The impact of base image selection on the energy efficiency of containerized applications in Docker

MSc Computer Science

Bailey Tjiong

TU Delft

The impact of base image selection on the energy efficiency of containerized applications in Docker

MSc Computer Science

Thesis report

by

Bailey Tjiong

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on December 13, 2023 at 15:00

Thesis committee:

Thesis advisor:	Arie van Deursen
Supervisors:	Thomas Durieux
	June Sallou
External examiner:	Jie Yang
Place:	Faculty of EEMCS, Delft
Project Duration:	February, 2023 - December, 2023
Student number:	4474686

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Faculty of EEMCS · Delft University of Technology

Abstract

Containerization has become a fundamental component of software development and the continuous integration and continuous delivery (CI/CD) pipeline. However, the energy overhead of containerization solutions such as Docker results not only in higher costs but also has environmental consequences. Currently, the energy efficiency of containers is often neglected, since developers mainly focus on small image size and time efficiency. Furthermore, there is also a lack of understanding about the impact of base images on the container during runtime. Therefore, the objective of this thesis is to assess the impact of base image selection on the energy efficiency of containerized workloads. This is done by running an empirical study, in which we conduct experiments with a diverse set of real-world workloads and base images. The results of this study show that the base image can have a significant impact on the energy efficiency of the container. However, the magnitude of this impact in practice depends on the workload. For some workloads (i.e. databases) this impact is more significant in practice than for others (i.e. gaming servers). While there is no single best or worst image across all workloads, Alpine is often the least energy-efficient option. Furthermore, the execution time of the container is only correlated to the energy consumption for CPU-intensive workloads. Therefore, besides image size and time efficiency, it is important to consider energy efficiency in the selection criteria as well when considering sustainable practices in software engineering.

Preface

As my time as a student at TU Delft comes to an end, I proudly present this thesis as the final chapter of my academic journey. The topic of this thesis arose from my interest in environmental sustainability. With help from my supervisors, we combined this interest with software engineering and the result is this study. I hope this study can contribute to the sustainable software engineering field, even if it is just a little bit.

I would like to thank my daily supervisors, June Sallou and Thomas Durieux, for sharing their time and assisting me during my thesis. The guidance, knowledge, and resources they provided were crucial in completing my Master's thesis. I would also like to thank Arie van Deursen, for giving me the opportunity to do my thesis at SERG, and Jie Yang, for participating as an external committee member for my thesis defense.

Finally, I have learned a lot during my time at TU Delft, and will use these skills in the next chapter of my life. Furthermore, I am thankful for the wonderful people I met during my time here, from whom I also learned a lot. These people have helped me directly or indirectly with achieving my academic goals. Finally, I am grateful for my parents, who made it possible for me to start, and now finish my education.

Bailey Tjiong
Delft, November 2023

Contents

List of Figures

List of Tables

1 Introduction

Containerization has become an integral part of the development of software, as well as of the continuous integration and continuous delivery (CI/CD) pipeline. Moreover, nowadays the container orchestration platform Kubernetes is a crucial part of DevOps workflows as it provides flexibility and reliability in the CI/CD pipeline, which makes containers essential as well. As containerization is a virtualization approach, the main benefits of containers include isolation and hardware independence [1]. This allows developers and engineers to work in a standardized environment, and therefore streamline the development lifecycle and CI/CD workflow. Furthermore, the lightweight nature of containers results in lower overheads than virtual machines (VMs) [2, 3]. These benefits of containerization allow for better scalability and portability.

Docker is considered the de facto standard for containerization [4, 5], and is still the most used container runtime according to a recent usage report [6]. Furthermore, the yearly StackOverflow survey shows that Docker is the most popular developer tool among professional developers [7]. The popularity of Docker is mainly due to its convenient process of creating and managing containers, which closes the gap between development and operations [8]. The containerization process in Docker starts by declaring the Dockerfile, which is a descriptive file containing instructions and configurations to build a new container image. These instructions always include a base image (e.g. Ubuntu or Alpine), which is an already-existing image on which the newly created image extends. The built container image is a template with the defined instructions and is used to create and subsequently run containers.

Although the lightweight environment of Docker uses the available resources more efficiently than VMs [9], Docker containers still have a measurable energy overhead [10]. As the number of running containers increases [6], the overall energy consumption will increase as well. High energy consumption does not only lead to higher costs, but it also has environmental consequences. Since energy consumption contributes significantly to carbon emissions [11], the discipline of Sustainable Software Engineering (SSE) focuses on developing energy-efficient software. Considering the importance of Docker in software engineering, it is necessary to optimize the energy efficiency of Docker containers to make an impact on sustainable development.

In recent years several studies have investigated the energy efficiency of Docker, and how it compares to bare metal Linux [10] and other virtualization solutions [12, 5, 13]. The general consensus of these studies is that Docker is usually more energy efficient than other virtualization solutions. Even though these experiments show the energy efficiency of the Docker technology itself, the factors that impact the energy efficiency of Docker containers are not taken into consideration. A recent study investigates the energy efficiency of different containerized workloads [14]. While this study motivates the optimization of energy-efficient containers, it only compares the energy consumption of specific workloads, and does not provide a general understanding to developers and engineers.

An important part of containerizing a workload is selecting a base image for the container image. The base image is usually selected based on the image size or the time efficiency. However, there is currently a lack of knowledge about the impact of the base image selection on the container during run time. Since there are many different images available and used for these workloads [15, 16, 17], investigating the base

image as a possible factor that affects the energy efficiency of containers will improve the understanding of container optimization.

This thesis will fill this knowledge gap by assessing the impact of base image selection on the energy efficiency of containerized workloads, and in particular Docker containers, by conducting an empirical study. For this empirical study, we will conduct experiments with a diverse set of workloads and base images. These workloads will reflect real-world scenarios, such as machine learning inference, web application traffic, and database transactions. The energy consumption, and CPU utilization, of these containerized workloads will be monitored, to evaluate the impact of the base images on the energy efficiency of these workloads.

1.1 Problem statement

The objective of this thesis is to assess the impact of base image selection on the energy efficiency of containerized workloads, as shown in Figure 1.1. This objective follows from two problems we identified in the current landscape. First, the energy efficiency of containers is often neglected, since developers usually focus on small image sizes and time efficiency. Second, it is difficult to choose the correct base image among the plethora of images that are available on container registries (e.g. Docker Hub), as there is a lack of understanding about the impact of the base image on the container during runtime. Understanding the impact of base image selection will motivate the research into optimizing the energy efficiency of Docker containers, and ultimately encourage developers and engineers to take energy consumption into account when selecting a base image for containers.

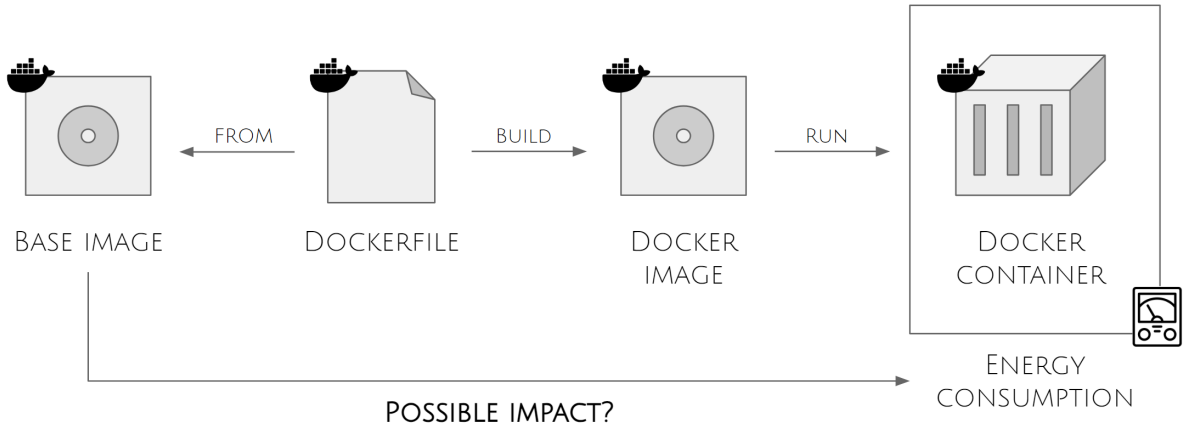

Figure 1.1: Objective of this thesis

1.2 Research questions

This thesis will fill the knowledge gap by answering the following research questions, which also assist in defining the experiments we will conduct to assess the impact of base image selection:

RQ1

Does the base image selection have an impact on the energy efficiency of the container?

The main focus of this study is to determine if a base image affects the energy efficiency of a containerized workload, to provide a better understanding of factors that impact the energy efficiency of Docker containers. We will assess this by monitoring the energy consumption of a container, using different base images. By using the energy consumption data of the different images, we will determine the significance of the impact of base images on the container's energy efficiency.

RQ2

Does the impact of base image selection vary for different types of workloads?

Docker containers are used for many different use cases and workloads. As there is a plethora of images available on container image registries, it is expected that some base images are more optimized for certain workloads. To confirm this, it is necessary to evaluate the impact of base images for a diverse set of workloads. We will answer this question by conducting the experiment for several workloads that reflect real-world scenarios. Subsequently, we will assess the differences in energy consumption between the base images across these workloads.

RQ3

Is there a trade-off between current selection criteria and energy efficiency?

We define execution time and image size as the current selection criteria for base images. Rosa et al. [18] found that smaller images have a higher number of adoptions. A study conducted by Zhang et al. [19] shows that developers are concerned with the build latency of images. Since this thesis focuses on the container during runtime, we will instead use execution time. When considering sustainable practices in software engineering, developers also need to consider the energy consumption of containers. However, the prioritization of energy efficiency might come at the cost of execution time or image size.

RQ3.1 Is the execution time of the containerized workload correlated with the energy consumption?

Besides energy consumption, we will also monitor the timestamps of the runs. It is likely that the base images lead to differences in performance for the containers, resulting in shorter or longer execution times. This research question is motivated by the possibility that a shorter execution time does not necessarily result in lower energy consumption. If this is indeed the case, it will motivate developers and engineers to not solely focus on the elapsed time of a containerized workload, but also take the overall energy consumption into account. We will evaluate the correlation between the elapsed time and energy consumption for each run. Furthermore, the ranking of the base images in terms of time (i.e. fastest) and energy consumption (i.e. lowest) is taken into consideration as well.

RQ3.2 Does a small image also result in low energy consumption of the container?

The different base images are not the same in size, as they will have their own standard libraries. The popularity of the Alpine base images shows that developers care about a small image size. However, it is not known if a small image size also correlated to the energy efficiency. If this is not the case, simply choosing the smallest base image might not be the best option if energy efficiency is taken into consideration. Therefore, we will investigate whether the size of the image is directly related to the energy consumption of the corresponding containers. More specifically, we will assess if using the smallest base image (i.e. Alpine), also results in a more energy-efficient container.

1.3 Contributions

This thesis will provide the following contributions to the field:

1. **Empirical study on the energy efficiency of containerized applications**

The empirical study will provide insights into the impact of base images on the energy consumption of containerized workloads. Furthermore, it will promote further research into optimizing the energy efficiency of Docker containers, and motivate developers and engineers to take energy consumption into account when choosing a base image.

2. **Dataset of energy consumption measurements for various workloads using different base images**

The obtained energy consumption data from monitoring the workloads are publicly available to increase the transparency of the experiments conducted in this thesis.

3. **A diverse set of workloads representing real world scenarios**

The Dockerfiles of the workloads used in the experiments, for the different base images, are publicly available. These containerized workloads can be used for future research or benchmarks.

4. **Pipeline of measuring the energy consumption of containerized applications**

The pipeline used for the experiments is available as well and can be used to reproduce the findings in this thesis.

1.4 Outline

The remainder of the thesis is structured as follows. Chapter 2 provides the relevant background information. Chapter 3 presents an overview of the literature related to this study and identifies the research gap. Chapter 4 introduces the methodology of the experiments and Chapter 5 presents the collected data from the experiments. The results are further discussed in Chapter 6. Chapter 7 discusses the implications of the results, as well as the limitations and future work. Finally, Chapter 8 concludes the thesis.

2 Background

As the goal of this thesis is to motivate developers and engineers to consider the energy efficiency of containers, this chapter will start with an introduction to measuring the energy consumption of software. Next, we will explain virtualization techniques to show the differences between containerization and VMs. This is followed by a section specifically about Docker, a container-based virtualization solution, which we will be using in our experiments.

2.1 Measuring energy consumption

With the growing concerns about climate change, the importance of sustainability is increasing as well. Since software is everywhere and used by everyone, we should also think about the sustainability of software. While the impact of software on the environment might not be obvious, since it is digital, it still has a significant impact. The machines on which software is running need to be supplied with energy, which has the highest share of carbon emissions [20, 21].

The discipline that is concerned with these problems is called Sustainable Software Engineering (SSE). SSE strives to create software that can provide value in the long run without hindering its surrounding environment [22]. Environmental sustainability, which is a branch in SSE, is concerned with minimizing the impact the development of software has on our planet [22]. In order to achieve environmental sustainability it is important to build carbon efficient applications. However, it is complex to measure the amount the carbon emitted by applications, partly because it is dependent on the energy generation source [23]. While measuring the power consumption and energy efficiency of a container, or software in general, might not be a straightforward task, there are several methods that can provide accurate measurements or approximations. These methods include both physical power meters and software tools for energy profiling. In the context of software engineering, energy efficiency is often defined as [24, 25]:

$$\textit{energy efficiency} = \frac{\textit{work done}}{\textit{energy consumption}} = \frac{\textit{performance}}{\textit{energy consumption}}$$

2.1.1 Physical power meters

Using physical power meters is the most accurate method for measuring the power consumption of an entire machine [26, 27]. Usually, the power meter is placed between the power socket or power source, and the System Under Test (SUT), for which we want to measure the energy consumption. The power meter will then measure the instantaneous power usage of the SUT.

Previous studies about the energy consumption of containers have used several physical power meters. The *Watts Up? PRO* does not report the power usage directly, however, it provides samples of the voltage and current draw [28]. Another tool that accurately measures the power consumption directly, is the *Raritan* Power Distribution Unit (PDU). This PDU reports the mean value of the instantaneous power over a time period, which is defined as active power [12].

While physical power meters are the most accurate method, setting them up correctly can be a difficult task [27], since they often need to be calibrated for each different machine. Furthermore, the power

meter itself will not show the energy consumption of specific domains, but will rather report the energy consumption of the entire machine. Therefore, there are also software-based energy profilers that do not require physical power meters or other dedicated hardware.

2.1.2 Software-based energy profilers

Over the last decade, many alternatives for physical power meters have been proposed, often based on a power model that estimates the power or energy consumption. Since the results are based on estimation, these tools will not be as accurate as physical power meters. However, they are easier to set up and use, and often enable the user to measure the energy consumption of single hardware resources (e.g. CPU cores) instead of the whole machine. For this study, we will use the energy consumption data directly from the corresponding AMD registers. To provide an overview of readily available alternatives, we will also cover several general-purpose energy profilers for Linux. The profilers obtain the measurements from model-specific registers as well. Therefore, we will also describe Intel's Running Average Power Limit, which is a commonly used interface that stores energy consumption counters in these registers.

Probably the easiest way to measure the energy consumption in Linux, is by using the `perf` command, also known as *perf_events* [29]. *Perf_events* is built on Intel's RAPL, and is primarily compatible with Intel architectures [27]. *Perf_events* is (semi-)compatible with AMD architectures, as they provide similar register contents [30]. Among other things, *perf_events* is able to measure the energy consumption of a single application run. Other options to measure power consumption include *Powerstat*[1], which is a tool to measure the power consumption of compatible Intel systems, more specifically a machine that supports the RAPL interface [31]. *Powerstat* will display the statistics regarding power consumption at the end of a run, which includes the average, standard deviation, and minimum and maximum values of the data. Since *Powerstat* provides the power consumption in Watts, the energy consumption in Joules must still be calculated afterward by multiplying the average power (in Watts) by the elapsed time (in seconds) [27]. Another tool to measure power consumption is *PowerTOP*, which, unlike *Powerstat*, is compatible with both AMD and Intel architectures. Similar to *Powerstat*, the energy consumption must still be calculated afterward from the power consumption statistics.

Running Average Power Limit

Most of these software energy profilers use Intel's Running Average Power Limit (RAPL). RAPL is an interface, provided by Intel processors since the Sandy Bridge architecture [32], used for measuring and limiting energy consumption from several power domains [33]. RAPL supports energy reporting for (a selection) of the following power domains [32]:

1. **Package (PKG):** This domain reports the energy consumption of the entire socket, which includes all cores, integrated graphics, last level caches, and the memory controller.
2. **Power Plane 0 (PP0):** This domain, also known as the core domain, reports the energy consumption of all cores on the socket.
3. **Power Plane 1 (PP1):** This domain, also known as the graphic domain, reports the energy consumption of the GPU on the socket.
4. **DRAM:** This domain reports the energy consumption of the RAM attached to the memory controller.
5. **PSys:** This domain reports the power specifications of the entire system-on-chip (SoC).

The power domains, which are illustrated in Figure 2.1, that are supported on a machine depend on the processor model. Where modern Intel processes will support all domains, older or non-Intel processes might only support a selection of these power domains.

Model-specific registers

The energy consumption counters provided by RAPL or similar interfaces are stored in model-specific registers (MSRs). These MSRs are registers provided by the processor that enable system software to interact with several of its features (e.g. performance monitoring). This interaction includes reading or writing to these MSRs [33]. Therefore, it is possible to obtain energy consumption data directly from these MSRs. In the *MSR_RAPL_POWER_UNIT* units for power, energy, and time can be read. Furthermore, the RAPL energy data for the PKG and PP0 domains can be accessed through *MSR_PKG_ENERGY_STATUS*

[1] `https://github.com/ColinIanKing/powerstat`

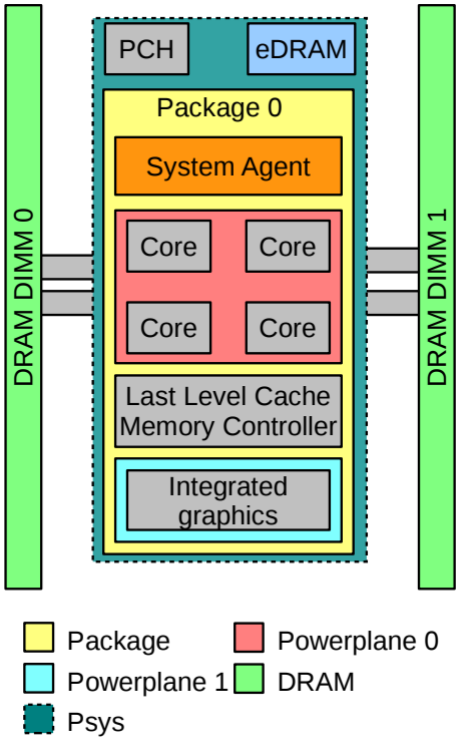

Figure 2.1: Overview of the power domains [32].

and *MSR_PP0_ENERGY_STATUS*, respectively. For several AMD processors similar MSRs are available: *AMD_MSR_PWR_UNIT*, *AMD_MSR_CORE_ENERGY* and *AMD_MSR_PACKAGE_ENERGY*.

2.2 Virtualization

The term virtualization is described by *VMware* as the separation of a resource for a service from the underlying physical delivery of that service [34]. Virtualization has become a widely used technology in many different areas; it provides isolation and hardware independence and increases the ability to scale [1]. Nowadays, when referring to virtualization, there are two main approaches: hypervisor-based and container-based.

Figure 2.2 illustrates the differences between hypervisor-based and container-based virtualization. Hypervisor-based virtualization allows the abstraction from the hardware, by emulating the complete hardware of a system. Consequently, this virtualization approach is often associated with virtual machines (VMs) [35]. These VMs use software that enables running a guest operating system, such that it represents a real machine [34]. The emulation of the hardware is achieved with a hypervisor, which is a software layer that acts as an interface between VMs and the hardware of the underlying system [36]. This hypervisor enables the execution of multiple VMs on a single machine.

VMs, or hypervisors in general, have been the virtualization technology of choice for managing, packaging, and deploying applications for a long time. However, over the past decade, containers have been increasing in popularity, since they promise lower overheads than VMs [2, 3].

2.2.1 Containerization

In contrast to hypervisor-based virtualization, container-based virtualization (or containerization) does not emulate the complete hardware of a system. Instead, it performs an operating system (OS) level virtualization [37]. This enables multiple containers to share the OS kernel of the underlying system [36]. This underlying system is called the container host [38]. Containers only contain the application, the OS, and the corresponding libraries and dependencies, which makes them more lightweight than VMs.

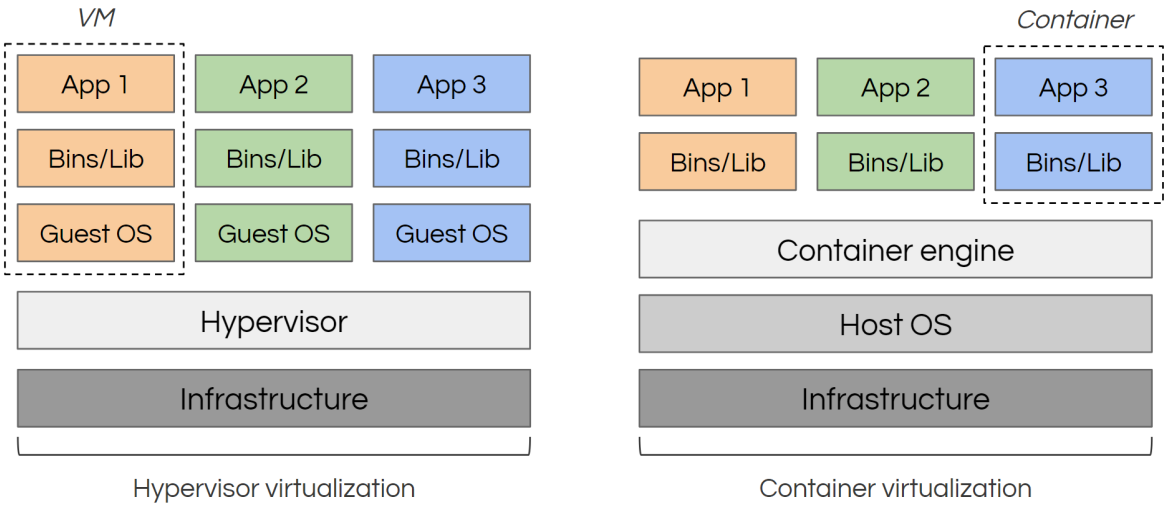

Figure 2.2: Comparison between hypervisor-based and container-based virtualization

The most well-known container-based virtualization platform, and also the platform that made containerization itself relevant [12, 39], is Docker. In order to properly containerize applications, Docker uses several features of the Linux kernel to isolate resources, in particular namespaces and control groups. These two features are the basis of Docker and other container runtimes based on Linux containers, which are essentially Linux processes.

Namespaces
The main idea of namespaces is to isolate Linux processes from each other. The usage of namespaces allows a set of resources to be seen exclusively by a set of processes that has the same namespace. In other words, each process is assigned a namespace, which indicates the resources this process can see and use. These resources can appear in multiple namespaces. Docker and similar container runtimes create these namespaces in order to isolate containerized applications. There are different kinds of namespaces within the Linux kernel [40].

Control groups
Control groups (cgroups) are a Linux kernel feature that allows allocating resources among defined sets of processes. These resources include CPU utilization, memory utilization, network bandwidth, and disk I/O, among other things. Besides limiting the use of resources, cgroups also allow prioritizing processes when there is resource contention. Ultimately, cgroups enable container runtimes to allocate available hardware resources of the host machine among the containers [41].

2.2.2 Docker
Docker is a platform based on open-source technologies, that provides a convenient way to develop, distribute, and run containerized applications. While Docker was originally a project that was built on Linux Containers (LXC), since version 1.0 it has used its own environment in *libcontainer*. Where LXC initializes an operating system for each container, Docker uses a single operating system environment, provided by the Docker Engine. Consequently, whereas LXC was mostly used to isolate an operating system in a container, the primary usage of Docker is to isolate a single application in a container. Currently, Docker is still the most used container engine [6], which is the main reason we will be using it for our study.

Docker Engine
Docker Engine is the actual technology that is used to build and manage containerized applications [42]. Docker Engine has a server-side, the Docker daemon (*dockerd*), and a client-side, the Docker command line interface (CLI). The client communicates with the daemon via a REST API. This daemon listens to API requests and is capable of creating and managing images, containers, networks, and volumes. Besides initiating the building of new images from Dockerfiles, the daemon can also pull existing images from

registries to the Docker host, which is the server on which the daemon is running. The daemon is similar to the hypervisor for VMs, as it allocates resources to containers and isolates containers from the host machine and other containers. However, the main difference between the daemon and a hypervisor is that the daemon does not separate guest operating systems for the containers [43].

The Docker daemon relies on *containerd* (see Figure 2.3), which is the high-level container runtime, to use the Linux kernel namespaces and control groups [44]. It manages the entire container lifecycle of the host system, which includes starting and stopping containers, and pulling and pushing images [45]. In order to create and run containers, *containerd* uses the low-level container runtime *runC*, which adds a fully compatible API on top of *libcontainer* [46]. This container runtime is compliant with the OCI[2] specification, which defines the open industry standards for containers and container runtimes. *RunC* prepares the runtime for containers, which also includes creating the namespaces. It only lives while starting the container process, after which it exits. The layer between *containerd* and *runC* is the *containerd-shim*. The shim is used to abstract the low-level runtime *runC*, and lives as long as the container process [47].

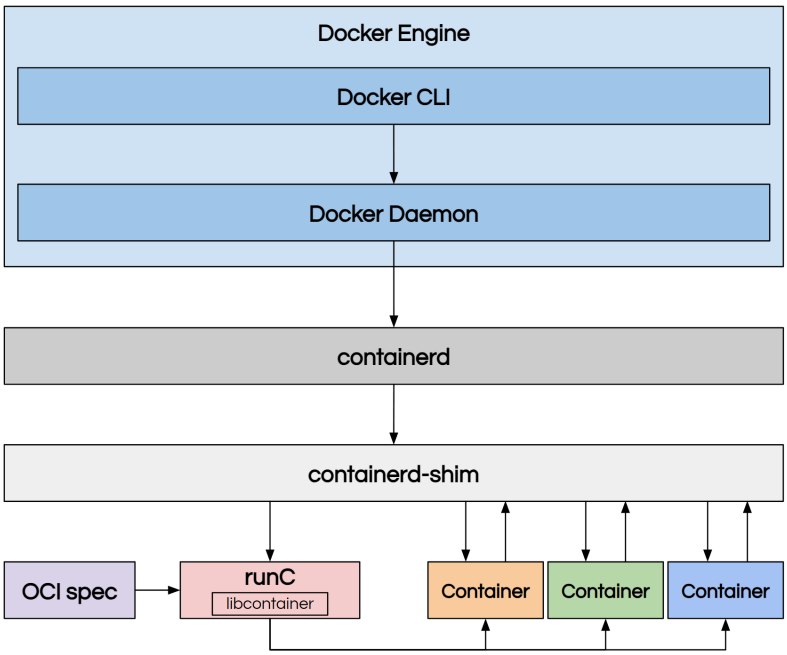

Figure 2.3: High-level overview of the layers in the Docker architecture.

Dockerfile, image, and container

The Dockerfile and (container) image are essential parts of the process of containerizing an application, as they determine what the container will look like. Figure 2.4 shows the relationship between a Dockerfile, Docker image, and Docker container.

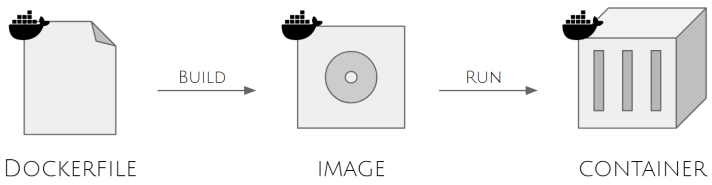

Figure 2.4: Relationship between the Dockerfile, image, and container

A Dockerfile is a text file that contains instructions and configurations. Each instruction creates a layer in order to build a new image, which is used as a template for containers [48, 49]. The instructions in the

[2] https://opencontainers.org/

Dockerfile are executed in chronological order, where the first instruction must be the *FROM* instruction [50]. The *FROM* instruction indicates the base image from which the new image will be built. In order to copy local files or entire directories to the container, the *COPY* instruction can be used. The *RUN* instruction contains commands that construct the container and is usually used to install libraries and dependencies that are needed inside the container. There can be multiple *RUN* instructions in the Dockerfile. Finally, the *CMD* instruction defines the command that must be executed when the container is running. There should only be a single *CMD* instruction in the Dockerfile, and if this is not the case, only the last *CMD* is applied [49]. While there are more frequently used Dockerfile commands, most Dockerfiles contain at least the aforementioned commands.

A Docker image is a template with instructions in order to create and subsequently run a container [42]. An image consists of multiple layers in hierarchical order. These layers represent changes or additions to the image, and are read-only. A benefit of the layer mechanism is that it allows reusing layers if there are multiple images or containers that contain the same layers [51]. Furthermore, when changes are made to the Dockerfile, only the layers that are affected by the changes are rebuilt. The base layer, which is also the oldest layer, usually contains the operating system (e.g. Debian Linux). Consequently, many popular language- or application-specific images are based on OS base images (see Figure 2.5).

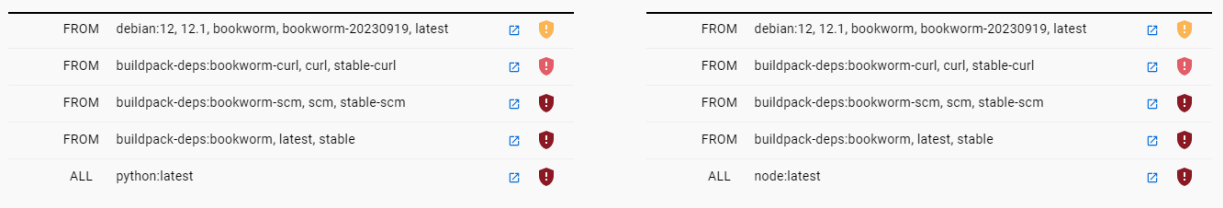

Figure 2.5: Image hierarchy of the latest *Python* (left) and *Node* (right) images.

A container is a runnable instance of a container image. Containers are the actual virtual environments containing all the packages and dependencies that are defined in the Dockerfile. When an image is used to create a container, a writeable container layer is added on top of the existing layers. Essentially, the difference between a container and a container image is the writable layer on top of the read-only layers of the image. This writeable layer contains the changes that are made during the runtime of the container. However, when the container is deleted, this writable container layer is deleted.

