miss.

We see WSEB as a dedicated benchmarking framework for web servers and containers, and GMT as a general whole-machine platform. The choice depends on the question. WSEB already covers much of what GMT offers, including its own comparison and plots, and the rest, such as a carbon score, is a feature that sits on top of the energy it already measures. WSEB is also extensible, so new workloads and new kinds of measurements can be added without changing the core. Future work includes feeding WSEB energy into a standardised carbon score through the Impact Framework [40], an automated pipeline that repeats runs and reports confidence intervals, tighter control of the host to cut variance across machines, and an optional hardware power meter to give the local setup a physical reference.

Declaration on Generative AI

The authors utilized AI tools for content enhancement and carefully reviewed and verified all AI-generated additions.

References

[1] J. G. Koomey, Growth in data center electricity use 2005 to 2010, 2011. URL: https://alejandrobarros.com/wp-content/uploads/old/4363/Growth_in_Data_Center_Electricity_use_2005_to_2010.pdf, accessed: 2025-05-03.

[2] G. Pinto, F. Castor, Energy efficiency: A new concern for application software developers, Communications of the ACM 60 (2017) 68–75. doi:10.1145/3154384.

[3] S. U. Lee, N. Fernando, K. Lee, J.-G. Schneider, A survey of energy concerns for software engineering, Journal of Systems and Software 210 (2024) 111944. doi:https://doi.org/10.1016/j.jss.2023.111944.

[4] A. Hindle, Green software engineering, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 5, 2016, pp. 46–55. doi:10.1109/SANER.2016.60.

[5] S. Paul, A. Saha, M. Arefin, T. Bhuiyan, A. A. Biswas, A. W. Reza, N. Alotaibi, S. Alyami, M. A. Moni, A comprehensive review of green computing: Past, present, and future research, IEEE Access 11 (2023) 87445–87494. doi:10.1109/ACCESS.2023.3304332.

[6] E. Capra, C. Francalanci, S. A. Slaughter, Is software "green"? application development environments and energy efficiency in open source applications, Information and Software Technology 54 (2012) 60–71. doi:https://doi.org/10.1016/j.infsof.2011.07.005.

[7] G. A. Garcia-Mireles, Exploring sustainability from the software quality model perspective, in: 2018 13th Iberian Conference on Information Systems and Technologies (CISTI), 2018, pp. 1–4. doi:10.23919/CISTI.2018.8399319.

[8] A. G. et al., Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—green software measurement model (gsmm), Future Generation Computer Systems 155 (2024) 402–418. doi:https://doi.org/10.1016/j.future.2024.01.033.

[9] I. Corporation, Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389, Technical Report, Intel Corporation, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html, accessed: 2025-05-05.

[10] Y. Gao, F. Wu, Y. Shu, W.-c. Feng, An evaluation of intel's rapl power limiting, Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED) (2016) 1–6.

[11] Hubblo-org, Scaphandre - an energy monitoring agent, https://github.com/hubblo-org/scaphandre, 2025. Accessed: 2025-05-03.

[12] Green Coding Solutions, Green metrics tool, https://metrics.green-coding.io/index.html, 2026. Accessed: 2026-05-14.

