Measuring Energy Consumption during Continuous Integration of Open-Source Java Projects

Master's Thesis

Robert Arntzenius

Measuring Energy Consumption during Continuous Integration of Open-Source Java Projects

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Robert Arntzenius
born in Heemstede, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Measuring Energy Consumption during Continuous Integration of Open-Source Java Projects

Author: Robert Arntzenius
Student id: 5643740

Abstract

Continuous Integration (CI) is a widely used quality-assurance measure within software development. It empowers developers to spot bugs and integration issues early in the development cycle and helps to maintain a coherent codebase, both in terms of quality and styling, even in open-source environments. But CI might have a hidden cost. Projects need to be built and tested continuously throughout the development cycle. It is not uncommon for projects to have multiple commits per day, reaching thousands of commits per year, with each commit having one or multiple build and test cycles. In this thesis, over 200 open-source Java projects were measured with the aim of making developers more aware of how much energy these builds can take and the measures that can be taken to reduce energy consumption where possible.

Thesis Committee:

Chair:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. P. Pawełczak, Embedded Systems Group

Preface

I want to express immense gratitude to my supervisor Professor Zaidman, who helped me after having spent over two months at an impasse, hopelessly looking for a subject that fit my interest and ability. You guided me throughout this thesis giving valuable feedback at every turn and never putting much pressure which gave me confidence in my own abilities. I'm very proud of the research we were able to do.

I also want to thank Dr. Pawełczak for finding the time to read and judge this thesis. Your evaluation and insights are greatly appreciated.

This thesis marks the end of my journey as a student. I began studying Computer Science in 2017 without having seriously considered any other studies. Honestly, it was an intuitive decision that turned out to be a perfect fit. Now, in 2024, I can finally conclude this chapter and move away from the academic world to begin my working career.

Robert Arntzenius
The Hague, the Netherlands
September 27, 2024

Contents

List of Figures

Chapter 1

Introduction

The software development industry seems ever-growing. We can see this in both the growing number of software developers worldwide [35] and the market growth and projections of the industry as a whole [34]. With this growth and all sorts of industries looking to digitize and electrify their respective processes, the world's collective energy consumption keeps increasing as well [13, 22]. Industries electrifying and digitizing their processes is often a step towards carbon neutrality [38, 26]. However, we should not forget that today's electricity sources are still largely non-renewable

The world's combined energy mix as of 2023 still adds up to only forty percent of our energy mix being low-carbon sources, the other sixty percent being resourced from mostly coal (35.5%) and gas (22.5%) [31]. Depending on where we are in the world, the electricity we use ranges from less than 30 grams of CO_2 equivalents per kilowatt-hour to over 1000g CO2e per kWh with the worlds average being 481g CO2e per kWh according to data from 2023 [31]. This means that we need to think carefully about how we spend this energy because there is a major cost to the energy we generate today and for the foreseeable future.

Together with the technological innovations and growth of the software industry, we have also become more reliant on the quality of this software. The importance of software reliability can hardly be overstated given all major industries including healthcare, travel, and global economics depend on software in one way or another. While this has been known for a long time, we have recently seen how vulnerable this makes us when a bug oversight in a software update of CrowdStrike led to over 4200 flights being canceled, hospitals canceling non-emergency surgeries, and payment systems being affected [28, 30]. This is an example of how vulnerable we are to failure in the software development industry in an extreme case, however, there are of course many ways the software industry successfully prevents these mistakes from reaching the end-user.

In software development, one of the ways software quality is ensured is by continuously building and testing the application throughout the development lifetime of a project [7]. This allows faulty code to be spotted at an early stage before it reaches any release version. While tests can be performed manually, in modern best practice it is most often done automatically on (dedicated) servers where not only tests are performed, but also other types of quality assurance measures, such as static analysis, branch coverage analysis, code-format checking, dependency checks and integration testing. The continuous application of such

a collection of processes we call Continuous Integration (CI) [15]. CI allows developers to continuously integrate small changes into major software systems while maintaining the reliability of these systems. This does come at the cost of having to build and test these large complex software systems typically multiple times a day and for many different versions to be able to perform this quality assurance, which can be very resource-intensive.

So where does this leave the software development industry? On one hand, CI is a safety measure that the software development industry has embraced for years now. It is estimated that over forty percent of open-source projects make use of CI [17, 19]. It allows us to find bugs earlier and helps large codebases to be coherent, both in terms of coding style and quality. On the other hand, CI might also have a hidden cost that comes from its resource-intensive components such as building and testing. What is the cost of this process to our environment and is this a sustainable way to reach the goal of reliable code? Is there a way to lighten the load of CI or is its energy consumption just the cost of the quality it produces?

1.1 Research questions

This section will give an overview of the questions we intend to answer with this research. Our goal is to understand the energy cost that comes with the essential parts of CI: building and testing. We of course want to be able to measure how much energy is being used by software development projects. In order to target a more reasonable and specific scope we decided to only look at open-source Java projects since Java is still one of the most used languages in software development [9]. As there is no such thing as a "typical" Java project the main question we are interested in is:

> **RQ1.** How much energy is used by an open-source Java project during the build and test phases of CI?

This is the question the rest of the thesis and the subjects discussed are shaped around. We cannot answer this question for all Java projects that exist, but this thesis aims to give an overview of a sizable representative subset. Another aspect of CI we want to understand is how energy use is distributed across the runtime of its CI. We typically separate CI into phases that all contribute to the overall quality assurance of the project. For example, you can distinguish between retrieving the repository, building the application[1], and testing the application and describe these as different phases of the CI pipeline. This leads us to our second research question:

> **RQ2.** Which phases of the CI pipeline use more and which use less energy?

Lastly, we want to investigate the parts of a project that make it more or less energy-efficient. We can do this by looking at overlapping characteristics in and resource allocation of projects that consume a lot of energy. We also will be diving deeper into the impact

[1] We will be using the terms *building* and *compiling* interchangeably throughout this research.

that caching has on energy usage and whether this could reduce the energy spent when continuously building and testing these projects in a significant way.

RQ3. Which characteristics of a Java project impact its energy use the most?

RQ4. How does dependency caching affect the runtime and energy usage of large Java projects?

Answering these four questions will provide deeper insight into the scale at which energy is being used during CI, where the energy is used, what characteristics of a project cause it to use this energy, and lastly, whether we can reduce this amount.

1.2 Contributions

The most important contribution this thesis provides is mainly for developers to gain insight into the energy consumption of CI in general. Software developers typically have little knowledge about energy consumption [29]. Chowdhury et al. and Verdecchia et al. both point out that software engineers need to have awareness about and feedback on energy consumption before they can adjust their programming practices and behaviour [10, 36]. This research is especially useful because of its relatively large scale and because the topic of energy consumption in the software development industry is of growing importance. Besides this and providing answers to the research questions, this thesis also produces an automated setup that can be used to replicate the experiment. The code for the experiment can also be adapted to fit the purpose of a wide range of future research [3].

1.3 Thesis overview

This thesis is structured as follows. In Chapter 2 we will review the necessary background on CI and energy measuring, especially regarding CI servers. Following that, Chapter 3 explains the experimental setup designed to answer our research questions. In Chapter 4 we review the experiments' results and make observations on the gathered data. We follow this up with a discussion in Chapter 5, providing a deeper analysis of the results. In Chapter 6 we will briefly go over some work related to our research after which we discuss some threats to validity and limitations in Chapter 7. Finally, in Chapter 8, we draw conclusions and discuss future work.

Chapter 2

Background

To provide proper context on the subject of this thesis, this chapter will give an in-depth overview of Continuous Integration (CI); its current state in modern software development, the benefits it provides to the developer and end-product, and how it became industry standard. We also explain the intricacies of energy measuring and how these apply to our research.

2.1 History of CI

As explained in Chapter 1, in software development software quality is often ensured by continuous thorough testing throughout the development lifetime of a project. This has the main purpose of spotting faulty code at the earliest stage possible before it reaches any release version. These tests can be performed manually, but in modern best practice, it is most often done automatically on dedicated servers together with other types of quality assurance measures, such as static analysis, branch coverage analysis, code-format checking, regression testing, dependency checks, and integration testing are applied as well. The continuous application of such a collection of processes we call Continuous Integration (CI).

CI has existed for a long time in one form or another—for example with the introduction of automatic build tools such as C/C++ Makefiles—but it was in the late 1990s when the book *Extreme programming Explained* was published which introduced CI as part of the software development methodology of Extreme Programming [15, 2]. At the time, pre-release integration was the norm. The software development methodology aimed to improve software quality and responsiveness to changing customer requirements. Quality assurance measures are automated and moved to dedicated machines where the project is built and tested in a clean environment. This makes building the software easier and more repeatable. In the publication, the methodology emphasizes short development cycles, automated testing, and frequent releases [4]. The motivation Beck describes for these elements is that they provide continuous and more specific feedback which allows developers to be aware of and correct failing tests within a very short time frame.

From its origin in Extreme Programming, we see that CI is in many ways bound to Agile practices. This combination works very well because both CI and Agile methodologies are

defined with short release cycles in mind [37]. Short here is of course a relative term and some Agile methods have stricter definitions on this than others. For instance, Extreme Programming explicitly states that the system is integrated and built **many times a day**. Most Agile methods in the right context (where short development cycles are reasonable) can use CI and some, like Scrum, KanBan and of course Extreme Programming almost always use CI in software development [27, 1].