In order to store data outside a container, Docker provides several options. *Volumes* are the preferred mechanism for persisting data used and generated by containers [52]. When a container is started, Docker mounts declared volumes onto the filesystem of the container. Changes will be applied to the volume instead of the writeable container layer, if the container writes to a path beneath a volume mount point. These changes will still be available after stopping the container. Another option to persist data is to use a *bind mount* [53], which mounts a file or directory on the host into a container. While volumes create a new directory within the storage directory of Docker on the host, a bind mount references a file or directory by its absolute path on the host machine. Finally, *tmpfs mounts* allow containers to create files outside the writeable container layer [54]. However, in contrast to volumes and bind mounts, tmpfs mounts are only persisted in the memory of the host, and therefore temporary. Figure 2.6 shows the difference between the aforementioned options.

Docker Hub

Docker Hub is the largest public registry for Docker images. This registry contains over 100,000 public container images from independent software vendors, open-source projects, and individual developers. In total more than 20 billion images are downloaded from Docker Hub each month. Docker Hub provides *Docker Official Images*, which are a curated set of essential base images. This set includes base images for operating systems, programming languages, and other essential tools (e.g. databases). Before an image is accepted as an official image, proposed Dockerfiles are reviewed according to defined guidelines. These guidelines can be found in the Docker Official Images documentation on GitHub[3], and ensure

[3] `https://github.com/docker-library/official-images`

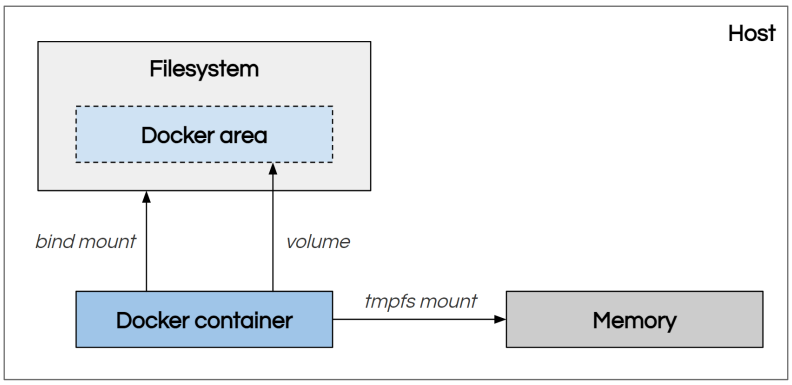

Figure 2.6: Difference between volumes, bind mounts and tmpfs mounts

that the Dockefiles adhere to best practices[4]. Furthermore, official images should primarily focus on free open-source software, and should support multiple architectures. To ensure that an official image contains the latest updates and security fixes, they should be actively rebuilt.

Docker Compose

Docker Compose is a tool for defining and running multiple containers. These containers are defined in a YAML configuration file and are called services. Compose can rebuild, start, and stop all services that are defined in the configuration. Furthermore, it is possible to configure the dependencies of the services in this YAML file. The main idea of Compose is to create one or multiple isolated environments on a single host. Each isolated environment has its own network setup by default. This ensures that each container in the environment is connected to this network, and can communicate with other containers in the network, by using the container name as hostname. Docker Compose is especially useful for development and automated testing environments, as it provides an isolated environment that can be created and destroyed in a convenient manner [55].

[4] `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`

3

Related work

This chapter will give an overview of work in the field that is related to this study. While optimization of the energy efficiency of Docker is still a relatively new topic, there are several studies that are relevant to this topic. These studies include both the energy consumption of Docker and the optimization of Docker artifacts. Furthermore, research about optimizing energy efficiency in the software engineering field shows the relevance of reducing the energy consumption of software, and therefore also of containers.

3.1 Optimization of Dockerfiles

In the last few years, there is a decent amount of research done regarding the optimization of Dockerfiles, images, and containers, resulting in recommendations, and propositions for optimization tools. Rosa et al. [56] propose approaches and techniques to assess and improve the quality of Dockerfiles and Docker images. They followed this study up with another study on fixing Docker smells, where they presented a plan to investigate what smells receive more attention from developers and if developers are willing to accept changes aimed at fixing smells. Another study by Zhang et al. [19] examines how evolutionary trajectories and architecture factors affect the Dockerfile quality and the build latency of the Docker image. Among other things, they found that more image layers and larger image layer sizes may result in longer build latency.

There are also approaches that help developers with the process of writing Dockerfiles, while maintaining the quality. Zhang et al. [4] tackles the problem of automatically recommending base images through Dockerfile configuration. Finding a proper base image manually is a difficult but important task, since base images may have an effect on the quality, efficiency, and size of the resulting images. The proposed approach characterizes the Dockerfile with AST paths and trains a path-attention model to recommend a base image. *Parfum* is a tool, proposed by Durieux [57], that also is based on an AST parser. However, this tool focuses on detecting Docker smells, and providing suggestions for repairing the smells. These suggestions are based on rules that follow best practices for Dockerfiles. Another tool to improve Dockerfiles is *Dockerlive*, proposed by Reis et al. [58], which tries to tighten the feedback loop between the developer and the system, by providing a live development environment for writing Dockerfiles. Among other things, this tool provides continuous indication of image build and runtime errors, container resource statistics, and image build and container log output.

The aforementioned approaches and tools mainly focus on the quality of Dockerfiles and the consequences it has on the size and build latency of the images. In contrast to our study, these approaches do not investigate how this relates to the energy consumption of the runnable instance of the image.

3.2 Energy consumption of virtualization environments

The sustainability of virtualization environments is something researchers are concerned with since there are studies about the energy consumption of these virtualization environments such as Docker. Santos et al. [10] published one of the first papers that investigates the energy consumption of Docker. They compared ”off-the-shelf” Docker images with bare metal Linux, conducting experiments with different

workloads. They found that in all cases Docker has a significantly higher energy consumption.

However, when applications require different environments, bare metal Linux might not be the best solution, which creates the need for virtualization. Therefore, it seems more logical to compare Docker to other virtualization solutions. Morabito focuses on this comparison in the paper on the power consumption of virtualization technologies [12]. They compare container-based solutions with hypervisor-based solutions (i.e. virtual machines), with regard to power consumption. They found that both solutions perform similarly in an idle state, in a CPU performance test, and in a memory performance test. However, the container-based solutions perform better in the network performance test, since network packets in hypervisor environments have to be processed by extra layers.

A similar study is performed by Tadesse et al. [5]. However, in this paper, they compared specific hypervisor-based and container-based solutions, namely VirtualBox and Docker, respectively. They found that Docker has lower CPU usage and power consumption. They obtained similar results as Morabito for the CPU, memory, and network performance tests. However, here they found that for VirtualBox a higher fraction of the CPU is used for virtualization overhead, which means that less of the CPU is used for actual work. Based on the experiments they found that CPU usage is the main driver of power consumption in virtualization environments, with a linear relationship between them.

As mentioned in Section 2.2, virtualization itself has proven to be a viable solution for managing and deploying applications, since it provides isolation and increases the ability to scale. The studies about the energy consumption of virtualization environments show that containerization is in general the more energy-efficient solution compared to virtual machines. This makes containerization the more interesting virtualization solution in terms of sustainability. Hampau et al. empirically assessed the energy consumption of three different containerization strategies for AI, including Docker [13]. For the experiment, they used 4 computer-vision algorithms with the three containerization strategies, and monitored the resource utilization and energy consumption. They found that the portable and contained Docker approach results in a fast execution time and reduced energy consumption.

In the experiments of these previous studies, virtualization environments and containerization solutions are the independent variables. Therefore, they do not investigate possible factors within containerization that impact energy efficiency. A recent study investigating the energy consumption of several containerized workloads was conducted by Warade et al. [14]. These containerized workloads include web server containers (i.e. Nginx vs. Apache) and database containers (i.e. Mongo vs. Postgres). Furthermore, the energy consumption during the start-up phase of several popular containers was evaluated. Their results showed that the energy efficiency of the workload differs for each container. Although this study is similar to ours, it does not use the same workload for different base images, but rather compares app-specific container images.

3.3 Energy monitoring and profiling

While the previous studies described in Section 3.2 performed their analysis with physical power meters, there are also software-based approaches that use the system hardware to measure the energy consumption of containers. Brondolin et al. propose a monitoring tool in the system itself [59]. This tool is able to assign precise information about power consumption to each thread and container running in the system, while adding negligible overhead in the running system.

Fieni et al. [60] propose a power meter for containers: *SmartWatts*. *SmartWatts* automatically adjusts its power model whenever deviation from the ground truth is detected. It consists of a sensor (client-side), which is run on target nodes that need to be monitored, and a power meter (server-side), which is a remote service that can be deployed whenever needed. *SmartWatts* builds upon RAPL, to collect baseline measurements for CPU and DRAM power consumptions, and Linux's *perf_events* interface, to capture the Hardware Performance Counters (HwPC) events to estimate per-container power consumption from resource-specific power models.

Another approach to collecting energy and performance data from containerized applications is proposed by Silva-de-Souza et al. [61]. They introduced *Containergy*, which is a software-based profiling tool. *Containergy* combines software containers, control groups (cgroups), hardware performance counters, and dynamic voltage and frequency scaling (DVFS). This combination is used to isolate the application from other running tasks on the target system. After the container is set up, the tool first collects raw profiling

data in runtime, and then it assembles the asynchronous application data and the raw performance data. Finally, this combined data is parsed and processed, after which a comprehensive profiling dataset is generated.

Software-based energy profilers do not need any external hardware, which makes measuring the energy consumption of containers easier and therefore more accessible to developers. While these profilers are usually less accurate than physical power meters, they can still provide meaningful insights to optimize the energy efficiency of containerized applications. Using software-based profilers, we can determine if certain differences between containers have an impact on the energy consumption. An important difference between containers is their base image. There are many different base images that developers can choose from, and therefore it could be interesting to investigate how the choice of this base image affects the energy consumption of the container.

3.4 Docker image comparison study

Although there are many base images to choose from, there are not many studies about the differences between these images and how they impact the performance of the container. These base images are available in container image registries such as Docker Hub. Ibrahim et al. [15] investigate the differences among five types of images (Cassandra, Java, Mongo, MySQL, and Nginx) on Docker Hub, with the goal of helping developers choose an image suitable for their needs. In their experiments, they look at the differences in terms of installed libraries, size and resource efficiency, and security vulnerabilities.

Zerouili et al. [62] carried out a large-scale study specifically of Debian-based images on DockerHub. Their analysis is based on 140,498 Debian images, which include both official and community images, with *Stable*, *OldStable*, and *Testing* Debian distributions. In their comparison, they consider five dimensions of technical lag: package lag, time lag, version lag, vulnerability lag, and bug lag. The goal of this study is to provide evidence that developers can benefit from identifying technical lag, in order to mitigate risks.

Another approach to recommending base images is proposed by Zhang et al. [4] in, which is also mentioned in Section 3.1. This paper introduces an automatic approach for recommending base images based on the Dockerfile.

Finally, a recent study by Rosa et al. [18] investigated what factors influence the adoption of Docker images. While energy efficiency is not included as a factor, this study provides an overview of the current selection criteria for base images.

These approaches can help developers to choose a base image that is suitable for their needs. However, these studies do not take the possible differences in energy efficiency into account. Since there are often many base images that provide the same functionality, it is also beneficial to investigate if there are differences in terms of energy efficiency, especially from a sustainability standpoint.

3.5 Energy efficiency in other SE fields

Although this thesis focuses on the energy efficiency of containers, studies about energy efficiency in other software engineering fields can offer useful insights, especially since there are not many studies about the energy efficiency of containers.

Palomba et al. [63] conducted a study on how code smells affect the energy consumption of mobile applications. They investigate to what extent code smells influence the energy efficiency of mobile applications, and whether refactoring these smells directly improves the energy efficiency. Their experimental results show that 94% of the most consuming methods in their dataset were affected by at least one code smell, which might indicate that there is a relationship between code smells and energy efficiency of methods. Furthermore, for several code smells, the corresponding method consumed significantly less energy after they were refactored.

Tsirogiannis et al. [24] were the first to report on the power profile of database operations (e.g. scans, hash joins, and sorts). They conducted experiments with Postgres and a commercial DBMS, to observe energy efficiency effects of varying software and hardware knobs. In their results, they found that CPU power is not linear with CPU utilization and that in most of the experiments, the most energy-efficient configuration was also the best performing.

Another study conducted by Capra et al. [64] examines the relationship between the use of traditional development environments and the energy efficiency of software. The motivation for this study was the observation that software contributes significantly to the overall energy consumption of IT.

3.6 Research gap

The studies about energy efficiency in software engineering show that researchers are concerned with optimizing the energy consumption of software. This is also the case for containerization and Docker in particular, as research has been conducted about monitoring the energy consumption of containers. Although there are also studies about the optimization of Docker artifacts, they barely focus on the optimization of the energy consumption of containers. A recent study by Warade et al. [14] acknowledges this problem and provides a starting point for research in this direction. However, this study only compares specific workload containers (e.g. Mongo and Postgres) in their experiments. In contrast, this thesis will focus on how different base images affect the energy consumption of certain workloads in Docker. The impact of base image selection on the energy efficiency of Docker containers is currently a gap in the existing literature. Ultimately, this thesis will provide further insights into factors that impact the energy efficiency of containers.

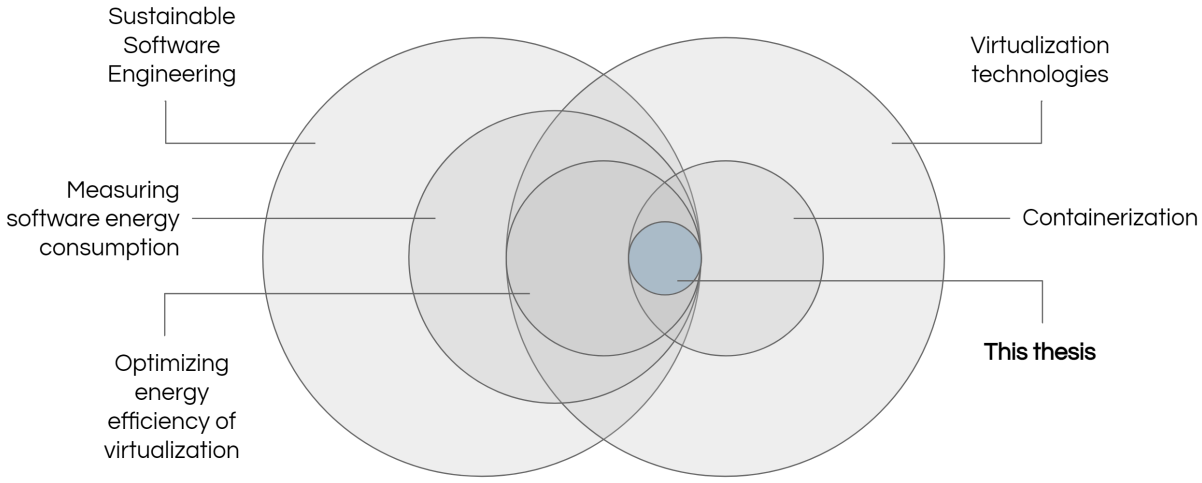

Figure 3.1: Research gap and the focus of this thesis

4

Methodology

This chapter will describe the experimental design of the study, including the procedure of measuring the workloads and the experimental setup. Furthermore, we will describe the base images used in the experiment as well as the workloads run in the containers. Finally, the evaluation after obtaining the data from the experiment will be explained.

To assess the impact of a base image on energy efficiency, we will run several container workloads (see Section 4.3), which reflect real-world use cases of Docker. Each container workload will use the same set of selected base images (see Section 4.2). During the run time of a workload, the energy consumption of all individual CPU cores, and the GPU is measured. Additionally, the CPU utilization, CPU frequency, CPU temperature, and GPU utilization are monitored as well. The measurement of the energy consumption for each workload is done 30 times for every base image in the set. This is done to mitigate external factors affecting the reproducibility of our experiment. After the measurements are obtained, the data is analyzed to determine (significant) differences between the images.

As mentioned in Section 2.1, energy efficiency for software is often defined as the ratio between performance (or work done) and energy consumption. However, since the workloads (i.e. work done; performance) are the same for each base image, we can directly evaluate the energy efficiency by looking at the energy consumption. Ultimately, the goal of the experiments in this study is to obtain a better understanding of energy efficiency optimization of Docker containers, and the role of base images in this.

4.1 Experimental design

In this study, we will use a software-based energy profiler to measure the energy consumption of the workloads within the containers during the experiments. As stated in Section 2.1, software-based energy profilers are not as accurate as physical power meters, however, they offer the advantage of an easier setup and often make it possible to measure the energy consumption of an isolated core, on which the application is running. Moreover, our focus in this study is not to obtain precise energy measurements, but to assess the relative differences in energy efficiency between base images.

Besides the total energy consumption, a requirement for the energy profiler is to provide energy consumption per hardware resource. This is necessary since the experiments are done on a single machine. This means that the possible background processes (e.g. client-side) of the workload are executed on the same machine as the workload itself. By having individual measurements per physical core and allocating a physical core to the workload, it is possible to determine the energy consumption of the workload without interference from the background processes. Furthermore, the energy profiler should have measurements per interval. These measurements can then be used to evaluate power usage or energy consumption over time. Taking these requirements into consideration, we decided to use an energy profiler provided by Durieux[1]. The main reason for this decision is that most commonly used profilers do not meet all requirements, or are not fully compatible with the test machine, which uses an

[1] https://durieux.me/

AMD architecture. The energy profiler we used for the experiments extracts the energy consumption data directly from the model-specific registers (i.e. AMD_MSR_PWR_UNIT), as described in Section 2.1.2.

4.1.1 Energy consumption measurements

Each experiment in this study will follow the same procedure. At the start of each workload experiment, the images for the containers are built from the predefined Dockerfiles; one for each OS base image. We used Parfum[2] to ensure that these Dockerfiles adhere to the best practices. Since different Linux distributions might use different package managers, we tried to keep the differences between the images minimal by using equivalent packages that are available for the respective distributions.

Before running the workload, it is essential to first warm up the machine. This warm-up prevents initial executions from appearing more energy-efficient than subsequent executions of the same workload. This is because there is a strong relationship between energy consumption and the temperature of the hardware [65]. As part of the warm-up procedure, we ran the workload for each base image without measuring the energy consumption. This is also done to ensure that the first runs of the experiment are not affected by starting up Docker containers for the first time. Additionally, to properly warm up the machine, it is recommended to run a CPU-intensive workload for at least 5 minutes [65] before conducting the first execution of each workload experiment. The prime number test from Sysbench [66] is used to stress the CPU and consequently warm up the machine. The Sysbench test is run on all available threads for 6 minutes.

After the warm-up, the workload is run and measured 30 times for each image, in random order. Since energy measurements can be inconsistent, it is important to repeat each measurement multiple times. The recommended minimum number of measurements is considered to be 30 [65], and ideally, the samples form a normal distribution. This 'magic number' originates from the Central Limit Theorem (CLT), which states that as the sample size gets larger the distribution of the sample means will be approximately normal. With a few exceptions, it is usually considered that the sample size has to be at least 30 to use the normal approximation [67].

As mentioned before, the execution of the workload using the different images will be run in random order (e.g. *ubuntu* $\rightarrow$ *alpine* $\rightarrow$ *debian* $\rightarrow$ *alpine* $\rightarrow$ *centos* $\rightarrow$ ...). Running the experiments in random order will help mitigate possible biases from external factors. For example, there could be a difference in room temperature over time, which would affect all base images instead of a single base image if the execution order is random. In between each execution, there will be a short pause of *20* seconds to prevent residual energy from the previous execution from interfering with the next one. Furthermore, it is important that this pause is not too long, which could result in the machine cooling down and providing inconsistent results.

Since the experiments were run on a single machine, the workload containers are run on isolated cores, while the other containers for clients and other processes are run on the remaining cores (see Figure 4.1). If the workload allows the usage of multiple threads, the workload is run on two threads. These two threads are on the same physical core, while the other cores are idle or used by possible background processes of the workload. If the workload only uses one thread, the other thread on the same physical core is still reserved for this workload, and is therefore idle. By reserving physical cores instead of single threads, we can obtain the estimated energy consumption by measuring the energy consumption of that specific core, without other processes interfering with the measured workload. Furthermore, we lock the CPU frequency (i.e. 4700 Mhz) of this core, to ensure that this frequency stays consistent for all runs.

4.1.2 Experimental setup

The experiments are run on a machine equipped with an AMD Ryzen 9 7900X processor (12 physical cores, 24 threads), 64 GB RAM (2x32GB DDR5 @ 5.600MT/s), and a 1000W power supply. The dedicated graphics card of the machine is an MSI Geforce RTX 4090 (24GB GDDR6X memory), which will be used for the *llama.cpp* workload with the CUDA image. The machine has a x86_64 architecture and Ubuntu 22.04.3 as the operating system, with Linux kernel version 6.2.0. For the containerization and execution of the workloads, we used version 24.0.5 of Docker Engine.

[2] `https://durieux.me/docker-parfum/`

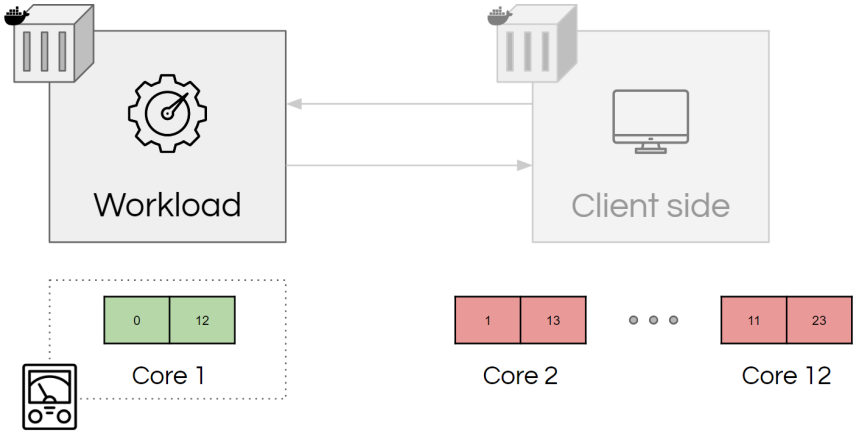

Figure 4.1: Allocated core for the workload

4.1.3 Reproducibility

To summarize the entire experiment, each workload will follow the following procedure:

1. Build the images from the provided Dockerfiles
2. Warm up the machine by running the workload for each base image and a prime number test for 6 minutes
3. Run and monitor the workload for each base image in randomized order, with:
    - 20s sleep between each run
    - the power measurements for each core and timestamps recorded every 100ms
    - the containerized workload on a separate core
    - a total of 30 runs for each base image
4. Calculate the total energy consumption of each run using the measurements

The Python scripts and the corresponding Dockerfiles and workloads can be found on GitHub[3]. The measured energy data, and the summaries of the 30 runs per base image can be found on Zenodo[4].

4.2 Image selection

While language base images have significantly increased in popularity over the last few years [16, 17], we will be primarily focusing on operating system (OS) base images in our experiment. One of the reasons is that we will focus on general use cases, to compare different base images, while the workload stays the same. When looking at language base images, it might be difficult to compare different language versions (e.g. Java and Python) of an application with each other, since the programming language itself also has an impact on energy efficiency. Likewise, the comparison of images of specific database technologies (e.g. Mongo and Postgres) will not provide general insights into the impact of base images. This makes it more difficult to quantify the effect of the base image choice on the energy efficiency of containerized workloads.

Another reason is that most of these application-specific base images (e.g. Python, NGINX, and Redis) are based on OS base images regardless, as mentioned in Section 2.2.2. Furthermore, multiple OS base images are still in the top 10 of most used base images in Dockerfiles [16, 17], which indicates these images are still relevant for most use cases. The base images used in the experiments are selected based on the following requirements:

- The image should be a Linux distribution since the underlying host OS is a Linux distribution
- The image is a *Docker Official Image* and published by a *Verified Publisher* on Dockerhub

[3] `https://github.com/btjiong/docker-energy`

[4] `https://zenodo.org/records/10220979`

- The image is still widely used (at the moment of the experiment); this can be shown from either a recent study [6, 16, 17], or the number of pulls per week on Dockerhub (>500,000 pulls/week)

These criteria ensure that the selected base images provide a general understanding of their impact on the energy consumption of containers. Furthermore, the images in this study are still relevant as they are widely used by developers, and are reliable and consistent as they are Docker Official Images. Using these criteria we decided to use the following set of OS base images:

1. **Debian:** Debian is a free-to-use open-source operating system, which uses the Linux kernel [68]. It is one of the most popular Linux distributions, and many other Linux distributions are built on Debian, as shown in Section 2.2.2.
2. **Ubuntu:** The latest known studies on large sets of Dockerfiles and images [16, 17] show that the Ubuntu base image is still the most used base image under developers. Ubuntu is an open-source operating system, built on the Debian architecture and infrastructure [69].
3. **CentOS:** While the CentOS image is no longer receiving support and maintenance, it remains one of the most commonly used OS base images [6, 17]. CentOS is a Linux distribution and is originally derived from the Red Hat Enterprise Linux (RHEL), which is often utilized in cloud infrastructures [70].
4. **Alpine:** The Alpine base image is the second most used OS base image [16, 17], mostly because of its small image size. Alpine Linux is a lightweight Linux distribution for general purposes and is built around busybox[5] and musl libc[6].

For the experiments, we used *Debian 12.0*, *Ubuntu 22.04*, *CentOS 8*, and *Alpine 3.18*. These OS base images cover a wide range of Docker base images since many programming language and tool-specific base images are based on these as well. We selected the latest version of the base images at the time of conducting the experiment. As Ubuntu is a Linux distribution based on Debian, it is interesting to see if these images perform similarly as well. In contrast, CentOS is based on Red Hat and might perform differently. Finally, Alpine is a completely different Linux distribution and is often used for its small image size. The selection of base images with different image sizes will help to determine whether smaller images also lead to more energy-efficient containers.

Furthermore, for some workloads, we also measured specialized base images as an extra comparison. Comparing the energy efficiency of these specialized base images with the OS base images can provide insights into whether these base images are actually better optimized. For the *llama.cpp* workload we used the official *CUDA* base image to measure the machine learning inference while using the GPU. Furthermore, for the *Cypress Real World App* workload, we used the official *Node 16* and *Node 16 Alpine* base images since the workload is a *Node.js* application.

4.3 Workloads

To ensure that the evaluation of the energy efficiency of Docker containers is meaningful, it is critical to select appropriate workloads that accurately reflect real-world scenarios. This is necessary to obtain results that offer valuable insights for Docker container use cases in practice.

There are several requirements that must be met to guarantee that the workloads are an accurate representation of real-world scenarios. Firstly, the workloads must be generalizable and can cover a broad range of use cases rather than solely focusing on a specific application. Secondly, the workloads should be easily adjustable, allowing for the simulation of varying demand levels. Finally, it is crucial that the workloads are repeatable, in order to reproduce the results consistently under the same conditions.

In this study, we selected seven workloads. These workloads include a machine learning inference workload, two web application workloads, an online gaming workload, two database workloads, and a video transcoding workload. This selection of workloads is based on their real-world use case, and the prevalence of their related images in Dockerfiles [17]. An overview of the workloads can be found in Figure 4.2. The specifications and details of each workload will be elaborated in greater detail in the following sections.

[5] `https://busybox.net/`

[6] `https://musl.libc.org/`

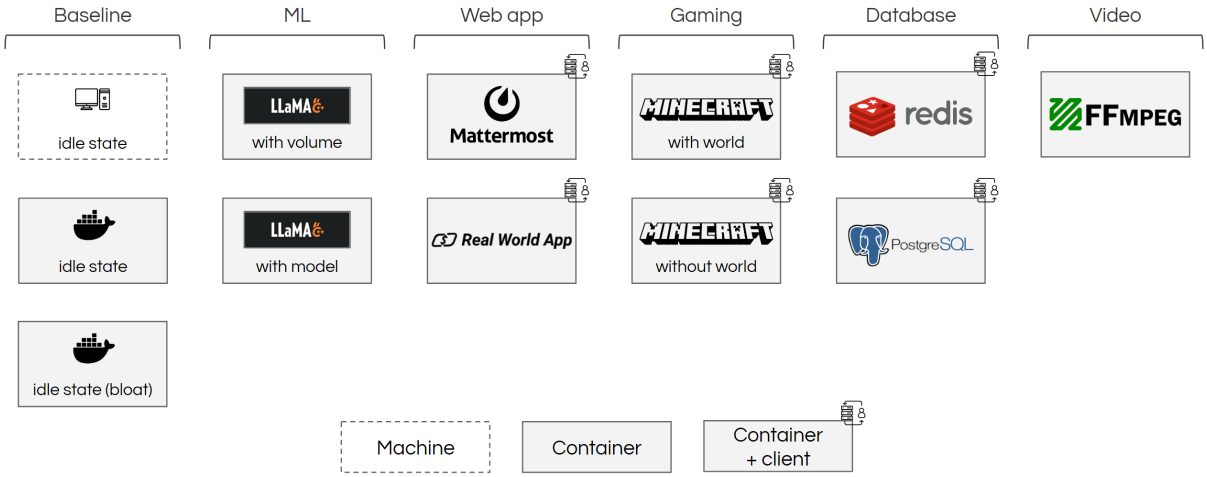

Figure 4.2: Overview of the workloads

4.3.1 Baseline

As the baseline, we used several idle workloads. Firstly, the energy consumption of the machine in an idle state is measured. Secondly, the energy consumption of the machine is measured, while running a container in an idle state. This measurement is done for both an empty container, which only consists of the base image, and a bloated container, which also includes the *llama.cpp* model. For all baselines, the idle state is measured for 2 minutes. These baselines are conducted to confirm the low energy overhead of Docker containers found in previous studies [12, 10, 14]. Furthermore, showing that base images consume similar amounts of energy in an idle state will motivate the decision to investigate a variety of workloads.

4.3.2 llama.cpp (machine learning inference)

The popularity of machine learning applications has been growing steadily over the past decade. Especially the introduction of *ChatGPT*[7] has spiked the interest of the general public in the use of large language models (LLM). Similar to this, Meta recently introduced *LLaMA* (Large Language Model Meta AI), which is a smaller foundational LLM. Smaller foundation models allow researchers who do not have access to many resources to still perform studies with these models [71].

For our experiments we will be using *llama.cpp*[8], which is a port of *LLaMA* in pure C/C++. Since the base images may use different default compilers, building *llama.cpp* during the image build can result in different inference results. Therefore, we build *llama.cpp* before building the image and copy the binary files into the image. This ensures that the inference results are always the same, which means that the measurements between the different base images can be compared with each other.

