# An empirical study on the Performance and Energy Consumption of AI Containerization Strategies for Computer-Vision Tasks on the Edge

Raluca Maria Hampau[1], Maurits Kaptein[2], Robin van Emden[3], Thomas Rost[3], Ivano Malavolta[1]
[1]Vrije Universiteit Amsterdam, The Netherlands
[2]Tilburg University, The Netherlands
[3]Scailable, The Netherlands
r.hampau@student.vu.nl, m.c.kaptein@tilburguniversity.edu,
robin.van.emden@scailable.net, thomas.rost@scailable.net, i.malavolta@vu.nl

## ABSTRACT

***Context***. From healthcare to agriculture and manufacturing, Artificial Intelligence (AI) is encountered everywhere. The rise of use cases of AI catered towards the Edge, where devices have limited computation power and storage capabilities, motivates the need for better understating of how AI performs and consumes energy.
***Goal***. The aim of this paper is to empirically assess the impact of three different AI containerization strategies on the energy consumption, execution time, CPU, and memory usage for computer-vision tasks on the Edge.
***Method***. In this paper we conduct an experiment with the used containerization strategy as main factor, with three treatments: ONNX Runtime, WebAssembly, and Docker. The subjects of the experiment are four widely-used computer-vision algorithms. We then orchestrate a series of runs where we deploy the four subjects on different generations of Raspberry Pi devices, with different hardware capabilities. A total of 120 runs (per device) are recorded to gather data on energy, execution time, CPU, and memory.
***Results***. We found a statistically significant difference between the three containerization strategies on all dependent variables. Specifically, WebAssembly proves to be a valuable alternative for devices with reduced disk space and computation power.
***Conclusions***. For computer-vision tasks with limited disk space and RAM memory requirements, developers should prefer WebAssembly for deployment. The (non-dockerized) ONNX Runtime resulted to be the best choice in terms of energy consumption and execution time.

## 1 INTRODUCTION

Artificial Intelligence (AI) and Internet of Things (IoT) are two of the most prominent areas in technology research and innovation, and are predicted to grow in the future [4]. Combining AI and IoT results in a paradigm called Artificial Intelligence of Things (AIoT) [37] which comes with its own set of challenges and opportunities. The common belief nowadays is that large amounts of data and powerful machines are required to run AI applications. AI has become increasingly popular in our daily lives and is beginning to shape various sectors in the industry. Different areas related to AIoT in today's society (e.g., healthcare [10] and manufacturing [26]) need specific solutions, optimizing for how much computation power and storage space is consumed and needed.

It is important to assess the potential implication of the fast-paced development of AI on our society, in a scenario where most interest in the subject of AI focuses towards improving the predictive accuracy of models [33]. The Edge Computing paradigm [32] has emerged to mitigate some of the shortcomings caused by using a cloud-based approach for IoT deployments: privacy concerns, limitations caused by lack of connectivity and possible latencies. In this context, Artificial Intelligence on the Edge [11] consists in building an AI model (model training and prediction/inference) at the edge of the environment on resource-scarce devices (such as sensors). AI applications on the Edge have become increasingly popular, mostly because of their high performance, privacy preservation, and potential independence from network connectivity [22]. In this paper we consider the configuration where an already trained model is deployed on the target machine. To that extent, when having a trained model, a developer can leverage different containerization strategies to perform inferences. In AI, an inference engine is a component that applies logical rules to gain new information from the trained knowledge base. In this context, Scailable[1] is a software startup providing a concrete implementation of the principle *train once – deploy everywhere*, where model training and model prediction are considered as completely independent phases of the lifecycle of an AI model [17]. In this paper we focus on *computer-vision tasks* (e.g., object detection, image identification, image classification) since they are comparable with each other and they are widespread across different application domains.

The **goal** of this study is to investigate the impact on performance (i.e., execution time, CPU usage, and memory consumption) and energy consumption of containerization strategies for AI-based computer-vision models on the Edge. To accomplish our goal, we run pre-trained models on physical devices, comparing the collected measures for three deployment strategies: the ONNX Runtime, Scailable's WebAssembly (WASM) engine, and Docker.

The experiment is conducted on two RaspberryPi devices (model 3B+ and model 4) and consists in 120 runs (per device) = 4 (models) x 3 (containerization strategies) x 10 (repetitions). The results present

[1]https://scailable.net


*EASE 2022, June 13–15, 2022, Gothenburg, Sweden*

ACM ISBN 978-1-4503-9613-4/22/06.
https://doi.org/10.1145/3530019.3530025

significant differences for all metrics between at least two of the measured containerization strategies. We observe that the ONNX Runtime and its dockerized version are best for energy consumption and execution time, while Scailable's WASM engine uses the lowest amount of memory and disk space.

The **main contributions** of this study are: an empirical investigation on the performance and energy consumption of off-the-shelf AI deployment strategies, a discussion of the obtained results, and the replication package of the study containing all data, the source code for running the experiment, and the scripts for data analysis. The target audience of this study are AI/ML researchers and software engineers interested in the sustainability aspects of AI development and its potential impact on the environment.

## 2 BACKGROUND

**AI-based Computer vision on the Edge**. In Computer vision, *inference* consists in an analysis (as a human) of visual data that leads to information on what the said data contains. It is considered a valuable area for research in Artificial Intelligence and in Computer Science in general, given the dynamics of the market [1]. AI on the Edge comes with its challenges, opportunities and a novel set of use cases catered to environments with bandwidth and latency constraints [2]. Computer vision includes tasks such as object detection, image identification and image classification. In healthcare [10], we encounter application providing cancerous moles detection in skin images or finding symptoms in x-ray and MRI scans. In agriculture, Edge AI could be employed for plant health monitoring [30]. In manufacturing, Edge sensor devices can evaluate whether a component is defective [26].

In the remainder of this section we give an overview of each deployment strategy so to provide additional context to the reader.
**ONNX Runtime**. ONNX is an *open ecosystem for inter-operable AI models*, which means that it enables better collaboration between well-known AI frameworks and tools. Given its open-source nature, as well as its support and performance in the field of ML [3], the ONNX Runtime is considered a good candidate for this study. This containerization strategy has no additional layers apart from the inference engine, so we use it as the baseline containerization strategy in our experiment. With the arrival of Microsoft's ONNX Runtime, a high-performance inference engine is now available to make use of the the ONNX format. An *.onnx* file of a trained model contains the needed information to be able to switch between platforms and instantiate a full data processing pipeline (via the ONNX Runtime). To perform predictions on Edge devices, developers simply need the *.onnx* file and the input data.
**Docker**. Docker is an open-source containerization solution ensuring a consistent and easily portable environment (container) to enhance the overall collaboration and development experience [25]. These features, as well as its already wide usage, not only in the AI/ML field, but in general in computer science and industry [5] make it a valuable approach for the experiment. In this study we deploy an exact replica of the ONNX Runtime configuration (the out-of-the-box installation of the *onnxruntime* Python package) mentioned in the previous section inside a Docker container. While execution performance is expected to be similar [36] (in spite of adding a virtualization layer), a difference in power consumption might result in a potential drawback of this deployment scenario.
**WebAssembly**. WebAssembly is a binary instruction format designed as a portable compilation target [14]. While being more frequently employed in most modern browsers, there are a number of *standalone runtimes* that provide software containerization on any target CPU architecture. Scailable promises faster execution time and smaller memory footprint when separating training from prediction generation [17]. We use Scailable to compile ONNX models into WASM binaries. The platform toolchain provides easy blackbox conversion of a fitted model from *.onnx* format to a dedicated, fully portable and highly optimized *.wasm* binary. Thus, the Scailable engine used in the experiment is essentially a small footprint, standalone pre-compiled runtime for an ARM64 CPU, based on wamr[2]. The engine supports both *.wasm* and *.aot* files (equivalents of the *.onnx* files mentioned before), and we used the wamr compiler to convert our *.wasm* files to their *.aot* version to ensure best possible performance throughout the measurement. The motivation behind this choice is that Ahead-of-Time (AOT) compilation turns the WASM bytecode to a native module and decreases the amount of computation necessary during run time by performing a series of optimizations.

