An Empirical Evaluation of the Energy and Performance Overhead of Monitoring Tools on Docker-based Systems

Madalina Dinga[1], Ivano Malavolta[0000−0001−5773−8346][1], Luca Giamattei[0000−0003−3767−4036][2],
Antonio Guerriero[0000−0002−8104−3832][2], and Roberto Pietrantuono[0000−0003−2449−1724][2]

[1] Vrije Universiteit Amsterdam, The Netherlands
m.dinga@student.vu.nl, i.malavolta@vu.nl
[2] University of Naples Federico II, Naples, Italy
{luca.giamattei,antonio.guerriero,roberto.pietrantuono}@unina.it

Abstract. *Context.* Energy efficiency is gaining importance in the design of software systems, but is still marginally addressed in the area of microservice-based systems. Energy-related aspects often get neglected in favor of other software quality attributes, such as performance, service composition, maintainability, and security.
Goal. The aim of this study is to identify, synthesize and empirically evaluate the energy and performance overhead of monitoring tools employed in the microservices and DevOps context.
Method. We selected four representative monitoring tools in the microservices and DevOps context. These were evaluated via a controlled experiment on an open-source Docker-based microservice benchmark system.
Results. The results highlight: *i)* the specific frequency and workload conditions under which energy consumption and performance metrics are impacted by the tools; *ii)* the differences between the tools; *iii)* the relation between energy and performance overhead.

1 Introduction

In recent years, the motivation to reduce energy consumption by conservation and efficient use has grown significantly. It has become not only a means for gaining control over costs but, most importantly, a way of reducing the carbon footprint of economic and human activity. This is reflected across all industries, including software development [16]. Nevertheless, energy consumption has only recently come to attention in literature [5]. With the advent of microservice-based systems coupled with agile (specifically DevOps) practices, a great focus is put on continuous monitoring: teams need feedback from the system running in the field, in order to get measures about systems performance, security, reliability, to track the status of microservices, to timely detect issues, and act consequently. However, this requires the deployment and operation of (sometimes complex) monitoring tools running alongside the microservices, which in turn might contribute to the overall energy consumed by the system.

This study aims to raise awareness on this matter by assessing the impact on energy consumption and performance overhead of monitoring tools employed in microservice-based systems. We limit the scope to systems running in Docker containers since energy is highly dependent on the platform. This helps to separate its effect from the main factor (i.e., the tools). Docker was chosen because it is one of the most popular container-based virtualization solutions [11].

To achieve our goal, we answer the following research questions. **RQ1**: What is the impact of using different monitoring tools on energy efficiency of Docker-based systems? **RQ2**: What is the impact of using different monitoring tools on performance of Docker-based systems?

We set up an extensive empirical study in which we select 4 monitoring tools run alongside a Docker-based system, and we measure energy consumption at machine level and several performance indicators (CPU, RAM, network, execution time). We then statistically analyze the results to understand the impact of the monitoring tools on both aspects under different conditions.

The contribution and results of this study are relevant to *i)* Docker-based systems developers, as they offer a better understanding on how to integrate monitoring tools within their applications in an energy-efficient manner; *ii)* the tools' maintainers, as they highlighting the impact of their monitoring systems on energy and performance, and showing potential improvements; *iii)* researchers working on microservices and DevOps, as they push toward addressing the problem of an efficient monitoring that has to trade off energy consumption for the need of gathering as much relevant information as possible to ensure quality.

The full replication package of the study is available at https://github.com/S2-group/icsoc-2023-energy-perf-monitoring-docker-rep-pkg.

2 Background

As defined by Fowler and Lewis, the **microservices** architectural style is "an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [1]. **Docker** [11] is one of the most recurrent technologies for implementing microservices [4]. It is a lightweight virtualization platform for packaging software solutions into self-contained and independently-deployed units (i.e., containers). Among others, with Docker, teams can develop and deploy their (loosely-coupled) microservices independently from each other, make faster and more frequent releases, and test their microservices in autononomy.

Being inherently distributed, microservice-based systems require specialized **monitoring tools**, such as Netdata[3], Prometheus[4], etc. Without losing generality, a monitoring tool is typically composed of one or more DBs for storing the collected metrics (e.g., the Time Series Database in Prometheus), a dashboarding platform for querying the DBs and/or showing the collected metrics to the user (e.g., Grafana), an alerting subsystem for sending notifications to the

[3] https://www.netadata.cloud

[4] https://prometheus.io

user (e.g., the rule-based Alert Manager component in Prometheus), and a set of **monitoring components** (e.g., Prometheus exporters) – typically one for each microservice – that collect metrics about their associated microservice (e.g., average CPU usage) and make those metrics available to the DBs. Despite their undoubted usefulness for the observability of the system, the just-mentioned monitoring components do not contribute directly to the functionalities of the system; still, they are deployed together with the microservices being monitored and compete for the same hardware resources. In this study, we are interested in quantifying the overhead that the just-mentioned monitoring components produce in terms of performance and energy consumption of the monitored system.