While the core concept of CI has stayed relatively stable, in modern software development the integration cycle does not always stop with CI, but often continues as far as the release phase; the whole process is then called CI/CD where CD stands for either Continuous Delivery or Continuous Deployment. These concepts serve as an extension of CI where the end goal is for the main integration branch to always be in a state that is ready to be released. The key difference is that in Continuous Delivery, this ready-to-be-released state is where the automation ends and in Continuous Deployment the project is automatically released to production in short cycles. These extensions to the CI pipeline obviously fit the philosophies that were already present in the original publication of Extreme Programming.

The growth of CI in popularity also caused the infrastructure to apply CI to a project to be very accessible. There are many options for CI services like *Jenkins*, *TravisCI*, *CircleCI*, and since 2018 even from GitHub with *GitHub Actions* to be deployed on dedicated servers from the likes of Amazon, Microsoft or Google. This allows less experienced developers to set up CI very easily. These CI services use workflow files that describe the steps that are taken in the project's CI. It also allows with only a few lines of code for the CI cycle to be executed in different language versions, on different operating systems, or both [16].

2.2 Benefits of CI

Many developers and software development companies swear to the methodologies that surround CI and this is not without reason. One of the benefits of CI is that complex integration can take a lot of time and it is hard to estimate beforehand how long it will take. Doing integration in continuous small cycles removes this unknown element from a project's development. Small incremental integration is typically very predictable and thus removes many of the risks that can form regarding delivery dates. For many software projects, this predictability can be a great asset.

CI is also useful specifically for open-source projects because these lack a specific team of trusted developers and need to check if the code is up to standard with every integration. Here CI allows for a sometimes loose collection of developers to have a consistent codebase, both in terms of quality and writing style, while also allowing the developers to make small or incremental additions that are immediately integrated into the codebase. This might be why it is observed by Hilton et al. that there is a likely correlation between a project's usage of CI and its popularity on GitHub.

In addition to these advantages, CI also promotes early detection of bugs and integration issues. By automating the testing and integration process, CI ensures that problems are identified and addressed promptly, reducing the chances of major issues arising later in development. This leads to faster feedback loops and more reliable code, ultimately im-

proving overall software quality and reducing the time and effort needed for debugging and troubleshooting.

2.3 Considerations of CI

When CI is used well by a team of developers it can have great benefits as seen in Section 2.2, but there are ways CI can be over-relied-on, wasteful, and ineffective. In contrast to other styles of integration, CI is by definition done in short release cycles and frequent integration into a single main branch. This is not the case for other integration methodologies such as feature branch integration which allows for a wide range of release schedules to adhere to its core concepts. This can be quite wasteful. For one, when a team relies on how responsive their CI is, it can impact the work-time efficiency of development.

However, when we talk about CI being a wasteful process, we are mostly referring to the resources it uses and the cost of those. As we have stated, CI is by definition a process that is continually applied, most often referring to at least once a day, but in its original definition even many times a day. This can lead to a lot of computational power that gets invested in this process. Besides this, there are very few downsides to CI, which leads to a process that has very obvious benefits and a mostly unseen downside.

2.4 Measuring energy consumption

Measuring the energy consumption of software is not as simple as it may sound at first glance. It involves a lot of decisions on what to measure and with what tools. Since software is never a completely isolated instance—there is always your operating system running in the background for example—there is no definitive way that software ought to be measured. Often energy consumption is measured by the use of energy profilers. These are software tools that estimate the power consumption of a system or even specific processes. These profilers rely on estimation models that measure the load of certain system components, most importantly the CPU, and then estimate based on the specific system what energy use the system has and what portion the software is responsible for.

Because of their ease of use, these energy profilers are the most used way of measuring energy consumption in studies about software, but they are not as reliable as hardware power monitors since these profilers either only consider certain components—specifically the CPU usage—or they rely on estimation models [11, 20]. The estimation models have been shown to differ significantly depending on which profiler is used [23], which leads to the conclusion that they can not be relied on exclusively.

We typically distinguish between black-box and white-box measuring [24]. These refer to the insight we have into the energy cost of specific tasks. White-box measurements allow us to see exactly which elements of the process are responsible for how much energy consumption. In black-box measuring you have no idea where the energy is used, only the total. This form of energy consumption measurement can be very useful through a lens of comparison like regression testing as done in *E-Compare: Automated energy regression testing for software applications* by Hagen [18].

In addition to the challenges mentioned, measuring energy consumption in software is further complicated by factors such as workload variations, background processes, and even hardware characteristics. All of these can influence the accuracy of energy measurements. Moreover, software optimization techniques aimed at reducing energy usage might have varying impacts depending on the specific environment in which they are deployed, making it difficult to generalize findings.

2.5 Servers and power consumption

When we want to make claims about total energy usage, we need to consider on which system(s) the software is being run. In the case of CI, this is mostly done on servers, typically either private servers or cloud-based CI servers. In both cases, the servers often run CI services from *GitHub Actions*, *Jenkins*, *TravisCI*, or *CircleCI*. One of the biggest obstacles of measuring the energy consumption of these servers is that we do not know what type of hardware is used. This varies a lot between different servers and pricings listed by these servers. Open-source projects typically use cloud-based servers, the most common options for these are Amazon Web Services, Google Cloud, and Microsoft Azure.

Servers are computer systems designed to allow many different users to perform tasks simultaneously. They are equipped to be extremely powerful to make this possible but consist of the same basic hardware components. These server components are designed for way greater loads and draw way more power. Interestingly, a primary concern while designing servers in many instances is its energy efficiency because this reduces the expenses of keeping the server running. However, while the hardware manufacturers are always trying to optimize their components, when you look at the actual component specifications they typically do not have the most impressive efficiencies compared to some laptop or desktop CPUs.

Servers often use CPUs from the AMD EPYC or Intel Xeon series. These CPUs are very powerful compared to normal CPUs and can reach thread counts of 192. In speed, they can beat any desktop CPU in every task that can be multi-threaded, but with incredible power comes incredible power-draw. It is not uncommon for modern server CPUs to reach a Thermal Design Power (TDP) of 300 Watts or higher[1]. When looking at the TDP per thread, we see efficient laptop and desktop CPUs have similar capabilities and sometimes even beat high-end server CPUs. Some examples are shown in Table 2.5.

There is one critical aspect that makes the comparison even harder to estimate: servers have a lot more overhead in terms of energy consumption. The most important extra power consumption comes from the cooling systems that are needed for these major servers. The measure of how much energy is used effectively is called Power Usage Effectiveness (PUE). For large data centers like those of Amazon, Google, and Microsoft, this PUE value typically lies between 1.1 and 1.5, which means that for every Watt of power that goes to a system component, 0.1–0.5W goes to external factors such as cooling [8].

[1]Thermal Design Power describes the amount of heat measured in Watts that a component can withstand. This is not the same as power-draw, but the two are closely related.

Table 2.1: CPU specifications* and Watts per thread.

CPU	*Class*	*Clock speed* (GHz)	*#threads*	*TDP* (W)	*Watts per thread***
Intel Core i5-13600T	desktop	1.8	20	35	**1.75**
AMD Ryzen 7 6800U	laptop	2.7	16	28	**1.75**
AMD EPYC 7702	server	2.0	128×2	200×2	**1.56**
Intel Xeon Platinum 8570	server	2.1	112×2	350×2	**3.13**
AMD EPYC 9684X	server	2.6	192	400	**2.08**
AMD Ryzen Threadripper PRO 3995WX	server	2.7	128×2	280×2	**2.19**

* Specifications taken from `https://www.cpubenchmark.net`
** Does not account for clock speed. Lower is more energy efficient.

Looking at these aspects of servers and comparing them to those of a smaller PC leads us to believe that an efficient PC, while generally taking more time to execute a task, will use less energy on average for the same task than a server would and will continue this research under this base assumption.

Chapter 3

Experimental setup

To get a well-formed understanding of the current state of CI regarding its energy consumption we need to conduct experiments. These experiments aim to gather energy consumption measurements on a list of software projects during the build phase. This chapter will provide insight into the important decisions made while designing the experimental setup and give an overview of the elements involved in the automated process. The chapter is divided into sections, each reviewing one of these parts in detail. The chapter concludes with Section 3.6 giving an overview of the complete experimental setup.

3.1 Goal of experiments

The experiments should provide reliable energy consumption data on a list of software projects that form a representative sample of open-source software in general. This means we require the projects to be ongoing and use CI to maintain code quality and reliability. Besides a sizable sample of projects, an automated setup for executing and measuring the CI is needed. This experimental setup needs to be as automated as possible and should prioritize repeatability. Each measurement should be repeated five times to ensure the values are consistent and, as is usual in energy measuring experiments, there needs to be a wait time implemented between experiments to ensure this repeatability [12].

While finding these projects' total energy consumption data will provide interesting metrics by themselves, we also set the goal to segment the energy metrics by the individual phases of the CI pipeline where possible. This will allow us to look deeper into which parts of the CI process consume the most and least energy. We will also need an additional alternative experimental setup with enabled caching after a first successful build to see how this influences runtime and energy use.

With these goals in mind, the experiments will allow us to answer three of our research questions:

RQ1. How much energy is used by a Java project during the build and test phases of CI?

RQ2. Which phases of the CI pipeline use more and which use less energy?

RQ4. How does dependency caching affect the runtime and energy usage of large Java projects?

Experiments to answer our third research question will be described later on in Section 5.5.

3.2 Hardware setup

While power consumption in software projects is generally measured with energy profilers because of their ease of use, they are not as reliable as hardware power monitors since these profilers either only consider certain components—specifically the CPU usage—or they rely on estimation models [11, 20], while hardware power monitors have access to the actual total power that a system uses to very high precision. This makes energy profilers very useful for relative measurements like comparing different versions of similar projects to each other. For our purposes however, we also want the absolute measurements to be as accurate to the actual power consumed as possible. For this reason, we decided on a power monitor.