It is important to note that in this experiment we use (i) the off-the-shelf configuration of each deployment strategy and (ii) the default setup of the Python API for performing inferences, which leverages the CPU as an execution provider. As a consequence, there are some relevant differences between how the three treatments interact with the processor of the Raspberry Pis, specifically: (i) ONNX Runtime and Docker use all available 4 cores of the Raspberry Pis, whereas WebAssembly uses only one core by default and (ii) ONNX Runtime and Docker use Single instruction, multiple data (SIMD) instructions, which are known to improve the performance of parallel computation, also for vision-based tasks [20], whereas WebAssembly does not use SIMD. We are aware that an ad-hoc configuration of each of the three treatments could have lead to different results, but in this experiment we aim at being as representative as possible of the most recurrent conditions where computer-vision tasks operate, instead of investigating on the optimal configuration for each of containerization strategy (which would require a completely different study design). Thus, we decided to favour representativeness by leaving the three containerization strategies with their default configuration across all runs.

## 3 EXPERIMENT DESIGN

The replication package[3] of the study contains: (i) the raw measures; (ii) the R scripts for data analysis; (iii) the scripts to automatize the experiment execution; (iv) the subjects of the experiment.

### 3.1 Goal and research questions

The **goal** of the experiment is to analyse AI containerization strategies for the purpose of evaluating how they impact energy consumption and performance from the point of view of AIoT developers and researchers, in the context of computer-vision tasks in Artificial

[2]https://github.com/bytecodealliance/wasm-micro-runtime
[3]https://github.com/S2-group/EASE-2022-energy-ai-edge-containers-rep-pkg

Intelligence on Edge. To achieve such goal we answer the following research questions.

**RQ1: What is the impact on energy consumption of using different containerization strategies for AIoT computer-vision tasks?** This question aims to empirically assess the possible differences in terms of energy consumption between the three deployment strategies (ONNX Runtime, Docker, WebAssembly).

**RQ2: What is the impact on performance of using different containerization strategies for AIoT computer-vision tasks?** This question aims at empirically assessing the possible differences in terms of timing and resource usage of the three containerization strategies mentioned before.

Answering the research questions formulated above will guide developers and researchers in the field to better understand and asses the impact of how and where AI is deployed on IoT devices. The results of this paper are valuable for developers that are debating on what would be the best choice for performing inferences in the context of AI containerization strategies on the Edge.

## 3.2 Subjects selection

The ONNX Model Zoo[4] is used as a reference when choosing a suitable and representative sample of subjects for the experiment. The set of pre-trained, state-of-the-art models of the ONNX Model Zoo have been successfully used in the literature [6, 23]. In this study we focus on the computer-vision tasks and we use the models in Table 1. To support the generality of the results, we ensure that the models are quite heterogeneous in terms of scope, dataset and size. We aim to use small, but fairly complex models, suitable for an IoT device. To allow conversion of a model from *.onnx* format to *.wasm* format, the ONNX variant should adhere to the following constraints: ONNX version 1.7.0, opset 12, file-format 7[5]. The Scailable python package[6] provides a validation function to aid in the conversion process. For the sake of replicability, all subjects are already pre-trained and we provide the same inputs to each of them. Even though assessing the accuracy of the models is out of scope of this study, the interested reader can inspect their output in our replication package.

**Table 1: Subjects of the experiment**

| Model | Scope | Dataset | Size |
|---|---|---|---|
| MNIST [21] | Image Classification | MNIST | 70k/30k |
| Emotion [7] | Image Classification | Emotion FER | 4k/1.8k |
| CIFAR10 [19] | Image Classification | CIFAR-10 | 4k/2k |
| YOLOv4 [8] | Object Detection | COCO | 700/300 |

## 3.3 Variables and statistical hypotheses

The *independent variable* (input) of the measurement is the **containerization strategy** and has three possible treatments: onnxr (ONNX Runtime), wasm (Scailable's engine), and docker (Docker), previously described in Section 2. All three are applied on the set of subjects from Table 1. Thus, it is relevant to mention the *balanced characteristic* of the experiment: the same exact number of subjects are assigned to each treatment. The *dependent variables* (output) are the **execution time (seconds**, **CPU usage (percentage), memory utilisation (megabytes)** and **energy consumption (kilojoules)**.

To answer RQ1 we construct the following null hypothesis:

• **Energy consumption**: considering $\mu_i$ as the median of the energy consumed by the deployment strategy $i$, the null hypothesis is formulated as follows:

$$H_0^{energy} : \mu_{onnxr} = \mu_{wasm} = \mu_{docker}$$

The corresponding alternative hypothesis is formulated as:

$$H_1^{energy} : (\mu_{onnxr} \neq \mu_{wasm}) \vee (\mu_{wasm} \neq \mu_{docker}) \vee (\mu_{docker} \neq \mu_{onnxr})$$

For RQ2, we construct a pair of null and alternative hypotheses for each dependent variable related to performance:

• **Execution time**: being $\mu_i$ the median of the execution time of the deployment strategy $i$, the null hypothesis is formulated as:

$$H_0^{exec} : \mu_{onnxr} = \mu_{wasm} = \mu_{docker}$$

The corresponding alternative hypothesis is formulated as:

$$H_1^{exec} : (\mu_{onnxr} \neq \mu_{wasm}) \vee (\mu_{wasm} \neq \mu_{docker}) \vee (\mu_{docker} \neq \mu_{onnxr})$$

• **CPU usage**: being $\mu_i$ the median of the CPU percentage utilised by the deployment strategy $i$, the null hypothesis is formulated as:

$$H_0^{cpu} : \mu_{onnxr} = \mu_{wasm} = \mu_{docker}$$

The corresponding alternative hypothesis is formulated as:

$$H_1^{cpu} : (\mu_{onnxr} \neq \mu_{wasm}) \vee (\mu_{wasm} \neq \mu_{docker}) \vee (\mu_{docker} \neq \mu_{onnxr})$$

• **Memory usage**: being $\mu_i$ the median of the memory utilised by the deployment strategy $i$, the null hypothesis is formulated as:

$$H_0^{mem} : \mu_{onnxr} = \mu_{wasm} = \mu_{docker}$$

The corresponding alternative hypothesis is formulated as:

$$H_1^{mem} : (\mu_{onnxr} \neq \mu_{wasm}) \vee (\mu_{wasm} \neq \mu_{docker}) \vee (\mu_{docker} \neq \mu_{onnxr})$$

For each dependent variable, the null hypothesis claims that there is no significant difference between the three containerization strategies, while the alternate hypothesis claims that for at least one pair of two treatments there is a significant difference. The hypotheses are not implying any direction on the studied phenomenon, so that no bias is introduced.