The collection of performance-related measures (e.g., CPU load, memory utilization, network requests) is relatively straightforward for Docker-based systems, primarily thanks to already-existing monitoring components wrapping Linux utilities such as SAR[5]. Differently, **the measurement of power consumption** (and thus energy) requires extra effort and technical skills. Some well-known tools for monitoring energy are PowerPack (physical measurement), RAPL (software-based measurement), or PowerAPI (software library).In this study, we favour accuracy and measure the power consumption at machine-level using a well-known physical power meter called *Watts Up Pro* [8].

3 Related Work

To our knowledge, there is no comprehensive study about the energy consumption of monitoring tools in the context of microservices.

Heward et al. [7] look into the performance impact of service monitoring for web applications. They explore various architecture designs for monitoring the web traffic. One conclusion is that a colocated proxy used for monitoring is much more efficient than a proxy located on a different machine. Similar to our study, this paper assesses the impact of monitoring tools, however it does not consider their impact on energy efficiency.

Foutse et al. [9] assess the impact of cloud patterns on performance and energy consumption and provide a series of guidelines for implementing energy-efficient cloud-based applications. Their results focus around the environmental impact of microservice-based systems, showing that migration to a microservices architecture can improve performance, while reducing energy consumption. The study does not take into account monitoring.

A related study [10] investigates the use of SmartNIC's, a low-power processor, for improving server energy-efficiency without latency loss, in the context of microservices. They propose E1, an execution platform for SmartNIC-accelerated servers, which, according to the authors, can significantly improve cluster energy-efficiency up to 3×, with minimal latency cost, for common microservices. The paper focuses on improving the energy efficiency of microservice-based systems, however it does not take into account the potential overhead of monitoring tools.

[5] https://linux.die.net/man/1/sar

Santos et al. [14] compare the energy consumption of applications running in Docker containers to those running bare-metal. The authors demonstrate that Docker increases the energy consumption even if the system under test is idle. The effect is caused by the activity of the Docker daemon (dockerd), a service that permanently runs on the host and orchestrates the containers. While this paper focuses on the energy footprint of Docker, our study explores how various monitoring tools running alongside Docker-based systems impact energy.

4 Study design

4.1 Experimental subjects: *Monitoring tools*

Tools selection. We searched for open-source monitoring tools on GitHub, looking at the ones reporting DevOps as topics in their description. The tools have been selected according to the following requirements:

– Compatible with microservices applications running in Docker containers.
– Do not require integration at the application level (i.e. code instrumentation), and in general aiming to:
  - avoid introducing unnecessary confounding variables, such as communication overhead due to interaction between additional components,
  - avoid increasing the deployment of the integrated applications,
  - aid replication of the experiments.

– Capable to collect metrics at container level.

Table 1: Monitoring tools

Monitoring Tool		
ELK Stack	**Website**	https://www.elastic.co/elastic-stack/
	Github	https://github.com/elastic/elasticsearch
	First and last release*	2/8/2010 and 07/29/2022
	Stars on Github*	60,700
Netdata	**Website**	https://www.netdata.cloud/
	Github	https://github.com/netdata/netdata
	First and last release*	9/26/2015 and 08/11/2022
	Stars on Github*	60,300
Prometheus	**Website**	https://prometheus.io/
	Github	https://github.com/prometheus/prometheus
	First and last release*	2/25/2015 and 08/13/2022
	Stars on Github*	43,800
Zipkin	**Website**	https://zipkin.io/
	Github	https://github.com/openzipkin/zipkin
	First and last release*	6/3/2016 and 1/27/2022
	Stars on Github*	15,600

*The date of access for the most recent release and for the number of stars on Github is 8/15/2022.

With these requirements, we aim to aid the replicability of the experiments and facilitate the interpretation of the results avoiding biases. Then, we ranked the tools in terms of stars and selected the top four, reported in Table 1. Specifically, the ranking included three metric-based tools (ELK Stack, Netdata, and Prometheus) and one tracing-based tool (Zipkin).

Benchmark system. As part of the setup, we select a well-known Docker-based microservice application, TrainTicket [6] (TTS) [17], and integrated it with the monitoring tools. TTS is a medium-size benchmark system containing 24 microservices related to business logic, out of 41 in total, implemented in different languages. It has been previously used in several experimental studies and is representative for industrial multi-container Docker applications through to its size, granularity, and variety of microservices [3].