[13] Y. Gharbi, Web Server Benchmarks: Energy-Efficient Benchmarking Framework for Yaws, Nginx, and Apache in Containerized Environments, Release V0.2.1, https://github.com/joegharbi/web-server-benchmarks/releases/tag/V0.2.1, 2026. Accessed: 2026-03-07.
[14] Y. Gharbi, BEAM Web Server Benchmarks, Release v1.5, https://github.com/joegharbi/BEAM-web-server-benchmarks/releases/tag/v1.5, 2026. Accessed: 2026-05-14.
[15] A. Melnikov, I. Fette, The WebSocket Protocol, RFC 6455, 2011. URL: https://www.rfc-editor.org/info/rfc6455. doi:10.17487/RFC6455.
[16] W. Lasocha, M. Badurowicz, Comparison of websocket and http protocol performance, Journal of Computer Sciences Institute 19 (2021) 67–74. URL: https://ph.pollub.pl/index.php/jcsi/article/view/2452. doi:10.35784/jcsi.2452.
[17] J. Armstrong, Making reliable distributed systems in the presence of software errors, Royal Institute of Technology, Stockholm, 2003. URL: https://erlang.org/download/armstrong_thesis_2003.pdf.
[18] F. Cesarini, S. Thompson, Erlang Programming: A Concurrent Approach to Software Development, O'Reilly Media, Inc, 2009.
[19] J. Valim, The Elixir Team, The Elixir programming language, https://elixir-lang.org/, 2026. Accessed: 2026-05-19.
[20] Y. Z. et al., On the opportunities of green computing: A survey, 2023. URL: https://arxiv.org/abs/2311.00447. arXiv:2311.00447.
[21] S. P. E. Corporation, Specpower_ssj2008, SPEC (2008). https://www.spec.org/power_ssj2008/.
[22] J. von Kistowski, M. Meißner, K.-D. Lange, J. A. Arnold, Server Energy Efficiency Benchmarks, Springer Nature Switzerland, Cham, 2025, pp. 247–265. URL: https://doi.org/10.1007/978-3-031-85634-1_11. doi:10.1007/978-3-031-85634-1_11.
[23] E. Nahum, T. Barzilai, D. Kandlur, Performance issues in www servers, IEEE/ACM Transactions on Networking 10 (2002) 2–11. doi:10.1109/90.986497.
[24] A. M. Faber, M. Gupta, C. H. Viecco, Revisiting web server workload invariants in the context of scientific web sites, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, 2006, p. 110. doi:10.1145/1188455.1188570.
[25] The Apache Software Foundation, Apache HTTP server project, https://httpd.apache.org/, 2024. Accessed: 2024.
[26] W. Reese, Nginx: the high-performance web server and reverse proxy, Linux Journal (2008). URL: https://www.linuxjournal.com/article/10108, accessed: 2025-05-05.
[27] Nine Nines, Cowboy - small, fast, modern HTTP server for Erlang/OTP, https://ninenines.eu/docs/en/cowboy/2.9/guide/, 2024. Accessed: 2025.
[28] C. Wikstrom, Yaws: Yet another web server, in: Erlang Workshop, 2002.
[29] S. Vinoski, Yaws: Yet another web server, IEEE Internet Computing 15 (2011) 90–94. doi:10.1109/MIC.2011.100.
[30] C. Pahl, Containerisation and the paas cloud, IEEE Cloud Computing 2 (2015) 24–31. doi:10.1109/MCC.2015.51.
[31] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.
[32] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, S. d. R. S. de Souza, Container-based performance evaluation: A survey and challenges, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 398–403. doi:10.1109/IC2E.2018.00075.
[33] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, C. A. F. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013, pp. 233–240. doi:10.1109/PDP.2013.41.
[34] P. Muzumdar, A. Bhosale, G. P. Basyal, G. Kurian, Navigating the docker ecosystem: A comprehensive taxonomy and survey, Asian Journal of Research in Computer Science 17 (2024) 42–61. doi:10.9734/ajrcos/2024/v17i1411.
[35] Docker, Inc., Docker documentation, https://docs.docker.com/, 2024. Accessed: 2024.

[36] M. S. Rahman, Efficiency and Performance of WebSockets and Server-Sent Events, Master's thesis, Blekinge Institute of Technology, 2016. URL: https://www.diva-portal.org/smash/get/diva2:874674/FULLTEXT01.pdf.

[37] M. Tomasetti, An analysis of the performance of websockets in various programming languages and libraries, SSRN Electronic Journal (2021). URL: https://ssrn.com/abstract=3778525. doi:10.2139/ssrn.3778525.

[38] Green Software Foundation, Green software foundation, https://greensoftware.foundation/, 2026. Accessed: 2026-06-09.

[39] Green Software Foundation, Software carbon intensity (SCI) specification, https://sci.greensoftware.foundation/, 2024. Accessed: 2026-06-09.

[40] Green Software Foundation, Impact framework, https://if.greensoftware.foundation/, 2026. Accessed: 2026-06-09.

[41] Y. Gharbi, M. Tóth, I. Bozó, Measuring and analysing erlang's energy usage, in: 2024 47th MIPRO ICT and Electronics Convention (MIPRO), 2024, pp. 1993–1998. doi:10.1109/MIPRO60963.2024.10569977.

[42] Y. Gharbi, I. Bozó, M. Tóth, Green computing for erlang, in: 3rd Workshop on Resource AWareness of Systems and Society (RAW 2024), 2024. URL: https://ceur-ws.org/Vol-3867/paper3.pdf, empirical analysis of Erlang energy behavior in different environments.

[43] Y. Gharbi, M. Tóth, I. Bozó, Analysing the energy usage of the erlang beam, in: SQAMIA 2024 Workshop, 2024. URL: https://ceur-ws.org/Vol-3845/paper16.pdf, focus on BEAM runtime behavior and JIT impact on energy and execution time.

[44] Y. Gharbi, M. Tóth, I. Bozó, Benchmarking Energy Efficiency in Web Servers with a Focus on Erlang: A Framework-Based Comparative Study, Submitted to SN Computer Sciences (2026).

[45] Y. Gharbi, Fair index benchmarks: Identical erlang and elixir web servers with matching workload scenarios for tool comparison, https://github.com/joegharbi/fair-index-benchmarks/releases/tag/v1.0.0, 2026. Release v1.0.0; Accessed: 2026-06-16.

[46] Y. Gharbi, Fair index benchmarks results: Energy measurements from the web server energy benchmark and the green metrics tool, https://github.com/joegharbi/fair-index-benchmarks-results/releases/tag/v1.0.0, 2026. Release v1.0.0; Accessed: 2026-06-16.