In view of all of the above, the experiment follows the *randomized complete block design* for the *1 factor - more than 2 treatments* design type [35], where each subject is allotted to all the treatments, and the pairs of <*subject, treatment*> are performed in a randomized order, to exclude any possible bias caused by the order in which the measurement is performed. It is worth mentioning that the Raspberry Pi devices are used as a *blocking factor*.

## 3.4 Data analysis

To analyze the outcome of the experiment, the following set of statistical methods are applied. We begin by performing preliminary observations on the measures using a series of plots to assess the distribution of data. Under the initial assumption that the collected data is not normally distributed we perform the Shapiro-Wilks test and visualize the data through Q-Q plots for each of the collected metrics. We can then decide if we further apply parametric or non-parametric tests to test the hypotheses.

[4]https://github.com/onnx/models
[5]https://github.com/onnx/onnx/blob/master/docs/Versioning.md
[6]https://pypi.org/project/sclblonnx/

Upon proving that the data is not normally distributed, we use the Kruskal-Wallis[29] test the check if there is a significant difference between at least two of the groups (hypothesis testing). This test is especially recommended in cases where the independent variable has two or more levels [35]. For both the Shapiro-Wilks and the Kruskal-Wallis test we compare the obtained p-value with the significance level of $\alpha = 0.05$. To avoid false discoveries, the Holm-Bonferroni correction [19] is applied on each pair of resulting p-values from the Kruskal-Wallis test.

To allow us a better understanding of the differences between the containerization technologies, beyond the interpretation of the p-values, we employ Cliff's Delta effect size test [9]. These results, along with the distributions pictured using violin plots will give us further insights on how different is one strategy compared to another, and by how much.

## 4 EXPERIMENT EXECUTION

The subjects are executed on a *Raspberry Pi 4 Model B* (8GB RAM, 64-bit Quad-core Cortex-A72 ARMv8-A processor) and a *Raspberry Pi 3 Model B+* (1GB RAM, 64-bit Quad-core Cortex-A53 ARMv8 processor), both running Ubuntu Server 20.04 as OS. We use Ubuntu Server since it is efficient in terms of disk space and RAM memory (makes it suitable for edge IoT devices) and it does not introduce any potential overhead caused by the GUI (it runs headless).

We then install the necessary packages for the experiment (and for each containerization strategy). For the ONNX Runtime, we install the required python packages and make no other additional customization steps. We already have the *.onnx* files for the trained models, that we obtained during subject selection phase. The Scailable engine is a standalone binary that does not require supplementary setup, besides compilation. However, we convert the *.wasm* files to their *.aot* variant using the wamr compiler (wamrc - release `WAMR-08-10-2021`): *wamrc -o model.aot model.wasm*. We preliminary test to check the performance of the *.aot* version, looking at the prediction time to ensure that using this file format is the faster approach for our experiment (the idea behind this step is to match ONNX's time as closely as possible, to have comparable runs). For the Docker deployment we use the *python:3.9-slim* image as a base, on top of which we perform the same steps as for the ONNX setup. The resulting SD card with the installed OS, the input dataset and the required scripts for automatization is employed for both devices used in this study.

To accurately perform model inferences, we conduct a preparatory setup where we decide on the appropriate input size for each subject (see Table 1). We then choose the containerization strategy that performs worst in terms of execution time on the devices (based on initial experimentation and preliminary runs) and we fixed the number of inferences so that the duration of each run of the experiment is approximately 60 seconds. As a result, we have different input sizes between the two Raspberry Pis (RPis). We can observe in Table 1 that for the RPi3 (the Size collumn, right side) we need in general less than half of the input size used for the RPi4.

Given that the Scailable binary we used accepts only serialized protobuf TensorProtos (*.pb*) as input, we perform the necessary preprocessing of all the images before the experiment is performed, save all of them as *.pb* files and copy them to the measured device. It is important to mention that the ONNX Runtime accepts this format as well, which makes the .pb format the preferred and natural approach for our experiment.

For energy recordings we use Monsoon's High Voltage Power Monitor (HVPM). The power meter enables voltage up to 13.5V and provides up to 6A of continuous current, at 5KHz sampling rate. It is considered a good candidate for this type of measures (total energy consumption) given its precision and previous usage in the field in similar scientific studies [18][34]. We measure the current consumed over a period of time (profiling time), given an input value for voltage. We start from the formula below, where the energy consumption over time *E(t)* is equal to the integral of power *P(t)* over time. The power at a certain point in time is equal to the product of voltage and current.

$$E(t) = \int P(t)dt = \int v(t) * i(t)dt$$

For each run, a csv file containing the unix timestamp (in milliseconds), the current and the voltage is obtained. We compute the integral (area under curve) using the trapezoidal rule. The Macbook Pro device is used to gather this data so that we minimize potential overhead on the system under test.

To get the performance measures, both *top* and *pidstat* can be used for profiling in GNU/Linux based systems. For that we build a bash script which periodically records the values for the target process based on its process ID (PID). The target process is the *python3* process performing inferences using *onnxruntime* package for the ONNX and Docker deployment, and *sclbl-bin* for the Scailable solution. The sampling rate can be easily set by modifying the script. For our experiment we take a snapshot of our process every 100ms, which consists in a sampling rate of 10Hz. Although both tools measure CPU and memory usage, the difference lays in how the former is computed and as per their documentation:

- *top (tcpu)* - task's share of the elapsed cpu time, percentage of total cpu time *(range: 0 - 100 x number of CPUs)*
- *pidstat (pcpu)* - total percentage of cpu time used by the task *(range: 0 - 100)*

Both tools produce the same values for memory usage. So, we analyse exclusively the measures recorded by pidstat, namely:

- *Resident Set Size (rss)* - amount of RAM memory being currently used by the process
- *Virtual Memory Size (vsz)* - amount of memory a process may hypothetically access

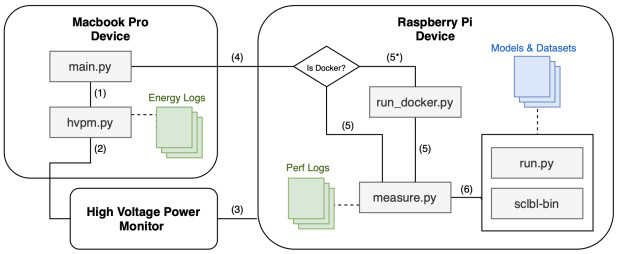


**Figure 1: Overview of the measurement infrastructure**

Execution time is collected via a timer that is started right before the launch inference task and gets stopped when the execution

of the model ends. To that extent we employ the *default_timer* from the *timeit* python package. Both of the output log files for the performance as well as the execution time reside on the test device and are manually copied to the Macbook Pro at the end of the measurement execution.