Integration. The integration of TTS with the monitoring tools follows the most basic configuration and deployment described in the documentation of the tools. ELK stack is integrated using Metricbeat[7], a lightweight shipper for host and service metrics. Metricbeat is deployed directly on the host and it monitors all of the deployed Docker containers. Metrics are stored in Elasticsearch and can be visualised in Kibana - both running in separate Docker containers. Frequency is the time interval (in seconds) at which metrics are sent to the Elasticsearch cluster. A snapshot of Metricbeat metrics is generated every second for high level, 5 seconds for medium level and 10 seconds for low level.

Netdata, similar to Metricbeat, has an agent that discovers all available control groups (cgroups) on the host system and collects their metrics. The collection frequency has the same progression as for ELK stack (1/5/10 seconds).

Prometheus is integrated with cAdvisor (Container Advisor)[8] to monitor the running containers. cAdvisor has the same approach as Metricbeat and Netdata – it gathers container metrics, such as CPU and memory through cgroups. Frequency is configured the same way (1/5/10 seconds) as the previous two tools.

As for Zipkin[9], the integration with the TTS is made with Java Sleuth[10]. It is configured using the *PercentageBasedSampler*, i.e., only a given proportion of traces are stored. The frequency is changed using probabilistic sampling: only a configurable percent of the traces are processed and stored. The setting for high is 100%, for medium 50%, and for low 25%. Further details about the integration are in the replication package: each integration has its own *Compose* file defining the services, networks, and volumes required to run the tools alongside the TTS.

4.2 Goal and research questions

The goal of the experiment is expressed via the Goal-Question-Metric approach [2]: to *analyze monitoring tools, for the purpose of evaluation, with respect to their energy and performance overhead, from the point of view of developers and tool maintainers, in the context of Docker-based systems.* The RQs are:

RQ1: What is the impact of using different monitoring tools on the energy efficiency of Docker-based systems?

RQ2: What is the impact of using different monitoring tools on the performance of Docker-based systems?

[6] https://github.com/FudanSELab/train-ticket

[7] https://www.elastic.co/beats/metricbeat

[8] https://github.com/google/cadvisor

[9] https://zipkin.io/

[10] https://spring.io/projects/spring-cloud-sleuth

Several performance indicators are considered pertaining to resource consumption and execution time, which will be analyzed individually. These are: *percentage of CPU utilization* while running at user level; *load average*, computed as the average number of runnable or running tasks (R state), and the number of tasks in uninterruptible sleep (D state) over the last minute; *percentage of used memory (RAM)*; *number of input datagrams successfully delivered per second* to IP user-protocols, and *total number of input datagrams received per second*, including those received in error; *execution time* in seconds.

4.3 Experiment variables

To mitigate the mono-operation bias and to accurately represent the runtime overhead of the tools, we consider the following **independent variables**.

Monitoring tool, five levels: the *baseline*, where we run the TTS deployed without any monitoring tool, plus the above-mentioned tools, Elasticsearch, Netdata, Prometheus, and Zipkin. The tool is deployed along with the TTS.

Frequency: the scrape interval, in the case of tools that collect metrics (Elasticsearch, Netdata, Prometheus), and the sampling interval, in the case of the tracing tool (Zipkin). It is treated as a blocking factor with three levels (high, medium, low). Ratio measures are transformed to ordinal ones based on the minimum allowed scrape interval and maximum allowed sampling rate among the tools. Based on this, level "high" corresponds to 1 second for metric collection tools and 100% sampling rate for tracing tools, level "medium" is 5 seconds and 50% and level "low" is 10 seconds and 25%, respectively.

Workload: the number of virtual users that stress the system during the test. It is treated as a blocking factor with three levels (high, medium, low). The mapping to ordinal scale considers the capabilities of the system as follows: level high corresponds to the highest number of users supported such that the tests are completed successfully (Table 2).

Deployment: the strategy used for deploying the system. This factor is fixed, in order to separate its effect from the main factor. The monitoring tools, next to the TTS are deployed on a single Ubuntu machine using Docker Compose V2 for running the containers on Docker platform.

The **dependent variables** are: **Energy efficiency** (total energy consumed (Joules) by TTS during a load test), and the above-defined performance metrics (**CPU usage**, **CPU load**, **RAM usage**, **Network traffic**, **Execution time**). The null hypotheses for RQ1 and RQ2 state that a dependent variable does not significantly differ when using different monitoring tools. The proper hypothesis

Table 2: Ratio values corresponding to treatments for every monitoring tool

Tool	**Frequency**			**Workload**		
	Low	**Medium**	**High**	**Low**	**Medium**	**High**
ELK Stack	10s	5s	1s	10	20	40
Netdata	10s	5s	1s	10	20	40
Prometheus	10s	5s	1s	10	20	40
Zipkin	25%	50%	100%	10	20	40

tests will be used depending on the data characteristics. Table 2 shows the ratio values for the co-factors (frequency and workload).