The power monitor we use is the AVHzY CT-3. This is a small power monitoring device, which can throughput power via a USB-C in and output which can be measured on the device itself or you can connect it to a computer via (micro-)USB. The power monitor can be read in two ways, via the proprietary software or by use of its C# library[1]. In terms of accuracy, the device is listed to have a voltage-current resolution accuracy of 0.0001V 0.0001A 0.1%+2d. In practice, this means that the error is about 0.1% for a single measurement for both the current and the voltage alike. As we take more and more measurements, the relative error will shrink as we will see in Section 5.2.

Since we need accurate energy data provided by a power monitor solution and we need isolated measurements, the only feasible way forward is by using a local machine on which the CI will be simulated. For this, we use a mini-PC without an internal battery so we can directly measure the power intake. The specific device we decided on is the MinisForum EM680 [2]. This power-efficient device fits the established criteria and is well-equipped to build and execute most resource-intensive projects, primarily because of its 8-core, 16-thread Ryzen 7 6800U CPU and 16GB RAM. This is the same CPU as was listed in Table 2.5, which we concluded to be very energy-efficient.

To get the most clean isolated measurements possible, we need to provide the system with a lightweight OS with minimal background processes. For this, we decided to use Ubuntu. To make sure the system has all basic necessities for building Java projects, we provided it with four Java versions managed by an installation of *SDKMAN!*, *Python2*, *Python3*, and *Docker*; the specific versions are listed in Table 3.2. We set up a separate user on the system with only the basic permissions but did add it to the Docker group, which allows it to use Docker. These restrictions ensure that we have a lot of control over where to locate any residual cached information that could impact the reliability of the results.

[1] Both can be found on this forum: `https://forum.avhzy.com/forum.php?mod=viewthread&tid=190`

[2] Detailed specifications of the MinisForum EM680 can be found at `https://store.minisforum.com/products/minisforum-em680`

Table 3.1: Manually installed software and versions.

Ubuntu	22.04.4 LTS (GNU/Linux 5.15.0-119-generic x86_64)
Java	8.0.402-tem 11.0.22-tem 17.0.10-tem 21.0.2-tem
SDKMAN!	script: 5.18.2 native: 0.4.6
Docker	26.1.1

3.3 Repository list

Before we describe the CI process we will be applying to our projects, we have to define what projects we will be measuring. Our list of repositories is based on a list of locally buildable Java projects from the results of Khatami and Zaidman [25]. The original set of projects the study used was aimed to be made up of "projects that are likely to be in a position to make optimal use of quality assurance practices," which aligns with our goals of being a representative set of Java projects that use CI for quality assurance. From their original set of 1454 Java projects, they found 202 Gradle projects and 457 Maven projects to be buildable locally.

Gradle and Maven are build tools for Java application development that handle all the building, deployment, and testing of the application and allow these complex processes to be executed with short commands that are independent of the project. This is very useful in our case because it allows for a very simple command pipeline that will work for all applications of a specific build tool. For example, the command `gradle build` will completely build and test an application from scratch. This also allows us to differentiate between the major phases of the CI. While we could make a setup that allows for a single list with both Gradle and Maven projects, we decided to divide them into two separate lists.

This means that we only need to gather very basic information about each of the repositories, some of which we already have from the original projects list. The info we need about each repository for the automated setup to be applied in particular is the repository owner and name, a commit hash of a working version of the application, the build tool (Gradle or Maven), and the required Java version. These will allow for the setup to go through the whole list without interference. From Khatami and Zaidman we already have the repository owner and name. It also provides us with commit hashes and the build tool used. However, to keep the data more relevant, we opted for more recent commits. For this we used a simple Python script that scrapes GitHub for commits and returns the hash of the most recent commit that passes all workflows, this gives us an indication of whether the specific commit works correctly assuming there is a CI workflow present.

Finding the assumed Java version for a project was revealed to be far from trivial. To find this information, we tried building all projects once on the system. Each project was tested on Java 8, the ones that failed then on Java 11, and so on for versions 17 and 21.

While this method is far from perfect, it does result in a list where all projects are tested and do actually build and run the tests correctly on the system in our setup. Our resulting lists consist of a total of 122 Gradle and 82 Maven projects.

3.4 CI setup

A CI pipeline design is often unique to the specific project it belongs to and can consist of many components; typically this includes building the project, unit testing, integration testing, and static analysis. The particular CI configuration of a project is described in a YAML file; a human-readable configuration file that lays out steps that can be taken from a clean containerized environment. Containerization is used most importantly because it allows for parallel, repeatable testing while maintaining flexibility. This means that a server can have many containers running simultaneously, each with its own project, and even have these containers behave like different operating systems within the containerized environment.

These advantages of containerization are important for large-scale servers with little oversight of what kinds of software will be deployed on their system. Our use case is very different however. For example, while for a server it is essential that the system is secure and cannot be breached by any malicious user, the vulnerability of our system is not as much of a concern as it does not have any valuable data on it and we will not be deploying multiple projects on the system simultaneously. It has also been shown that containerization and virtualization lead to increased energy use [32]. While we know that most CI servers do use containerization, we want to make a conservative estimate, so with these considerations in mind, we designed our setup to be run on the system itself without the use of containerization. Some projects may involve containerization in their CI implementation. Because this is essentially part of the project's implementation, we allow these containers. We decided to use the same design for a CI pipeline for each project with the essential components: compiling and testing.

The CI pipeline uses a simple bash script which uses the build tools in three steps: first, it clones the repository, then it compiles the application and lastly, it runs tests. Cloning a repository is trivial and as mentioned previously the build tools Maven and Gradle need only single commands for compiling and testing respectively. The specific commands for Gradle are `./gradlew assemble` to compile and `./gradlew check` to run tests, for Maven these are `./mvnw compile` and `./mvnw test` respectively. These steps make for a short bash script of only a few lines per step. This bash script needs only slight alteration based on the information of the specific repository detailed in Section 3.3. The script is therefore generated with these details filled in by a Python script.

Lastly, to make sure each experiment starts from scratch, a cleanup script was written. The goal of this script is to remove all caching files and directories; notable caching locations are the project directory, Docker containers, and the caching directories located in the home directory: `.cache`, `.gradle` (for Gradle builds) and `.m2` (for Maven builds). To achieve this goal, the cleanup script, `cleanup.sh`, removes all files and folders from the home directory except for the following: itself, the python script mentioned above, the hid-

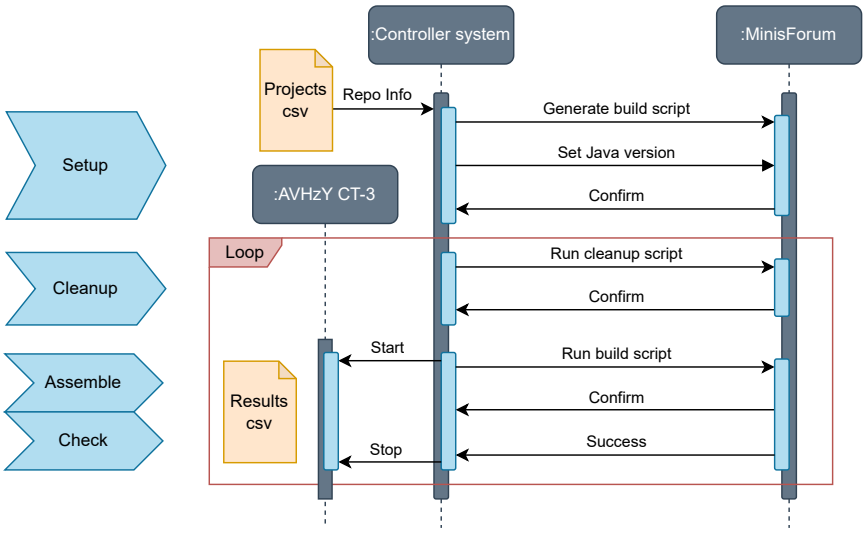

Figure 3.1: Communication sequence diagram of the experiment for one project.

den directories `.sdkman` and `.ssh`, the session-specific `.profile` file and finally any files with 'bash' in the name like `.bashrc` and `.bash_history`. It also specifically cleans up Docker so no residual files are kept to guarantee all experiments start on an equal footing; this is the only part of the system that gets cleaned outside of the home directory since the user has no elevated permissions aside from being in the docker group. In Section 3.7 we will explain our alternate setup that allows caching.

3.5 Communication

Measuring energy consumption with a power monitor device requires a system that controls this device and reads it out. This can of course be done on the system we are measuring but it is not part of the CI process and does draw power and system resources that can otherwise be used by the CI. This is why we use a different system to communicate with the power monitor. This system has to know which repository is being run, when it starts and finishes, and the success status of the experiments to guide measurements. Because of this, we decided to make this system the controlling force behind the experiments. It sends out commands to start the CI pipeline for a specific project and reads out the power monitor while waiting for the process to finish.

This communication between these systems is via SSH. Sending bash commands via SSH can be done in two ways: singular separate commands or a continuous shellstream. Since we need to set up the environment with the correct Java version and have continuous feedback of the active CI phase we opted for a shellstream implementation. A communication channel is kept between systems through which commands are sent and messages are received. Instead of relying on the predictability of the CI output, we prepared a messag-

ing system for the controller to have up-to-date info on the current state of the CI system. These messages are three special character strings of length six that mean *success*, *failed*, and *confirm* respectively.

As seen in Figure 3.1, constant communication between the systems is kept; messages and commands between systems are represented by arrows. The controller always waits for confirmation before moving on to the next step. As you can see in the diagram, the controller system can send (complex) commands to the MinisForum—like the command for generating the build script with the relevant information about the repository—and the MinisForum then responds via the shellstream's output with specific code words. The diagram also shows the clear connection between the phases of the experiment and the messages that start, end, and separate these phases.