The Macbook Pro device is the orchestrator of the experiment and communicates with both the measured devices and the power monitor to start the experiment and record the raw measures (see Figure 1). The main script firstly initializes the HVPM device using its Python API and sets the value of the output voltage to *5V*. We then wait for 2 minutes to make sure that the measured device has powered up. After that, we start to run the models and record the metrics. We iterate through a randomized *.csv* file containing all possible combination of treatments; specifically, each device will be the host for 120 runs = 4 (models) x 3 (containerization strategies) x 10 (repetitions). Additionally, we leave a period of 60 seconds of cool-down (to mitigate the threat of maturation) between subsequent runs. One iteration goes through the following steps pictured in Figure 1:

• 1: the *main.py* script will run the current iteration configuration <model, strategy> and start by first communicating with the power monitor through *hvpm.py*;
• 2: a separate thread using *hvpm.py* starts periodic sampling using the HVPM;
• 3: the power monitor communicates directly with the device to both provide input voltage and record measures;
• 4: back in *main.py*, based on the strategy needed for this run a decision needs to be made on the script to be run next;
• 5: for docker, we separately start the container before running *measure.py (5*)*; for all strategies, the *measure.py* script will first start the performance profilers on the device and log the raw data;
• 6: at this point all profilers are setup and we run the subject using as inputs the model file and the dataset already available on the device; to start the engine we use *run.py* for docker and onnxr (as they both use the same Python package) and the sclb-bin for wasm.

Due to the intrinsic variability of energy and performance measurements, we take the following precautions: (i) between each run we have an idle period of 60 seconds to cool down both the test device and the power monitor; (ii) the measurement of each trial is repeated 10 times per subject; (iii) we randomize the order of both the input of each task and the execution of each run.

## 5 RESULTS

We firstly overview the raw measures, and then we present the results. The following abbreviations are used in the tables and figures: *en* - energy consumption, *ex* - execution time, *rss* - resident set size (memory), *vsz* - virtual set size (memory), *pcpu* - cpu measures collected via pdistat, *tcpu* - cpu measures collected via top.

### 5.1 Overview of collected measures

We use a series of Q-Q plots to visually assess the distribution of data. Looking at the graphs the data does not seem to be normally distributed for none of the metrics. However, to get a further indication of the distribution of the obtained data we apply the Shapiro-Wilks test. Given that all obtained p-values are lower than the chosen significance level of $\alpha = 0.05$, we conclude that the data is not normally distributed.

Tables 2 and 3 provide an overview of the obtained data. Specifically, we can look at the median column to given an initial assessment on how each treatment performed compared to the others. Additionally, in the violin plots in Figure 3 the width of the violin represents the frequency of samples in a specific region.

**Table 2: Summary statistics on the Raspberry Pi 4**

| | | min | max | med | mean | std | cv |
|---|---|---|---|---|---|---|---|
| en (kJ) | onnxr | 2. | 9 | 5.50 | 4.87 | 1.92 | 0.39 |
| | wasm | 9 | 15 | 11 | 10.82 | 1.30 | 0.12 |
| | docker | 3 | 8 | 7 | 5.20 | 2.35 | 0.44 |
| ex (s) | onnxr | 25 | 48 | 39 | 36.67 | 7.35 | 0.20 |
| | wasm | 53 | 67 | 57 | 56.75 | 3.11 | 0.05 |
| | docker | 25 | 48 | 38.5 | 37.07 | 8.05 | 0.21 |
| rss (Mb) | onnxr | 14.60 | 501.33 | 417.40 | 433.85 | 47.53 | 0.10 |
| | wasm | 5.91 | 875.55 | 250.66 | 328.34 | 179.82 | 0.54 |
| | docker | 179.35 | 506.19 | 422.49 | 437.68 | 48.44 | 0.11 |
| vsz (Mb) | onnxr | 8.32 | 183.89 | 100.04 | 118.43 | 37.48 | 0.31 |
| | wasm | 0.004 | 782.39 | 157.72 | 235.39. | 179.65 | 0.76 |
| | docker | 26.03 | 183.50 | 100.16 | 117.83. | 37.28 | 0.31 |
| pcpu (%) | onnxr | 0 | 15.85 | 0.35 | 1.02 | 2.30 | 2.24 |
| | wasm | 0 | 16.34 | 0.28 | 0.69 | 1.72 | 2.48 |
| | docker | 0 | 10.62 | 0.37 | 0.71 | 1.18 | 1.67 |
| tcpu (%) | onnxr | 0 | 200 | 200 | 191.88 | 23.04 | 0.12 |
| | wasm | 0 | 100 | 100 | 97.82 | 6.42 | 0.06 |
| | docker | 0 | 213.3 | 200 | 194.60 | 22.24 | 0.11 |

**Table 3: Summary statistics on the Raspberry Pi 3**

| | | min | max | med | mean | std | cv |
|---|---|---|---|---|---|---|---|
| en (kJ) | onnxr | 2 | 9 | 7 | 6.125 | 2.22 | 0.36 |
| | wasm | 10 | 13 | 11 | 11.2 | 0.95 | 0.08 |
| | docker | 3 | 10 | 7 | 6.55 | 2.33 | 0.35 |
| ex (s) | onnxr | 27 | 52 | 44 | 42.35 | 8.08 | 0.19 |
| | wasm | 57 | 66 | 60 | 60.4 | 2.49 | 0.04 |
| | docker | 28 | 56 | 45 | 43.425 | 8.65 | 0.19 |
| rss (Mb) | onnxr | 14.60 | 517.77 | 412.29 | 425.65 | 52.04 | 0.12 |
| | wasm | 5.91 | 534.84 | 195.72 | 245.5 | 101.89 | 0.41 |
| | docker | 167 | 506.192 | 417.38 | 429.87 | 52.34 | 0.12 |
| vsz (Mb) | onnxr | 8.12 | 190.78 | 91.74 | 109.17 | 37.86 | 0.34 |
| | wasm | 0.004 | 441.936 | 102.828 | 153.1 | 100.77 | 0.65 |
| | docker | 17.39 | 184.21 | 94.65 | 112.19 | 37.47 | 0.33 |
| pcpu (%) | onnxr | 0 | 1.95 | 0.34 | 0.40 | 0.33 | 0.81 |
| | wasm | 0 | 1,13 | 0.22 | 0.25 | 0.19 | 0.76 |
| | docker | 0 | 1.69 | 0.3 | 0.35 | 0.29 | 0.81 |
| tcpu (%) | onnxr | 0 | 320 | 193.3 | 182.91 | 31.03 | 0.16 |
| | wasm | 0 | 100 | 93.3 | 92.24 | 12.59 | 0.13 |
| | docker | 0 | 213.3 | 193.3 | 183.72 | 32.29 | 0.17 |

All p-values obtained from the Kruskal-Wallis test are lower than our significance threshold ($\alpha = 0.05$). Given that we have multiple entities belonging to the same entity: *rss* - *vsz* for memory usage and *pcu* - *tcpu* for CPU usage, we perform the Holm-Bonferroni correction [16] on each pair of resulting p-values from the Kruskal-Wallis test to avoid false discoveries. The results of the correction show no potential impact on the significance of our results.

We then perform a size effect estimation on the data, to compare the differences between pairs of containerization strategies. The results are presented in Table 4 and later discussed.