For the inference, we will use the same parameters as in the example provided by *llama.cpp*[9]. Furthermore, we will use two additional parameters to indicate the seed and lock the memory, to ensure the conditions for each execution are as equal as possible. These are the parameters that are used for each execution:

- -p (prompt): ”Building a website can be done in 10 simple steps:”
- -n (tokens): 1024
- –seed: 12345678
- –mlock
- -t (threads): 2

We will measure a version of the image that uses a volume for the model and another version that

[7] https://openai.com/blog/chatgpt
[8] https://github.com/ggerganov/llama.cpp
[9] https://github.com/ggerganov/llama.cpp

contains the model inside the image. In both cases, we used the *7B* model (13 GB), quantized to 4-bits (3.9 GB).

4.3.3 Mattermost (web application)

Mattermost is an open-source chat platform used for communication and collaboration within organizations and companies. We deploy a self-hosted *Mattermost* instance, which clients can use via the web application, as well as the desktop and mobile applications. For the experiment we will run a *Mattermost* instance in preview mode, using the *Mattermost Docker Preview Image*[10]. To ensure that only the energy consumption of the *Mattermost* instance itself is measured, we move the database to a different container. To simulate traffic on the *Mattermost* instance, we used a subset of the *Cypress*[11] end-to-end tests, provided by the *Mattermost* repository[12]. Since not all provided tests pass in the preview mode, we ensured that the selection of tests actually passes. Furthermore, the tests in this subset correspond to different use cases in *Mattermost*.

4.3.4 Cypress Real World App (web application)

Cypress' Real World App (RWA)[13] is a payment application for demonstration and educational purposes. It is similar to *Venmo*[14], since it is possible to create an account, add bank accounts, and send and retrieve money between friends. The application is developed by *Cypress*, to demonstrate real-world usage of Cypress component and end-to-end testing patterns and practices.

For this workload, the energy consumption of the server-side of the application is measured. When both the client- and server-side of the application are ready, the *Cypress* end-to-end tests that are provided in the repository are run, to simulate regular web behavior.

The *Node 16* base image is used for both the client-side container and the container with the end-to-end tests. We used version 16 because of compatibility reasons. In addition to the 4 OS base images for the workload container, we also measured the workload using the official *Node 16* and the *Node 16 Alpine* base images.

4.3.5 Minecraft server (online gaming)

Minecraft is a Java-based game developed by *Mojang Studios* and is the best-selling video game of all time [72]. *Minecraft* can be played both in single-player mode and multiplayer mode. For multiplayer it is possible to self-host a *Minecraft* server, by using the publicly available server[15]. For this workload the energy consumption of the server inside the Docker container is measured, similar to the experiment conducted by Tadesse et al. [5]. Using another Docker container, a client connects to this server, sends messages to the chat and collects grass blocks, which is automated using the *Mineflayer* package[16]. After 2 minutes the client disconnects from the server. We conducted two experiments for this workload: one where we created the *Minecraft* world during the runtime, and another one where we created the world before building the image.

4.3.6 Redis server (database)

Redis is an open-souce, in-memory data store that is usually used as a database, but can be used as a cache, message broker, and streaming engine as well. The workload we measured is the *Redis* instance inside the Docker container, which acts as a *Redis server*. When the server is ready, another container with the *Redis* benchmark[17] connects to the server. This benchmark simulates database traffic by sending a total of 1,000,000 requests, done by 50 parallel clients. The key space is set to 100,000 to enable possible cache misses, which occur in real-world workloads as well.

[10] `https://github.com/mattermost/mattermost-docker-preview`

[11] `https://www.cypress.io/`

[12] `https://github.com/mattermost/mattermost/tree/master/e2e-tests/cypress`

[13] `https://github.com/cypress-io/cypress-realworld-app`

[14] `https://venmo.com/`

[15] `https://www.minecraft.net/en-us/download/server`

[16] `https://github.com/PrismarineJS/mineflayer`

[17] `https://redis.io/docs/management/optimization/benchmarks/`

4.3.7 Postgres server (database)

Postgres, or *PostgreSQL*, is an open-source relational database. Similar to the *Redis server* workload, the energy consumption of the *Postgres server* is measured, which is a *Postgres* instance installed inside a Docker container. After the server is started, a container with *pgbench*[18], which is the official *Postgres* benchmark, connects to the database. The benchmark first initializes the test database, using a scale factor of 50, which creates 5,000,000 rows in the *pgbench_accounts* table. After this initialization, the benchmark simulates database traffic by running 50,000 transactions, done by 10 connected clients.

4.3.8 FFmpeg (video transcoding)

FFmpeg is an open-source collection of tools and libraries, designed for processing video and audio files. Among other things, *FFmpeg* can be used for transcoding media files, which is the conversion of formats for media files. Video transcoding plays an important role in optimizing the user experience of streaming services [73]. For the video transcoding workload we used the *Big Buck Bunny*[19] movie, which is one of the most popular open-source films and is often used for video codec development [74]. This video is in mp4 format, with a 1920x1080 aspect ratio and 60 frames per second (fps). We used *FFmpeg* to transcode this video to a new mp4 file with a 1280×720 aspect ratio and 24 fps.

4.4 Data analysis

Statistical analysis is crucial after obtaining the energy consumption measurements of software and calculating the total energy consumption of each run, as it can provide insights into differences in energy efficiency between groups. This analysis is used to determine whether the results of the measurements have statistical significance. Subsequently, this statistical significance can then help to assess if there is actual practical significance in the experiment. Figure 4.3 shows an overview of the data analysis.

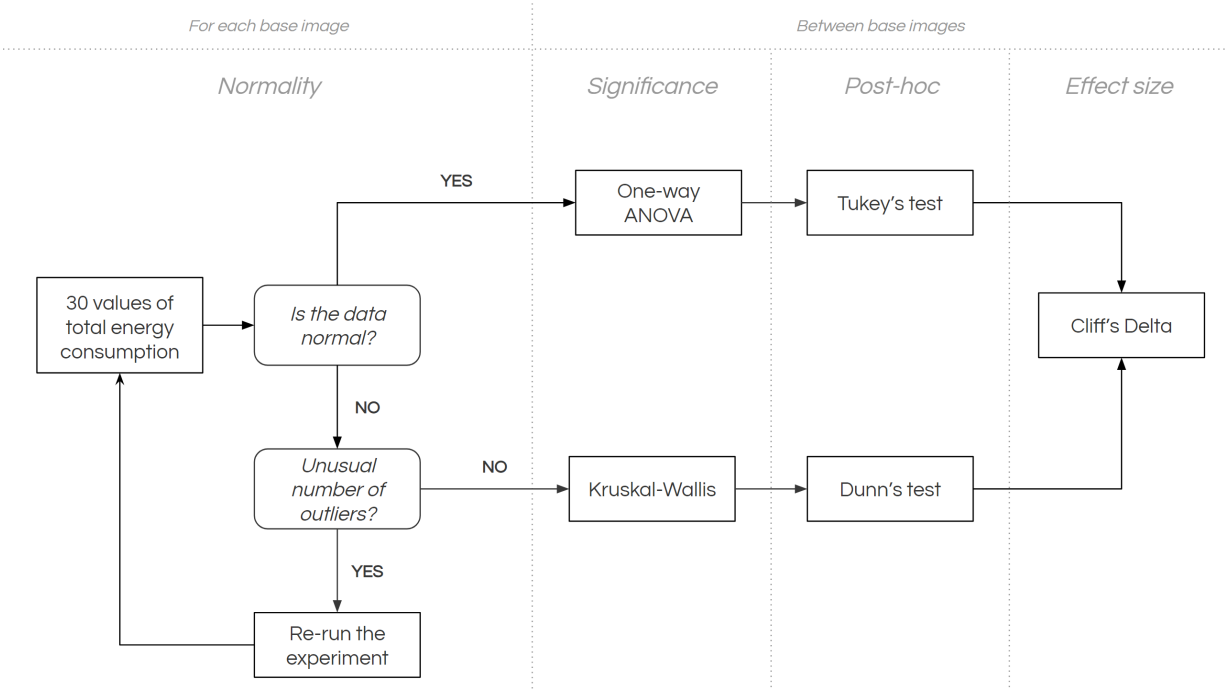

Figure 4.3: Overview of the data analysis

4.4.1 Normality tests

After obtaining energy consumption measurements of three or more groups (i.e. the base images) that need to be compared, it is important first to verify that the set of runs for each group corresponds to a normal distribution. If the distribution is not normal it could mean that the experiment is not deterministic

[18] https://www.postgresql.org/docs/current/pgbench.html

[19] https://peach.blender.org/

and therefore not (easily) reproducible [65]. Another reason could be that there are some measurements that were not executed properly (e.g. different conditions or errors), which may affect the distribution of the data.

To verify that the data is normal, the Shapiro-Wilk test is used, which is an appropriate method for smaller sample sizes (i.e. 30 samples) [75]. The null hypothesis of the Shapiro-Wilk test states that the sample data is drawn from a normal Gaussian probability, and we used a confidence interval of 95% [76, 77].

Since outliers in the data may affect the result of the normality test, the normality of the data can also be assessed using plots [78]. Violin plots, which are a combination of a kernel density estimate (KDE) plot and box plot, visualize the distribution of the data. These violin plots are used as an additional method for determining if the data is approximately normal, even if the normality tests say they are not. The KDE plots of a normal or Gaussian distribution will be symmetrical about its mean.

If the result of the normality test indicates that the data is not normal, and the plots are not approximately normal as well or show an unusual number of outliers, it is necessary to re-run the measurements. However, if the plots are approximately normal but the data is not normal according to the normality test, we used the Kruskal–Wallis test. On the other hand, if the normality tests showed that the data was normal, the regular one-way analysis of variance (ANOVA) test is used.

4.4.2 Statistical significance tests

If the data of all groups are indeed normal, these groups can be compared with each other. Rather than simply comparing the means, it is important to use a statistical significance test to determine if the results of the groups are statistically significant. For experiments with three or more groups, a t-test is not a suitable statistical significance test. Therefore, in our case, one-way ANOVA is an appropriate test. This test is used to determine if the means of three or more independent groups are significantly different. The null hypothesis of one-way ANOVA states that the sample means of the groups are equal, and we used a confidence interval of 95%.

To obtain reliable results for the one-way ANOVA, some data requirements must be met. Firstly, the groups are independent of each other (i.e. samples from one group do not influence samples from other groups). Secondly, the data samples of the groups must be (approximately) normally distributed. Finally, the variances must be (approximately) equal for all groups [79].

If the ANOVA test indicates that the means of the groups are equal, it is safe to conclude that there are no significant differences between the groups. If there are significant differences between groups, the actual significance of these differences between the groups can be obtained using post hoc tests.

However, if the data of the groups are not normally distributed, the Kruskal-Wallis test is used to compare them with each other. This Kruskal-Wallis test is the non-parametric equivalent of the one-way ANOVA test [80]. This means that the test is not limited by assumptions about the distribution (i.e. normality) [81]. The Kruskal-Wallis test is similar to the Mann-Whitney U test, but it can be used to test three groups or more. The null hypothesis of the Kruskal-Wallis test states that the mean ranks of the groups are equal [80], and for this test, we used a 95% confidence interval as well.

4.4.3 Post hoc tests

While the ANOVA tests can indicate the inequality between the groups, it does not provide information on which groups are significantly different. An appropriate method to obtain this information is Tukey's test, which is a commonly used test to determine the significance of differences between pairs of groups. If the data was not normal and the Kruskal-Wallis test was used to test the statistical differences between the images, Dunn's test was used to determine the differences between pairs of images. This Dunn's test is a non-parametric alternative to Tukey's test.

4.4.4 Effect size analysis

After concluding that there are significant differences between the groups, and which pairs of groups are significantly different, the magnitude of the differences should be determined. Since Cohen's d is mostly used for normally distributed data and not all data of the workloads might be normal according to the normality tests, Cliff's delta can be used instead to determine the effect size. Since Cliff's delta is a

non-parametric effect size, it can be used for both workloads with normally distributed and non-normally distributed data. Therefore it can be used as a consistent complementary measure for all groups to assess the actual significance of the differences between the base images. Cliff's delta is a value between *-1* and *1*, based on how often the values in the first distribution are larger than the values in the second distribution. That means that if in most cases the first distribution consumes less energy than the second distribution, Cliff's delta is negative, and if it consumes more Cliff's delta is positive.

To obtain the difference between groups in actual numbers (i.e. Joules), calculating the mean difference between groups is an option. However, if there are outliers in the data, the mean of a group might not provide an accurate representation of this group. Therefore, we will compare the median differences between the groups in our experiments.

5

Data collection

This chapter provides the obtained data for the base images and workloads described in Section 4.3. We will give an overview of the normality tests for all workloads. Furthermore, we will present violin plots for each workload for a visual comparison, accompanied by the median, mean, and standard deviation. For the actual results and analysis, we refer to Chapter 6.

5.1 Normality tests

The results of the Shapiro-Wilk tests can be found in Table 5.1. These results show that not every workload and base image has normally distributed data according to the normality tests. In fact, only for the time data of the *Redis server* workload, all base images have normally distributed data. Since normality tests might be sensitive to outliers, we also provide the distribution plots to assess whether the data is approximately normal. However, these normality tests are used to decide which statistical significance test to use for the workload. Therefore, we used the Kruskal-Wallis and Dunn's tests for all data, except for the time data of *Redis server*, for which we used the one-way ANOVA and Tukey's test.

While we tried to mitigate outliers by randomizing the order of runs, isolating the containerized workload on a single core, and locking the CPU frequency, they are still present in the data. These outliers might be caused by external factors (e.g. change in room temperature over time), or internal factors (e.g. inconsistent fan speed) that affect the temperature of the machine, and subsequently the energy consumption. However, if the distribution of the data is approximately normal, we considered the experiment data valid and decided to not re-run the workload.

Table 5.1: Overview of the normality tests (Shapiro-Wilk)

	Energy (J)				*Time (s)*			
	debian	ubuntu	centos	alpine	debian	ubuntu	centos	alpine
Docker baseline								
Bloated Docker baseline								
llama.cpp (without model)	✓	✓						
llama.cpp (with model)	✓		✓					
Mattermost	✓			✓				
Cypress Real World App		✓		✓				
Minecraft server		✓	✓	✓		✓		
Minecraft server (no world)	✓		✓	✓			✓	
Redis server		✓	✓		✓	✓	✓	✓
Postgres server	✓			✓				✓
FFmpeg			✓	✓			✓	✓

5.2 Data distributions

This section provides an overview of the data collected from the 30 runs for each base image in all the workloads. This includes violin plots and median, mean, and standard deviation values of the energy consumption and elapsed time data. The energy consumption and time values of each run are derived from the difference between the values of the first and last measurement samples. The raw data on which these plots and values are based can be found on Zenodo[1]. Plots of the power over time and the CPU usage over time for the median runs (i.e. runs that correspond to the median value in energy consumption) can be found in Appendix A and Appendix B, respectively. Appendix B also includes the Pearson correlation between power and CPU usage.

5.2.1 Baseline

Figure 5.1 shows the violin plots of the energy and time data for the 'empty' Docker container in an idle state. The corresponding median, mean, and standard deviation values can be found in Table 5.2. The tails of the plots show that there are some outliers in the data, which affect the result of the normality test. Other than those outliers, the data is fairly normal and the differences between the base images are minimal.

[1] `https://zenodo.org/records/10220979`

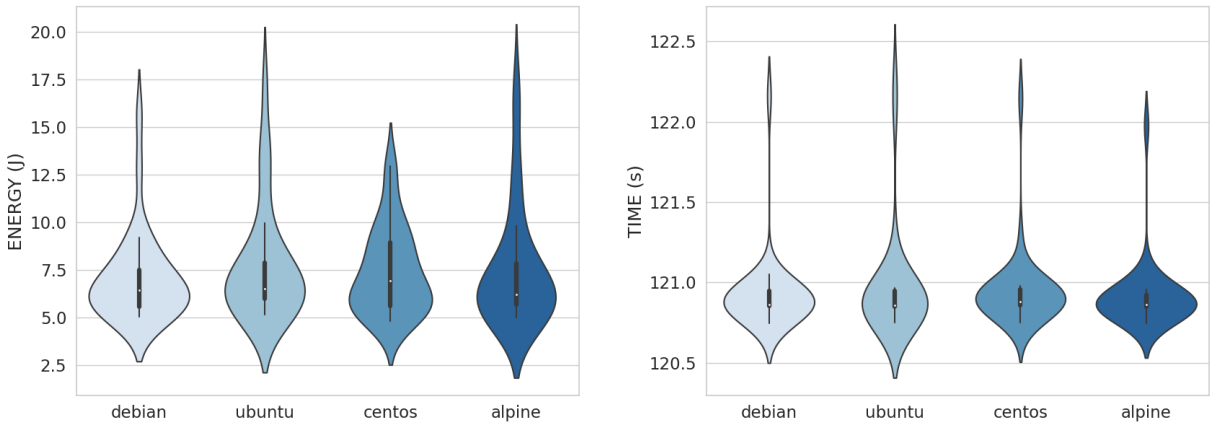

Figure 5.1: Docker baseline. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.2: Docker baseline. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	6.43	**7.07**	2.29	120.86	120.92	0.24
ubuntu	6.50	7.72	2.93	120.86	120.97	0.33
centos	6.94	7.41	2.21	120.86	120.95	0.24
alpine	**6.21**	7.51	3.06	120.86	**120.91**	0.21

A similar observation of the outliers in the data and minimal differences between base images can be found in the plots of the 'bloated' Docker baseline (see Figure 5.2). The values in Table 5.3 show that there is no significant difference between an empty and a bloated Docker container image.

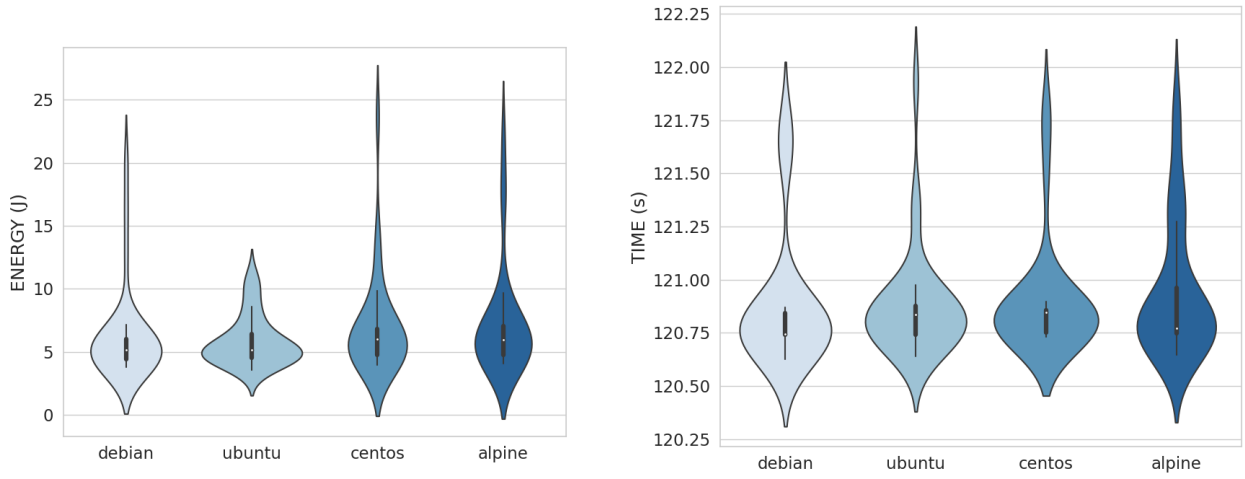

Figure 5.2: Bloated Docker baseline. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.3: Bloated Docker baseline. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	5.17	6.24	3.59	**120.75**	120.88	0.31
ubuntu	**5.14**	**5.87**	1.94	120.84	120.88	0.25
centos	6.01	6.95	3.92	120.85	120.90	0.27
alpine	5.99	7.21	4.24	120.77	120.94	0.31

5.2.2 llama.cpp

Figure 5.3 shows that all four base images perform similarly in both energy consumption and time for the *llama.cpp* workload with a volume. For the CentOS data, it is clear that there are a few outliers. These outliers also increase the standard deviation, as shown in Table 5.4. However, the mean and median are still relatively close to each other, which indicates that the distribution is fairly symmetrical.

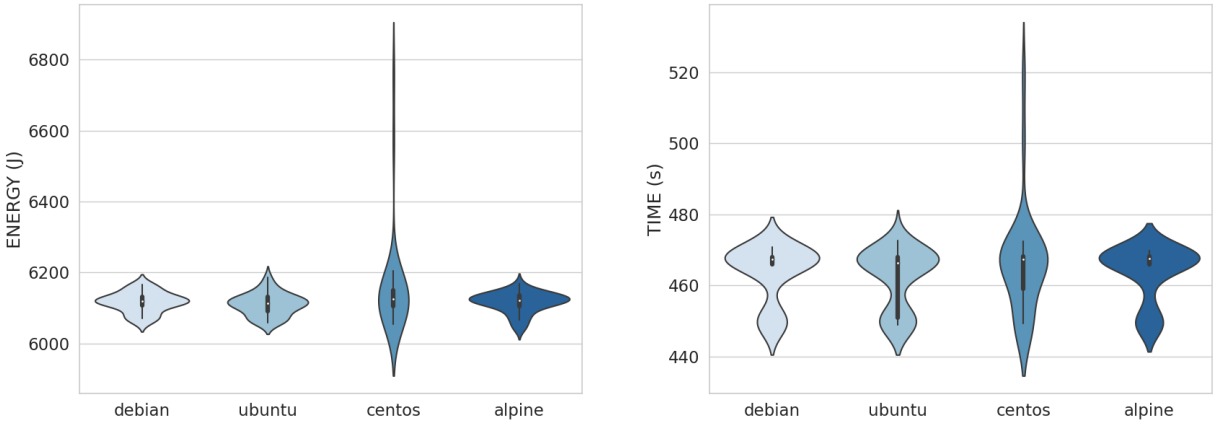

Figure 5.3: llama.cpp with volume. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.4: llama.cpp with volume. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	6119.72	6117.05	27.91	467.41	463.59	7.74
ubuntu	**6112.86**	**6110.58**	30.90	**466.40**	**462.24**	8.18
centos	6125.96	6157.81	141.48	467.41	466.04	14.29
alpine	6121.95	6115.86	28.57	467.56	463.71	7.46

As shown in Figure 5.4 there are also no differences for the *llama.cpp* workload with the model included. Here, the data of Ubuntu and Alpine contain some outliers. If we compare the values of this workload (see

Table 5.5) with the values of the workload with a volume, we see that using a volume for the model only affects the container image size, and does not affect the energy consumption or execution time.

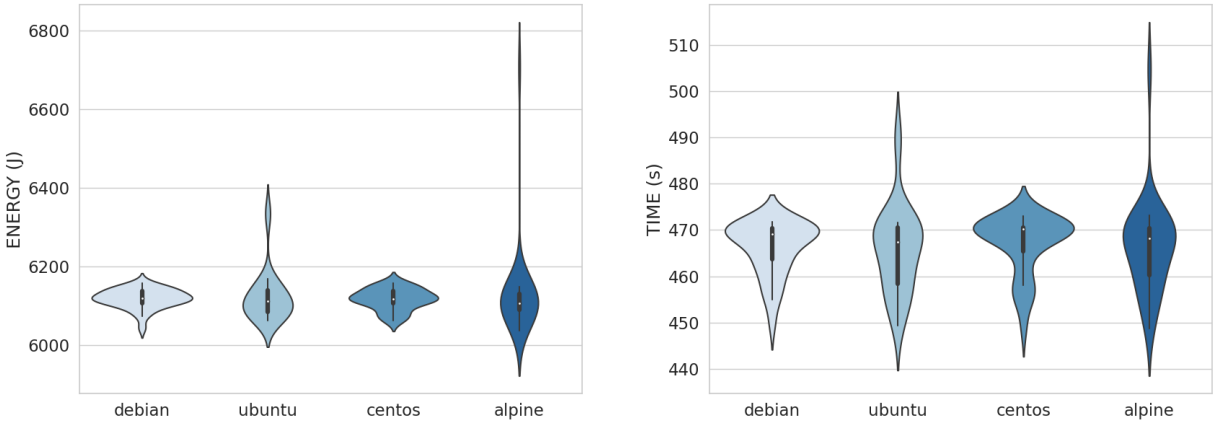

Figure 5.4: llama.cpp with model. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.5: llama.cpp with model. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	6120.19	6119.11	24.19	469.20	466.19	5.66
ubuntu	6111.50	6123.30	64.05	**467.45**	**465.82**	9.30
centos	6118.35	**6117.37**	26.35	470.24	467.08	6.20
alpine	**6107.40**	6126.50	111.27	468.13	466.16	9.89

5.2.3 Mattermost

The differences between the *Mattermost* images are visualized in the violin plots in Figure 5.5. These plots show that both Debian and Ubuntu take slightly longer to perform the *Mattermost* workload, compared to CentOS and Alpine. Similarly, Debian and Ubuntu also consume more energy during this workload. The median values in Table 5.6 show that Alpine executes this workload the fastest, while also having the lowest energy consumption.

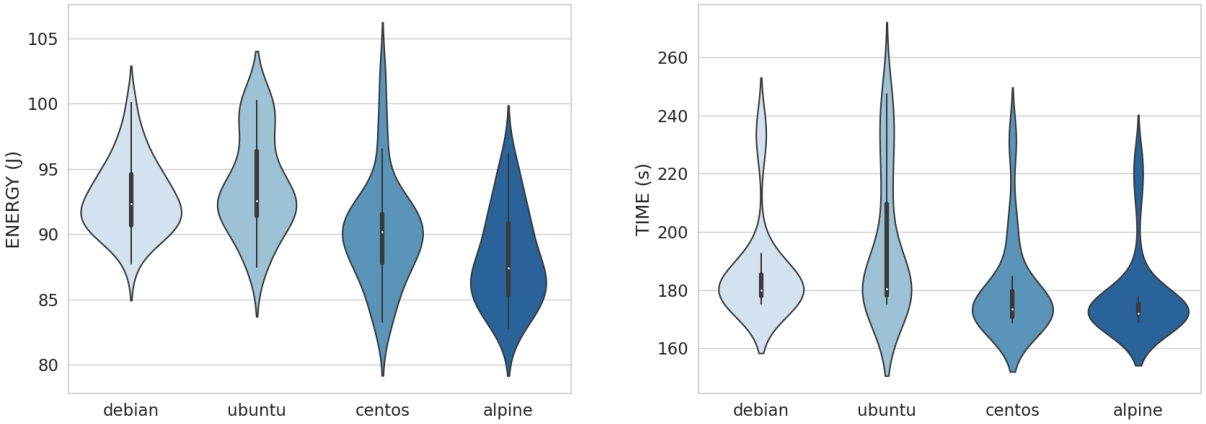

Figure 5.5: Mattermost. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.6: Mattermost. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	92.35	92.82	2.75	180.01	186.20	16.34
ubuntu	92.53	93.78	3.67	180.41	195.02	23.91
centos	90.17	90.10	3.97	173.45	180.19	16.36
alpine	**87.39**	**88.13**	3.53	**172.09**	**177.88**	14.38

5.2.4 Cypress Real World App

The violin plots of the energy consumption in Figure 5.6 and the median values (see Table 5.7) clearly show that Alpine consumes more energy than the other base images for the *Cypress Real World App* workload. This is in contrast with the other web application, *Mattermost*, where Alpine was the most energy efficient. In terms of time needed to finish the workload, all base images perform similarly. For all base images, there seem to be some outliers, especially for the elapsed time, affecting the normality tests.

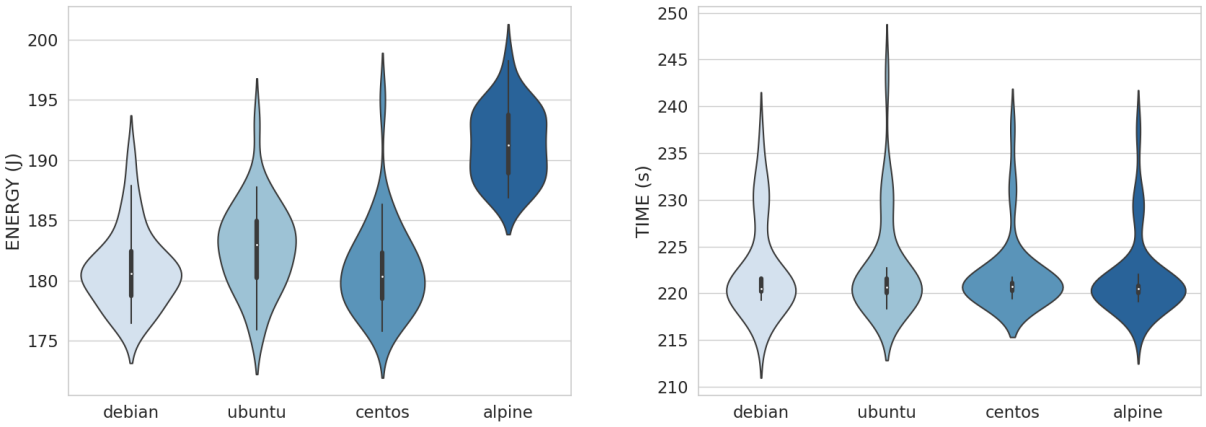

Figure 5.6: Cypress Real World App. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.7: Cypress Real World App. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	180.57	181.02	3.22	220.52	222.51	4.90
ubuntu	183.01	182.82	3.59	220.63	222.75	5.32
centos	**180.38**	**180.95**	3.75	220.73	222.04	4.00
alpine	191.26	191.34	2.95	**220.48**	**221.66**	4.13

5.2.5 Minecraft server

Figure 5.7 shows that for the *Minecraft server* workload with world creation, there is a difference in energy consumption to some extent between each base image. It is clear that Debian has the lowest energy consumption, and Alpine the highest. This is confirmed by the values in Table 5.8. An interesting observation is that both Debian and Ubuntu seem to take longer than CentOS and Alpine while having a lower energy consumption.