3.6 Experiments overview

This section will provide a comprehensive overview of the completed experimental setup. There are three main hardware devices to consider:

MinisForum A mini-PC on which CI will be simulated on a lightweight system. To minimize irrelevant power consumption, no peripherals are connected to the system. It is only connected to power. It does, however, have a wireless network connection available.

Power monitor A dedicated USB power monitor device called AVHzY CT-3, which measures power consumption with great accuracy.

Controller system A desktop system that sets up the experiments on the MinisForum via SSH and reads out the power monitor.

The final setup looks something like Figure 3.2. Here we can see the three main hardware components and how they are connected. The controller system is directly connected to the power grid and the MinisForum is only connected through the AVHzY CT-3. From the Controller system, communication with the power monitor is set up to be read out once every millisecond and an SSH shellstream is created to communicate with the MinisForum. We continue by setting up the message strings for *success*, *failed*, and *confirm* as global variables on the MinisForum. From here, we take a repository from our list, which provides all relevant information to set up and start the CI.

The sequence for a single repository is as follows. On the controller system, we create a file to write our data to. This file is named `<owner>-<repo><iteration>` so, for example, the name `googlecloudplatform-spring-cloud-gcp4` would be the fourth experiment for the project *spring cloud gcp* from *googlecloudplatform*. Via SSH we call our Python script with the repository owner, name, and commit hash, which generates a `buildscript.sh`. Also, we make sure we can execute this bash script by modifying its permission flags. We further set up the environment on the MinisForum by setting the current Java version to the required version. We want to start with a clean system without file caching, so we run our cleanup script, after which the MinisForum sends a *confirm* message

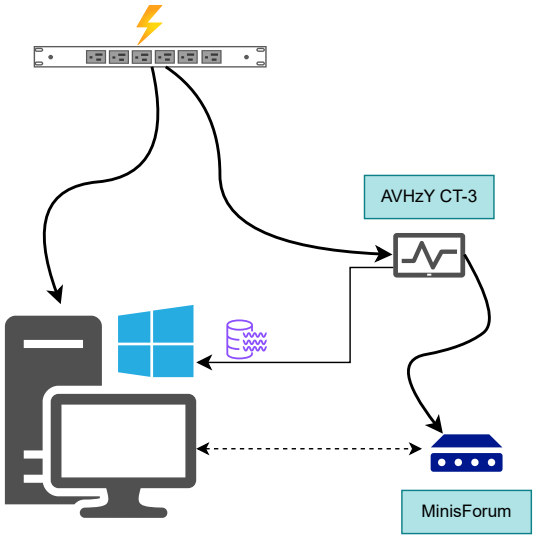

Figure 3.2: Hardware setup schematic.

to let us know the setup is complete. We synchronize the systems by waiting for the confirmation and then we let the system sleep for 20 minutes to make sure it is idle; this is a lot longer than the wait time suggested by Cruz, but since there is no golden rule we decided to keep it on the safe side. From here, we call the start the measure and call `buildscript.sh`.

On the MinisForum, the build script immediately starts cloning the repository and when finished, moves into that directory and checks out the specified commit. From here, it looks for wrappers of both Gradle and Maven. The wrappers are called `gradlew` and `mvnw` respectively. For Gradle the CI is separated into the commands `./gradlew assemble` and `./gradlew check`. Similarly, for Maven the commands are `./mvnw compile` and `./mvnw test`. After the first of the two commands has finished, the exit status is checked. On success, a 0 is returned, in this case, the script sends a *confirm* message back to the controlling system. That notifies the controlling system that the CI will move to the next phase. This second phase moves very similar to the first but ends in a *success* message instead of *confirm*.

When the success message is received, the measurements stop and the file is closed. All repositories are tested five times, so when it finishes it either does another run without changes, starting from the cleanup script, or we set up the system for the next repository. If for any reason the CI fails at any phase, a *failed* message is sent. Every time this happens, the setup allows for one more try to make sure it is not just a fluke. If two fails happen consecutively, the experiment continues with the next repository. In the end, this should result in reliable energy measurements of a complete list of repositories, each with five successful CI runs with separation between the compiling and testing phases.

3.7 Alternate setup with caching

The experimental setup thus far has been to answer our first two research questions, but we need to slightly alter our setup to answer our fourth research question:

> **RQ4.** How does dependency caching affect the runtime and energy usage of large Java projects?

As discussed in Section 3.4, in our main experimental setup we remove all caching by basically emptying our home directory and clearing our Docker installation to make every build start from scratch. However, modern CI servers often do allow caching to some degree on their servers. The main caching that is performed is called dependency caching. This allows build tools to store all kinds of caching data such as dependencies, build artifacts and task outputs to be used for later executions.

Gradle distinguishes between three types of caches: project-specific build caches, user-specific build caches, and remote build caches. Remote build caches are used on CI servers to allow cached results to be shared between multiple developers, CI servers, or build environments. Project and user-specific build caches are stored in sub-directories named `.gradle` found in separate locations on a system, the former being located in your project's root directory, the ladder in the user's home directory.

For our experiments we will specifically not remove the caches that are stored in the home directory, so we will not remove the `~/.gradle` and `~/.m2` directories. This will allow Gradle and Maven to cache data for projects to hopefully make builds faster and thus less energy-intensive.

Chapter 4

Results

Over the span of about three months, the experiments described in Chapter 3 were performed. This resulted in energy-measurement data describing over 200 Java projects. Because this is such a large number of repositories to describe in two graphs, we decided to split both the Gradle and Maven results into multiple graphs ordered by their energy use because it allows for more detailed graphs on each repository. This chapter will go over the results of these experiments and make observations on the data that was gathered.

The results will be distilled from over 25GB of data. So before we take a look at the big picture results we can look at some examples of data from a single project. Figure 4.1 shows the results of one experimental run of both the assemble and test phase of Picard from Broadinstitute[1]. The figure shows the current in amperes (A) and voltage in volts (V) throughout the CI runtime. From these, we can find all metrics of interest. We can combine the voltage and current to get the power draw at any time t, and with the power over time, we can calculate the energy used over a span of time Δt either in joules (J) or (Wh). To have some consistency, we will mostly use Wh and kWh when quantifying energy, because these units are closely related to how we measure and see energy in real-world applications.

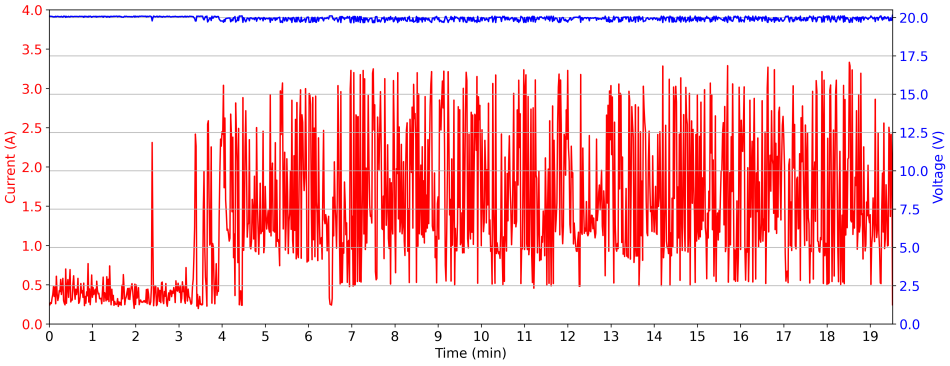

Figure 4.1: Example of Gradle experiment down-scaled 1000 times.

[1] "Picard is a set of command line tools for manipulating high-throughput sequencing (HTS) data and formats such as SAM/BAM/CRAM and VCF." URL `https://broadinstitute.github.io/picard/`

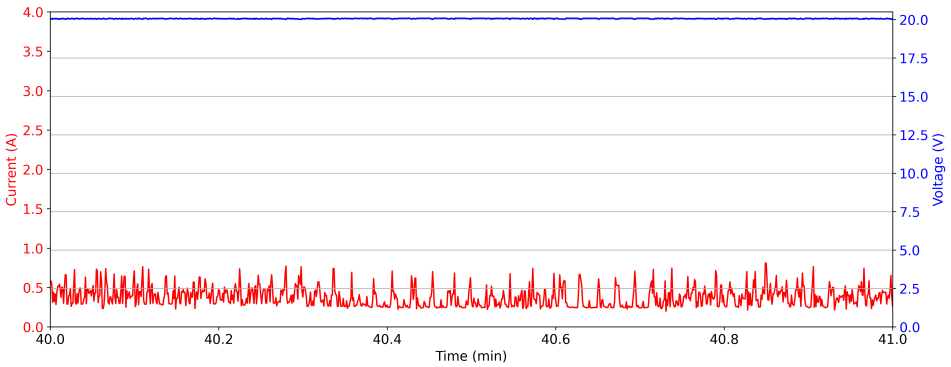

Figure 4.2: Example of Gradle experiment zoomed-in to single second (low intensity).