The scatter plots in Figure 2 present how the collected measures evolve during the execution of all runs of the experiment. In the scatter plots we can clearly see that energy consumption grows linearly for all treatments (first column in Figure 2); this is expected since the energy consumption is defined as the area under the curve of the power values (see Section 4). The measures for CPU usage (second and third columns in Figure 2) also exhibit interesting trends. Firstly, the values of `pidstat` tend to grow over time on

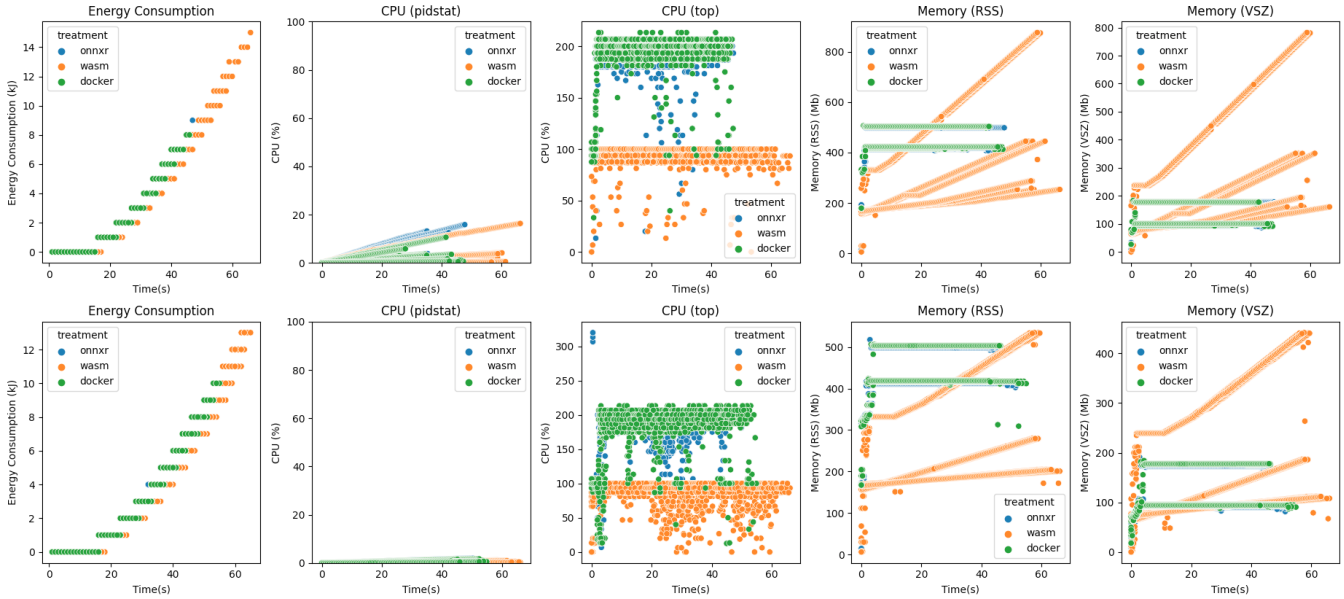


**Figure 2: Scatter plots of the measures collected from the RPi4 (top) and the RPi3 (bottom)**

Rpi4 whereas they tend to be constant on Rpi3, we are planning to replicate a dedicated experiment to shed light on this phenomenon. Also, the values of `top` clearly show that the `onnxr` and `docker` containerization strategies are using multiple cores, whereas `wasm` uses only one core; this result is expected since we are using the default configurations of the three containerization strategies (see Section 2). Moreover, the values collected via `top` tend to have higher variability and completely different values among the three containerization strategies, with *wasm* clearly using only one core, as opposed to *onnxr* and *docker* using also two or more cores (more on this later). For what concerns memory usage (fourth and fifth columns in Figure 2), we can clearly observe that *wasm* tends to consume a higher amount of memory over time, as opposed to *onnxr* and *docker* which tend to have a more stable memory usage across the whole duration of the run. We hypothesize that the growth of memory usage in *wasm* is due to the wamr compiler producing binaries that do not properly free memory across different inferences; as a follow-up of this study, we will inspect the source code of the wamr compiler to trace the root causes of this behaviour and replicate the experiment using different wamr releases.

Figure 6 presents a breakdown of the collected metrics per subject. The figure shows a more in depth view on the subjects and provides a first glimpse at what models should potentially benefit from optimizations. In general, we can observe and further support the statement that results on energy consumption are impacted by the execution time, as the distributions of the two variables are noticeably similar for each model.

## 5.2 Results on energy consumption (RQ1)

Tables 2 and 3 and Figure 3 allow us to make preliminary observations on the obtained results. Looking at the mean and median values, the energy consumption seems to be the lowest for the ONNX containerization strategy and highest for the WASM engine.

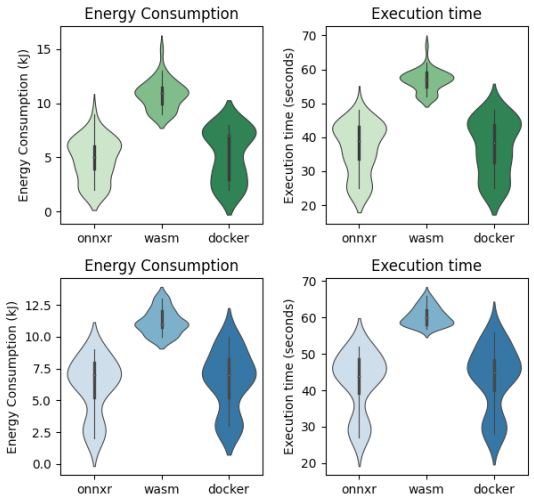


**Figure 3: Violin plots of the measures collected for energy & execution time from RPi4 (green, top) and RPi3 (blue, bottom)**

To test the hypothesis, the p-values computed for the energy consumption: $3.27e^{-18}$ for Raspberry Pi 4 and $5.90e^{-18}$ respectively for Raspberry Pi 3, are far below the set significance level and thus allow us to reject the null hypothesis $H_0^{energy}$. Therefore, the results show that there is a significance difference between at least two of the containerization strategies.

## 5.3 Results on performance (RQ2)

According to our experiment setup, onnxr and docker are the containerization strategies with the lowest execution time, on both

devices. Comparatively, the measures for wasm tend to be more consistent around the mean value (Figure 3). CPU wise, measures are lowest for the wasm deployment, while the similarity persists for onnxr and docker. Finally, the wasm containerization strategy uses less memory overall (rss + vsz), than docker and onnxr. The violin plots from Figure 3 show that there is tendency for the measures to reside around the median value.