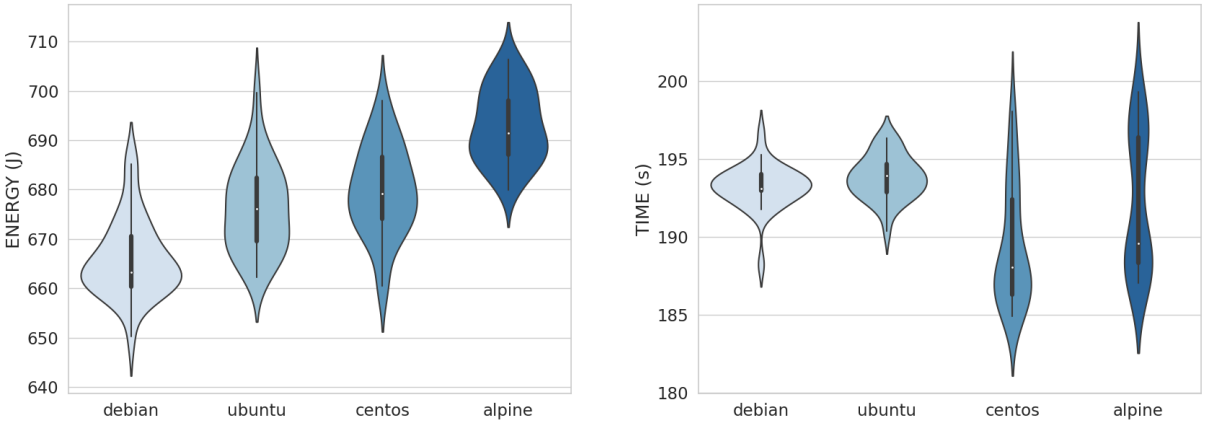

Figure 5.7: Minecraft server with world. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.8: Minecraft server with world. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	**663.26**	**666.01**	7.68	193.08	193.22	1.39
ubuntu	676.05	676.20	8.76	193.90	193.80	1.39
centos	679.25	679.82	8.95	**188.04**	**189.30**	3.72
alpine	691.47	692.87	7.26	189.59	192.11	4.35

The violin plots in Figure 5.8 illustrate the differences between the base images for the *Minecraft server* workload without world creation during run time, which reduces the execution time by approximately 40 seconds. Similar to the workload with the world creation, both Debian and Ubuntu take longer to finish the workload than CentOS. However, the differences in energy consumption seem to be relatively smaller if the world creation is not included. This means that most of the differences in energy consumption arise during the world creation. Ultimately, Table 5.9 shows that Alpine still has the highest energy consumption.

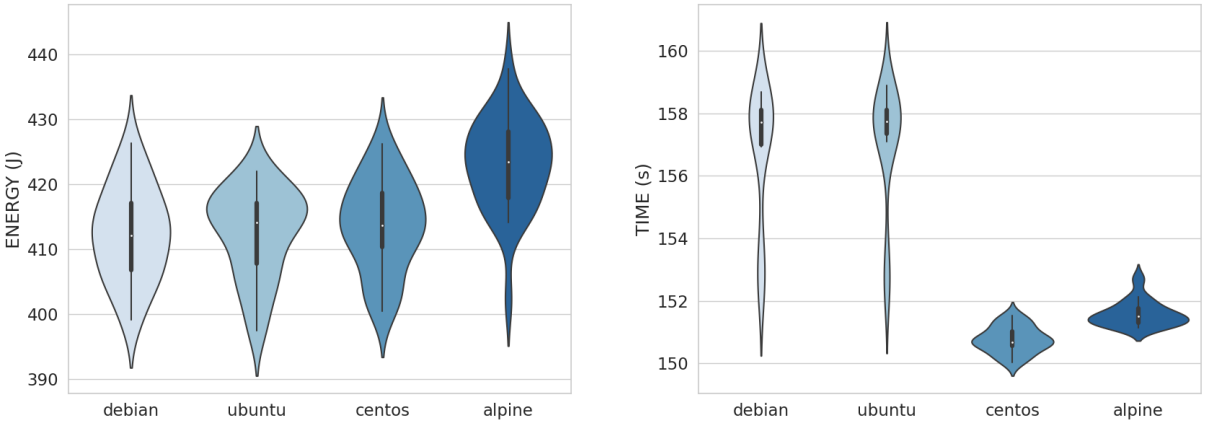

Figure 5.8: Minecraft server without world. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.9: Minecraft server without world. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	**412.14**	**412.35**	7.12	157.71	156.74	2.13
ubuntu	414.13	412.48	6.70	157.73	157.03	1.95
centos	413.66	413.63	6.88	**150.68**	**150.76**	0.41
alpine	423.43	423.07	6.97	151.50	151.59	0.40

5.2.6 Redis server

The plots of the *Redis server* workload in Figure 5.9 show that there are no big outliers in the data. Furthermore, the energy plot shows a clear difference in terms of energy consumption between Alpine and the other base images. Moreover, Debian and Ubuntu seem to consume a similar amount of energy. Alpine also needs significantly more time to finish the workload, compared to the other images, which are very close to each other. The high energy consumption and long execution time of Alpine are confirmed by the median and mean values in Table 5.10.

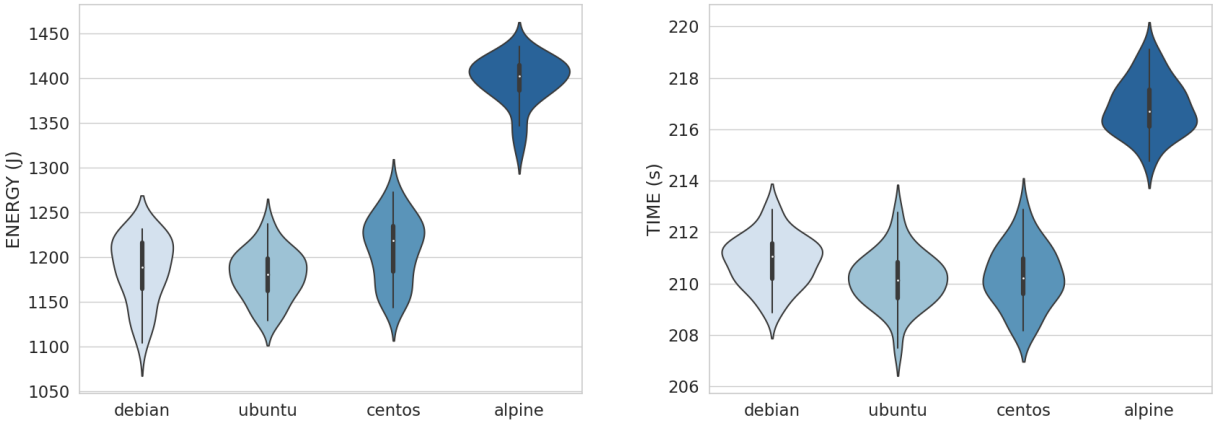

Figure 5.9: Redis server. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.10: Redis server. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	1189.28	1186.16	36.00	211.07	210.88	0.97
ubuntu	**1180.97**	**1180.25**	27.27	**210.12**	**210.17**	1.04
centos	1218.82	1209.31	35.77	210.22	210.26	1.17
alpine	1402.44	1398.03	26.26	216.69	216.86	1.03

5.2.7 Postgres server

Figure 5.10 shows that there are a few outliers in the Ubuntu data for the *Postgres server* workload. In terms of energy consumption, there are differences between all base images. Furthermore, the plots and the values in Table 5.11 clearly show that Alpine has the highest energy consumption, and CentOS the lowest.

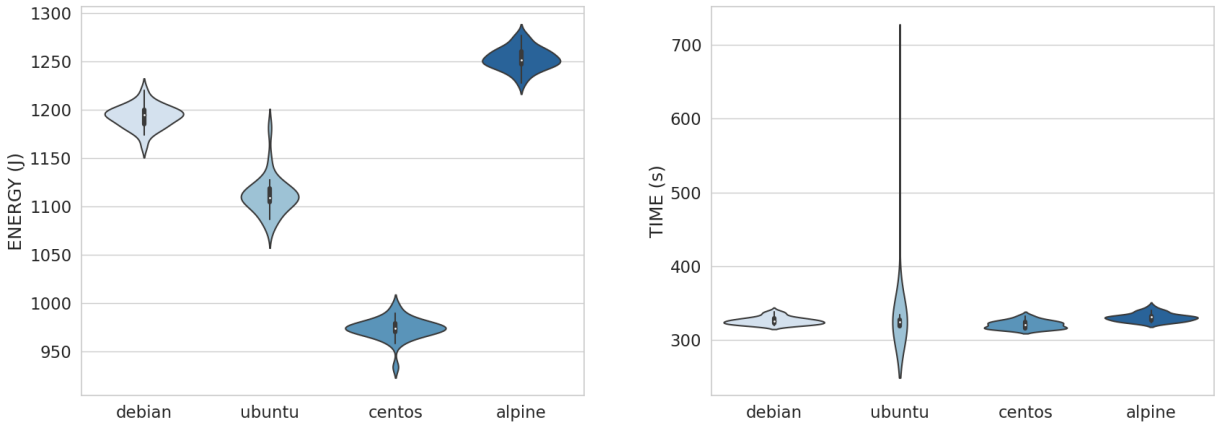

Figure 5.10: Postgres server. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.11: Postgres server. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	1194.65	1193.09	11.61	325.90	326.36	5.08
ubuntu	1109.08	1110.95	19.00	325.11	340.82	66.10
centos	**973.70**	**973.50**	10.86	**320.53**	**321.20**	5.35
alpine	1251.34	1252.85	11.07	331.07	331.10	5.47

5.2.8 FFmpeg

The plots of the data distribution of the *FFmpeg* workload and the corresponding values can be found in Figure 5.11 and Table 5.12, respectively. The energy plot clearly shows that Debian has the lowest energy consumption, and CentOS the highest. Ubuntu and Alpine consume a similar amount of energy. In terms of time, Alpine seems to perform the best, while CentOS takes the longest to finish the workload. The execution times of Debian and Ubuntu are very similar.

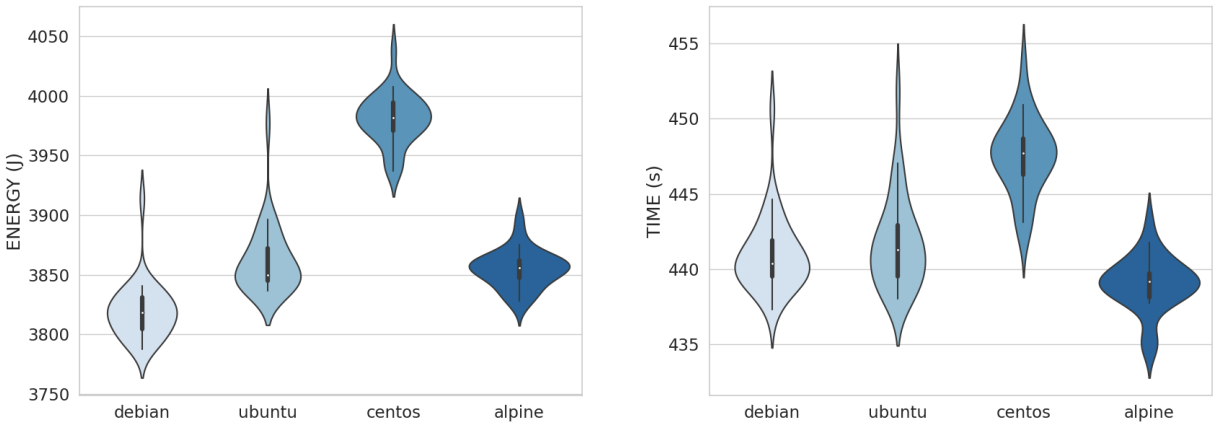

Figure 5.11: FFmpeg. Violin plots of the energy consumption (left) and elapsed time (right)

Table 5.12: FFmpeg. Overview of the median, the mean (μ), and standard deviation (σ) for energy and time.

	Energy (J)			*Time (s)*		
	median	μ	σ	median	μ	σ
debian	**3818.50**	**3819.13**	23.35	440.37	440.96	2.48
ubuntu	3849.87	3861.99	28.10	441.28	441.72	3.04
centos	3981.80	3980.33	20.94	447.71	447.51	2.48
alpine	3856.06	3854.92	15.72	**439.18**	**439.03**	1.78

6 Results

This chapter will discuss the results obtained from the collected data. Table 6.1 shows an overview of the results obtained from the energy measurements. The left side of the table shows the statistical significance of the energy and time data. The right side of the table provides the Pearson correlation between the energy and time data of the runs. Using these results we will answer the research questions of this thesis.

Table 6.1: Overview of the statistical analysis tests (One-way ANOVA or Kruskall-Wallis) and correlation between energy and time (Pearson). Values in **bold** indicate there is a correlation (moderate to very strong)

	Significant		*Correlation (Time / Energy)*			
	ENERGY (J)	TIME (s)	debian	ubuntu	centos	alpine
Docker baseline			−0.11	0.20	0.13	−0.15
Bloated Docker baseline			−0.04	0.18	0.01	0.11
llama.cpp (without model)			**0.84**	**0.83**	**0.91**	**0.76**
llama.cpp (with model)			**0.69**	**0.87**	**0.78**	**0.82**
Mattermost	✓	✓	**0.50**	**0.70**	**0.66**	**0.66**
Cypress Real World App	✓		**0.46**	−0.11	**0.43**	**0.52**
Minecraft server	✓	✓	**0.46**	**0.47**	0.22	0.25
Minecraft server (no world)	✓	✓	−0.01	0.02	**0.44**	**0.61**
Redis server	✓	✓	0.12	−0.03	0.34	0.08
Postgres server	✓	✓	0.25	**0.62**	0.12	0.07
FFmpeg	✓	✓	**0.89**	**0.92**	**0.96**	**0.87**

6.1 RQ1: Does the base image selection have an impact on the energy efficiency of the container?

The statistical analysis tests of the results (see Table 6.1) show that for the Docker baselines, where the containers are idle, there are no significant differences between the base images. Furthermore, the results of the two Docker baselines indicate that in an idle state having a bloated image does not affect the energy efficiency. The lack of significance in an idle state indicates that possible differences in energy consumption are due to the execution of the workload. Subsequently, this would mean that some images can perform a workload more efficiently than others.

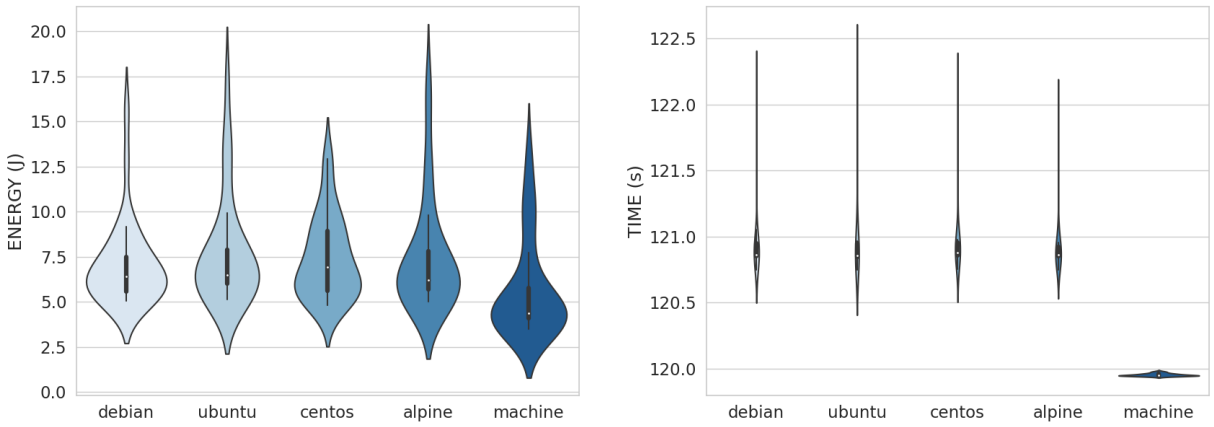

Figure 6.1: Machine baseline. Violin plots of the energy consumption (left) and elapsed time (right)

Compared to the machine in an idle state, the Docker container has a very low overhead, resulting in slightly higher energy usage, as shown in Figure 6.1. This is in line with previous studies on the performance of Docker [12, 10, 14]. Furthermore, containerized workloads take slightly longer, as the containers have a start-up time. This start-up phase of containers is a significant part of the energy consumption difference with the machine baseline (see Figure 6.2 and Figure 6.3).

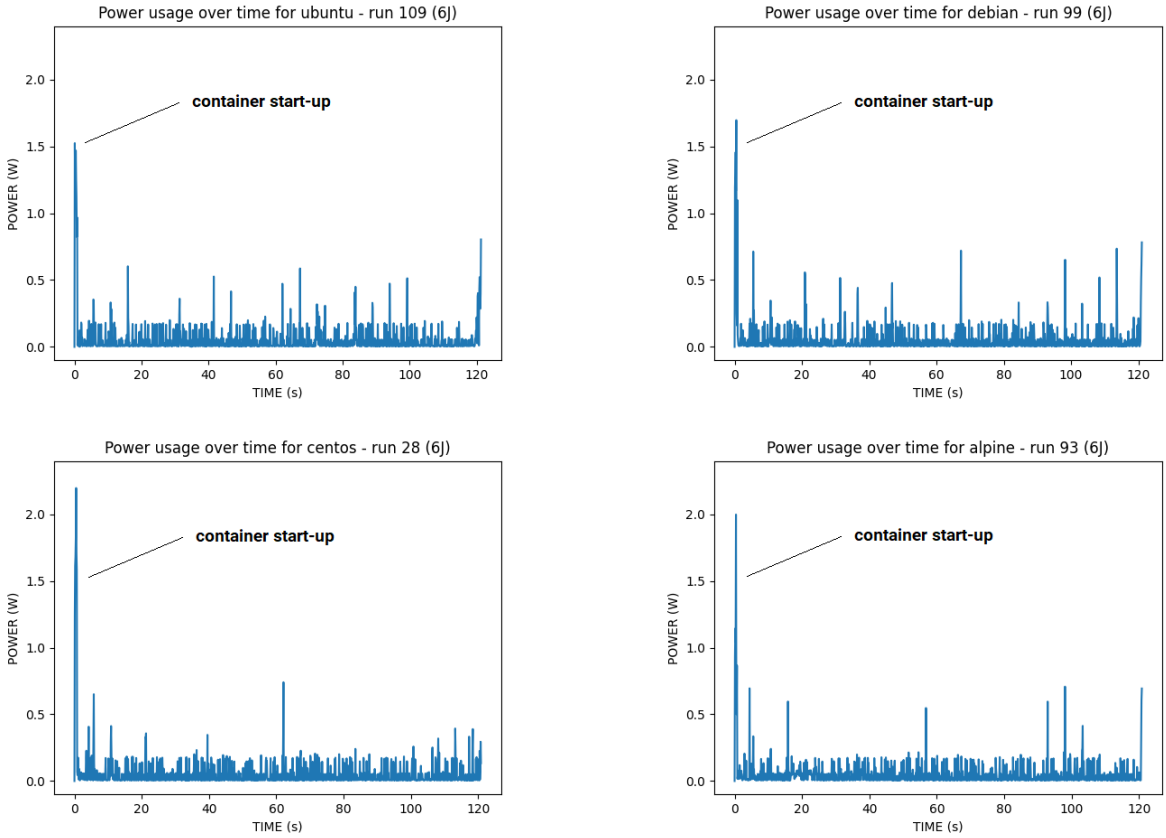

Figure 6.2: Power usage over time for the Docker baseline.

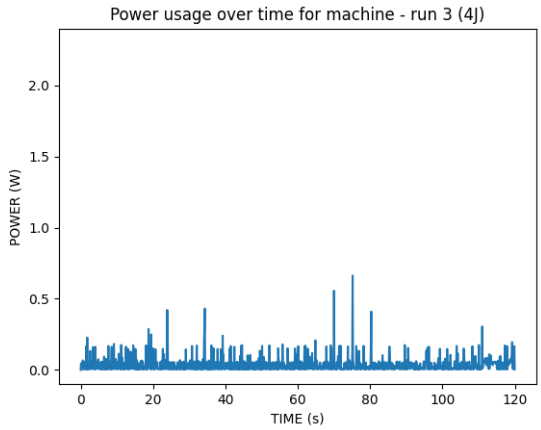

Figure 6.3: Power usage over time for the machine baseline.

While there are no significant differences in energy consumption between base images in an idle state, the main purpose of the experiments was to evaluate the energy efficiency of containers under workload. The results show that there is a significant difference in energy consumption between at least two base images for all workloads, except for the two *llama.cpp* workloads. For the *llama.cpp* workloads we used the same binary for each image, which is likely the main reason that the different distributions do not affect the workload performance of the container. In combination with the consistently high CPU usage (see Figure B.3), which is strongly correlated with the energy consumption (see Table B.3), the minimal difference in total energy consumption is not surprising.

As shown in Table 6.2, the Cliff's Delta is always large in the workloads where the difference is significant ($|\delta| > 0.43$ [82]). This large effect size means that one image is performing better than the other images in almost all runs. Although this effect size increases the certainty that there is an actual difference between the two images, it does not provide direct information on how large the difference in energy consumption is.

Table 6.2: Overview of the effect sizes (Cliff's delta) of the energy consumption differences. Values in **bold** indicate significance from the post hoc test. A negative value indicates the first image is more efficient.

Image pair		Mattermost	Cypress RWA	Minecraft	Minecraft (no world)	Redis	Postgres	FFmpeg
debian	ubuntu	−0.13	−0.33	**−0.63**	−0.05	0.16	**0.99**	**−0.90**
debian	centos	**0.50**	0.04	**−0.74**	−0.10	−0.36	**1.00**	**−1.00**
debian	alpine	**0.68**	**−0.96**	**−0.98**	**−0.72**	**−1.00**	**−1.00**	**−0.86**
ubuntu	centos	**0.57**	0.34	−0.24	−0.07	**−0.45**	**1.00**	**−0.97**
ubuntu	alpine	**0.73**	**−0.94**	**−0.85**	**−0.74**	**−1.00**	**−1.00**	0.05
centos	alpine	0.30	**−0.94**	**−0.72**	**−0.67**	**−1.00**	**−1.00**	**1.00**

Before conducting the experiments, it was expected that Ubuntu and Debian would be consuming a similar amount of energy, and that there would be a significant difference between CentOS and Alpine. In most cases with significant differences in the workload, there is indeed a significant difference between CentOS and Alpine. However, between Ubuntu and Debian there is a difference in energy efficiency in some cases. Overall, the base image selection can have a significant impact on the energy efficiency of the

corresponding container. However, the magnitude of the difference in energy consumption between base images depends on the workload. Therefore, whether the improvement in energy efficiency is actually worth switching base images for, might depend on the workload.

Summary RQ1

The base image selection can have an impact on the energy efficiency of the container under workload. Furthermore, the effect size is always large if there is a significant difference in energy consumption between base images.

6.2 RQ2: Does the impact of base image selection vary for different types of workloads?

As mentioned before, Table 6.1 shows that except for the baselines and *llama.cpp* workloads, the results of the remaining workloads are statistically significant. In order to assess how these workloads affect the impact of the base image selection, we compare the best and worst-performing base images in terms of energy efficiency. Table 6.3 provides the absolute and the percentage difference of the median between these two images. These differences do not always occur between the same pair of base images, which means that there is not a single most energy-efficient base image for all cases. Furthermore, the relative differences in energy consumption between base images are not consistent among all workloads.

Table 6.3: Overview of the best and worst base image in terms of energy efficiency, based on the statistical analysis. The last two pairs of columns show the absolute and the percentage difference, both for the median value of energy and time, respectively. A negative value for the time difference indicates the less energy-efficient image is actually faster.

			Energy		Time	
	Best	Worst	ΔJ	%	Δs	%
Mattermost	Alpine	Ubuntu	$5,14$	$5,88$	$8,32$	$4,83$
Cypress Real World App	CentOS	Alpine	$10,88$	$6,03$	$-0,25$	$-0,11$
Minecraft server	Debian	Alpine	$28,21$	$4,25$	$-3,49$	$-1,81$
Minecraft server (no world)	Debian	Alpine	$11,29$	$2,74$	$-6,21$	$-3,94$
Redis server	Ubuntu	Alpine	$221,47$	$18,75$	$6,57$	$3,13$
Postgres server	CentOS	Alpine	$277,64$	$28,51$	$10,54$	$3,29$
FFmpeg	Debian	CentOS	$163,30$	$4,28$	$7,34$	$1,67$

Although the percentage difference in the two web application workloads is similar (i.e. 5.88% and 6.03%), in the *Mattermost* workload Alpine is the most energy-efficient, whereas in the *Cypress Real World App*, it is the least energy-efficient. The difference between the workloads is that *Mattermost* is a single executable binary and *Cypress Real World App* is a *Node.js* application. While the 6% difference in energy efficiency seems minimal, it can be noticeable in practice. For example, when running the containerized web application for a longer period of time, or executing containerized tests in the CI/CD pipeline frequently, this difference in energy efficiency will definitely add up in the total energy consumption. Moreover, it is usual to run multiple containers instead of a single one in practice.

Since the *Cypress Real World App* is a *Node.js* application, we also tested how the OS base images performed against the official *Node* images. Both the *Node* image based on Debian and the *Node* Alpine image perform similarly to the Alpine base image. This means that the elapsed time is the same as the other base images, while the energy consumption is higher than Debian, Ubuntu and CentOS. The fact that both *Node* images consume a similar amount of energy is unexpected, since there is a significant

difference between the Debian and Alpine OS base image. The difference between the Debian base image and the *Node* image based on Debian is possibly caused by the version of the distribution. Due to compatibility reasons, we used *Node 16*, which uses Debian 10 (i.e. *buster*). In contrast, the Debian base image we used is Debian 12 (i.e. *bookworm*). This means that the version of the OS base image also affects the energy efficiency of the container.

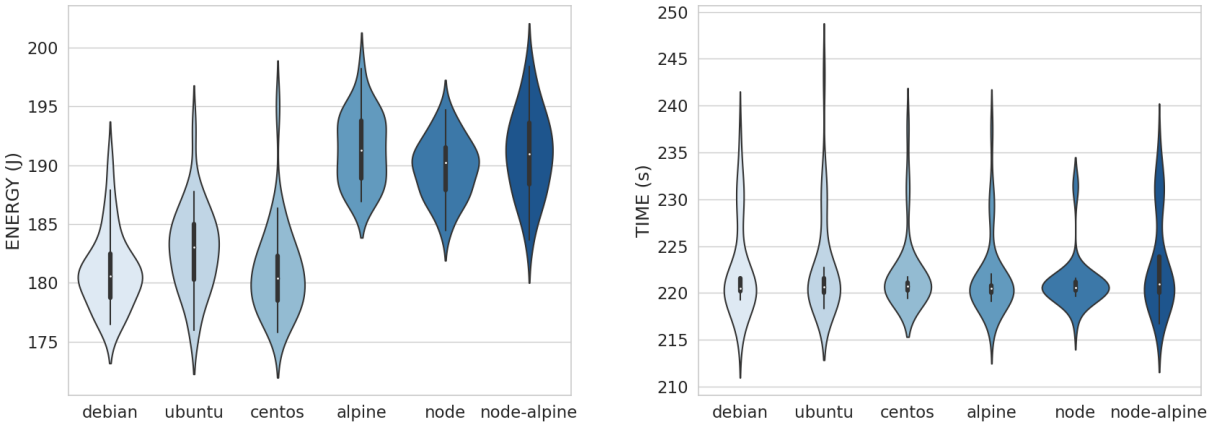

Figure 6.4: Cypress Real World App with official images. Violin plots of the energy consumption (left) and elapsed time (right)

The high energy consumption of Alpine is also prevalent in both *Minecraft server* workloads, for which the server is a Java application packaged in a JAR file. In both cases, Debian is the most energy-efficient. The difference in energy consumption is mainly caused by the start-up phases of the server. This is confirmed by the workload without the world creation, which shortens the start-up phase of the server. In this case, the effect size of the difference is still large (see Table 6.2). However, the magnitude of the difference in energy efficiency is less significant, as it decreases from $4,25\%$ to $2,74\%$. Therefore, the difference in energy efficiency might not be significant in practice. Whereas this *Minecraft server* only runs for approximately 2 minutes, real-world gaming servers usually run indefinitely, which means that the impact of the start-up phase will tend to be negligible.

We observe that the most significant differences in energy consumption appear in the database workloads, *Redis server* and *Postgres server*. In these cases, Alpine is clearly the least energy-efficient among the base images as well. In comparison with Alpine, there is not a practical difference between the other 3 base images while running *Redis server*, even though there is a relatively small statistical difference between Ubuntu and CentOS (see Table 6.2). This is not the case in the *Postgres server* workload. For this workload, there is a clear difference between all 4 base images. A major difference between *Redis* and *Postgres* is that all *Redis* data resides in RAM. Since the data is stored and retrieved from RAM, it can read and write much faster than relational databases (e.g. Postgres), which also rely on I/O performance. Interestingly enough Debian consumes significantly more energy than Ubuntu ($7,72\%$ difference) in the *Postgres server* workload. Furthermore, as shown in Table 6.3, CentOS is the most energy efficient in this case and consumes $28,51\%$ less energy than Alpine. This difference in energy consumption can be significant for applications with many database transactions. Furthermore, considering the relatively short workload time, choosing an energy-efficient base image in this case can be crucial in practice, where the database is run for a longer period of time. Ultimately, Alpine is the least optimal choice in terms of energy efficiency for containerized databases.

In contrast with the results of the other workloads, CentOS has the highest energy consumption while transcoding a video with *FFmpeg*. Similar to machine learning inference (see Figure B.3), video transcoding (see Figure B.12) is a CPU-intensive task. The CPU utilization for both these workloads is consistently near 100% for the entire duration of the workload. Since the CPU is undergoing near maximum stress for the whole workload and the power is strongly correlated with the CPU usage (see Table B.3 and

Table B.12), it makes sense that energy and time is strongly correlated for these workloads. However, while there were no significant differences in the *llama.cpp* workload, for *FFmpeg* there are differences in the energy efficiency. Although Ubuntu and Alpine consume a similar amount of energy and slightly more than Debian, they all have a significantly lower energy consumption than CentOS. This is mostly caused by the longer time CentOS needs to transcode the video, as there is a very strong correlation between time and energy in this case. As transcoding plays an important role in optimizing videos for streaming services [73], choosing a base image for a time-efficient container is crucial. The results show that selecting a slow image will have a significant impact on the total energy consumption in the long run.

Overall, there is no single best or worst base image in terms of energy consumption for all workloads. Consequently, different workloads might require different base images when considering energy efficiency. Furthermore, the magnitude of the differences in energy consumption is not consistent among all workloads. However, in most of the cases covered in this thesis, Alpine has higher energy consumption than the other 3 base images. Especially for the database workloads, Alpine is significantly less energy-efficient. Ultimately, while selecting the most energy-efficient base image depends on the workload, it is safe to say that Alpine is generally the least optimal option.

Summary RQ2

The type of workload affects the impact of base image selection on energy efficiency. There is no single best or worst base image across all workloads. Furthermore, for some workloads (i.e. databases) the impact of the selection is more significant in practice than for others (i.e. gaming servers).