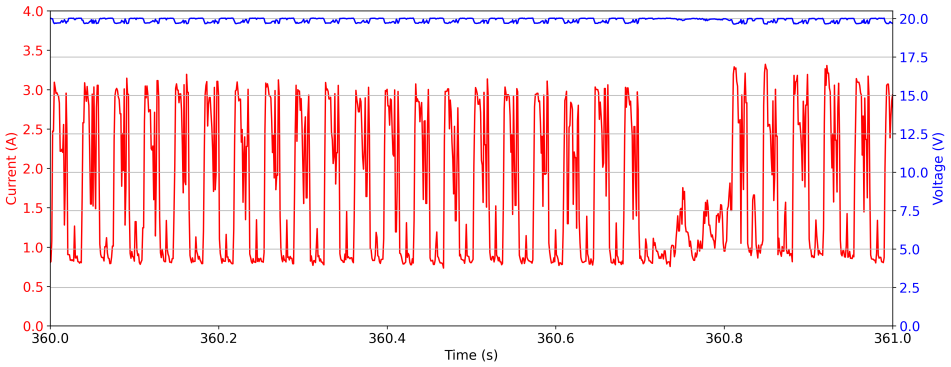

Figure 4.3: Example of Maven experiment zoomed-in to single second (high intensity).

As you can see clearly in the graph, the voltage over the entire runtime is stable at 20V, this is true for all measurements[2]. The current is everything but stable, rapidly moving up and down between about 0.25A and 3.5A. Note that because of the frequency of measuring, the data is down-scaled in the graph to make it more readable, taking only one data point per second instead of millisecond.

When we remove the down-scaling and zoom in on a single second in Figure 4.2 and Figure 4.3 we can see that even at millisecond precision, the current is constantly alternating[3]. We also see that the device can shift between low and high currents many times within a second, this makes high-frequency readings all the more important for accurate readings.

While these figures show what kind of data and precision we are working with, they are hardly readable and do not communicate the aspects we are most interested in. Therefore most of the figures following in this chapter will zoom into the total energy readings we can calculate from the data.

[2]The small dips you can see are from short bursts of high power draw where the current has large spikes as seen in Figure 4.3. Here the current approaches the threshold where the power delivery would be 65W. This threshold lies somewhere around $\frac{65W}{20V} = 3.25A$

[3]The Maven project in Figure 4.3 is JavaParser URL `https://javaparser.org/`

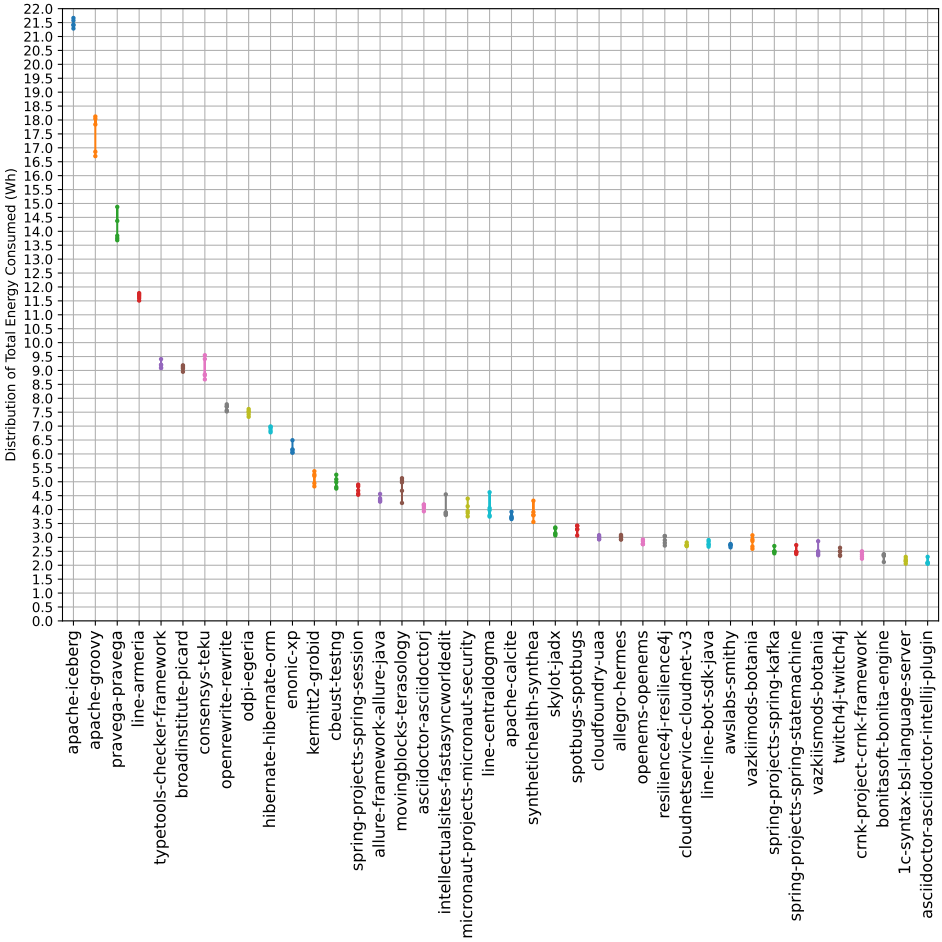

Figure 4.4: Total energy consumption from the upper third of Gradle projects ordered by energy intensity.

4.1 Energy consumption of Gradle Projects

To start, we look at the total energy spent for the complete CI build phases of the list of Gradle repositories. Because of the sheer number of repositories in our dataset, we decided to divide the repositories into three groups; this allows for more detailed graphs on each repository. Each plot considers one-third of the repositories ordered by the lowest data point of the project.

The highest consuming projects are shown in Figure 4.4, these projects gradually span from 2Wh to about 9.5Wh with four projects reaching higher values that have larger gaps between with a maximum of about 21.5Wh. The ranges between the measurements of most projects are very small ($< \frac{1}{2}$Wh) with some exceptions that span about one watt-hour and the two highest-consuming builds spanning about one and a half watt-hours. When we look further down the list in Figure 4.5 we can see this general consistency of measurements

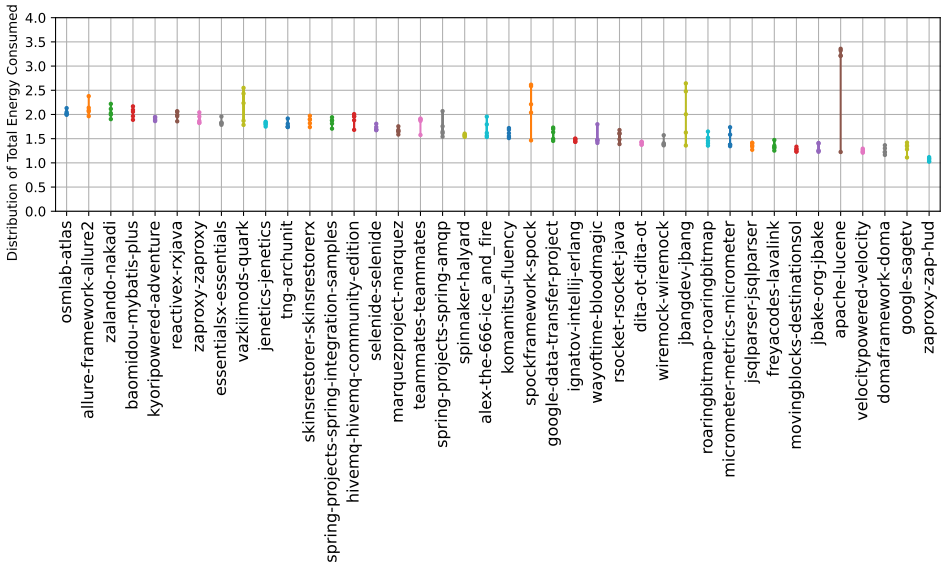

Figure 4.5: Total energy consumption from the middle third of Gradle projects ordered by energy intensity.

stays, but again with some exceptions: three builds have a range of about 1Wh and there is Apache Lucene with a span of over two watt-hours, which is larger than its lowest total.

Unfortunately, we do not know exactly why Lucene has this wide span. One thing that is interesting to note is that the measurements are also distributed in a peculiar way. There are four of the five measurements which are very close to each other (within a span of 0.2Wh) with the fifth being about 2Wh lower. The separation between these measurements and the large gap without any data points suggests that there might be a specific part of the CI that got skipped or delayed in either of the groups of data points. When looking into the data we can at least say that the disparity occurred during the test phase. We see that the first run of Lucene spent about 20 seconds in its testing phase, while the other three spent around 210 seconds. Also, we see that this extra time was not spent idle considering the four higher runs did consistently use around 40 Watts. This fits our hypothesis that some part of the testing phase was skipped rather than the other four runs experiencing a delay.

4.2 Energy consumption of Maven Projects

Now we can look at the results of the Maven experiment and make some comparative observations. When we look at Figure 4.7 and Figure 4.8 we also see that the general distribution of project averages looks about the same as the one we saw for the Gradle projects with the overwhelming majority of the projects having an average energy consumption of less than 5Wh and about half of the projects being around or below 2Wh. We also see that projects only rarely reach an energy consumption above 10Wh just like we observed in our Gradle experiments.

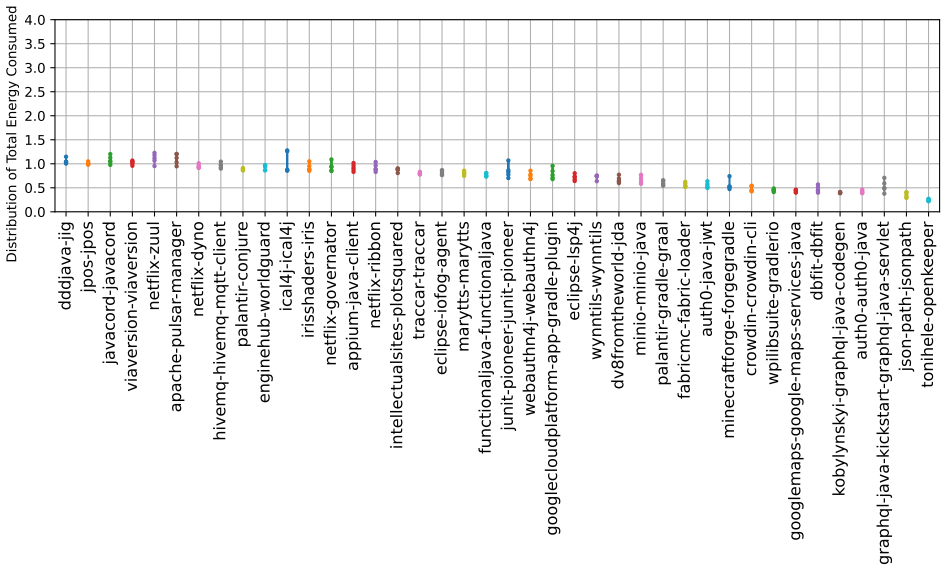

Figure 4.6: Total energy consumption from the lower third of Gradle projects ordered by energy intensity.