4.4 Experiment design

We alternate every possible combination (4 monitoring tools plus the baseline, 3 frequency levels, and 3 workload levels) of all of the levels across all independent variables, following a 5x3x3 full factorial design. We do not consider frequency in the case of the baseline treatment, since it does not apply in that case. This means we only have 3 runs for the baseline, for 3 levels of workload, leading to 39 trials in total, i.e., (5x3x3)-6. We aim to keep the monitoring tool effect at the core of the experiment, while also considering frequency and workload as factors that might influence energy efficiency. In order to mitigate their effect and to ensure an unbiased assignment, we analyze each combination of the co-factors separately, resulting in 9 different blocks. The results might differ depending on how energy and performance are affected at runtime when running the experiment under different frequency and workload conditions.

Each of the 39 runs is repeated 10 times and in randomized execution order, to mitigate the potential bias caused by the order in which tools are run.

4.5 Experiment execution

Testbed. The experiment is performed on a machine with a 64-bit Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz octa-core processor, 32 GB RAM, running Ubuntu Server 18.04 as operating system, which runs TTS and the monitoring tool. The server is fully dedicated to this experiment to reduce the chances of external factors contributing to the energy and performance measurements. The scripts orchestrating and running the experiments and the results of the energy measurements are run on two further separate machines to avoid bias.

Metrics collection. For energy measurements collection, we opted for measuring energy at machine level using a physical power meter. Specifically, the *Watts Up Pro* power meter is used to collect power measures from the monitored server, in watts (W), at one second intervals, then used to compute energy (J). For performance measurements, we use SAR, a system utility allowing for monitoring the resources of a Linux system, again with a one-second interval.

Experiment execution. Each run has a profiling time of 13.7 minutes on average, which may vary depending on the execution time of the load test. We add 3 minutes idle time between consecutive runs to guards against carryover effects (consecutive runs influencing each other) [15] and 10 minutes, for system initialization, which leads to 26.7 minutes to complete one run. We perform 10 runs for every trial (39 trials), resulting in 390 runs in total, executing for 10,413 minutes (more than 7 days). We set the execution time for a run to be at least 10 minutes, taking into account the frequency of 1 second at which Watts Up Pro collects energy measurements. This results in at least 600 measurements for a run, which allows to accurately compute the energy efficiency.

We orchestrated the experiment using Experiment Runner[11], a Python-based framework for automatically executing experiments targeting software systems. For each run, these steps are performed: (i) deploy of TTS along with the chosen monitoring tool, (ii) start monitoring energy and performance (with Watts Up Pro and SAR), (iii) interact with the system by triggering a load test script, (iv) stop monitoring once the load test has completed, (v) stop all processes related to TTS, or to the monitoring tool running alongside, (vi) clean up the system by removing all unused local volumes and restarting Docker Engine.
Workload. The load test script was obtained by merging together a set of scripts generated with K6[12], an open-source load testing tool. K6 can generate scripts for performance testing starting from the Swagger/OpenAPI specification of the REST APIs. We obtain 34 scripts for each of the 34 microservices which are integrated with Swagger. The scripts are included in the replication package of the study. Each of the scripts is stressing a different microservice by interacting with its API. Since the requests propagate through the entire system, the 7 remaining microservices which are not directly tested are also interacted with.

The 34 scripts are merged together into a single load test script which will be used during a run to stress the entire system. We perform multiple iterations of this script, with several virtual users (10, 20 and 40), to ensure that the duration of a run is at least 10 minutes. On average, each user performs the same amount of work (i.e., 34.5 iterations of the load test script in one run). The replication package contains: (i) the raw measures, (ii) the scripts for data processing and analysis and (iii) the scripts to automate the experiment execution.

5 Results

5.1 Results on energy efficiency (RQ1)

Figure 1 reports the energy consumed by the compared tools, with values ranging from 38,552 to 88,516 Joules. The coefficient of variation is between 21.3% and 26.8% and the standard deviation shows that the data is relatively

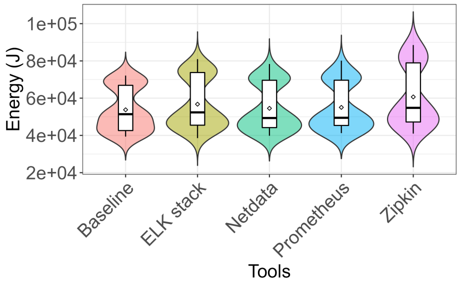

Fig. 1: Energy efficiency across monitoring tools

[11] https://github.com/S2-group/experiment-runner
[12] https://k6.io/

disperse (13,453 globally), which most probably comes from the difference among the frequency-workload blocks. Considering the mean values (the diamond in the box plot), there is a visible difference between the baseline (53,755 J) and running a tool alongside the TTS (54,543 J, 55,046 J, 56,760 J, 60,668 J, respectively for Netdata, Prometheus, ELK Stack, Zipkin). As expected, the tools have a footprint on energy, with Netdata being the most energy-efficient tool and Zipkin the least one. Median values (the bars in the plot) are slightly lower but confirm the ranking. The distributions are similar to each other, with Zipkin having the highest variance (ranging from 38,552 J to 88,516 J). All the distributions are highly bimodal, with two separate groups, suggesting an impact of the blocking factors, frequency and workload. Figure 2 reports the results by block, showing that Zipkin consumes more than other monitoring tools when the workload is high.