6.3 RQ3: Is there a trade-off between current selection criteria and energy efficiency?

We defined execution time and image size as the current criteria for selecting base images. The focus of this thesis is the energy consumption during run time, and therefore we did not consider the relationship between build time and energy consumption. In order to decide if there is a trade-off between energy consumption and the execution time, we assessed the correlation between time and energy of runs, and whether the fastest image is also the most energy efficient. For the trade-off between image size and energy consumption, we assessed whether the smallest image, Alpine, is also the most energy-efficient image. The findings of the experiments show that an energy-efficient container image can come at the cost of a longer execution time or a larger image size.

6.3.1 RQ3.1: Is the execution time of the containerized workload correlated with the energy consumption?

The expectation that a longer execution time results in a higher energy consumption is not always true. This is indicated by the Pearson correlation between the total time and energy consumption of the runs (see Table 6.1). However, it is clear that for *llama.cpp* and *FFmpeg* there is a high correlation between the elapsed time and energy consumption of the runs. These workloads are CPU-intensive, where the CPU utilization stays near 100% throughout the entire workload (see Figure 6.5). Consequently, as CPU usage is strongly correlated with energy consumption, naturally the elapsed time is the main factor of the energy efficiency in these cases.

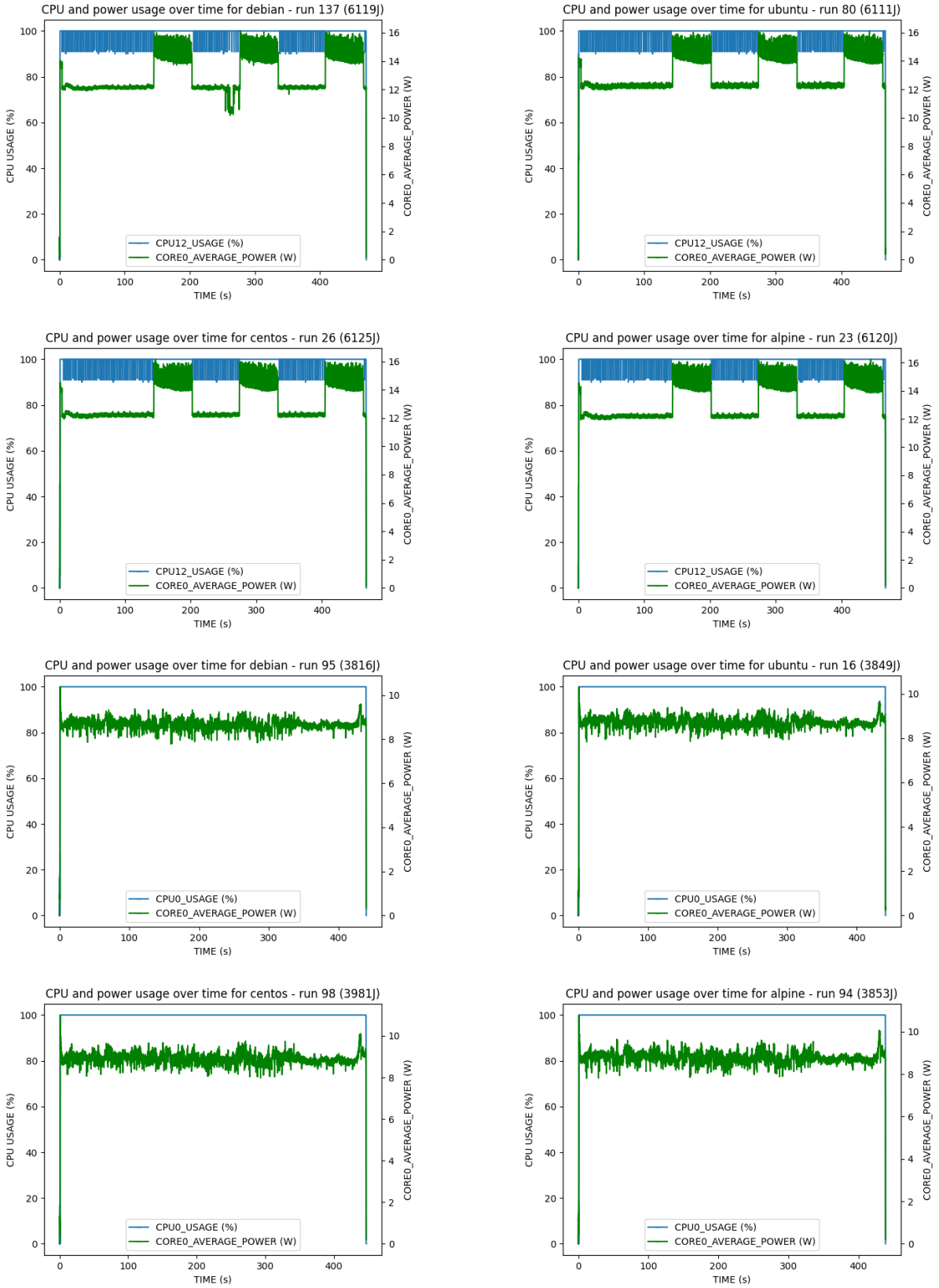

Figure 6.5: CPU and power usage over time of the median runs, for *llama.cpp* (top 4) and *FFmpeg* (bottom 4).

For the *llama.cpp* workload we also used the CUDA image, which uses the GPU for inference. The energy consumption values for the CUDA image are based on the energy consumption of the GPU and 1 core, as this core is still used by the container. Figure 6.6 shows that using a GPU with the *CUDA* image for this workload results in a shorter run time and a lower total energy consumption. However, the average Watt during this period is much higher when the GPU is involved, compared to inference with the CPU (see Figure A.5). In practice, it makes more sense to use the GPU for inference, as it is not only significantly faster but also has a lower total energy consumption.

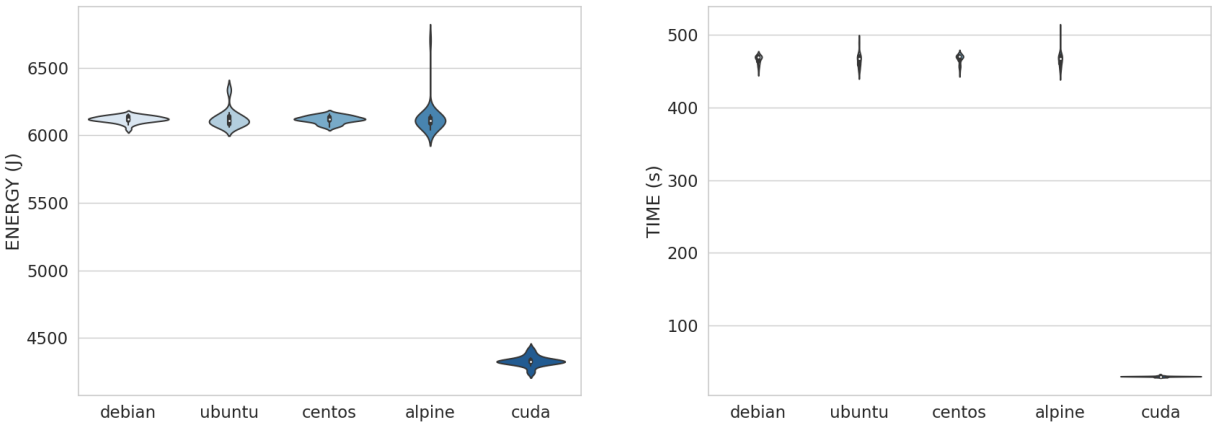

Figure 6.6: llama.cpp with GPU. Violin plots of the energy consumption (left) and elapsed time (right)

Figure 6.7 gives an overview of the kernel density estimate (KDE) plots for time and energy, for the workloads with statistically significant results. The plot for *FFmpeg* clearly shows the high correlation between execution time and energy consumption. However, while CentOS is overall the slowest base image and also the one with the highest energy consumption, the fastest base image, which is Alpine, is not the most energy-efficient in this case.

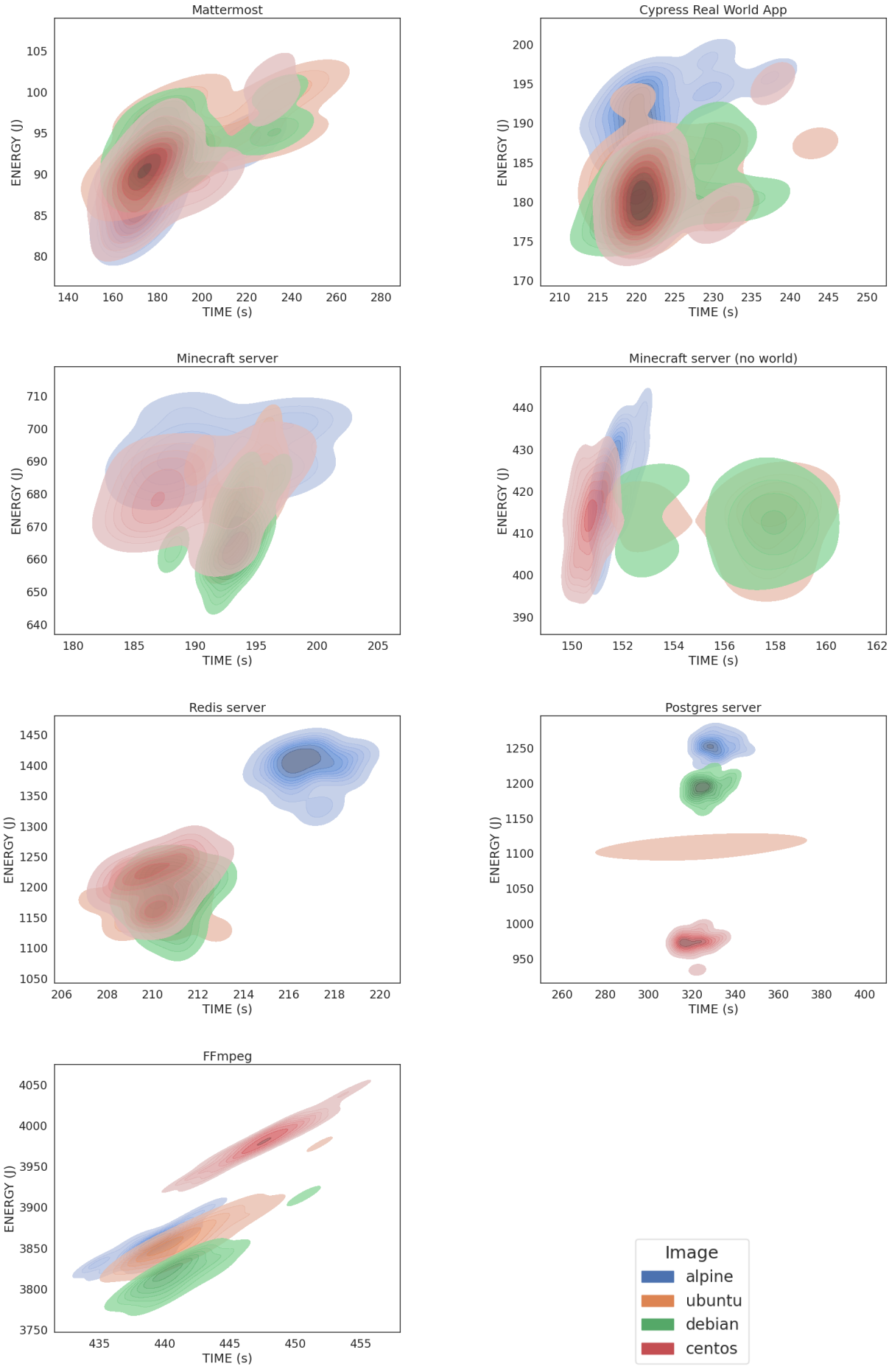

Figure 6.7: Bivariate KDE plots for time and energy.

The *Mattermost* workload shows a moderate to strong correlation between time and energy consumption as well. Furthermore, the differences in time and the differences in energy consumption are proportional to each other, which means that the slower images (i.e. Debian and Ubuntu) also consume more energy. Additionally, the effect size for the difference in elapsed time is almost equal to the effect size of the difference in energy consumption.

For the remaining workloads, we cannot say that there is a relationship between elapsed time and energy consumption, as Figure 6.7 shows no correlation. As identified before, a trend in these workloads is that the Alpine image results in the least energy-efficient container. Nonetheless, only for the database workloads (i.e. *Redis server* and *Postgres server*) is this higher energy consumption accompanied by a longer execution time, which is also clearly illustrated in Figure 6.7. For the other cases, the difference in time is not statistically significant. Table 6.3 shows that for the *Minecraft server without world* workload Alpine is even faster than Debian, which is the most efficient image in this case. Since the client is connected for the same period of time and the server start-up time is approximately the same for each base image, this longer execution time is likely caused towards the end of the workload. This is also confirmed by the power over time plots of Debian and Ubuntu in Figure 6.8, where there is a small period of time at the end of the workload with approximately 0W usage. This period of time is approximately 5 seconds, which is also the difference in total time between Alpine and the two Debian-based images. Therefore, it is certain that the difference in energy consumption is not a result of the longer execution time. This is also the case for *Cypress Real World App*, for which there is no difference in elapsed time between the base images.

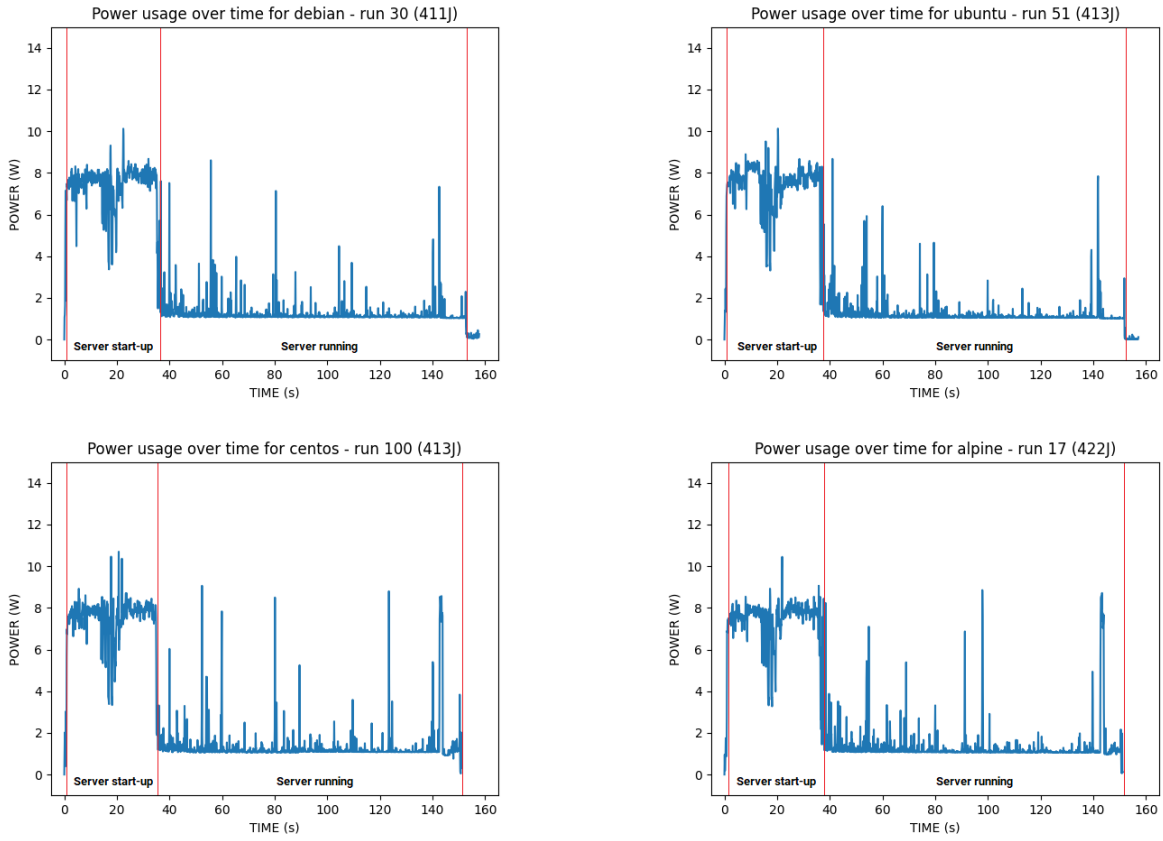

Figure 6.8: Power usage over time for the *Minecraft server without world* workload.

Ultimately, energy consumption is correlated to execution time for high CPU-intensive workloads. For the CPU-intensive *FFmpeg* workload the most energy-consuming container also has the longest elapsed time. This is also the case for both of the database workloads. However, for the other workloads, the least energy-efficient container is not the slowest container. Consequently, we cannot say that the lifespan of

the container is directly correlated to its energy consumption, as it depends on the workload. Therefore, when selecting a base image, it is beneficial to not only consider the time efficiency, but also to take the energy efficiency of the corresponding container into account.

6.3.2 RQ3.2: Does a small image also result in low energy consumption of the container?

To observe if an unnecessary large container image affects energy consumption, we monitored 'empty' containers, as well as 'bloated' containers, in an idle state. Figure 6.9 show that having a bloated container does not have a practical impact on the energy efficiency of the container. Similarly, we monitored the *llama.cpp* workload with containers that use a volume for the model, as well as containers that copied the model, resulting in a larger image. This larger image does not lead to higher energy consumption, as shown in Figure 6.10.

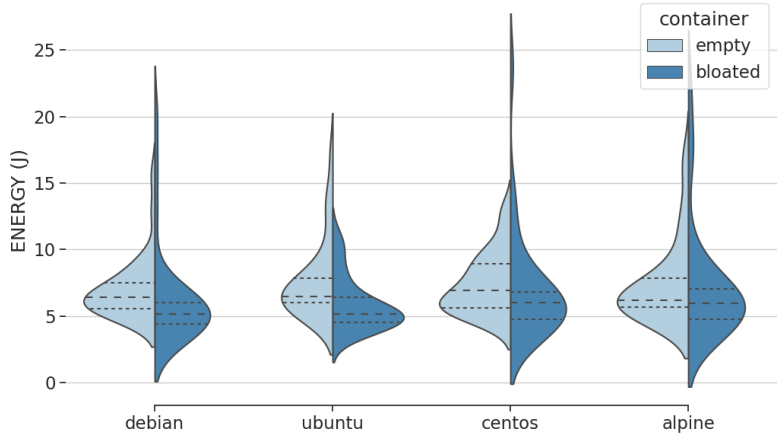

Figure 6.9: Docker baseline. Violin plots of the energy consumption for 'empty' containers (left) and 'bloated' containers (right)

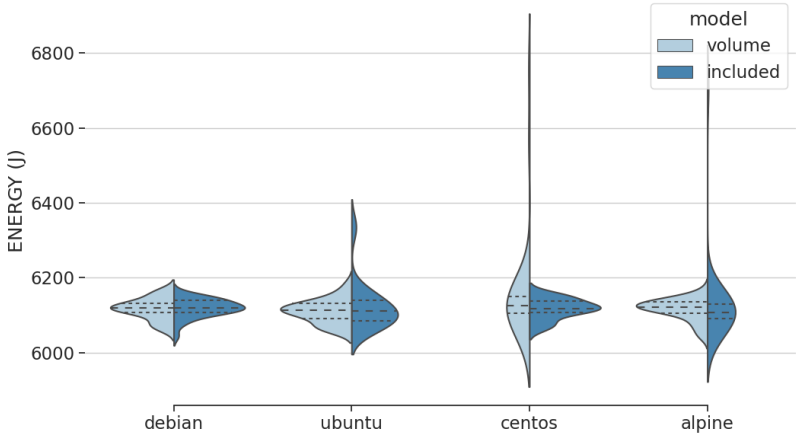

Figure 6.10: llama.cpp. Violin plots of the energy consumption for containers using a volume for the model (left) and containers with the model included (right)

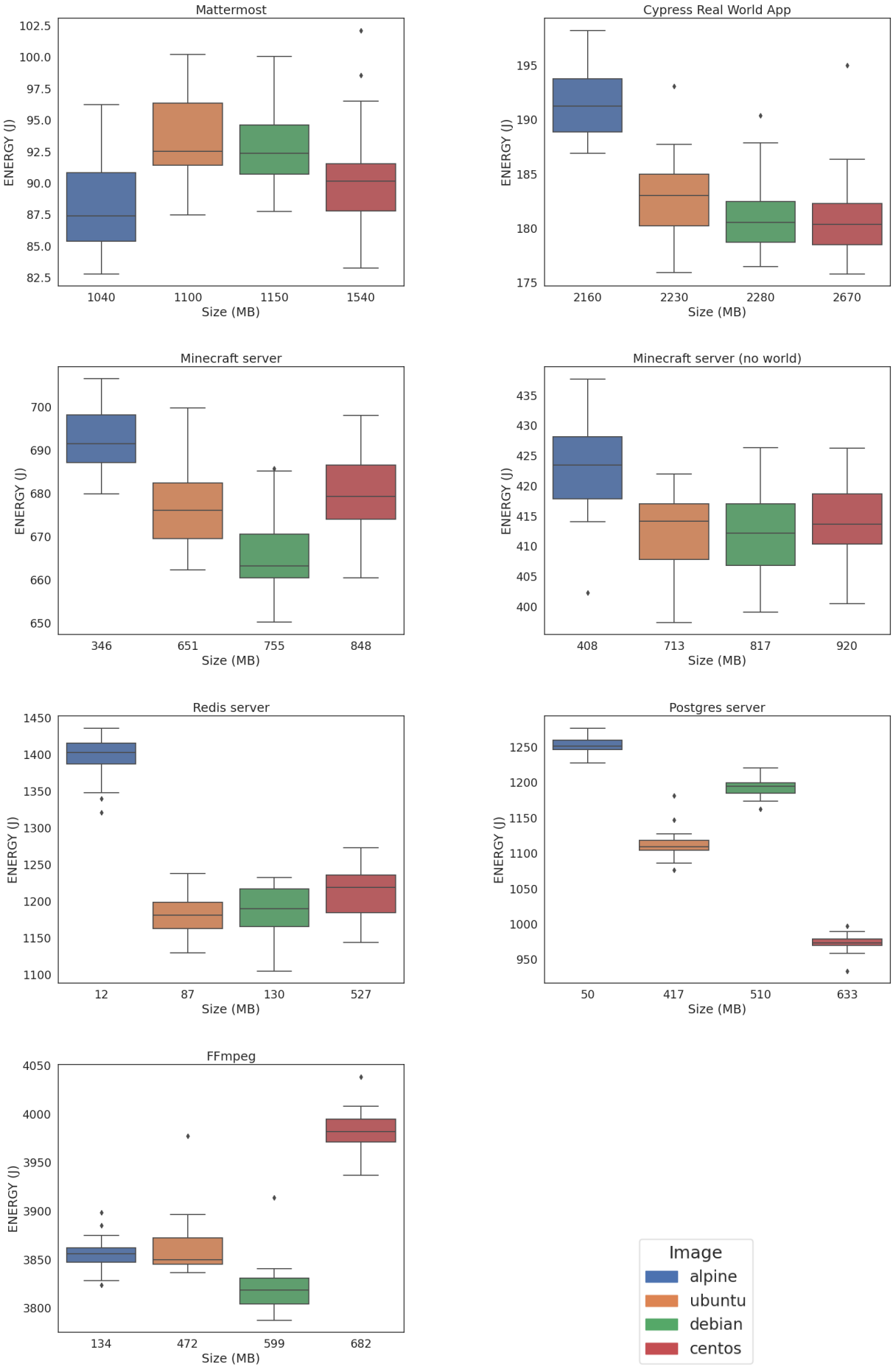

Figure 6.11: Box plots of the energy consumption data, ordered by image size.

The observation in Table 6.3 that Alpine is the least energy-efficient base image in most workloads, is illustrated in Figure 6.11, where the images are ordered by size. These plots also show that image size is not necessarily correlated to the energy consumption of the container. Moreover, the results of the *Bloated Docker baseline* and the *llama.cpp (with model)* workloads suggest that using a bloated image does not generate higher energy consumption. Since Alpine is a popular base image, it is important to notice that there is also a trade-off between small image size and low energy consumption.

Although the lightweight nature of Alpine can help with the quick build times of containers and better portability, it is possible that it is also the reason for its higher energy consumption. The small image size of Alpine is achieved by the fact that it is built around *musl libc* and *busybox* [83]. *musl* is a C standard library for Linux-based systems and is a lightweight alternative to *glibc*, which is used by Debian, Ubuntu, and CentOS. However, *musl* does not provide the same functionality as *glibc*. Workloads that use libraries that rely on C code will therefore be impacted by the differences between *musl* and *glibc* [84]. As seen before, for the workloads that use a precompiled binary (i.e. *llama.cpp* and *Mattermost)*, Alpine is not the least efficient base image. In these cases, *musl* does not impact the performance.

Summary RQ3

There can be a trade-off between current selection criteria and energy efficiency. This means that the most time-efficient base image is not always the most energy-efficient. Moreover, the smallest base image (i.e. Alpine) is often the least energy-efficient.

7 Discussion

This chapter will discuss the implications of the results of this thesis. Furthermore, the limitations and potential future work will be presented.

7.1 Implications

The results in this thesis indicate that the selection of the base image can have an impact on the energy consumption of the corresponding container. While the image has no impact on containers in an idle state, for actual workloads the impact is significant most of the time. Our findings show that the optimal choice for a base image, in terms of energy efficiency, depends on the containerized workload. Moreover, the magnitude of the difference in energy consumption is not consistent across all workloads. As a result, whether it is worth considering a more energy-efficient container image in practice depends on the case.

Consequently, developers need to do similar tests before deciding which base image to choose when targeting energy efficiency. For the web application workloads the execution time, frequency of running containers, and number of containers need to be taken into account to decide if the difference in energy consumption is significant in practice. These factors can also be considered for containerized gaming servers. However, since the energy difference is mainly caused by the start-up phase and real-world gaming servers usually run indefinitely, the difference in energy efficiency will be negligible. In contrast, for database workloads, the magnitude of the difference by itself is large enough to consider the base image selection.

The execution time of the container is only related to the energy consumption for cases where the CPU utilization stays consistently near 100% throughout the entire workload. This means that for the video transcoding workload selecting a base image on energy efficiency will also result in a time-efficient container. However, for other workloads, the execution time of the container is not directly correlated to its energy consumption. Therefore, developers should decide per case if the energy efficiency of the container is an acceptable trade-off for execution time.

The size of the base image does not impact the energy consumption directly. However, the experiments show that in many cases Alpine, which is the smallest base image, consumes the most energy. This high energy consumption is likely caused by Alpine's C standard library *musl*, which does not provide the same functionality as *glibc*. Therefore, based on these results we suggest that selecting Alpine is generally not optimal if the energy-efficiency is taken into account.

Additional results of the experiments in this thesis have shown that the version of the OS base image might also have an impact on the energy consumption of the container. We observed this in the *Node* web application workload, where the official *Node* image based on Debian 10 consumed more energy than the image based on Debian 12. Another interesting observation is the impact of a fixed CPU frequency. In the initial experiments, we did not lock this frequency. While this resulted in shorter execution times, the total energy consumption was considerably higher. Although this is not directly related to the main objective of this thesis, it can be considered in future research regarding the energy efficiency of software.

In conclusion, our findings indicate that the impact of base image selection depends on the workload. Since the execution time is not always correlated with energy consumption, time efficiency should not be the only factor to consider when striving for sustainable software. Furthermore, despite the small image size of Alpine, we advise developers to avoid this image if the energy efficiency of containers is a priority.

7.2 Limitations

Even though the results of the experiments provide insights into the impact of base image selection on the energy efficiency of containers, there are several threats to validity. First, the results of the experiments are based on one Docker version. While one could expect that the Docker version would not have a significant impact on the relative differences in energy consumption between the base images, we did not confirm this in our study.

Second, the experiments are performed on a single machine. This means that the same experiments on a machine with less efficient hardware could potentially produce different results. Furthermore, the additional background processes of the workloads were also run on the same machine, albeit on different cores. In practice, the workload itself might also switch between cores instead of being run on a single one. Moreover, only a single host OS was used for the experiments. As containerization also relies on the host OS, a different Linux distribution or even Windows or Mac as host OS could produce different results. Furthermore, as the experiments are only performed on an AMD processor, there is no certainty that the results on a machine with a different processor architecture will be the same.

Third, we can not assure that the room temperature was consistent throughout the entire experiment. To mitigate this threat, we did 30 runs for each base image in a random order. This means that there is no bias towards a specific base image.

To conclude, in order to ensure the validity of our experiments, we tried to limit the significance of these threats. We isolated a single core for the workload to ensure that the energy consumption data was not affected by other processes. Furthermore, we locked the frequency of the CPU to make each run as equal as possible. Finally, the 30 runs in randomized order ensure that potential outliers caused by external factors did not affect the overall results.

7.3 Future work

This thesis provides insights into factors that impact the energy efficiency of containerized workloads. As the findings of this study show that base image selection can affect the energy consumption of containers, we encourage further research into understanding energy-efficient containers. First, an extension of this study could be conducting experiments for other containerization solutions that are becoming more prevalent. Previous studies [12, 13] have shown that there can be differences in energy efficiency between containerization solutions, thus it could be interesting to see if the findings in Docker are similar for these solutions.

Next, the experiment with *Node* images showed that a different version of Debian can result in a difference in energy consumption. Older or newer versions of an OS base image might use different standard libraries that are more or less optimized for certain applications. Since older versions of base images are still used, mainly due to compatibility reasons, it could be interesting to further investigate this observation by considering multiple versions.

Finally, the focus of this study was to investigate the energy consumption during the run time of the container. Future work can also consider the total energy consumption of build time in combination with run time. This could provide additional recommendations for developers when selecting a base image since building the image is an important part as well.

8

Conclusion

Containerization has become a fundamental component of software development and the continuous integration and continuous delivery (CI/CD) pipeline. The energy overhead of containerization solutions such as Docker does not only result in higher costs, but also impacts the carbon footprint. As climate change is a global concern, it is also important to strive for greener software. This study suggests that developers and DevOps engineers should take base image selection into consideration when deploying Docker containers. Based on experiments that simulate real-world usage of Docker, this study provides insights on whether this consideration is worth taking.

The workloads in this study include a machine learning inference workload, two web application workloads, an online gaming workload, two database workloads, and a video transcoding workload. For each workload, we conducted an experiment in which we compared 4 different OS base images: Debian, Ubuntu, CentOS, and Alpine. At the start of the experiment, the machine is warmed up by running the workload for each base image and a prime number test. After this warm-up, we ran and monitored the workload for each base image 30 times in randomized order. Finally, the total energy consumption of each run is calculated.

RQ1

Does the base image selection have an impact on the energy efficiency of the container?