Another obvious thing we observe when we look at the figures is that the ranges between the data points of individual projects are a lot wider than with the Gradle data. This is not only for the extreme cases such as Eclipse Lemminx seen in Figure 4.8, but almost all projects have a noticeably wider range with only a few exceptions like DiscordRSV or MultiVerse-core. This inconsistency is not expected, especially since our experimental setup has only been changed to fit our Maven commands—using `mvnw compile` and `mvnw test` instead of `gradlew assemble` and `gradlew check`—and nothing else. In Section 5.3 we will try to uncover the reason why Maven behaves this differently between measurements.

4.3 Energy-time correlation

One interesting way to look at the data is to look at how the runtime of the CI is related to the energy consumption. Even though the amount of energy per unit of time fluctuates with the current, the total energy consumed and the runtime are obviously related since a longer runtime allows the CI more time to spend energy. Figure 4.9 and Figure 4.10 show scatterplots of how the runtime and energy consumption correlate in our Gradle and Maven experiments respectively. In these plots each measurement is represented by a dot on the graph, meaning there are five dots corresponding to a Gradle or Maven project respectively. We can read the runtime in minutes, the total energy in Watt-hours, and by combining those we can get the average power consumption in Watts. For example in Figure 4.9 we can see two measurements very close to [15,2] which corresponds to a total energy consumption of 2Wh, a 15 minute runtime, and using $P = \frac{E}{t}$ we find the average power to be: $\frac{2\text{Wh}}{\frac{15}{60}\text{h}} = 8\text{W}$.

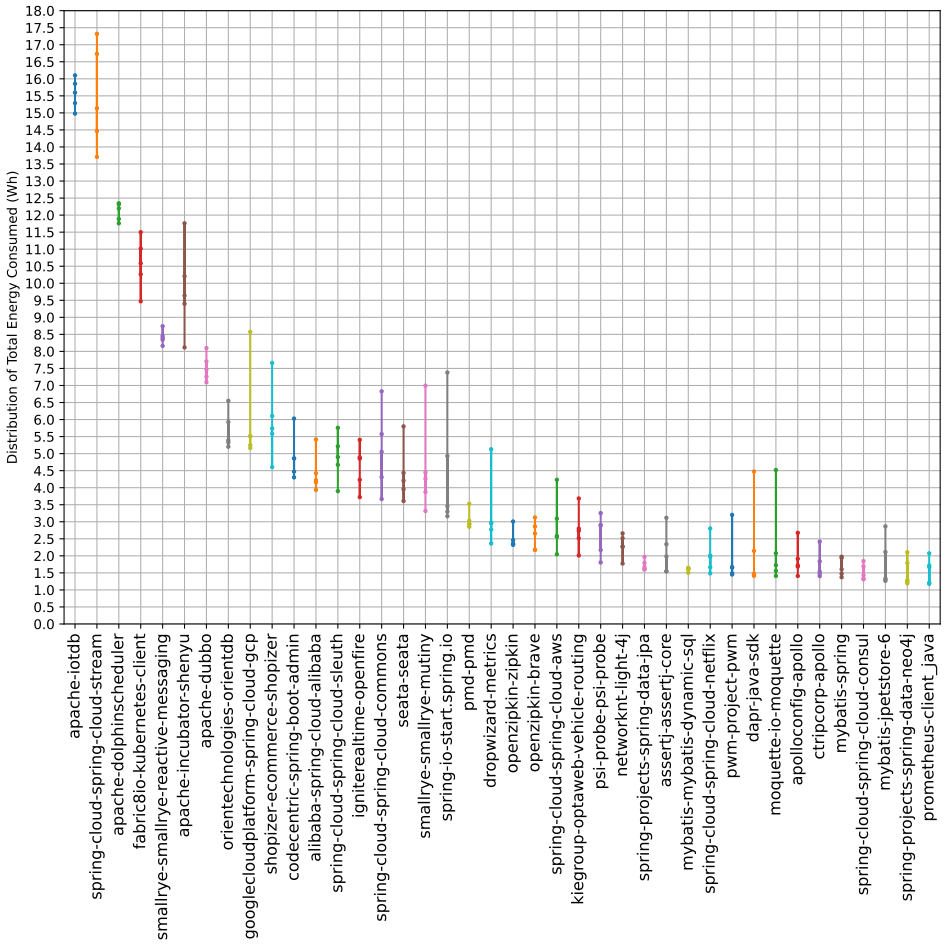

Figure 4.7: Total energy consumption from the upper half of Maven projects ordered by energy intensity.

One thing we find in both plots is that there are some natural restrictions that bound the ratio between energy and time. These bounds fit the power ratings of the hardware. The MinisForum has an idle state that uses a stable 5W, this can be seen as the hard lower bound. The upper bound is similarly decided by the limitations of the MinisForum, the processor's Thermal Design Power (TDP) is rated at $28W$, which means that its cooling can support up to 28 watts of heat produced by the system (primarily the CPU). While the TDP of the CPU gives us a measure, the value is not directly related to power-draw so it forms a softer restriction that serves as a barometer for how much power the device can withstand; the actual power draw that would produce this heat will in reality be higher than $28W$. Besides this, the TDP specifically considers the thermal capabilities of the processor, not the entire system, so the power draw of other components could also be a factor. The power adapter of the device can provide up to 65 watts of power, so that would be the actual hard limiting bound.

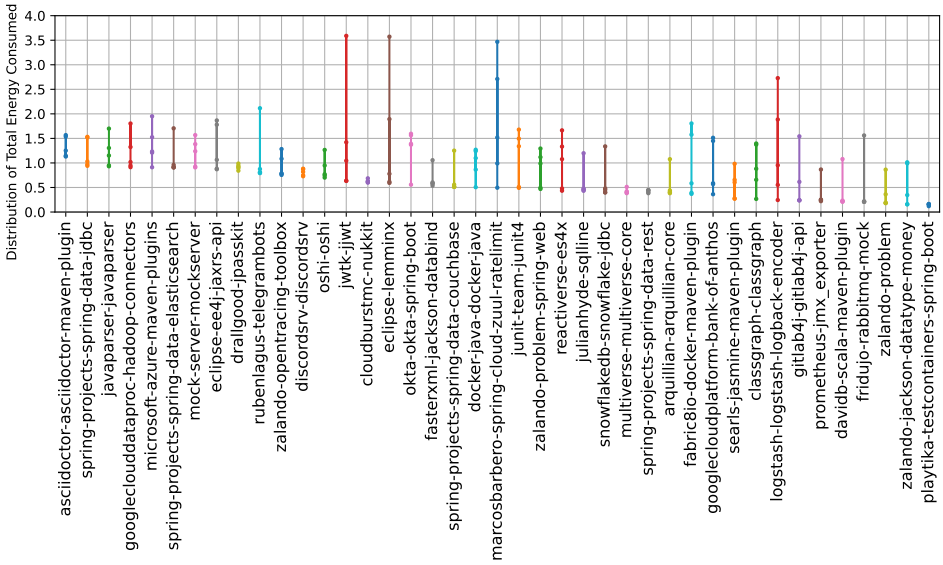

Figure 4.8: Total energy consumption from the lower half of Maven projects ordered by energy intensity.

We can see in the Gradle experiments that the upper bound is sometimes overshot, with most data points that do surpass it, staying close to the 28W bound. The five data points that form a group relatively far above the bound with a runtime between 25 and 30 minutes are the data points of Apache Groovy; they form the strongest exception with an average power of 36.91W. This suggests that Apache Groovy, over its approximate half-hour runtime, spends most time on intensive tasks like running intensive tests in parallel and only a short time on less intensive tasks such as retrieving dependencies and waiting periods. We will examine the highest consuming projects including Apache Groovy in more detail in Section 5.5.

In Figure 4.9 we find a relatively even spread between an average power draw of 8W and 30W with some exceptions reaching a slightly higher average power draw of up to about 37W. We also see that the majority of the data points—469 out of 620 (75.6%)—have their entire CI performed within a ten-minute runtime. About half of the remaining measurements fall between a runtime of 10 and 15 minutes (78 data points; 12.6%) and the other half of those have a runtime longer than 15 minutes (73 data points; 11.8%).

With Maven things look very different. In Figure 4.10 we can see the same graph for Maven[4]. The measurements all seem to hug the lower bound. This means that these measurements of projects spent a lot of their runtime idle. When we consider the runtime output, we know that time is spent retrieving and waiting for dependencies. The data points also seem to be less clearly clustered than we saw for the Gradle projects, which is expected looking at the spread we saw in Figure 4.7 and Figure 4.8.

[4]Note that we measured less Maven than Gradle projects and thus the graph is less dense.