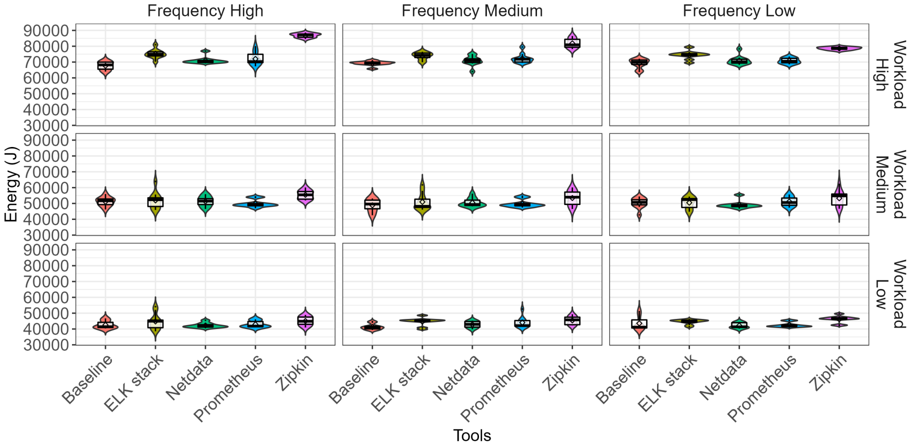

Fig. 2: Energy efficiency across all frequency and workload level combinations

To statistically analyze the data, we first run Shapiro-Wilks test to check for normality for each of the 9 blocks. Results are in Table 3, SW column, with significance level $\alpha = 0.05$. The p-value for testing the null hypothesis, stating that the energy sample is drawn from a normal distribution, is lower than 0.05 for

Table 3: Results of Shapiro-Wilk (SW) and Kruskal-Wallis (KW) tests for each frequency (F) and workload (W) block. Bold text denotes a significant difference ($\alpha = .05$)

Block	SW (p-value)	KW (p-value)	η^2	η^2 interpretation
F Low, W Low	**0.00113**	**0.00156**	0.3	large
F Low, W Medium	0.0939	0.21	0.0413	small
F Low, W High	**0.0157**	**3.77e-06**	0.59	large
F Medium, W Low	**0.0172**	**0.00157**	0.299	large
F Medium, W Medium	**0.019**	0.303	0.0189	small
F Medium, W High	**0.00228**	**1.17e-06**	0.645	large
F High, W Low	**9.25e-05**	0.154	0.0594	small
F High, W Medium	0.0826	**0.022**	0.165	large
F High, W High	**9.52e-05**	**4.58e-07**	0.69	large

7 blocks out of 9, even after applying various data transformations (logarithmic, reciprocal, square root and exponential). We conclude that data are mostly not normally distributed for energy, hence we proceed with non-parametric statistical tests. Specifically, we apply the Kruskal-Wallis test to determine if at least one of the monitoring tools differ from the others. The p-values are lower than the $\alpha = 0.05$ (Table 3, column KW) for 6 out of 9 blokcs. It means that for those blocks a significant difference in energy efficiency among monitoring tools is detected. The magnitude of variability in energy efficiency attributable to the monitoring tools, computed as the eta-squared statistic [12], η^2, is generally large, for all the statistically significant results (Table 3, last two columns).

Table 4: Results of the Wilcoxon test - frequency (F) and workload (W) combination of treatments (block). Bold text denotes a significant difference ($\alpha = .05$)

Tool	**Block**	**p-value**	**Cliff's δ**	**δ interpretation**
ELK stack	F Low, W High	**0.002**	-0.86	large
	F Medium, W Low	**0.015**	-0.72	large
	F Medium, W High	**0.733e-03**	-0.94	large
	F High, W High	**0.825e-03**	-0.96	large
Netdata	F High, W High	**0.014**	-0.68	large
Prometheus	F Medium, W Low	**0.015**	-0.70	large
	F High, W High	**0.026**	-0.60	large
Zipkin	F Low, W High	**0.458e-03**	-1.00	large
	F Medium, W Low	**0.009**	-0.84	large
	F Medium, W High	**0.458e-03**	-1.00	large
	F High, W High	**0.825e-03**	-1.00	large