The results show that the base image can have a significant impact on the energy efficiency of the container. For almost all the workloads the energy consumption results are statistically significant, which means that at least two base images are different in energy efficiency. Only the results for the baselines and the *llama.cpp* workloads are not statistically significant. The Cliff's Delta is always large in the workloads where the difference is significant ($|\delta| > 0.43$). This large effect size means that one image is performing better than the other images in almost all runs. Overall, the base image selection has a significant impact on the energy efficiency of the corresponding container.

RQ2

Does the impact of base image selection vary for different types of workloads?

The actual impact of the base image selection depends on the workload run in the container. The results show that the impact is the most significant for database workloads. The differences in energy consumption do not always occur between the same pair of base images. This means that in general, there is no single best or worst base image in terms of energy consumption across all workloads. However, in many cases, Alpine is the least energy-efficient base image.

RQ3

Is there a trade-off between current selection criteria and energy efficiency?

The time needed to finish the workload and the energy consumption are not necessarily correlated. For CPU-intensive workloads energy consumption is correlated to the elapsed time, leading to a proportional difference in time and energy consumption between the base images. However, for most workloads, there is no or a weak correlation between time and energy. Interestingly enough, there are cases where the faster base image consumes more energy than the slower image. Furthermore, the smallest base image (i.e. Alpine) is often the least energy-efficient option. Alpine is mainly used because of its lightweight nature, but it is important to notice that there is a trade-off between this small image size and an energy-efficient container. Therefore, when selecting a base image, it might be beneficial to not only look at the execution time and size of the containerized workload, but also to take the total energy consumption of the container into consideration.

To conclude, this study provides a starting point for research into the optimization of energy efficiency of containers, by evaluating the impact of base images. With the findings of this study, we hope to motivate further research into this topic. Furthermore, the pipeline, workloads, and raw dataset of the experiments are publicly available to reproduce these findings. Finally, we suggest that developers and engineers should not only take execution time and image size into consideration but energy consumption as well when selecting a base image for containers.

References

[1] Roberto Morabito et al. “Hypervisors vs. Lightweight Virtualization: A Performance Comparison”. In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[2] Prateek Sharma et al. “Containers and Virtual Machines at Scale: A Comparative Study”. In: *Proceedings of the 17th International Middleware Conference*. Middleware ’16. Trento, Italy: Association for Computing Machinery, 2016. DOI: 10.1145/2988336.2988337. URL: https://doi.org/10.1145/2988336.2988337.

[3] Kavita Agarwal et al. “Containing the Hype”. In: *Proceedings of the 6th Asia-Pacific Workshop on Systems*. APSys ’15. Tokyo, Japan: Association for Computing Machinery, 2015. DOI: 10.1145/2797022.2797029. URL: https://doi.org/10.1145/2797022.2797029.

[4] Yinyuan Zhang et al. “Recommending Base Image for Docker Containers based on Deep Configuration Comprehension”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 449–453. DOI: 10.1109/SANER53432.2022.00060.

[5] Senay Semu Tadesse et al. “Characterizing the power cost of virtualization environments”. In: *Transactions on Emerging Telecommunications Technologies* 29 (2018).

[6] Sysdig. “Sysdig 2022 Cloud-Native Security and Usage Report”. In: *Sysdig* (2022).

[7] Stack Overflow. *2020 Developer Survey*. 2022. URL: https://survey.stackoverflow.co/2022.

[8] Mubin Ul Haque et al. “Challenges in Docker Development: A Large-Scale Study Using Stack Overflow”. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’20. Bari, Italy: Association for Computing Machinery, 2020. DOI: 10.1145/3382494.3410693. URL: https://doi.org/10.1145/3382494.3410693.

[9] Babak Bashari Rad et al. “An Introduction to Docker and Analysis of its Performance”. In: *IJCSNS International Journal of Computer Science and Network Security* 173 (Mar. 2017), p. 8.

[10] Eddie Antonio Santos et al. *How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers*. 2017. DOI: 10.48550/ARXIV.1705.01176. URL: https://arxiv.org/abs/1705.01176.

[11] Hannah Ritchie et al. “CO□ and Greenhouse Gas Emissions”. In: *Our World in Data* (2020). https://ourworldindata.org/co2-and-greenhouse-gas-emissions.

[12] Roberto Morabito. “Power Consumption of Virtualization Technologies: An Empirical Investigation”. In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 2015, pp. 522–527. DOI: 10.1109/UCC.2015.93.

[13] Raluca Maria Hampau et al. “An Empirical Study on the Performance and Energy Consumption of AI Containerization Strategies for Computer-Vision Tasks on the Edge”. In: *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’22. Gothenburg, Sweden: Association for Computing Machinery, 2022, pp. 50–59. DOI: 10.1145/3530019.3530025. URL: https://doi.org/10.1145/3530019.3530025.

[14] Mehul Warade et al. “Monitoring the Energy Consumption of Docker Containers”. In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2023, pp. 1703–1710. DOI: 10.1109/COMPSAC57700.2023.00263.

[15] Md Hasan Ibrahim et al. “Too Many Images on DockerHub! How Different Are Images for the Same System?” In: *Empirical Softw. Engg.* 25.5 (Sept. 2020), pp. 4250–4281. DOI: 10.1007/s10664-020-09873-0. URL: https://doi.org/10.1007/s10664-020-09873-0.

[16] Changyuan Lin et al. “A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 371–381. DOI: 10.1109/ICSME46990.2020.00043.

[17] Kalvin Eng et al. *Revisiting Dockerfiles in Open Source Software Over Time*. 2021. arXiv: 2103.12298 [cs.SE].

[18] Giovanni Rosa et al. “What Quality Aspects Influence the Adoption of Docker Images?” In: *ACM Trans. Softw. Eng. Methodol.* 32.6 (Sept. 2023). DOI: 10.1145/3603111. URL: https://doi.org/10.1145/3603111.

[19] Yang Zhang et al. “An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency”. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. 2018, pp. 138–143. DOI: 10.1109/COMPSAC.2018.00026.

[20] Microsoft. *Principle 2: Electricity*. 2023. URL: https://learn.microsoft.com/en-us/training/modules/sustainable-software-engineering-overview/4-electricity.

[21] European Environmnet Agency. *Sectoral greenhouse gas emissions by IPCC sector*. 2016. URL: https://www.eea.europa.eu/data-and-maps/daviz/change-of-co2-eq-emissions-2#tab-chart_4.

[22] Luís Cruz. *Sustainable Software: What, Why and How.* 2023. URL: https://luiscruz.github.io/course_sustainableSE/2023/.

[23] Scott Chamberlin. *Measuring Your Application Power and Carbon Impact (Part 1)*. 2020. URL: https://devblogs.microsoft.com/sustainable-software/measuring-your-application-power-and-carbon-impact-part-1/?WT.mc_id=green-8660-cxa.

[24] Dimitris Tsirogiannis et al. “Analyzing the Energy Efficiency of a Database Server”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 231–242. DOI: 10.1145/1807167.1807194. URL: https://doi.org/10.1145/1807167.1807194.

[25] Gregor Erbach. “Understanding energy efficiency”. In: *European Parliamentary Research Service* (2015). URL: https://www.europarl.europa.eu/RegData/etudes/BRIE/2015/568361/EPRS_BRI(2015)568361_EN.pdf.

[26] Sara Bergman. *How to measure the power consumption of your backend service*. 2020. URL: https://devblogs.microsoft.com/sustainable-software/how-to-measure-the-power-consumption-of-your-backend-service/.

[27] Luís Cruz. *Tools to Measure Software Energy Consumption from your Compute*. 2021. URL: https://luiscruz.github.io/2021/07/20/measuring-energy.html.

[28] Giovanni Rosa et al. *Fixing Dockerfile Smells: An Empirical Study*. 2022. DOI: 10.48550/ARXIV.2208.09097. URL: https://arxiv.org/abs/2208.09097.

[29] Perf Wiki. *perf: Linux profiling with performance counters*. 2023. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[30] Vince Weaver. *Linux support for Power Measurement Interfaces*. 2023. URL: https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html.

[31] Colin King. *Powerstat manual*. 2017. URL: https://manpages.ubuntu.com/manpages/bionic/man8/powerstat.8.html.

[32] Kashif Khan et al. “RAPL in Action: Experiences in Using RAPL for Power Measurements”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3 (Jan. 2018). DOI: 10.1145/3177754.

[33] Intel Corporation. *Reading and Writing Model Specific Registers (MSRs) in Linux*. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/reading-writing-msrs-in-linux.html.

[34] VMware. *Virtualization Overview*. 2006. URL: https://www.vmware.com/pdf/virtualization.pdf.

[35] Michael Eder. “Hypervisor- vs. Container-based Virtualization”. en. In: (2016). DOI: 10.2313/NET-2016-07-1_01. URL: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf.

[36] IBM. *What is virtualization?* 2023. URL: https://www.ibm.com/topics/virtualization.

[37] IBM Cloud Team. *Containers vs. Virtual Machines (VMs): What’s the Difference?* 2021. URL: https://www.ibm.com/cloud/blog/containers-vs-vms.

[38] Scott McCarty. *A Practical Introduction to Container Terminology*. 2021. URL: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.

[39] Senay Semu Tadesse et al. “Energy Consumption Measurements in Docker”. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2017, pp. 272–273. DOI: 10.1109/COMPSAC.2017.117.

[40] Scott van Kalken. *What Are Namespaces and cgroups, and How Do They Work?* 2021. URL: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/.

[41] Red Hat. *Chapter 1. Introduction to Control Groups (Cgroups)*. 2023. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[42] Docker. *Docker overview*. 2023. URL: https://docs.docker.com/get-started/overview/.

[43] IBM Corporation. *Docker host and Docker engine*. 2021. URL: https://www.ibm.com/docs/en/linux-on-systems?topic=linuxonibm/com.ibm.linux.z.ldvd/ldvd_c_docker_host.htm.

[44] Docker Inc. *dockerd*. 2023. URL: https://www.docker.com/blog/extending-docker-integration-with-containerd/.

[45] Djordje Lukic. *Extending Docker’s Integration with containerd*. 2022. URL: https://www.docker.com/blog/extending-docker-integration-with-containerd/.

[46] Mihail Kirov. *Digging Into Runtimes – runc*. 2022. URL: https://blog.quarkslab.com/digging-into-runtimes-runc.html.

[47] Ivan Velichko. *Implementing Container Runtime Shim: runc*. 2022. URL: https://iximiuz.com/en/posts/implementing-container-runtime-shim/.

[48] Isaac Arogbonlo. *Differences Between a DockerFile, Docker Image, and Docker Container*. 2022. URL: https://cto.ai/blog/docker-image-vs-container-vs-dockerfile/.

[49] Jyoti Jha. *Understanding the Dockerfile Commands*. 2022. URL: https://dev.to/aws-builders/understanding-the-dockerfile-format-3cc6.

[50] Docker. *Dockerfile reference*. 2023. URL: https://docs.docker.com/engine/reference/builder/.

[51] Roman Glushach. *Docker Images: A Deep Dive into Container Technology*. 2023. URL: https://romanglushach.medium.com/docker-images-a-deep-dive-into-container-technology-43ea01b4d7e1.

[52] Docker Inc. *Volumes*. 2023. URL: https://docs.docker.com/storage/volumes/.

[53] Docker Inc. *Bind mounts*. 2023. URL: https://docs.docker.com/storage/bind-mounts/.

[54] Docker Inc. *tmpfs mounts*. 2023. URL: https://docs.docker.com/storage/tmpfs/.

[55] Docker Inc. *Docker Compose overview*. 2023. URL: https://docs.docker.com/compose/.

[56] Giovanni Rosa et al. “Assessing and Improving the Quality of Docker Artifacts”. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2022, pp. 592–596. DOI: 10.1109/ICSME55016.2022.00081.

[57] Thomas Durieux. *Parfum: Detection and Automatic Repair of Dockerfile Smells*. 2023. arXiv: 2302.01707 [cs.SE].

[58] David Reis et al. “Dockerlive : A live development environment for Dockerfiles”. In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2022, pp. 1–4. DOI: 10.1109/VL/HCC53370.2022.9833145.

[59] Rolando Brondolin et al. “DEEP-Mon: Dynamic and Energy Efficient Power Monitoring for Container-Based Infrastructures”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 676–684. DOI: 10.1109/IPDPSW.2018.00110.

[60] Guillaume Fieni et al. *SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers*. 2020. DOI: 10.48550/ARXIV.2001.02505. URL: https://arxiv.org/abs/2001.02505.

[61] Wellington Silva-de-Souza et al. “Containergy—A Container-Based Energy and Performance Profiling Tool for Next Generation Workloads”. In: *Energies* 13.9 (2020). DOI: 10.3390/en13092162. URL: https://www.mdpi.com/1996-1073/13/9/2162.

[62] Ahmed Zerouali et al. “A Multi-Dimensional Analysis of Technical Lag in Debian-Based Docker Images”. In: *Empirical Softw. Engg.* 26.2 (Mar. 2021). DOI: 10.1007/s10664-020-09908-6. URL: https://doi.org/10.1007/s10664-020-09908-6.

[63] Fabio Palomba et al. “On the impact of code smells on the energy consumption of mobile applications”. In: *Information and Software Technology* 105 (2019), pp. 43–55. DOI: https://doi.org/10.1016/j.infsof.2018.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301678.

[64] Eugenio Capra et al. “Is software “green”? Application development environments and energy efficiency in open source applications”. In: *Information and Software Technology* 54.1 (2012), pp. 60–71. DOI: https://doi.org/10.1016/j.infsof.2011.07.005. URL: https://www.sciencedirect.com/science/article/pii/S0950584911001777.

[65] Luís Cruz. *Scientific Guide for Reliable Energy Experiments*. 2023. URL: https://luiscruz.github.io/course_sustainableSE/2023/.

[66] Gentoo Authors. *Sysbench*. 2023. URL: https://wiki.gentoo.org/wiki/Sysbench.

[67] H. Pishro-Nik. *Introduction to Probability, Statistics, and Random Processes*. Kappa Research, LLC, 2014. URL: https://www.probabilitycourse.com/.

[68] Debian. *About Debian*. 2021. URL: https://www.debian.org/intro/about.

[69] Ubuntu. *Debian: Debian is the rock on which Ubuntu is built*. 2023. URL: https://ubuntu.com/community/governance/debian.

[70] Inc. Red Hat. *Red Hat Enterprise Linux*. 2023. URL: https://www.redhat.com/en/red-hat-enterprise-linux.

[71] Meta AI. *Introducing LLaMA: A foundational, 65-billion-parameter large language model*. 2023. URL: https://ai.facebook.com/blog/large-language-model-llama-meta-ai/.

[72] Zachary Boddy. *Minecraft crosses 300 million copies sold as it prepares to celebrate its 15th anniversary*. 2023. URL: https://www.windowscentral.com/gaming/minecraft/minecraft-crosses-300-million-copies-sold-as-it-prepares-to-celebrate-its-15th-anniversary.

[73] Anton Gavrailov. *The Role of Video Transcoding in Online Streaming Platforms*. 2023. URL: https://www.linkedin.com/pulse/role-video-transcoding-online-streaming-platforms-anton-gavrailov.

[74] Janko Roettgers. *How ‘Big Buck Bunny’ — a movie you’ve probably never heard of — became an internet legacy*. 2021. URL: https://www.protocol.com/big-buck-bunny-video-industry.

[75] Prabhakar Mishra et al. “Descriptive Statistics and Normality Tests for Statistical Data”. In: *Annals of Cardiac Anaesthesia* 22 (Jan. 2019), pp. 67–72. DOI: 10.4103/aca.ACA_157_18.

[76] Andrew P. King et al. “Chapter 7 - Inferential Statistics IV: Choosing a Hypothesis Test”. In: *Statistics for Biomedical Engineers and Scientists*. Ed. by Andrew P. King et al. Academic Press, 2019, pp. 147–171. DOI: https://doi.org/10.1016/B978-0-08-102939-8.00016-5. URL: https://www.sciencedirect.com/science/article/pii/B9780081029398000165.

[77] Kandethody M. Ramachandran et al. “Chapter 11 - Categorical data analysis and goodness-of-fit tests and applications”. In: *Mathematical Statistics with Applications in R (Third Edition)*. Ed. by Kandethody M. Ramachandran et al. Third Edition. Academic Press, 2021, pp. 461–490. DOI: https://doi.org/10.1016/B978-0-12-817815-7.00011-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128178157000117.

[78] Hae-Young Kim. “Statistical notes for clinical researchers: Assessing normal distribution (2) using skewness and kurtosis”. In: *Restorative dentistry & endodontics* 38 (Feb. 2013), pp. 52–54. DOI: 10.5395/rde.2013.38.1.52.

[79] Kent State University Libraries. *SPSS TUTORIALS: ONE-WAY ANOVA*. 2023. URL: https://libguides.library.kent.edu/spss/onewayanova.

[80] Yinglin Xia. “Chapter Eleven - Correlation and association analyses in microbiome study integrating multiomics in health and disease”. In: *The Microbiome in Health and Disease*. Ed. by Jun Sun. Vol. 171. Progress in Molecular Biology and Translational Science. Academic Press, 2020, pp. 309–491. DOI: https://doi.org/10.1016/bs.pmbts.2020.04.003. URL: https://www.sciencedirect.com/science/article/pii/S1877117320300478.

[81] Stephen W. Scheff. “Chapter 8 - Nonparametric Statistics”. In: *Fundamental Statistical Principles for the Neurobiologist*. Ed. by Stephen W. Scheff. Academic Press, 2016, pp. 157–182. DOI: https://doi.org/10.1016/B978-0-12-804753-8.00008-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128047538000087.

[82] Andras Vargha et al. “A Critique and Improvement of the ”CL” Common Language Effect Size Statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics - J EDUC BEHAV STAT* 25 (June 2000). DOI: 10.2307/1165329.

[83] Alpine Linux. *About Alpine*. 2022. URL: https://alpinelinux.org/about/.

[84] Martin Heinz. *Why I Will Never Use Alpine Linux Ever Again*. 2023. URL: https://martinheinz.dev/blog/92.