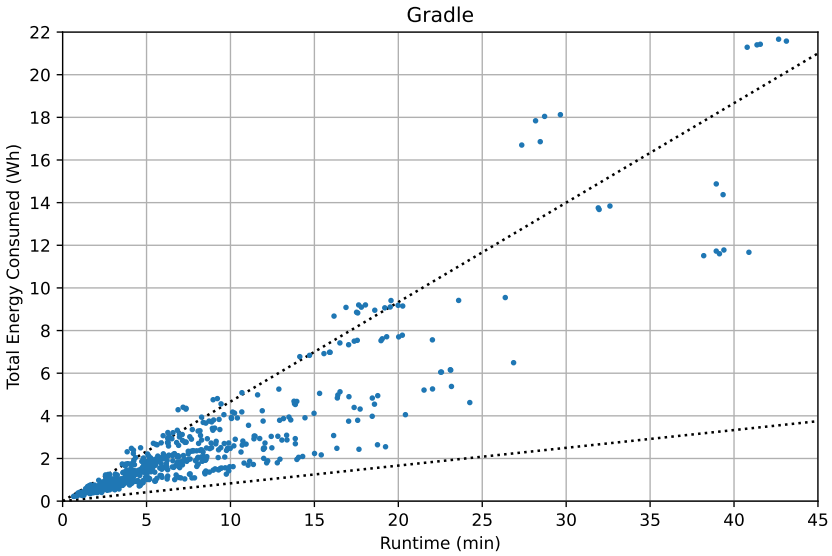

Figure 4.9: Scatter-plot of energy-runtime correlation for Gradle projects; dotted lines represent minimum power draw (low) and TDP (high) of the system respectively.

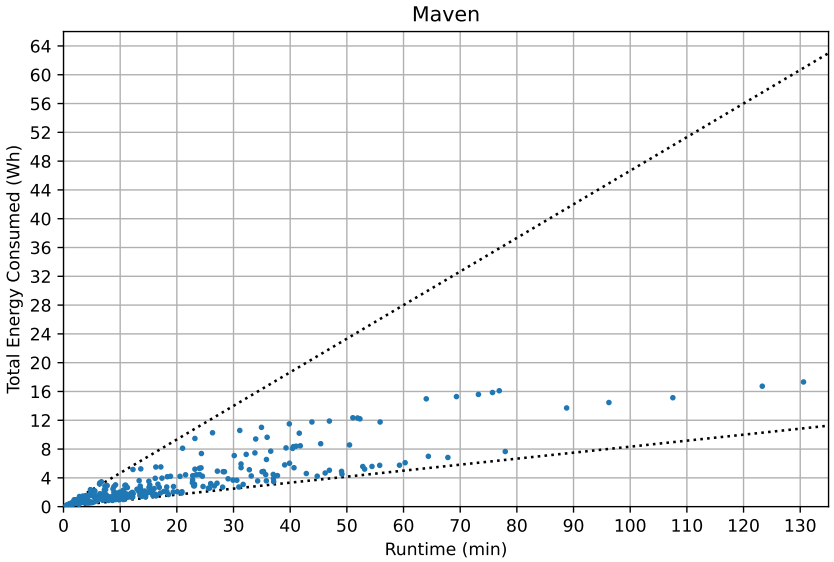

Figure 4.10: Scatter-plot of energy-runtime correlation for Maven projects; dotted lines represent minimum power draw (low) and TDP (high) of the system respectively.

4.4 Phase proportions

For this research, we were not only interested in the total energy use, but we also wanted to see how the energy usage is divided between the CI phases. We separated the data between the two phases we performed: compiling and testing. We can now see for each project what proportion of energy and time is spent in which phase. Every project is represented by a dot that indicates the average relative time spent compiling (area under the dot) or testing (area above the dot). For example *PlotSquared*, *WorldGuard* and *Twitch4j* all spend just shy of 90% of their total energy while compiling and only a bit over 10% of their energy testing. The entirety of Gradle projects can be seen in Figure 4.11 and the Maven projects in Figure 4.12.

One obvious observation we make when we look at both figures is that the distribution is quite even. While we can see there are more and less dense areas along the line, there are data points almost everywhere in between 9–99% and 8–96% for the Gradle and Maven projects respectively. The ordering of the data also closely follows the total one seen in the energy consumption graphs from Figures 4.4–4.8. This close alignment between the graphs tells us that the majority of the extra energy that is spent by larger projects is used during the testing phase, because of how the proportion of energy between these phases shifts.

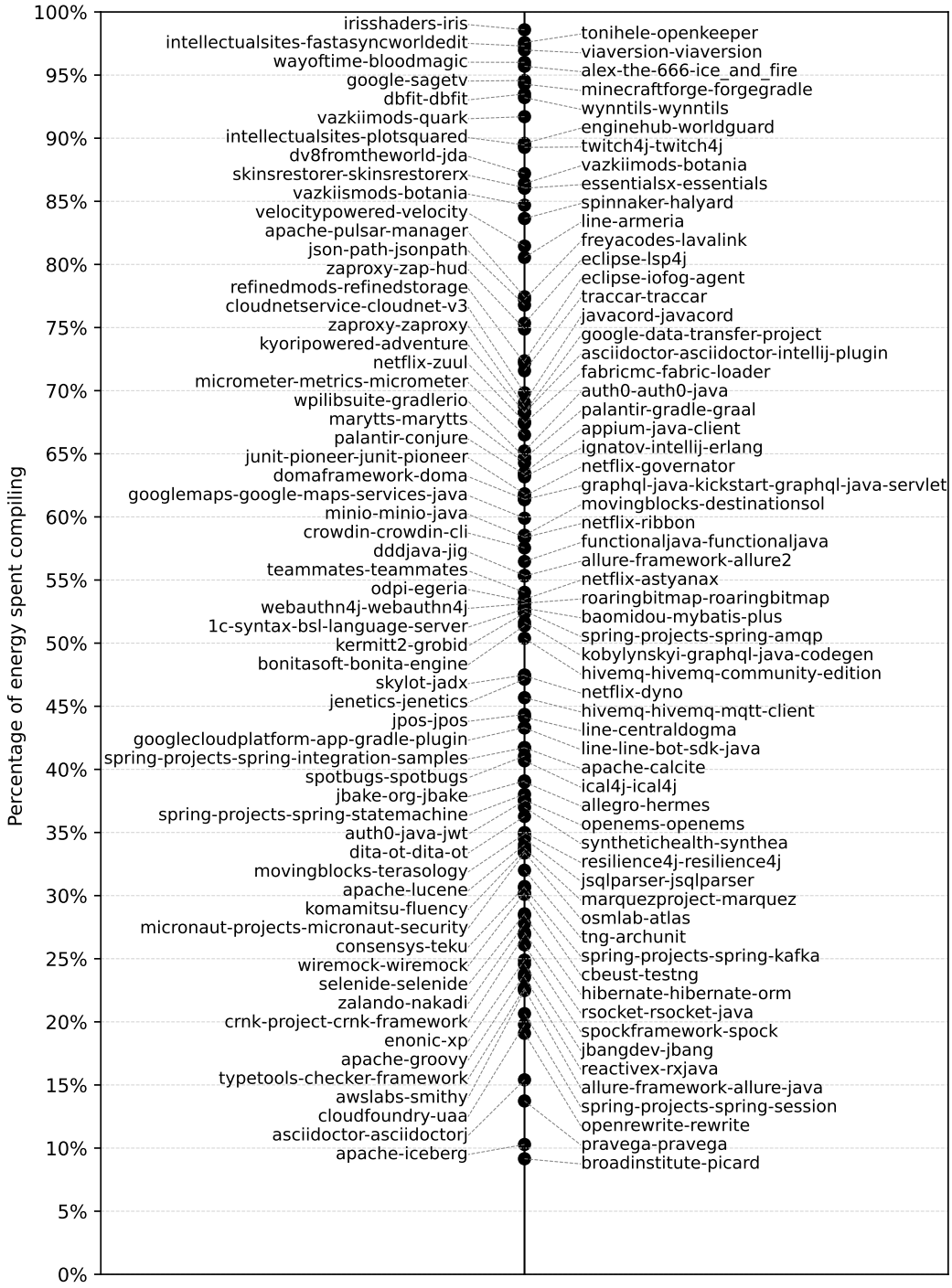

Figure 4.11: Percentages of energy spent compiling during CI for Gradle projects.