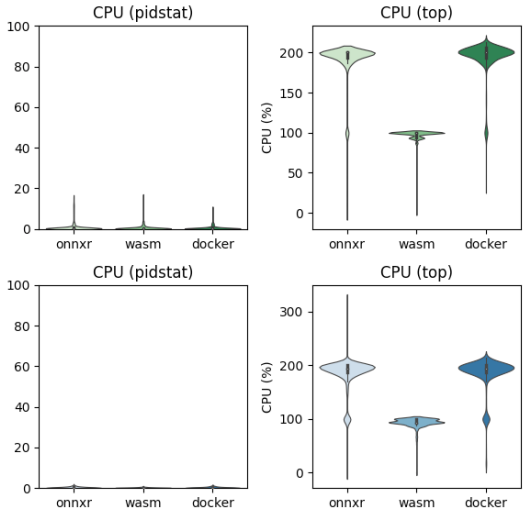


**Figure 4: Violin plots of the measures collected for CPU usage from RPi4 (green, top) and RPi3 (blue, bottom)**

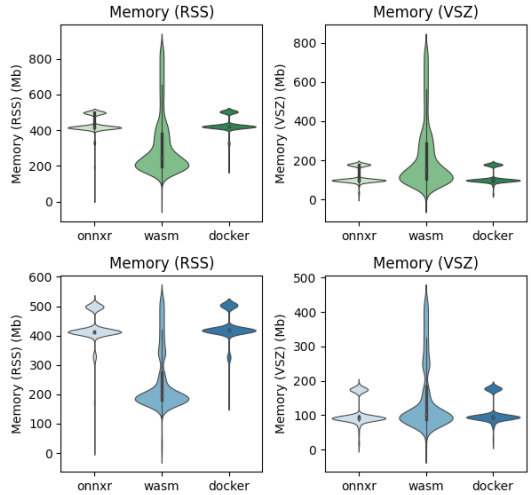


**Figure 5: Violin plots of the measures collected for memory usage from RPi4 (green, top) and RPi3 (blue, bottom)**

For this RQ we previously formulated a set of hypotheses for each dependent variable. The results of the Kruskal-Wallis test allow us to safely reject the null hypotheses for all dependent variables related to RQ2. As such, we conclude that there is a significant difference between at least two of the containerization strategies for execution time, CPU usage and memory utilization.

## 5.4 Effect size estimation

Finally, for each metric we do a complementary analysis by performing an effect size estimation on the results using Cliff's delta. We use Hess and Kromrey's [15] values for effect magnitudes: 0.147 - *small*, 0.33 - *medium*, 0.474 - *large*. Table 4 presents the interpretation of the results based on the obtained values from the test computation: *n - negligible, s - small, m - medium, l - large*. The following pairs are used for the comparison: *on-dk* for onnxr and docker, *on-ws* for onnxr and wasm and *dk-ws* for docker and wasm. Negative values for $\delta$ imply that, in general, samples from the distribution on the left member of the pair had lower values.

**Table 4: Results of the Cliff's Delta effect size test**

| | **rpi4** | | | **rpi3** | | |
|---|---|---|---|---|---|---|
| | *on-dk* | *on-ws* | *dk-ws* | *on-dk* | *on-ws* | *dk-ws* |
| en | -0.15 *(s)* | -0.99 *(l)* | -1.0 *(l)* | -0.11 *(n)* | -1.0 *(l).* | -0.97 *(l)* |
| ex | -0.06 *(n)* | -1.0 *(l)* | -1.0 *(l)* | -0.13 *(n)* | -1.0 *(l)* | -1.0 *(l)* |
| rss | -0.44 *(m)* | 0.57 *(l)* | 0.58 *(l)* | -0.52 *(l)* | 0.78 *(l)* | 0.79 *(l)* |
| vsz | 0.003 *(n)* | -0.44 *(m)* | -0.44 *(m)* | -0.42 *(m)* | -0.21 *(s)* | -0.14 *(n)* |
| pcpu | -0.02 *(n)* | 0.12 *(n)* | 0.14 *(n)* | 0.07 *(n)* | 0.25 *(s)* | 0.19 *(s)* |
| tcpu | -0.22 *(s)* | 0.95 *(l)* | 0.96 *(l)* | -0.04 *(n)* | 0.94 *(l)* | 0.93 *(l)* |

# 6 DISCUSSION

## 6.1 Overview of the obtained results

We start by looking closer at the comparison between *onnxr* and *docker*, especially because the Docker container employed in the experiment uses the same ONNX Runtime setup. Overall, the *onnxr* deployment has best results in terms of energy consumption (see Table 4). As initially predicted, the difference in execution time is negligible on both devices, while the extra layer of virtualization results in a higher consumption of energy for the *docker* deployment, although not a significant one. We must, however, take into consideration that this metric is computed over time. Thus, a higher duration in execution time can impact the energy consumption negatively (less is considered to be better). A reason on the higher value in energy consumption for WASM engine can be then perceived a result of its higher execution duration, compared to the other two.

The *wasm* containerization strategy performs best in terms of RAM memory consumed, observable both from the violin plots in Figure 5 and from the $\delta$ values from Table 4. When comparing *bm-ws* and *dk-ws* for rss, we get a large effect size estimation.

The execution time of wasm does not seem to be affected by the subject, and all values are fairly consistent and are gathered towards the median (Figure 3) on both devices. The *wasm* approach performs positively considering it uses only one core (the samples of tcpu for wasm provide a median and mean value close to 100, as can be seen in Tables 2 and 3). We are unable to gather much insightful information about the CPU usage (Figure 4) from our *top* recorded measures. However, upon analysis of the data recorded from *pidstat* (looking at the values from Table 4), out of the three deployments, the Scailable approach seems to perform best.

The disk space required for each containerization strategy can be another factor that could make a developer lean towards one of the three solutions, especially when extremely small device are concerned. The *onnxruntime* python package has 18.7MB (the *onnx* package itself has an additional 19.7MB), while some dependencies such as the *numpy* package contains as much as 90MB. Comparatively, the standalone sclbl-bin binary we used in our measurement

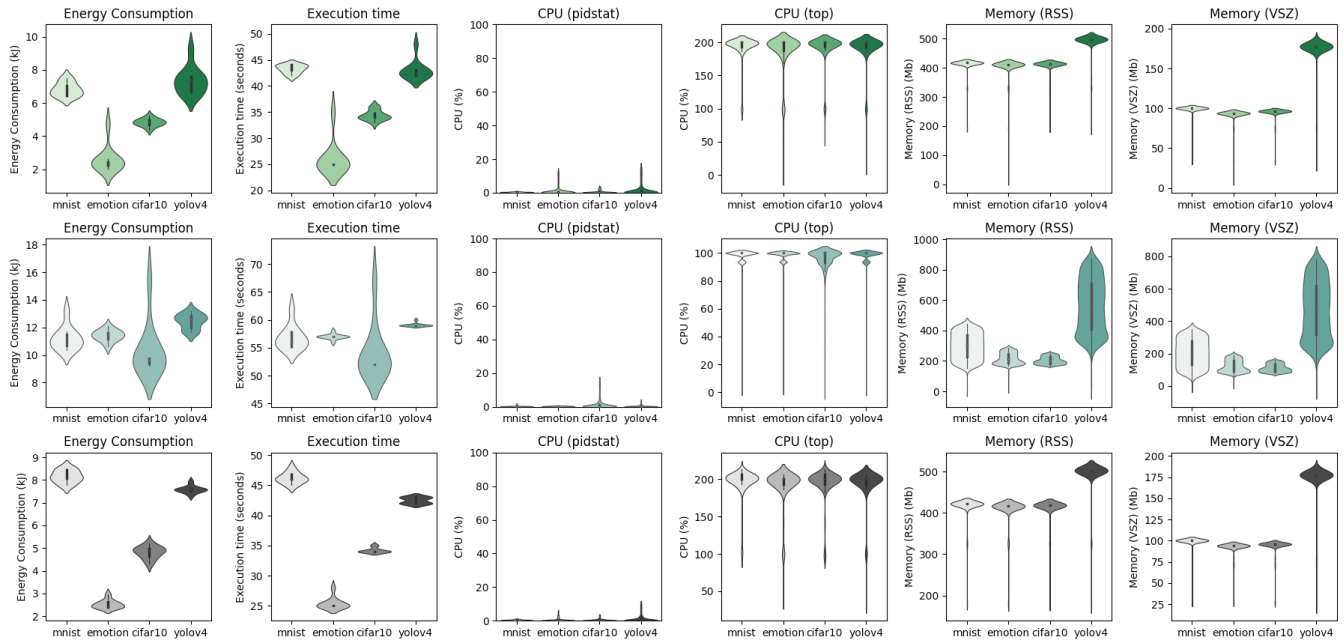


**Figure 6: Breakdown of the collected metrics for each of the 4 algorithms: onnxr (top), wasm (middle), docker (bottom)**

has 5.5MB. The Docker approach is by far the largest of the three, considering that the image we employed in our experiment has 535.1 MB in its compressed form.