A

Median results - Power usage

A.1 Baselines

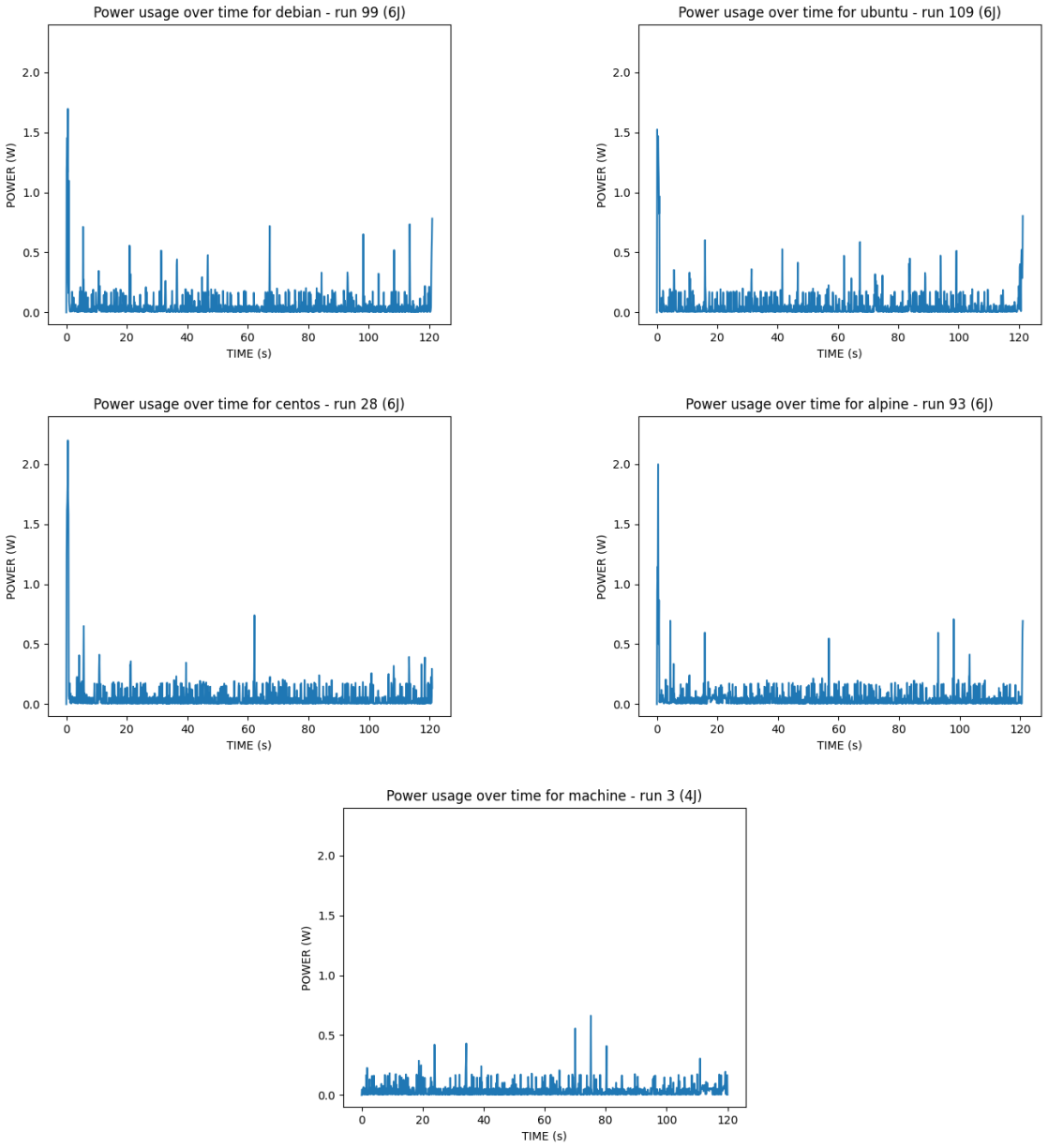

Figure A.1: Docker baseline and machine baseline. Power usage over time

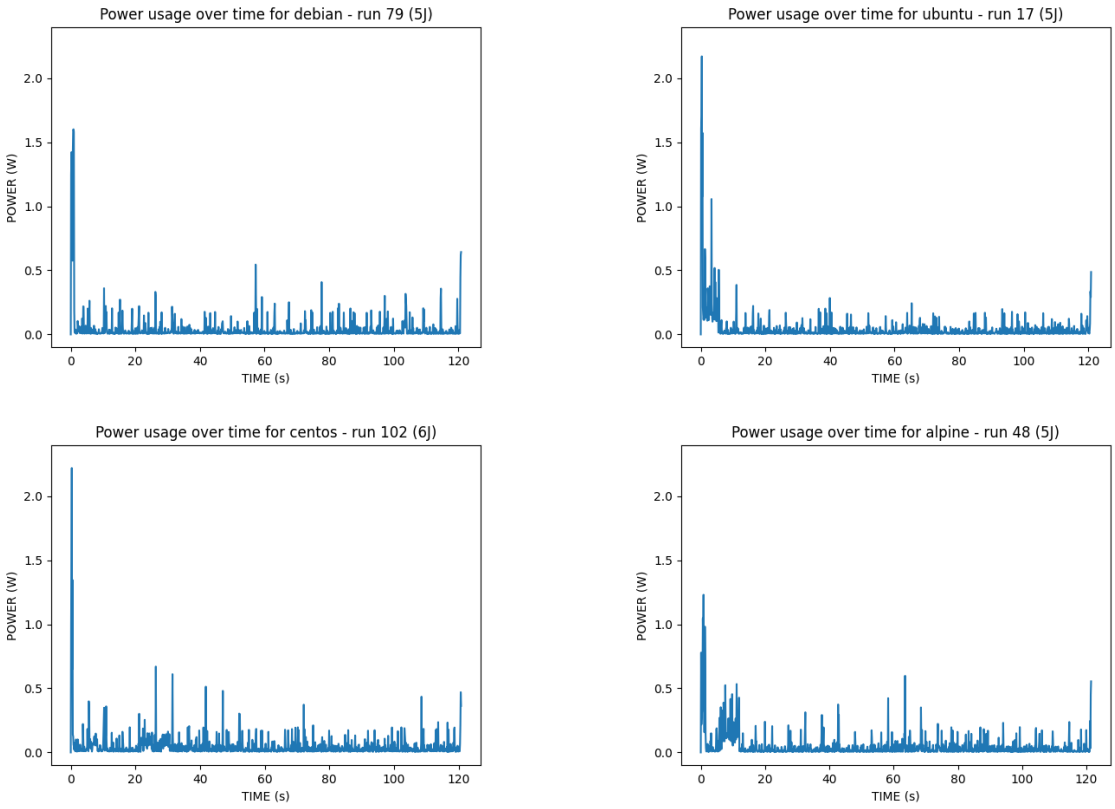

Figure A.2: Bloated Docker baseline. Power usage over time

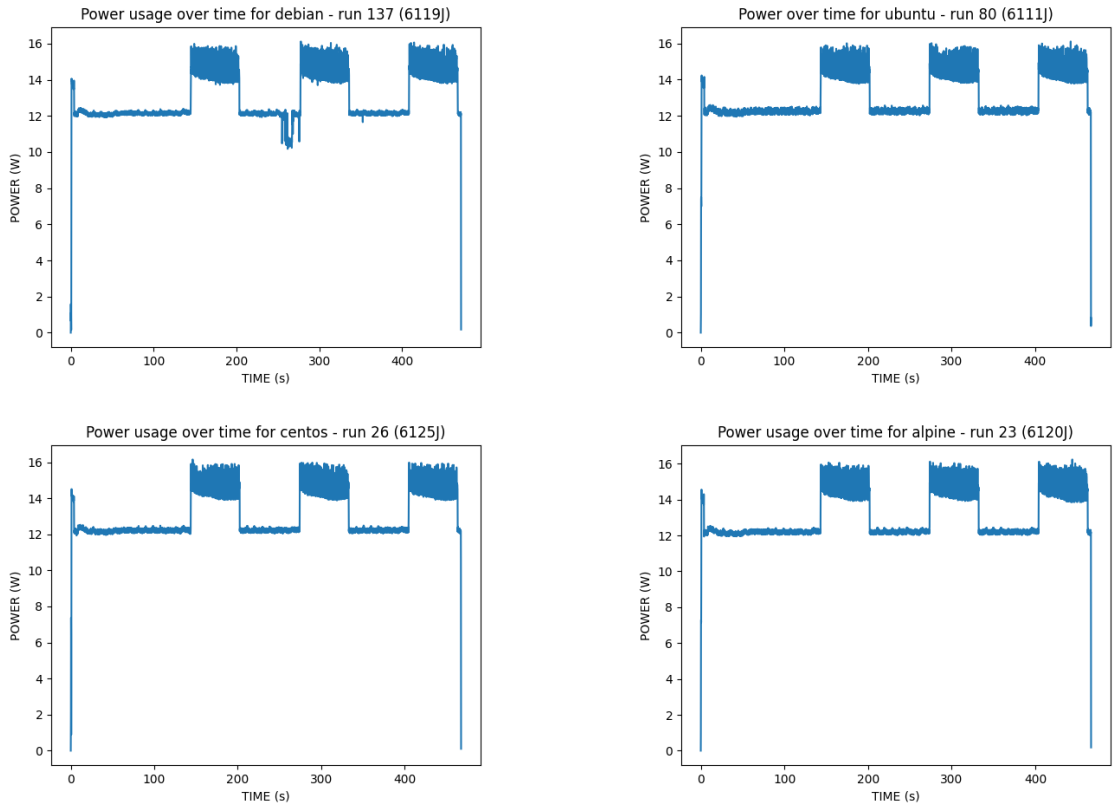

Figure A.3: llama with volume. Power usage over time

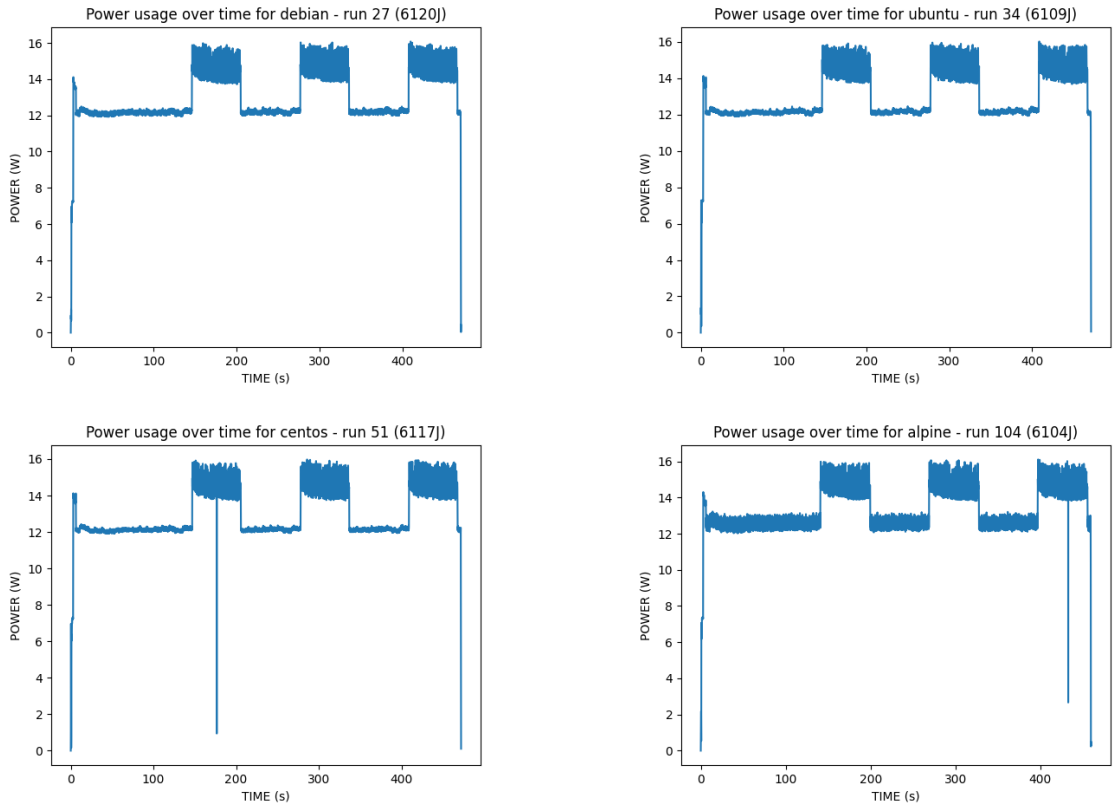

Figure A.4: llama with model. Power usage over time

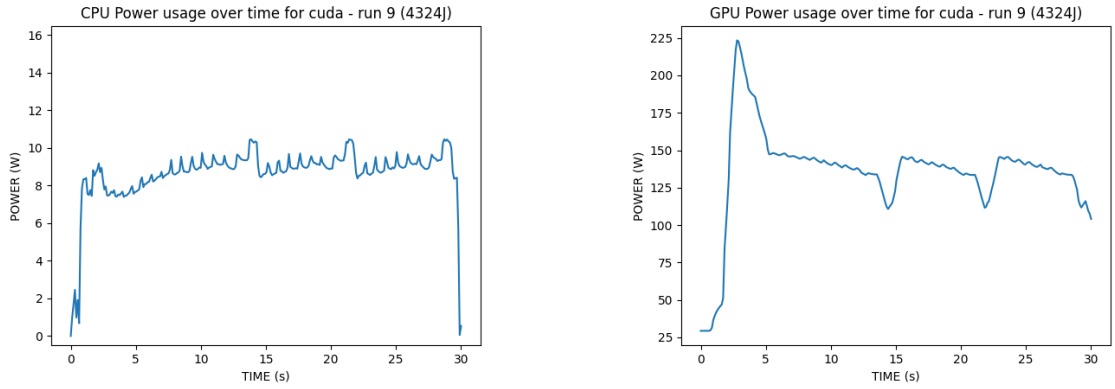

Figure A.5: llama with GPU. Power usage over time

A.2 Mattermost

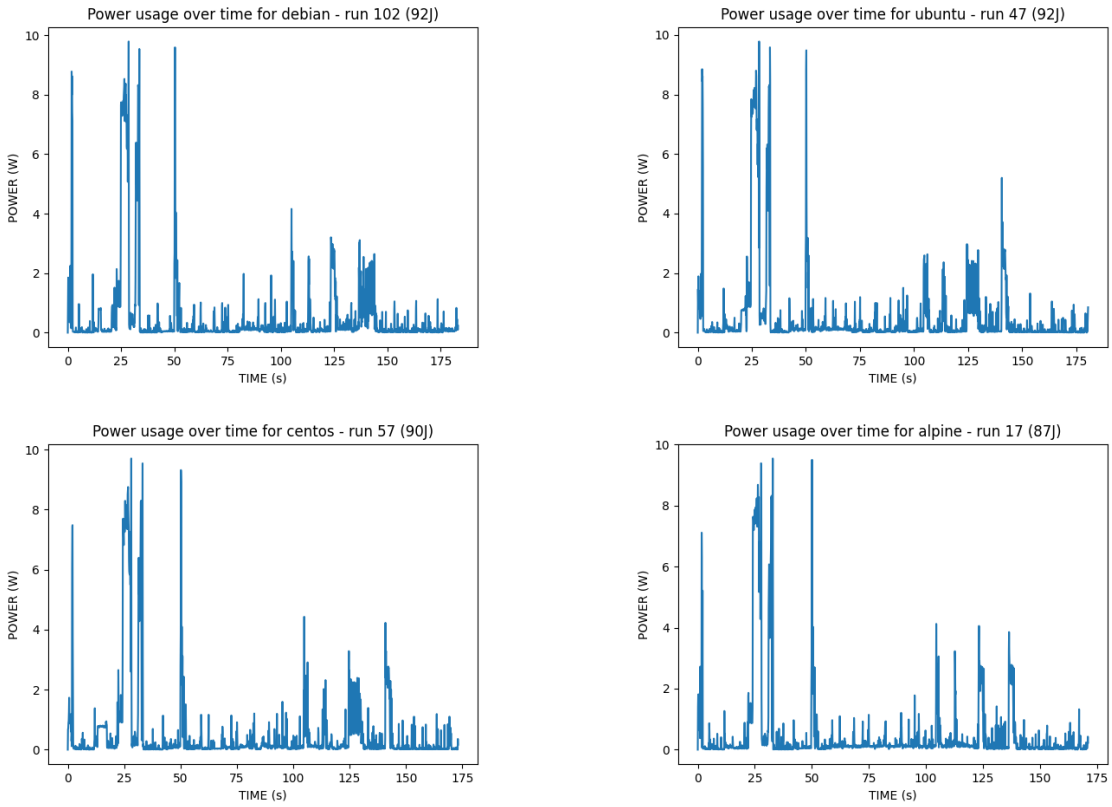

Figure A.6: Mattermost. Power usage over time

A.3 Cypress Real World App

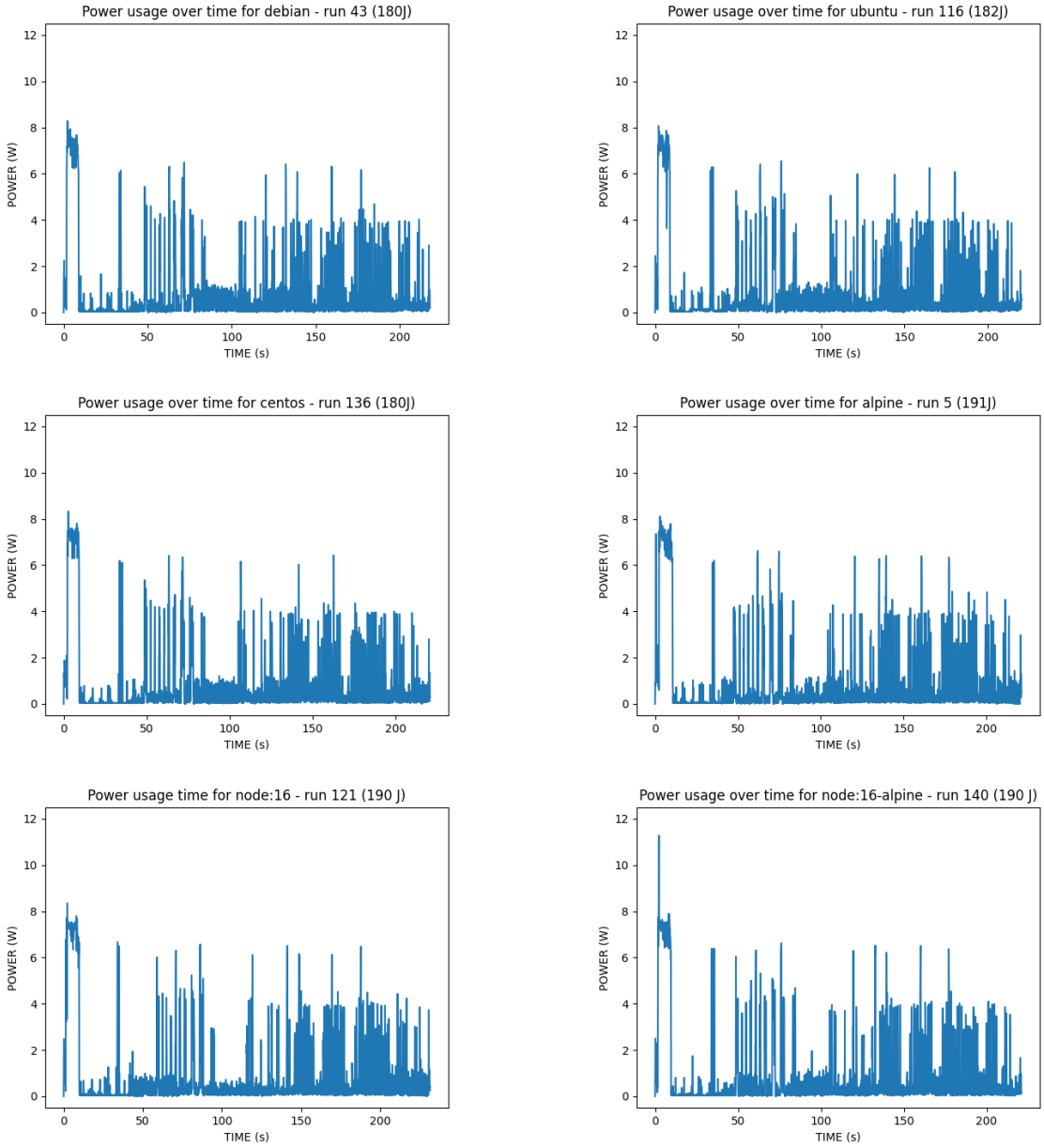

Figure A.7: Cypress Real World App. Power usage over time

A.4 Minecraft server

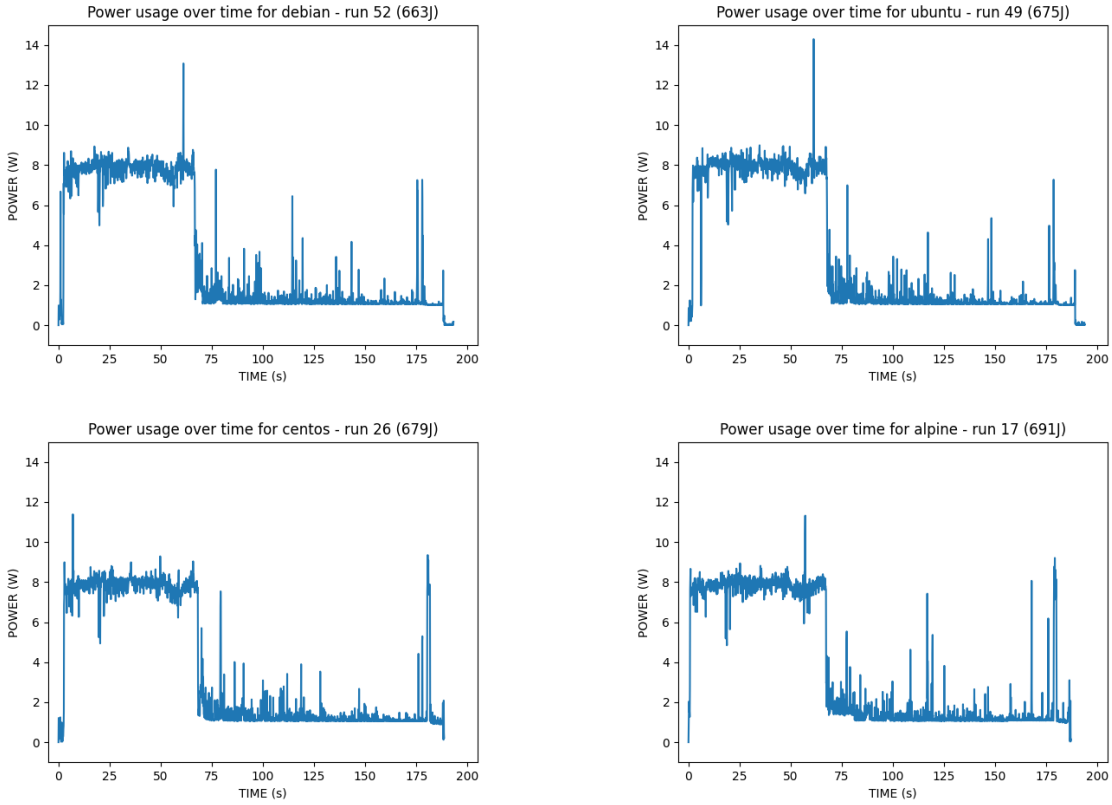

Figure A.8: Minecraft server with world. Power usage over time

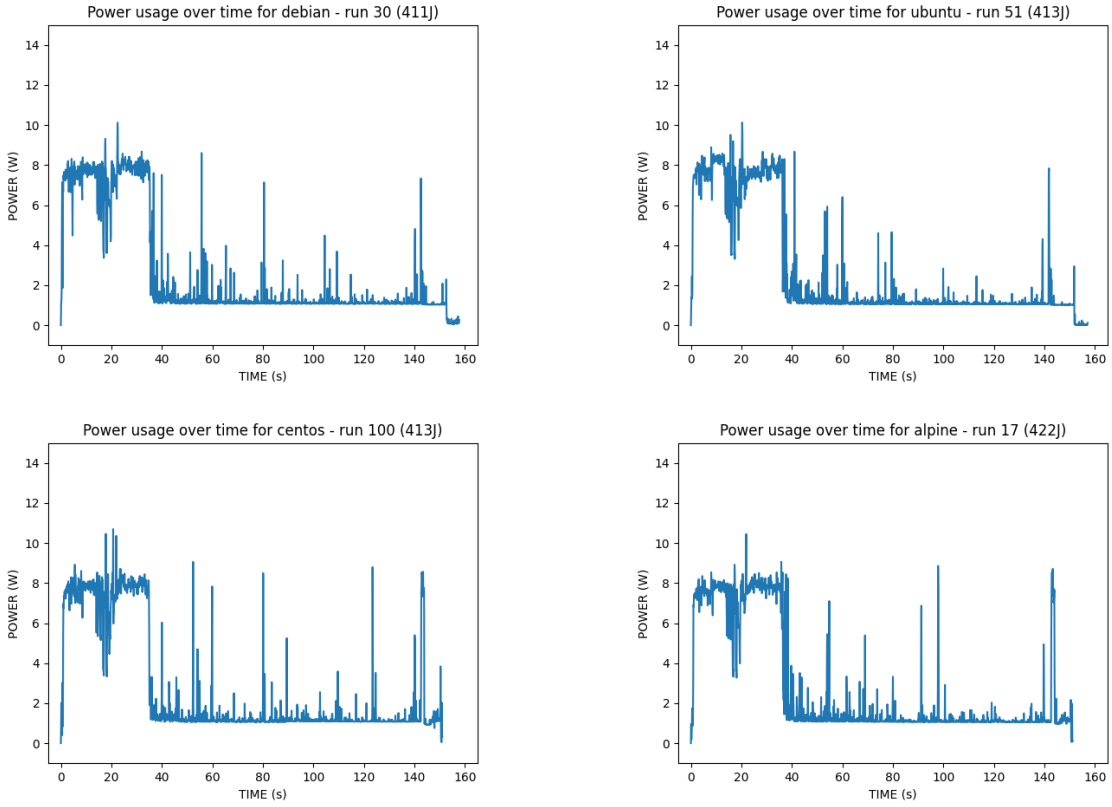

Figure A.9: Minecraft server without world. Power usage over time

A.5 Redis server

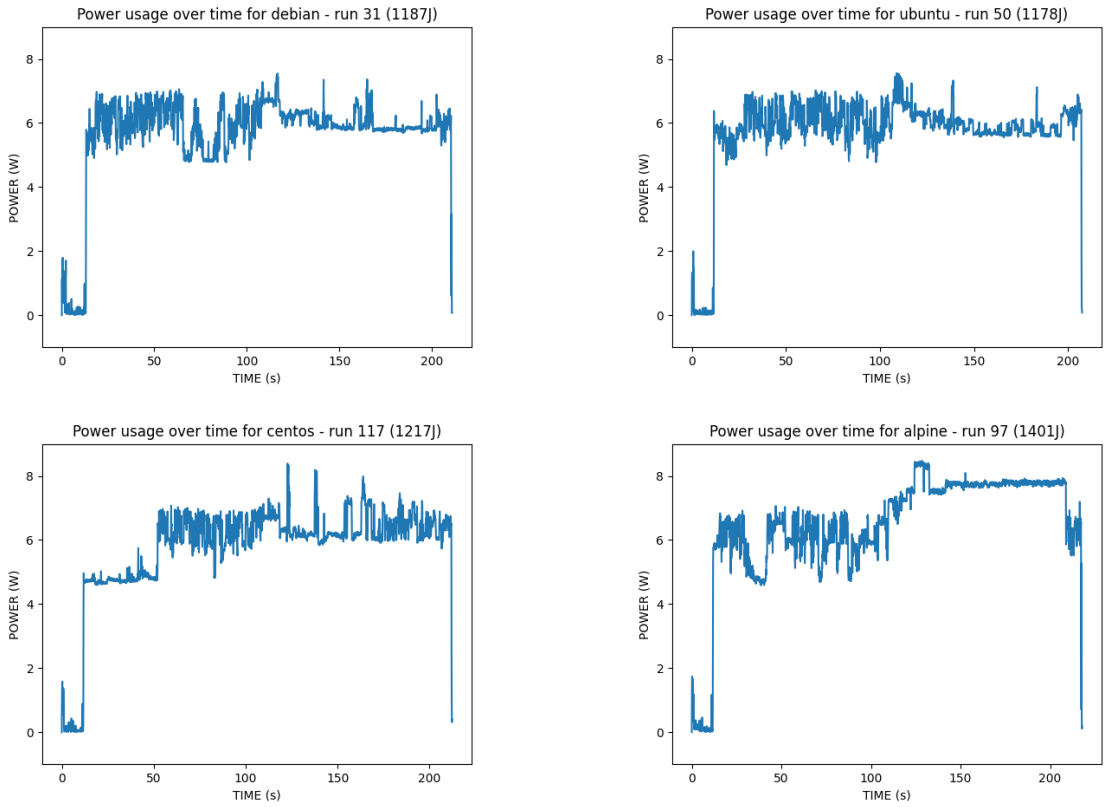

Figure A.10: Redis server. Power usage over time

A.6 Postgres server

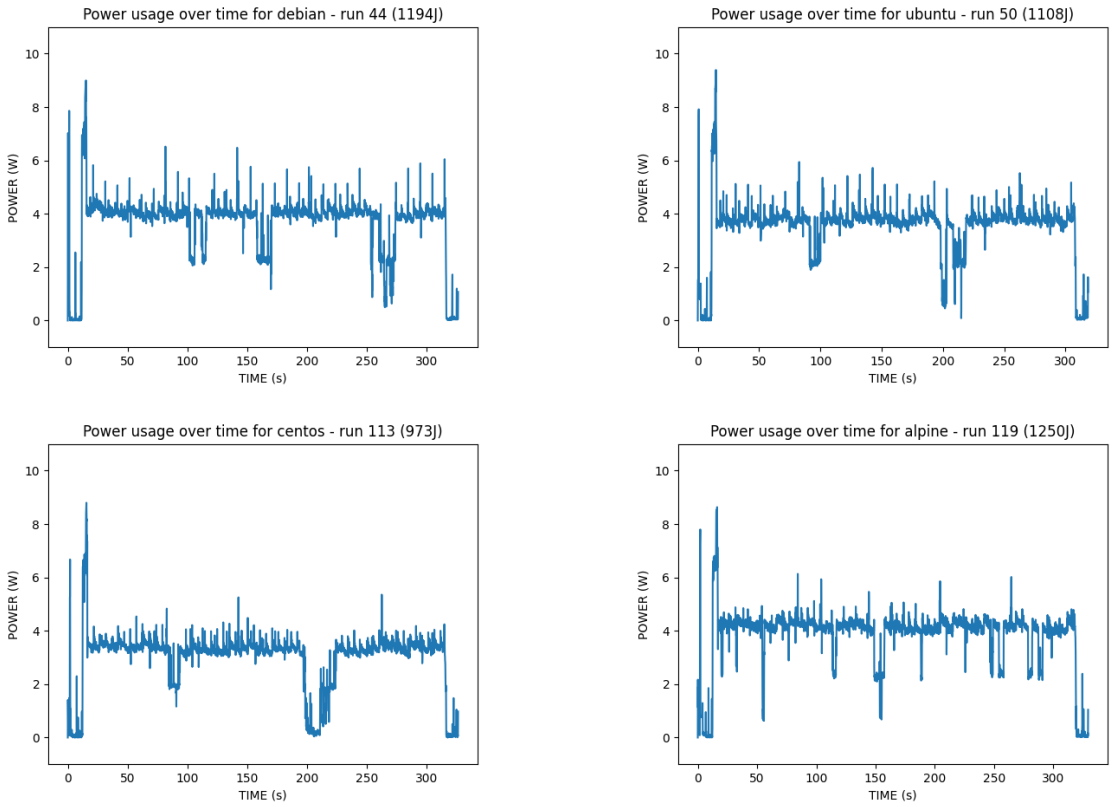

Figure A.11: Postgres server. Power usage over time

A.7 FFmpeg

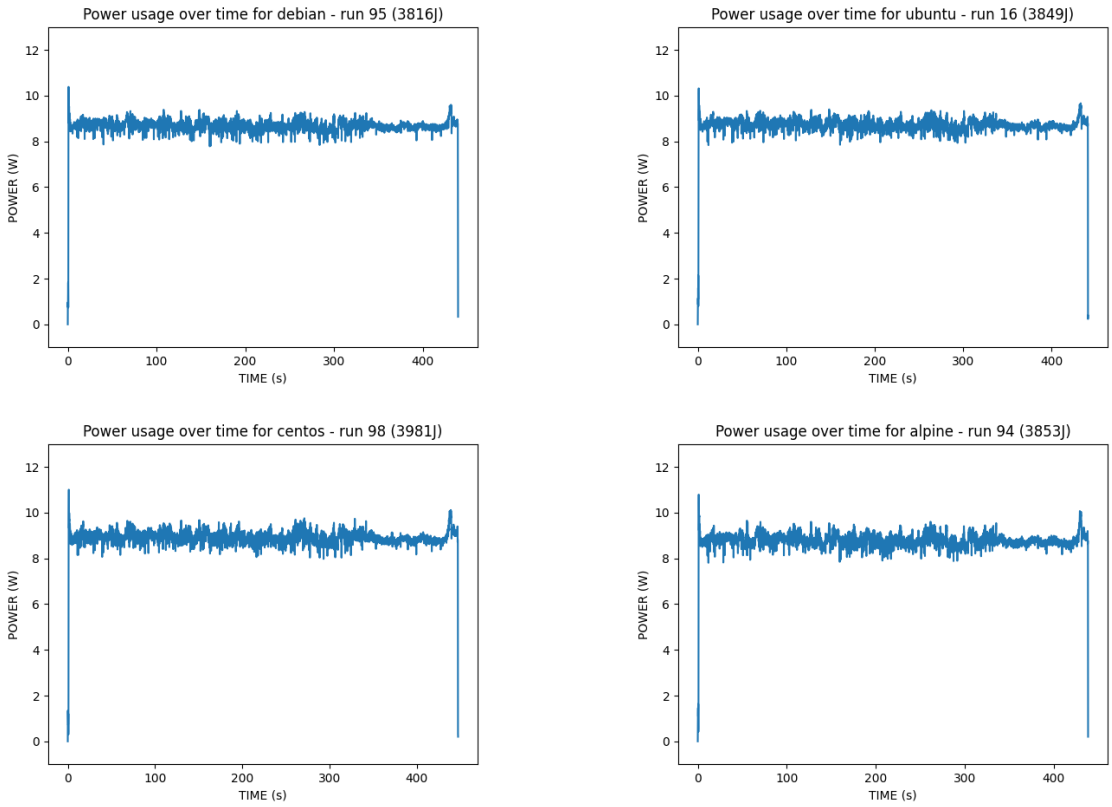

Figure A.12: FFmpeg. Power usage over time

B

Median results - CPU usage

B.1 Baselines

Table B.1: Docker baseline and machine baseline. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.45**
ubuntu	**0.44**
centos	**0.52**
alpine	**0.47**
machine	0.09

Table B.2: Bloated Docker baseline. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	0.36
ubuntu	0.18
centos	**0.44**
alpine	**0.46**

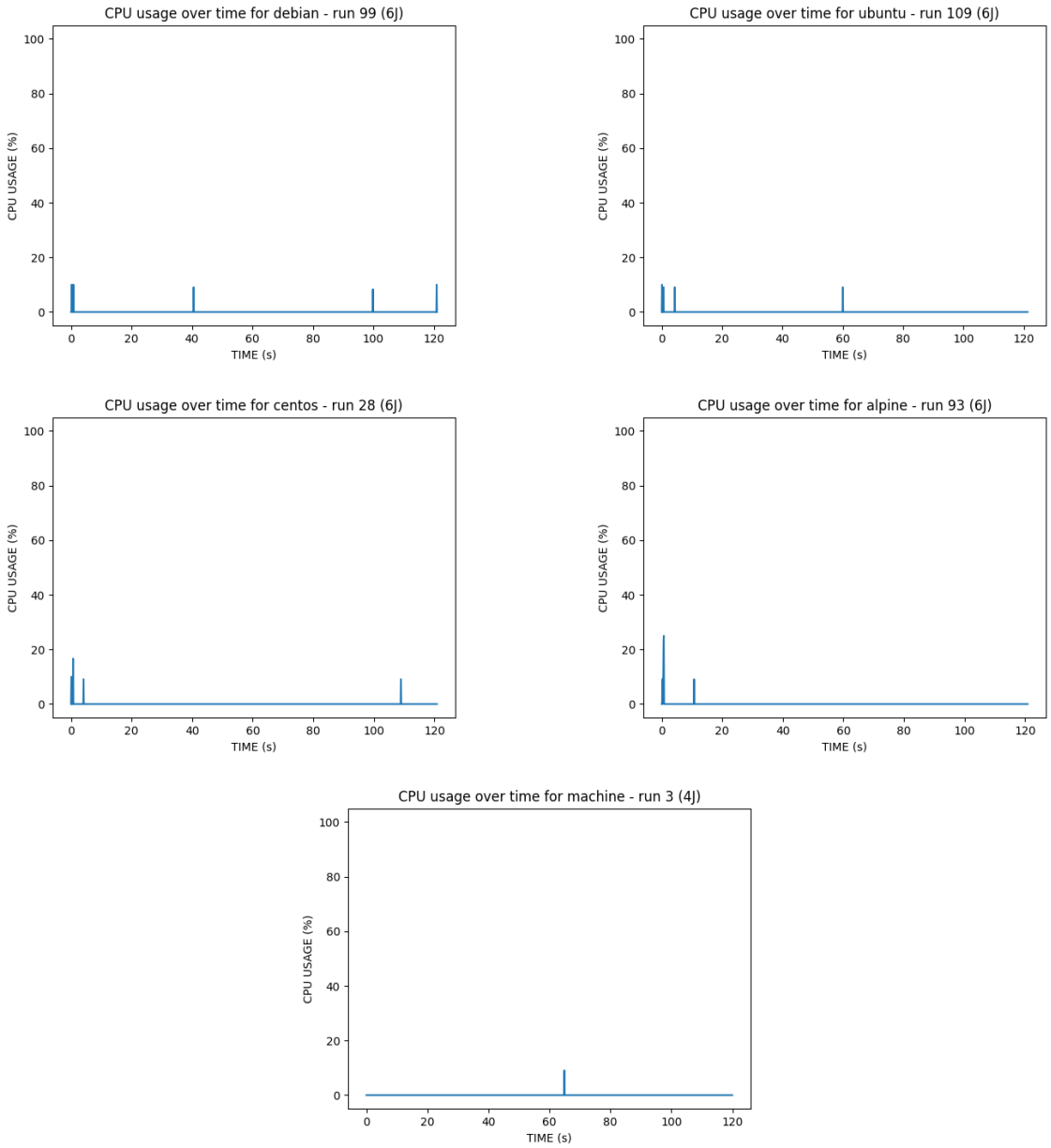

Figure B.1: Docker baseline and machine baseline. CPU usage over time

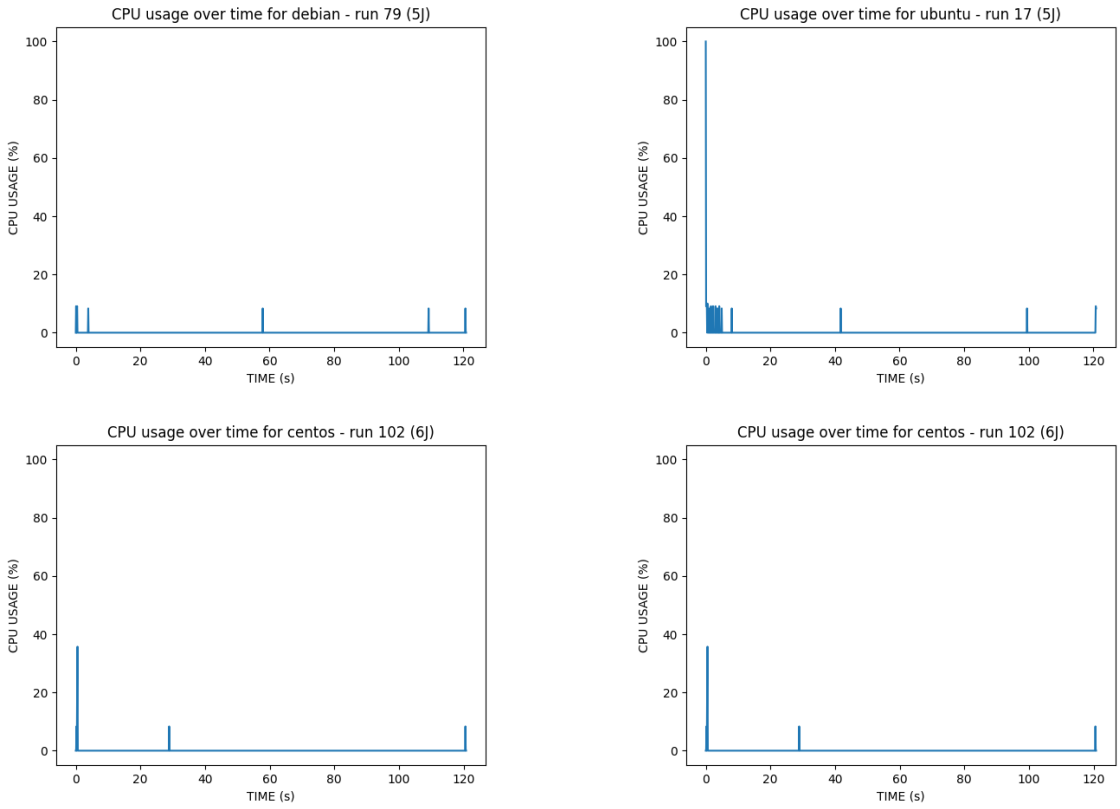

Figure B.2: Bloated Docker baseline. CPU usage over time

B.2 llama.cpp

Table B.3: llama.cpp with volume. Correlation between power and CPU usage.

Image	Correlation (Power / CPU 1 usage)	Correlation (Power / CPU 2 usage)
debian	0.39	**0.40**
ubuntu	0.35	0.36
centos	0.37	0.38
alpine	0.28	0.31

Table B.4: llama.cpp with model. Correlation between power and CPU usage.

Image	Correlation (Power / CPU 1 usage)	Correlation (Power / CPU 2 usage)
debian	**0.50**	**0.45**
ubuntu	**0.48**	**0.42**
centos	**0.46**	**0.40**
alpine	**0.47**	**0.54**

Table B.5: llama.cpp with GPU. Correlation between power and CPU usage.