Table 5: Dunn test ($\alpha = 0.05$) per block. **grey**: not significant , **green**: significant

vs		*Netdata*									*Prometheus*									*ELK stack*								
Prometheus		▨	▨	▨	▨	▨	▨	▨	▨	▨																		
ELK stack		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨									
Zipkin		🟩	▨	🟩	▨	▨	🟩	▨	▨	🟩	🟩	▨	🟩	▨	▨	🟩	▨	▨	🟩	▨	▨	▨	▨	▨	🟩	▨	▨	🟩
Config.	Workload	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*	*L*	*M*	*H*
	Frequency	*L*	*L*	*L*	*M*	*M*	*M*	*H*	*H*	*H*	*L*	*L*	*L*	*M*	*M*	*M*	*H*	*H*	*H*	*L*	*L*	*L*	*M*	*M*	*M*	*H*	*H*	*H*

As a significant difference between the tools exists, we perform a pairwise comparison between each monitoring tool and the baseline, by applying the Wilcoxon test across all blocks with Benjamini-Hochberg (B-H) correction for multiple-comparison protection. There is significant difference for every tool for at least one block (Table 4). The Cliff's delta for those blocks shows a large effect according to the interpretation by Romano [13]. Also, Table 5 reports the Dunn's test comparing each tool with each other. Both these tests confirm that Zipkin is the most-consuming tool, followed by ELK stack.

Result 1 - Monitoring tools significantly impact the energy efficiency of Docker-based systems, under several (6 out of 9) frequency and workload conditions. Not all tools have the same impact; Zipkin has the largest negative impact. A high workload contributes markedly to high consumption of all the tools, and exacerbates the difference between the tools.

5.2 Results on performance (RQ2)

Figure 3 reports performance. The replication package contains Tables with summary statistics of these results. The CPU usage percentage is 52.4% on average globally, with the lowest recorded values for ELK stack (49.0%) and the highest for Zipkin (57.6%). Apart from Zipkin, there seems to be a negligible difference between the baseline and the tools. For the CPU load average, Zipkin still shows the highest value, 18.4, while Netdata and Prometheus show the lower ones (16.6). RAM usage has a very low standard deviation. The mean is 70.7%, with the minimum average value for the baseline (65.1%) and the highest for ELK stack (99.2%). This means that ELK stack has a very high memory footprint, keeping the RAM usage close to the maximum capacity throughout the execution of the load test. This phenomenon might be an indication of the high energy consumption of ELK stack, where we obtained a significant difference for 4 out of 9 blocks (Table 4). Also, Zipkin tends to be more memory- and CPU-intensive than other monitoring tools like Netdata and Prometheus, and also in this case Zipking was shown to consume more energy than the baseline for 4 out of 9 blocks. In terms of execution time, looking at the mean values, it is not highly impacted by tools such as Netdata and Prometheus. ELK stack and Zipkin however have the highest execution time on average. A run has a duration of 13.7 minutes on average, with the minimum being 10 minutes (for Prometheus) and a maximum of 18 minutes (for Zipkin).

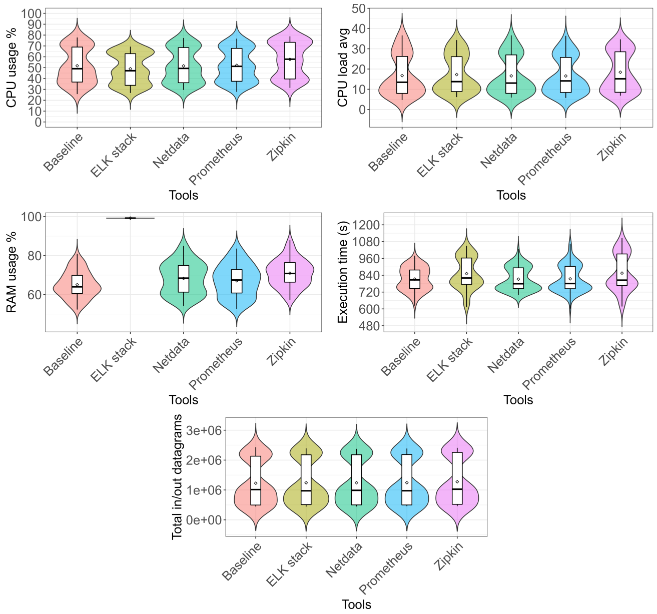

Fig. 3: Dependent variables across monitoring tools

Table 6: Results of the Shapiro-Wilk test for each frequency (F) and workload (W) block. Bold text denotes a significant difference ($\alpha = .05$)