## 6.2 Implications for developers and researchers

This study provides empirical evidence to researchers and developers to motivate the need to improve the utilization of available resources on Edge devices.

For Edge AI applications catered towards devices with scarce resources (either disk space or computation power), developers should aim to separate the training from prediction before deployment [13]. There are various options for containerizing the prediction step. Our results show that a WebAssembly approach might work best where RAM memory utilization is a concern. Moreover, Scailable's WASM approach facilitates conversion from *.onnx* to *.wasm*, which provides a straightforward (and portability oriented) method for deployment. The model can be first trained with any of the ONNX compatible AI frameworks (such as Keras[7], TensorFlow[8]) that allow the model to be saved as an *.onnx* file and later converted and launched into production inside a WASM container. The separation would prove especially valuable in cases where data is not meant to be sent to an external server (e.g., healthcare [10]). However, if the device has a high amount of resources available, the developer should aim to perform AI deployments using the ONNX containerization strategy for the lowest level of energy consumption. The Docker approach is also a good candidate for reduced energy consumption and fast execution time, on top of providing a portable and contained solution for deployment.

[7]https://keras.io
[8]https://www.tensorflow.org

As we discussed in Section 2, for the sake of representativeness in this experiment we are using the default configuration of the three containerization strategies. This design choice is noticeable also in the collected measures; if we look at the measures for energy and CPU usage in Figure 2, we have a nice example of interaction between energy and performance metrics. Specifically, we can notice that *wasm* (i) is the only containerization strategy using a single core and (ii) tends to have the longest execution times. Together, these two facts can explain why *wasm* tends to consume less energy than the other two containerization strategies, even if they both use multiple cores for executing their tasks. As an extension of this study, researchers are invited to replicate our experiment to shed light on which containerization strategy is more energy efficient/performant under different configurations.

# 7 THREATS TO VALIDITY

**Internal Validity**. The risk of how *history* might affect the results is reduced by performing the measurement all at once, in the same environment and using the same infrastructure, but in a randomized manner. To mitigate, we built scripts to automate the execution of the experiment (more broadly detailed in Section 4). In this scenario, *maturation* comes as another possible threat, since the runs are performed repeatedly (and sequentially) on the same object. To mitigate this threat, we set an interval of 60 seconds between runs. The *selection of subjects* (models) might constitute another potential threat as it was performed manually. Thus, the selected set of models could not be regarded as an accurate representative for the whole population. A first iteration in this process consisted in selecting a set of models from the ONNX Zoo, from each of the classes mentioned on the GitHub page, initially aiming to have two representatives from each class. However, we observed the lack of

already trained models for some of the classes and what is more, most of the models were not suitable for conversion from *.onnx* to *.wasm* format (as mentioned in Section 3). As a result to that, we ended up with a selection of 4 models from the Computer Vision (CV) category and the threat is reduced by aiming to diversify the CV tasks they performed. It is worth mentioning that both prediction engines (ONNX Runtime and WebAssembly) perform worse on first input received compared to subsequent inputs, which is mainly because of a required warm up inference. We did not perform any special treatment for that initial inference to better represent real life deployments and, since we randomize input order for every run, we believe it does not introduce significant bias.

**External Validity**. The *interaction of selection and treatment* is one of the external threats concerning this study, also mentioned in the previous section. This threat can be mitigated in the future by repeating the experiment on more model classes, as they become available and allow conversion, or even making the training of the new models as part of a future experiment. In general, threats to external validity are reduced by providing a relevant and realistic measurement environment to minimize the effect of the *interaction of setting and treatment*. In our study we support the mitigation of this threat by performing the experiment on both a less (Raspberry Pi 3) and more (Raspberry Pi 4) highly performing device. On both devices we use a relatively recent and modern version of Ubuntu Server and latest versions for all the tools and packages employed in the instrumentation process. What is more, the generalization factor can be improved by replicating the experiment on other edge devices, some especially made for AI deployments such as the Jetson Nano.

**Construct Validity**. The threat concerning *inadequate preoperational explication of constructs* is mitigated by having a good design of the experiment in accordance with the documentation [35], before the execution phase. In Section 3, which is specifically dedicated to this topic, we provide details on the matter by defining: goal, research questions, hypotheses, variables and the statistical methods. Additionally, we follow design principles concerning randomization, blocking and balance. Another possible threat is the *mono-operation bias* caused by having only one factor in our experiment. However, to reduce this threat by performing a number of 120 repetitions: 10 repetitions per combination of <subject, treatment>. Note that the relatively high standard deviation in our dataset might have been caused by having 10 repetitions per trial; this potential source of bias can be mitigated by increasing the number of such repetitions in a replication of the experiment.

**Conclusion Validity**. In our study, most of the p-values resulted from the applied statistical methods (such as Saphiro-Wilks and Kruskal-Wallis) are significantly lower that the chosen significance level of $\alpha = 0.05$. Additionally, we check the *assumptions* of the Kruskal-Wallis test (in Section 5) to ensure that those are not violated and to further support the correctness of our conclusions. Including all the data collected during the statistical analysis phase, especially the outliers, supports our claim of correctness by *not fishing* for a specific result. What is more, the replication package associated with this experiment allows the reproduction and validation of the outcomes presented in this paper.

# 8 RELATED WORK

Santos et al. [31] present a comparison of the energy consumption in and out of Docker containers for various workloads. The research highlights that the additional abstraction layer of Docker would introduce a significant overhead in terms of energy, compared to its bare-metal counterpart. An additional test was performed while the system under test was in an idle state, which further supports the claim and states that even having only the Docker service running on the system produces an increase in energy consumption. The authors of the paper motivate the increase as being caused by the activity of the Docker deamon (*dockerd*).

Alternative research provides a more detailed analysis on how WebAssembly performs compared to other containerization technologies. Napieralla [27] focuses on hardware constrained devices at the Edge to analyse the utility of WASM over Docker and native execution in IoT. A section dedicated to the performance on startup time proves WASM to be the better contender when compared to Docker. What is more, WASM performs better memory wise, but lacks in terms of CPU performance. The study does not take into account a comparison on the levels of consumed energy.

An investigation of energy consumption related to WebAssembly implementations is produced by Oliveira et al. [28]. The authors analyze the optimization potential of WebAssembly to improve the performance of JavaScript language in embedded systems. The results suggest that there is a trade off between battery consumption and performance, where WASM proves to be a viable (better execution time and reduced battery consumption) solution when compared directly to JavaScript. The results of our research provide similar insights on containerization comparison, while centralizing the measurements of a significant number of metrics (most studies focus either on energy consumption or CPU/memory utilisation, and not on both). Moreover, the novelty of our work consists in using AI models as a benchmark for the experiment.