Image	Correlation (GPU Power / GPU usage)
cuda	**0.57**

Image	Correlation (CPU Power / CPU 1 usage)	Correlation (Power / CPU 2 usage)
cuda	**0.63**	**0.89**

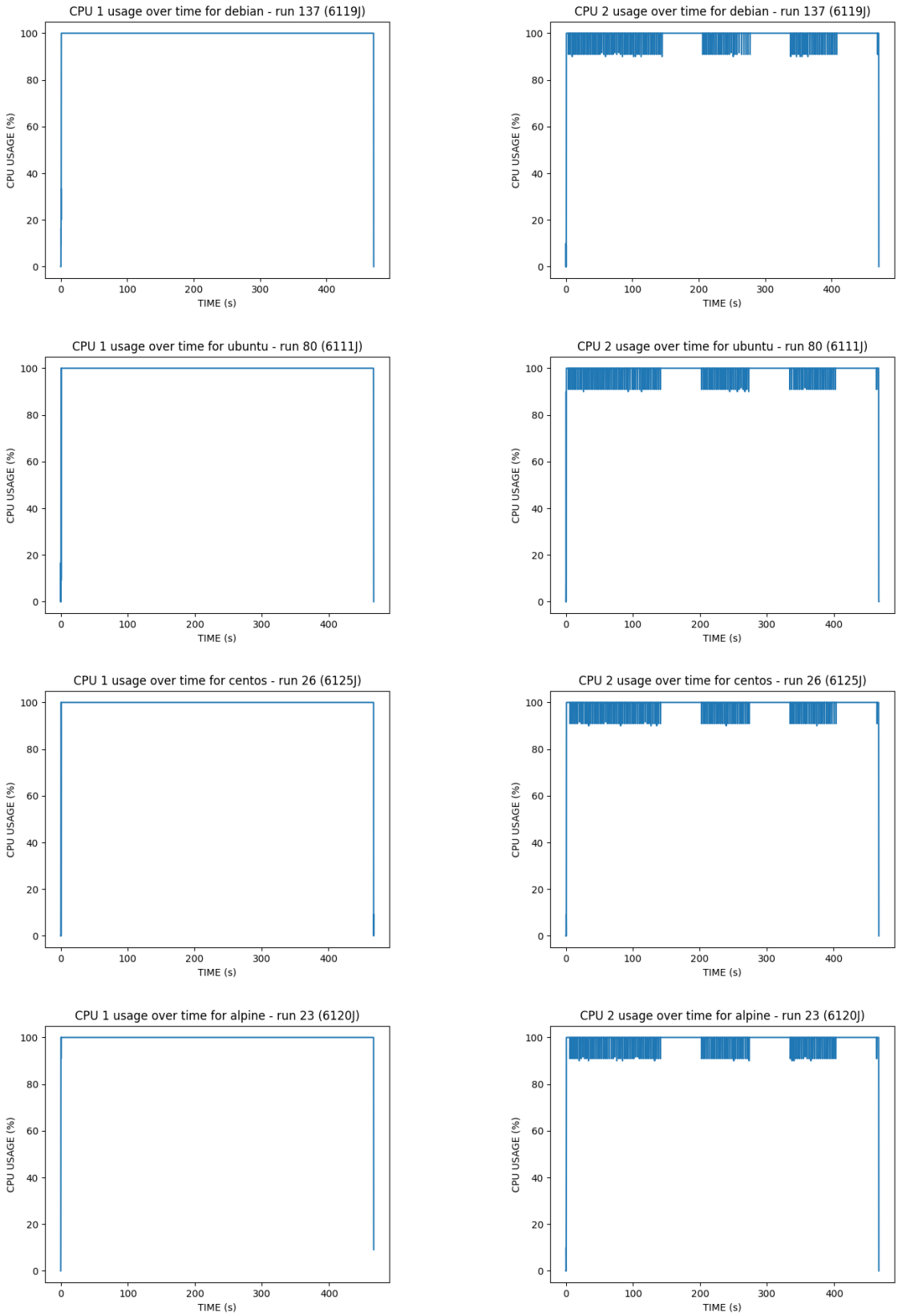

Figure B.3: llama with volume. CPU usage over time

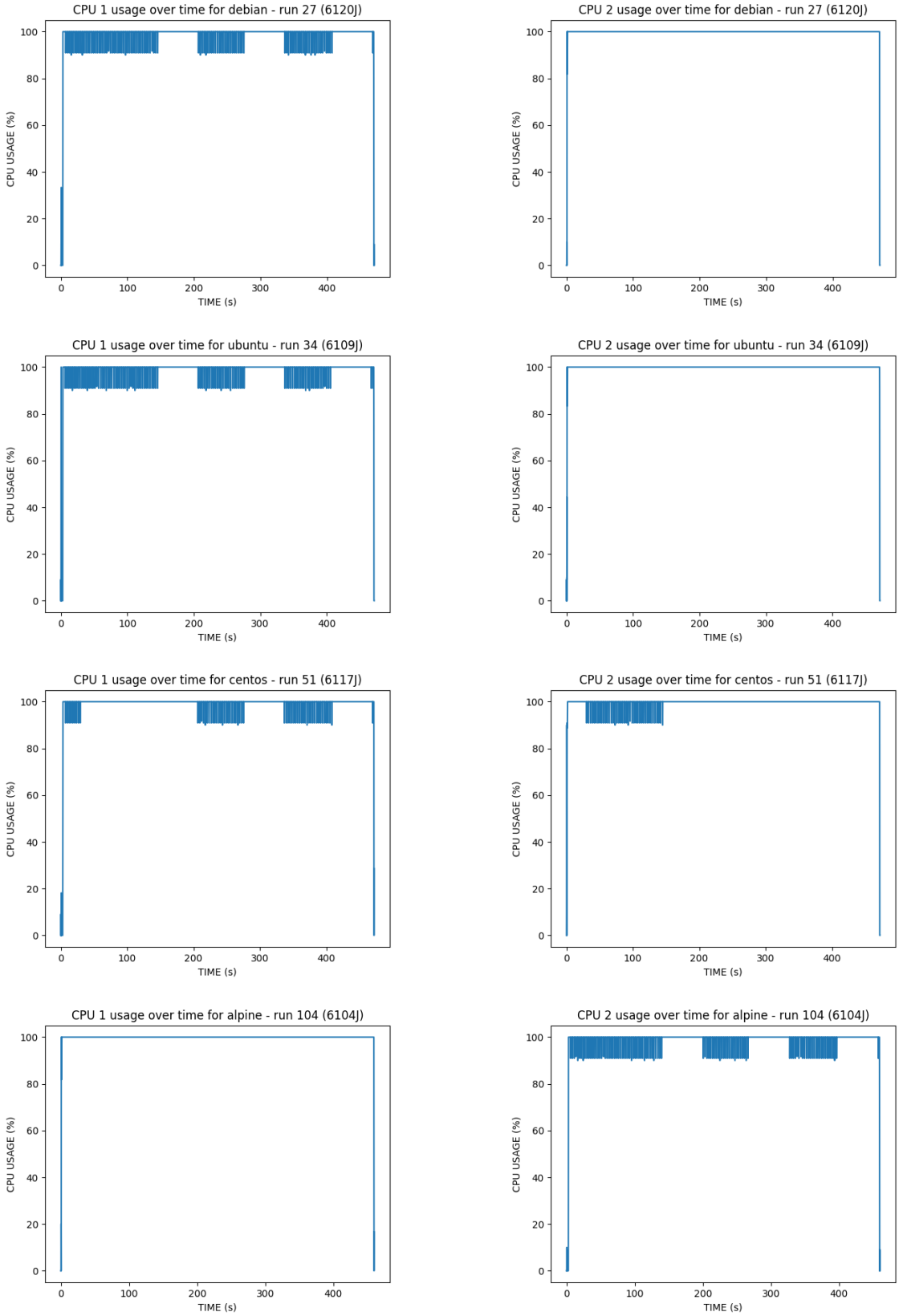

Figure B.4: llama with model. CPU usage over time

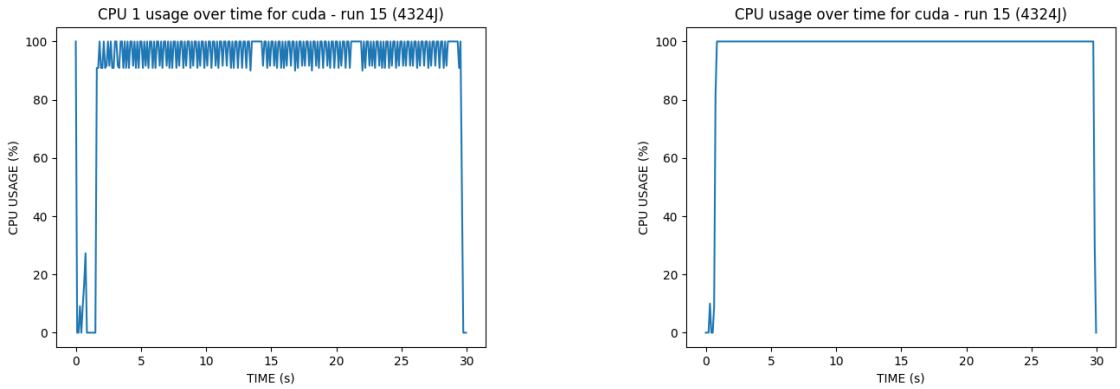

Figure B.5: llama with GPU. CPU usage over time

B.3 Mattermost

Table B.6: Mattermost. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.93**
ubuntu	**0.93**
centos	**0.93**
alpine	**0.93**

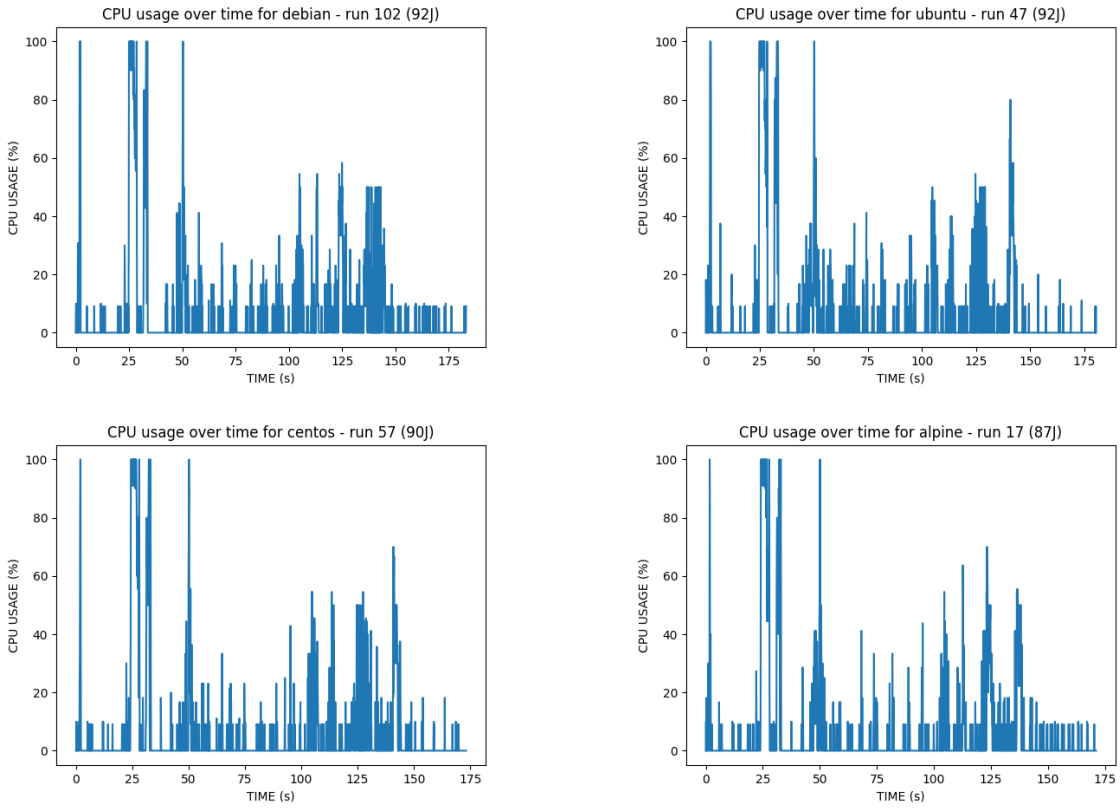

Figure B.6: Mattermost. CPU usage over time

B.4 Cypress Real World App

Table B.7: Cypress Real World App. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.86**
ubuntu	**0.86**
centos	**0.88**
alpine	**0.87**
node	**0.87**
node-alpine	**0.89**

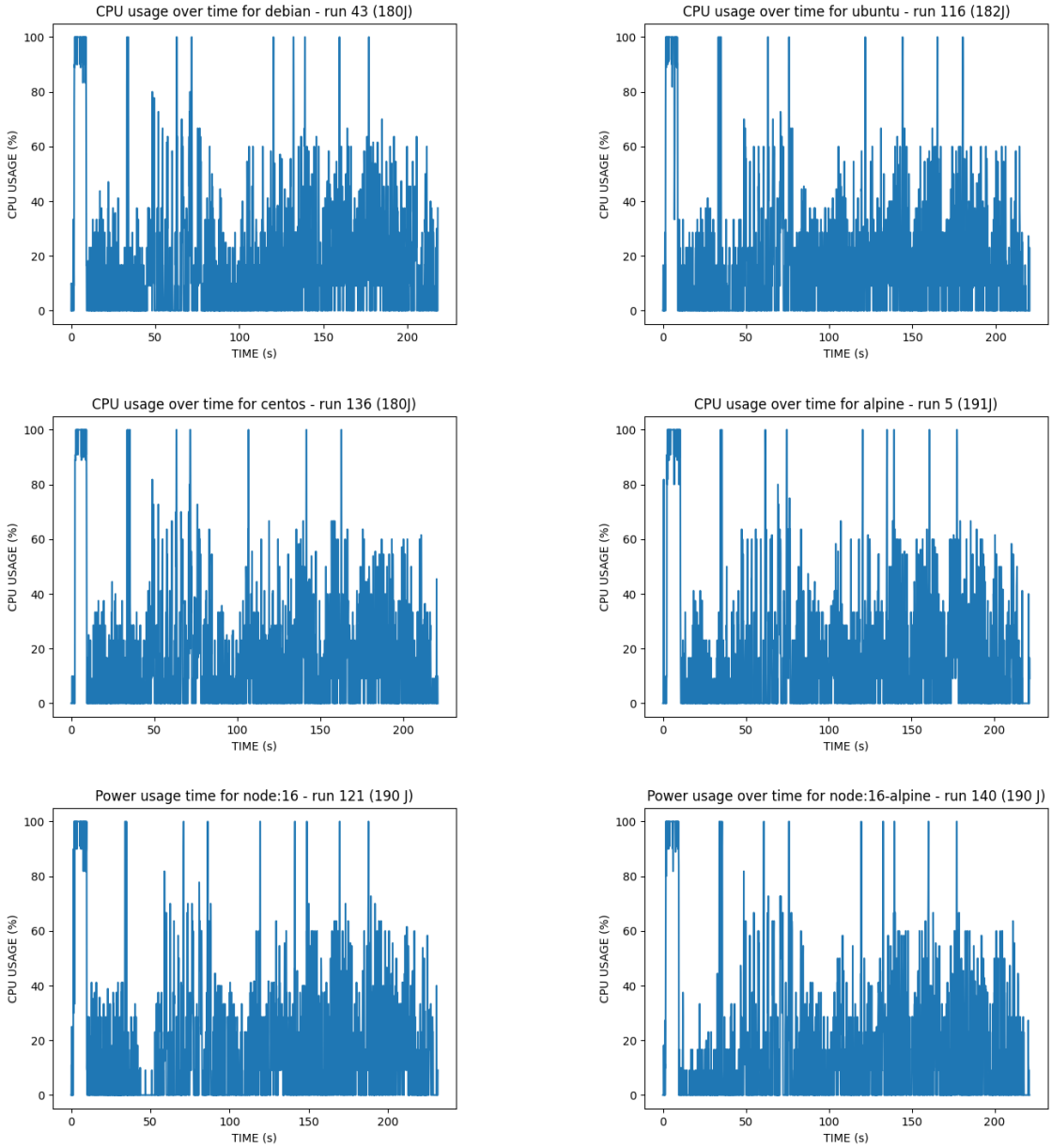

Figure B.7: Cypress Real World App. CPU usage over time

B.5 Minecraft server

Table B.8: Minecraft server with world. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.99**
ubuntu	**0.99**
centos	**0.99**
alpine	**0.99**

Table B.9: Minecraft server without world. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.98**
ubuntu	**0.98**
centos	**0.98**
alpine	**0.98**

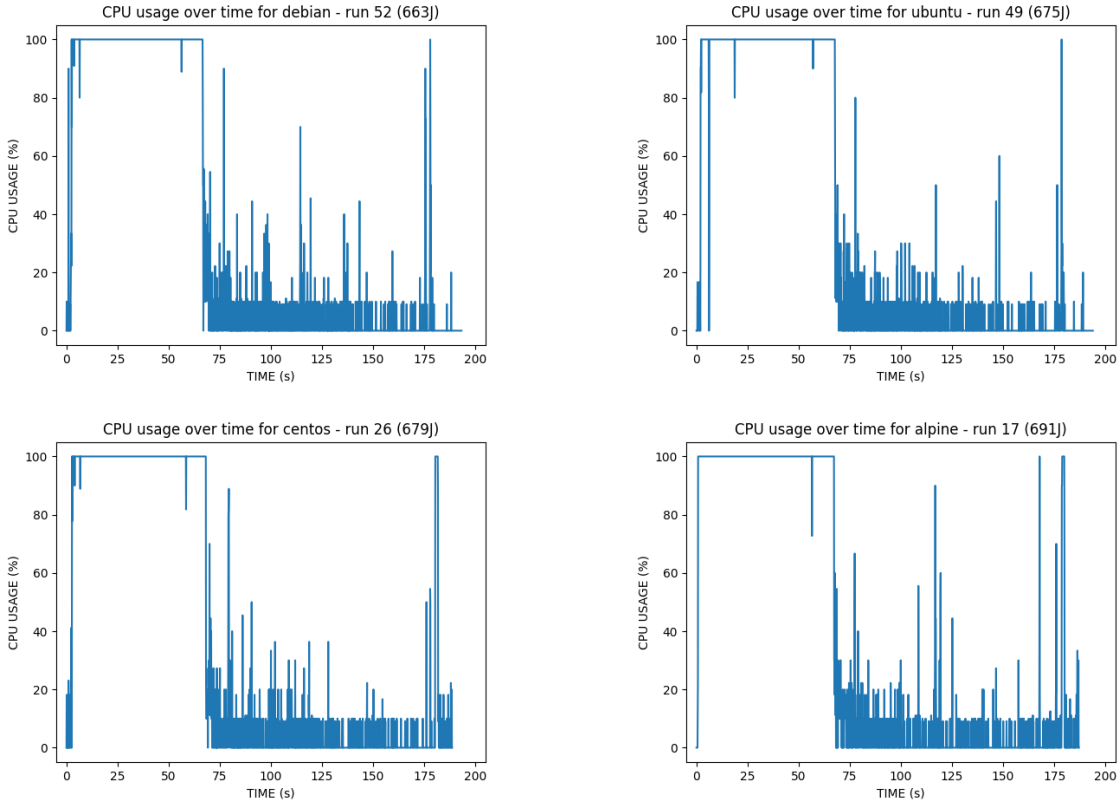

Figure B.8: Minecraft server with world. CPU usage over time

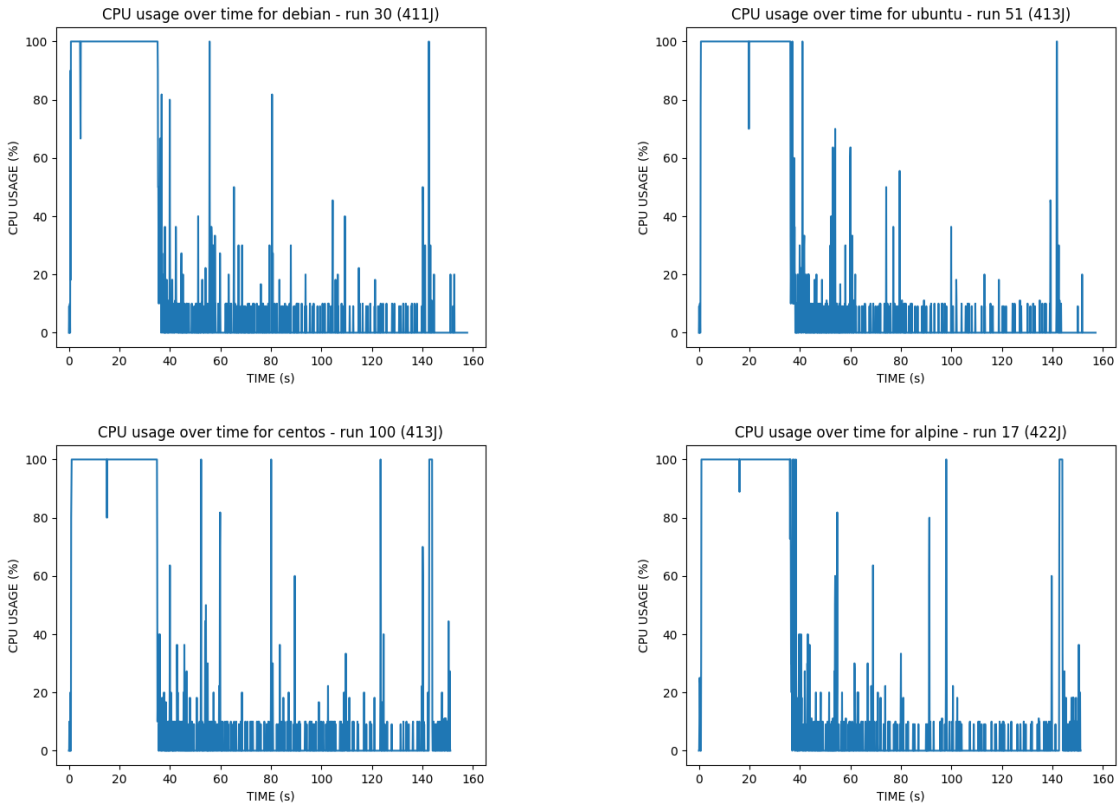

Figure B.9: Minecraft server without world. CPU usage over time

B.6 Redis server

Table B.10: Redis server. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.72**
ubuntu	**0.77**
centos	**0.59**
alpine	**0.79**

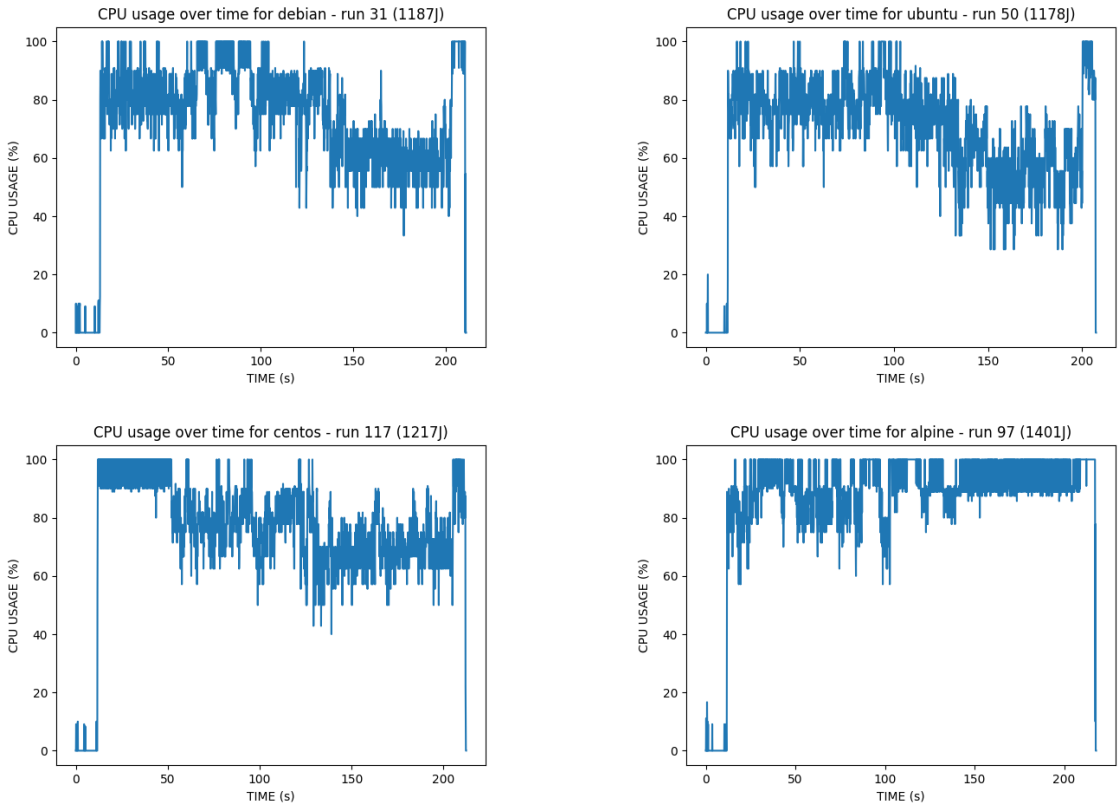

Figure B.10: Redis server. CPU usage over time

B.7 Postgres server

Table B.11: Postgres server. Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.91**
ubuntu	**0.90**
centos	**0.89**
alpine	**0.92**

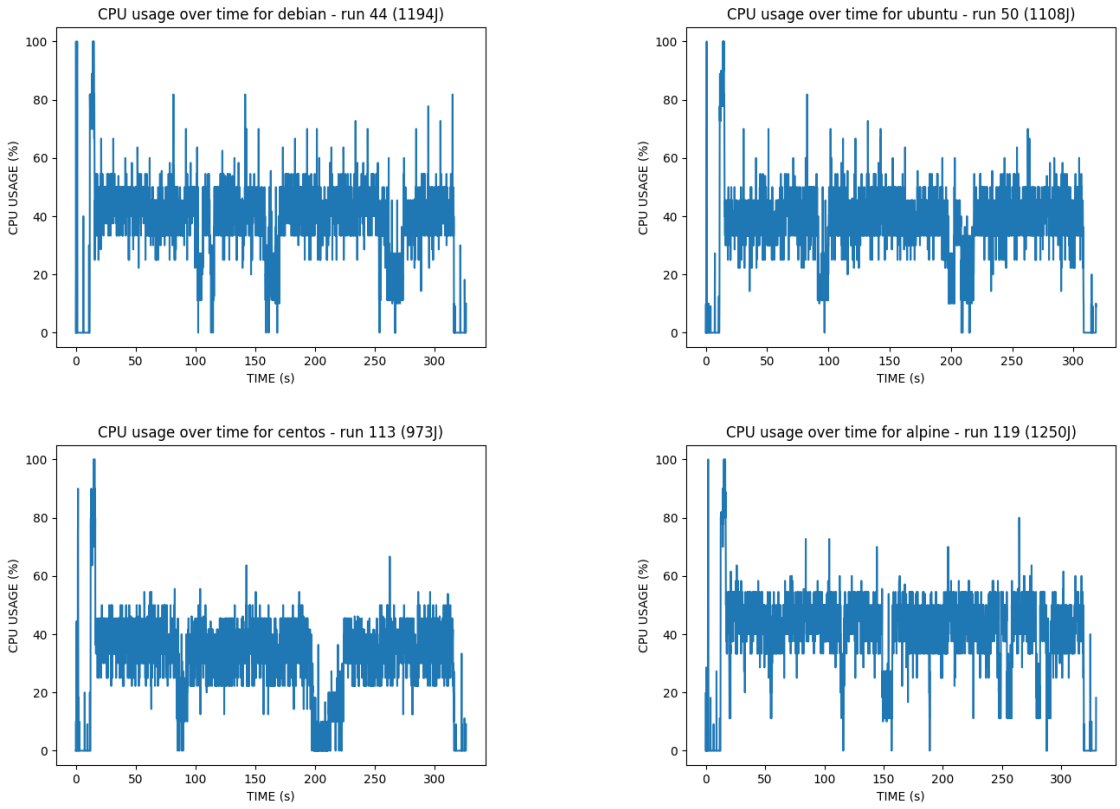

Figure B.11: Postgres server. CPU usage over time

B.8 FFmpeg

Table B.12: FFmpeg Correlation between power and CPU usage.

Image	Correlation (Power / CPU usage)
debian	**0.86**
ubuntu	**0.87**
centos	**0.86**
alpine	**0.84**

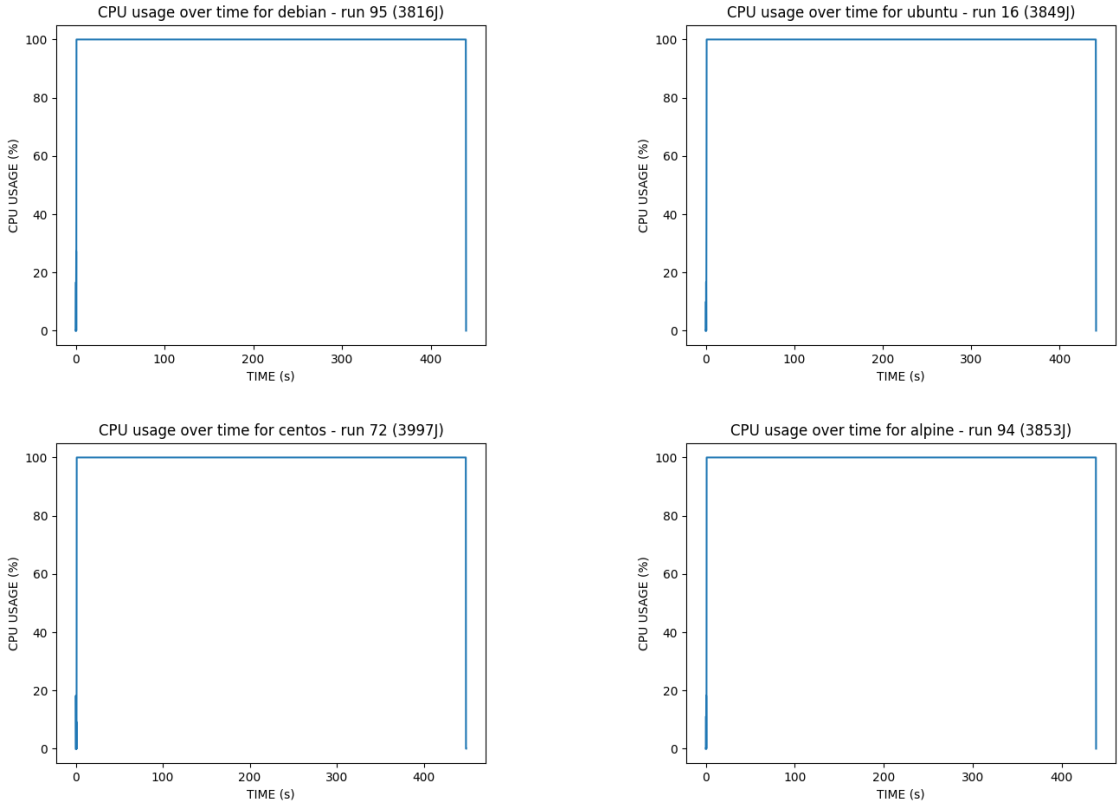

Figure B.12: FFmpeg. CPU usage over time