Block	CPU	Memory	Network	Load	Exec. time
F Low, W Low	**0.000412**	**1.07e-05**	**0.0015**	0.178	**2.81e-05**
F Low, W Medium	**0.000864**	**9.09e-06**	0.298	**3.88e-06**	0.585
F Low, W High	**0.014**	**1.95e-05**	0.323	0.575	0.0807
F Medium, W Low	0.435	**1.02e-05**	**1.94e-05**	0.212	**7.77e-05**
F Medium, W Medium	**0.00248**	**1.01e-05**	0.114	**3.68e-07**	**0.00395**
F Medium, W High	0.388	**8.72e-05**	0.533	0.639	0.59
F High, W Low	0.693	**5.94e-06**	**0.00158**	0.657	**0.000204**
F High, W Medium	**0.0466**	**1.71e-05**	0.066	**0.000453**	0.518
F High, W High	0.183	**5.43e-05**	**0.0041**	0.111	0.0317

Result 2 - Monitoring tools like Zipkin (for CPU) and ELK Stack (for RAM) increase the resources' usage and affect the execution times more than other tools. The same result was observed with total energy consumed.

The mean network traffic is similar for all the tools. This is expected, since the same amount of traffic is likely generated while running the load test. The distributions in all the cases except for RAM are bimodal, highlighting the variability among the different blocks.

The Shapiro-Wilk normality tests (Table 6) show that in most cases data are not normally distributed for any of the dependent variables.

As before, we run the Kruskal-Wallis test for each variables to detect a possible difference between the monitoring tools on performance. For CPU usage, CPU load, and RAM usage, the obtained p-values are significant forall the blocks. The magnitude of the difference attributable to the tool (eta-squared statistic) is *large* in 9 out 9, 2 out of 9, and 9 out 9 cases for CPU usage, CPU load, and RAM usage, respectively. For network traffic and execution time, the difference is significant only in 2 and 4 cases, respectively. The Tables per variable with KW p-values and η^2 results are in the replication package.

The Wilcoxon test across all blocks, with the B-H correction, compares each tool against the baseline. Results are hereafter summarized:

- **CPU usage**: In general, for ELK stack and Zipkin there is a significant impact of monitoring tools on CPU usage. The trend is more pronounced under a high workload. The p-values for 3 blocks allow to reject the null hypothesis that the median difference between the baseline and ELK stack is zero. The same stands for 6 blocks regarding Zipkin. This confirms the previous observations. Also, the Cliff's delta reveal always a large effect when the p-values are significant.
- **CPU load**: For Zipkin and Netdata there is at least one block where the impact on CPU load is significant. This means that these tools can influence CPU load under specific frequency and workload conditions.
- **RAM usage**: Except for Netdata, there is statistical significance for every tool, for at least one block. The Cliff's delta effect size is large for all these

blocks but one (on Prometheus). The results for ELK stack further confirm the previous observations that ELK stack has a very high memory footprint (under all workload-frequency conditions, RAM usage is close to 100%).

- **Network traffic**: In general, there is no statistical significance for network traffic (only in one block for Netdata, Prometheus and Zipkin the difference is significant). This is expected, as running a monitoring tool should not influence the network traffic generated by running the load test.
- **Execution time:** In case of ELK stack and Zipkin there is statistical significance and Cliff's delta estimates show large effect size. Again, a high workload increases the impact and exacerbates the differences: the longest execution time is for Zipkin under the high workload-high frequency configuration (more details in the replication package plots). For Netdata and Prometheus, there are no statistically significant results.

Result 3 - Monitoring tools influence the CPU usage, CPU load, RAM usage and execution time, under specific frequency and workload conditions. Zipkin and ELK stack have the largest impact, exacerbated when running under high workload conditions.

Table 7: Correlation coefficients

	CPU usage	CPU load	RAM usage	Execution time	Network
Pearson	**0.859**	**0.899**	**0.187**	**0.867**	**0.961**
Spearman	**0.829**	**0.851**	**0.290**	**0.796**	**0.954**

Finally, we analyze each dependent variable in relation to energy efficiency (Table 7). CPU usage, load average and execution time are strongly correlated with energy efficiency. Although this does not imply causation, their reduction can potentially improve energy efficiency. However, there is a relatively small correlation between RAM usage and energy efficiency. This also explains why ELK stack is not worse than Zipkin in terms of energy consumption, despite its intensive use of RAM. The correlation between network traffic and energy efficiency is also high, primarily due to the presence of 3 groups each corresponding to a level of workload – high network traffic leads to the highest energy values, while low network traffic leads to the lowest values.

Result 4 - CPU usage, CPU load average and execution time are strongly correlated with energy efficiency.