Energy consumption in machine learning is still a topic that needs extensive exploration. García-Martín et al. [12] aim to mitigate the lack of interest in the subject by collecting and summarizing the methodology required to produce results by focusing on models for energy estimation. Comparatively, in our experiment, we make use of direct energy measures.

A recent study [13] empirically assessed the energy consumption and run-time performance of two commonly-used deep learning frameworks (i.e., PyTorch and TensorFlow), with very different results between the training and inference phases. Our study is different from [13] since we focus on the Edge deployments (instead of server-side computation), on the containerization strategies (instead of the used DL frameworks), and on the inference phase (instead of both training and inference).

A similar study of AI on the Edge was performed by Zhang et al. [38] The research focuses on two trained models and highlights the performance of different ML frameworks on various hardware configurations. By dividing the inference process, the authors draw a conclusion on how the loading latency and frequency of a model could affect the time to generate predictions. Comparatively, in our experiment the models are loaded only once. Additionally, our study differs both in terms of the chosen subjects and the inference environment.

To our knowledge there is no comprehensive study about the energy consumption of the ONNX Runtime. An analysis on the ONNX Runtime was carried out only on time predictions between different machine learning frameworks or benchmarking specific operations [3]. Additional research on the topic of AI models performance on mobile devices has been performed, for example by Luo et al. [24]. However, the authors do not include any measures for energy and do not consider ONNX when building and analysing their proposed unified benchmark suite.

## 9 CONCLUSION

This paper aims to be an additional stepping stone in the field of AIoT, with a focus on the impact of three containerization strategies wrt energy consumption and performance. Scailable's WASM engine (although new to the field) proved to be a relevant contender to ONNX and Docker. Based on the results of our experiment, for Edge AI applications with strict disk space and RAM memory requirements developers should prefer WebAssembly for deployment over ONNX or Docker; the ONNX strategy should be the preferred choice when energy consumption and execution time are a concern.

As future work, we are planning to investigate on how energy correlates with (additional) performance-related metrics, replicate the experiment by using other AI-based tasks (e.g., conversational AI) and other configurations/hardware platforms.

## ACKNOWLEDGMENTS

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.

## REFERENCES

[1] 2016. *Computer Vision Hardware and Software Market.* https://www.businesswire.com/news/home/20160620005440/en/Computer-Vision-Hardware-and-Software-Market-to-Reach-48.6-Billion-by-2022-According-to-Tractica

[2] 2019. *What Is Edge Computing?* https://blogs.nvidia.com/blog/2019/10/22/what-is-edge-computing/

[3] 2020. *ONNX benchmarksr.* http://www.xavierdupre.fr/app/_benchmarks/helpsphinx/onnx.html

[4] 2021. *Computer Science Trends in 2021 to Look Out For.* https://www.computerscience.org/resources/computer-science-trends

[5] 2021. *Why is Docker so popular.* https://www.section.io/engineering-education/why-is-docker-so-popular/

[6] OTH Amberg-Weiden. [n.d.]. Characterization of Object Detection Performance in an Edge Environment. ([n. d.]).

[7] Emad Barsoum, Cha Zhang, Cristian Canton-Ferrer, and Zhengyou Zhang. 2016. Training Deep Networks for Facial Expression Recognition with Crowd-Sourced Label Distribution. *CoRR* abs/1608.01041 (2016). arXiv:1608.01041 http://arxiv.org/abs/1608.01041

[8] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *CoRR* abs/2004.10934 (2020). arXiv:2004.10934 https://arxiv.org/abs/2004.10934

[9] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[10] Thomas Davenport and Ravi Kalakota. 2019. The potential for artificial intelligence in healthcare. *Future healthcare journal* 6, 2 (2019), 94.

[11] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. 2020. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal* 7, 8 (2020), 7457–7469.

[12] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. 2019. Estimation of energy consumption in machine learning. *J. Parallel and Distrib. Comput.* 134 (2019), 75–88.

[13] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. 2022. Green AI: Do Deep Learning Frameworks Have Different Costs?. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering.*

[14] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 185–200.

[15] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association.* Citeseer, 1–30.

[16] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[17] Maurits Kaptein. 2020. *Exploiting the differences between model training and prediction.* https://towardsdatascience.com/exploiting-the-differences-between-model-training-and-prediction-40f087e52923

[18] Muhammad Umair Khan, Shanza Abbas, Scott Uk-Jin Lee, and Asad Abbas. 2021. Measuring power consumption in mobile devices for energy sustainable app development: A comparative study and challenges. *Sustainable Computing: Informatics and Systems* 31 (2021), 100589.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[20] Lionel Lacassagne, Daniel Etiemble, Ali Hassan Zahraee, Alain Dominguez, and Pascal Vezolle. 2014. High level transforms for SIMD and low-level computer vision algorithms. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector processing.* 49–56.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[22] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. 2018. Techology trend of edge AI. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT).* 1–2. https://doi.org/10.1109/VLSI-DAT.2018.8373244

[23] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. 2019. ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS).* IEEE, 214–218.

[24] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. 2020. Comparison and benchmarking of ai models and frameworks on mobile devices. *arXiv preprint arXiv:2005.05085* (2020).

[25] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[26] László Monostori. 2003. AI and machine learning techniques for managing complexity, changes and uncertainties in manufacturing. *Engineering applications of artificial intelligence* 16, 4 (2003), 277–291.

[27] Jonah Napieralla. 2020. Considering webassembly containers for edge computing on hardware-constrained iot devices.

[28] Fernando Oliveira and Júlio Mattos. 2020. Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. In *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais.* SBC, 133–138.

[29] Eva Ostertagova, Oskar Ostertag, and Jozef Kováč. 2014. Methodology and application of the Kruskal-Wallis test. In *Applied Mechanics and Materials*, Vol. 611. Trans Tech Publ, 115–120.

[30] Bhuvan Puri. 2021. IoT and AI-based Plant Monitoring System. *International Journal of Machine Learning and Networked Collaborative Engineering* 4, 3 (Jun. 2021), 135–142. http://mlnce.net/index.php/Home/article/view/154

[31] Eddie Antonio Santos, Carson McLean, Christopher Solinas, and Abram Hindle. 2018. How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers. *Journal of Systems and Software* 146 (2018), 14–25.

[32] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[33] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).

[34] Matteo Varvello, Kleomenis Katevas, Mihai Plesa, Hamed Haddadi, and Benjamin Livshits. 2019. BatteryLab, A Distributed Power Monitoring Platform For Mobile Devices. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks.* 101–108.

[35] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering - An Introduction.* Kluwer Academic Publishers.

[36] Pengfei Xu, Shaohuai Shi, and Xiaowen Chu. 2017. Performance evaluation of deep learning tools in docker containers. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM).* IEEE, 395–403.

[37] Jing Zhang and Dacheng Tao. 2021. Empowering Things With Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things. *IEEE Internet of Things Journal* 8, 10 (2021), 7789–7817. https://doi.org/10.1109/JIOT.2020.3039359

[38] Xingzhou Zhang, Yifan Wang, and Weisong Shi. 2018. {pCAMP}: Performance Comparison of Machine Learning Packages on the Edges. In *USENIX workshop on hot topics in edge computing (HotEdge 18).*