6 Threats to validity

Internal validity. To mitigate the *history* threat, where events occurring at the same time a treatment is applied could produce the effect, we repeat each run 10 times. Also, to avoid order effects, execution order is randomized. To avoid

carryover effects (i.e., consecutive treatments influencing each other), we stop the running systems and wait for 3 minutes before starting the next run. We also alternate the baseline with the other treatments, as a best practice to be able to verify any noticeable changes in absence of intervention. We address the *ambiguous temporal precedence* threat by ensuring replication of the exact sequence of independent variables manipulation, thanks to the Experiment Runner tool.
External validity. We select well-known monitoring tools for microservices, considering their popularity on GitHub. Clearly, different tools might impact energy and performance differently. To mitigate this risk, we applied a minimalistic setup, following the documentation to avoid introducing unnecessary confounding variables (due to additional components, for instance). We compared the tools on a widely used microservice benchmark application, Train Ticket; though, using another application would lead to different results. Future replications of this experiment will help in mitigating this potential source of bias.
Construct validity. We are confident about the integration of the monitoring tools, as we carefully tested each of them running alongside the system before running the experiment. The implementation is publicly available. Also, hardware power meters, like the one we used, are known to have high accuracy and do not influence the measured system [6].

7 Conclusions and Future Work

In this study we conducted an empirical assessment of the energy and performance overhead of monitoring tools on Docker-based systems. We obtained significant results in terms of energy and performance (CPU usage, CPU load, RAM usage, network traffic and execution time), under specific frequency and workload conditions. Not all the tools impact energy efficiency and performance in the same way, but we observed a high energy consumption and a high CPU, RAM and execution time for the same tools. The correlation analysis confirms the association for CPU and execution time, but not for memory, hence the latter is likely to have a smaller impact on energy. For a more granular analysis, to be able to detect energy hotspots in monitoring tools, we plan to deploy a software power meter in a future iteration, such as SmartWatts[13], that measures energy at container level.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

[13] https://powerapi-ng.github.io/smartwatts.html

References

1. Microservices (Jun 2023), https://martinfowler.com/articles/microservices.html
2. Basili, V.R.: Software Modeling and Measurement: The Goal Question Metric Paradigm. Computer Science Technical Report Series, CS-TR-2956 (1992)
3. Cortellessa, V., Di Pompeo, D., Eramo, R., Tucci, M.: A model-driven approach for continuous performance engineering in microservice-based systems. Journal of Systems and Software **183**, 111084 (2022)
4. Di Francesco, P., Lago, P., Malavolta, I.: Architecting with microservices: A systematic mapping study. Journal of Systems and Software **150**, 77 – 97 (2019)
5. Ergasheva, S., Khomyakov, I., Kruglov, A., Succi, G.: Metrics of energy consumption in software systems: a systematic literature review. IOP Conference Series: Earth and Environmental Science **431**, 012051 (02 2020)
6. Fahad, M., Shahid, A., Manumachu, R.R., Lastovetsky, A.: A comparative study of methods for measurement of energy of computing. Energies **12**(11), 2204 (2019)
7. Heward, G., Müller, I., Han, J., Schneider, J.G., Versteeg, S.: Assessing the performance impact of service monitoring. In: ASWEC 2010. pp. 192–201. IEEE (2010)
8. Hirst, J.M., Miller, J.R., Kaplan, B.A., Reed, D.D.: Watts up? pro ac power meter for automated energy recording (2013)
9. Khomh, F., Abtahizadeh, S.A.: Understanding the impact of cloud patterns on performance and energy consumption. J. Syst. Softw. **141**, 151–170 (2018)
10. Liu, M., Peter, S., Krishnamurthy, A., Phothilimthana, P.M.: E3: Energy-efficient microservices on smartnic-accelerated servers. In: USENIX. pp. 363–378 (2019)
11. Merkel, D., et al.: Docker: lightweight linux containers for consistent development and deployment. Linux j **239**(2), 2 (2014)
12. Pierce, C.A., Block, R.A., Aguinis, H.: Cautionary note on reporting eta-squared values from multifactor anova designs. Educ. Psychol. Meas **64**(6), 916–924 (2004)
13. Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J.: Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: FAIR. vol. 177, p. 34 (2006)
14. Santos, E.A., McLean, C., Solinas, C., Hindle, A.: How does docker affect energy consumption? evaluating workloads in and out of docker containers. Journal of Systems and Software **146**, 14–25 (2018)
15. Vegas, S., Apa, C., Juristo, N.: Crossover designs in software engineering experiments: Benefits and perils. IEEE Trans. on Softw. Eng. **42**(2), 120–135 (2015)
16. Verdecchia, R., Lago, P., Ebert, C., de Vries, C.: Green it and green software. IEEE Software **38**(6), 7–15 (2021)
17. Zhou, X., Peng, X., Xie, T., et al.: Benchmarking microservice systemsforsoftwareengineeringresearch. In: 40th ACM/IEEE Int. Conf. Softw. Eng. (ICSE) (2018)