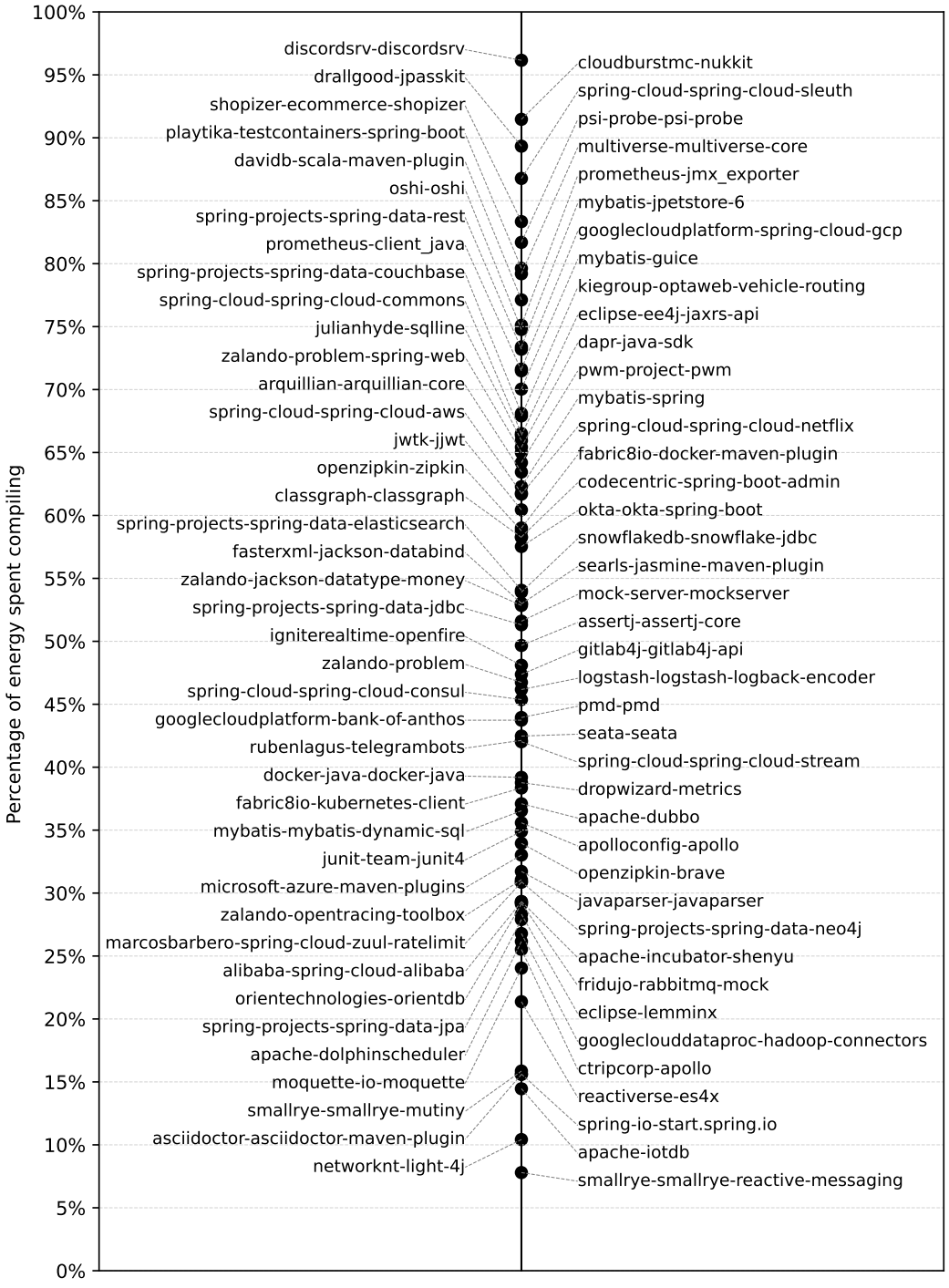

Figure 4.12: Percentages of energy spent compiling during CI for Maven projects.

Chapter 5

Discussion

Now that we have presented the results and made some observations, this chapter will continue with discussing the results to try and provide some deeper insight and find answers to our research questions. We will start discussing the consistency and reliability of our results. We will also provide a thorough analysis on the accuracy on the power monitor we used in our experiments. We continue by discussing more of our results and making estimations based on the data we gathered. We will conclude by analyzing the projects with our highest energy-readings and looking into the influences of dependency-caching on these projects.

5.1 Reliability Gradle results

Before we can interpret the data, we first need to ask whether the data shows signs of reliability. The most indicative element of the data is how much spread there is between data points of the projects. As we have seen in Figures 4.4–4.6, the data in general shows little spread between the different measurements implying the data is reproducible. The spread of data points of projects in general ranges between < 0.1Wh and 0.5Wh for projects with a relatively small runtime, with longer projects mostly ranging between < 0.1Wh and 1Wh; there are some exceptions to this however. The most pronounced exception is the Apache Lucene project (seen in Figure 4.5) spanning between 1.2Wh and 3.4Wh.

Another aspect of the data we can look at to find out whether our results are reliable is the shape of the distribution for individual projects. When performing energy measurements in a well-automated and consistent setup, we expect the data to be normally distributed [12]. However, because we only have five measurements per project we cannot say anything conclusive about the complete shape of these distributions. To gain some insight we can normalize the data and combine it to see if that approximates a Gaussian distribution. Please note that since we are working with very small groups of data, we *cannot* draw strong conclusions about the actual distributions of our experiment. This is because small sample sizes can lead to high variability in estimates of distribution characteristics like the mean and standard deviation. This test *can* be helpful for finding patterns in our standard deviations across all projects.

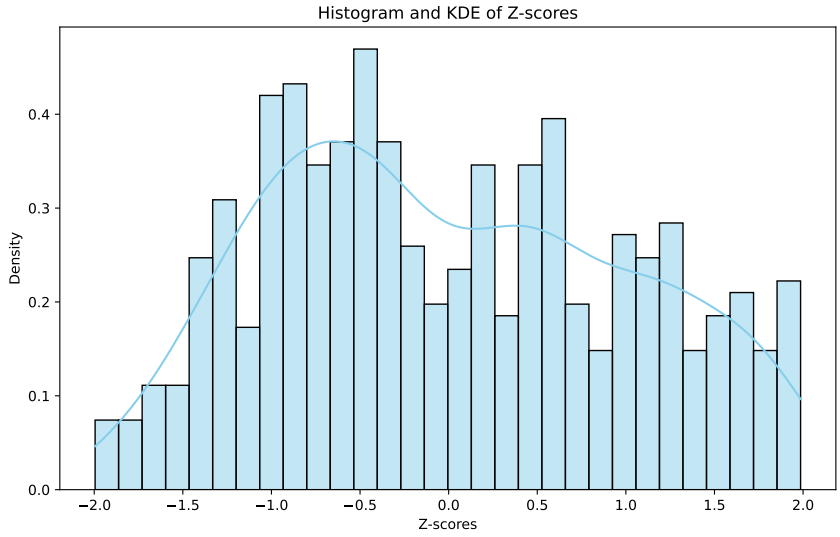

Figure 5.1: Z-score distribution of combined normalized data.

To standardize the data for their respective project we use Equation 5.1.

$$Z = \frac{x - \mu}{\sigma} \tag{5.1}$$

Where:

- Z denotes the Z-score.
- x denotes the actual value of the data point.
- μ denotes the mean of the five values from the respective project.
- σ denotes the standard deviation of the five values from the respective project.

When we apply this to all the data and combine it into one graph we hope it fits a standard normal distribution with a mean of 0 and a standard deviation of 1. When we look at Figure5.1 we can see that it is not an exact fit. If we apply the Shapiro-Wilk test statistic we learn that it does not adequately fit a normal distribution with a p-value of 4.98e$-$10 where it has to be above 0.05 [33]. While we can say that the distribution does not resemble a Gaussian, we can definitely see some of the main characteristics of a Gaussian. For one, all of the Z-scores fall between -2 and 2, which does follow a standard normal distribution where approximately 95% of all data points lie in this interval. We also find that even though the most observed Z-scores lie around -0.5, the mean and standard deviation are the same as a normal distribution with $\mu = 2.18\text{e}{-16}$, $\sigma = 1.0$.

In Section 3.4, we described the way we try to eliminate caching from our experimental setup. We can see in the consistency of the results, that this indeed has worked. This is further supported by the fact that the chronological order of measurements of the same project does not seem to have any correlation to the relative energy consumption of that measurement. With caching, you would expect the energy consumption and runtime of a project to be lower for the later projects, because they would have the benefit of the cached information to reduce the workload, but we do not see any such pattern in the measurements.

5.2 Hardware accuracy

In Section 3.2 we briefly mentioned the precision specification of the power monitor we used. The AVHzY $-$ CT3 has a voltage current resolution accuracy of 0.0001V 0.0001A $0.1\% + 2d$. Now that we have the data we can estimate the error from this specification. This estimate will only consider the device and not any other potential measuring errors like timing or environmental influence. When we assume the error is independent between measurements we can apply the propagation of uncertainty formula for addition to our total summed energy readings as seen in Equation 5.2.

$$\varepsilon_E = \sqrt{\sum_{i=0}^{N} (\varepsilon_i^P \cdot \Delta t_i)^2} \tag{5.2}$$

With:

ε_E	denoting the absolute energy error of all data points combined.
N	denoting the total number of data points.
ε_i^P	denoting the absolute power error for data point i.
Δt_i	denoting the duration of time interval i.

$$\varepsilon_P = \sqrt{(I \cdot \varepsilon_V)^2 + (V \cdot \varepsilon_I)^2} \tag{5.3}$$

With:

ε_P	denoting the absolute power error.
ε_I	denoting the absolute current error.
ε_V	denoting the absolute voltage error.
I	denoting the actual measured current.
V	denoting the actual measured voltage.

To avoid calculating the exact error for all projects, we decided to use these formulas to estimate an upper bound of this error. For this, we first need to know the range of each of the variables. We know from our data that the voltage is very consistent at 20V, so we can fill in $V = 20$ and $\varepsilon_V = 0.0202$. Our data shows the current typically ranges between 0.25A and 3.5A.

$$\varepsilon_{\text{MIN}}^P = \sqrt{(0.25 \cdot 0.0202)^2 + (20 \cdot 0.00045)^2} \approx 0.01\text{W}$$

$$\varepsilon_{\text{MAX}}^P = \sqrt{(3.5 \cdot 0.0202)^2 + (20 \cdot 0.0037)^2} \approx 0.10\text{W}$$

Filling this in we find that the power error ranges between approximately 0.01W and 0.1W. From this, we can find a bound function in terms of time. Since we know our Δt should be a constant 1ms, the only unknown is the runtime duration, which is directly represented by the number of data points N. This gives us an adapted version of Equation 5.2.

$$\varepsilon_E = \sqrt{N \cdot (\varepsilon_P \cdot \Delta t)^2} \tag{5.4}$$

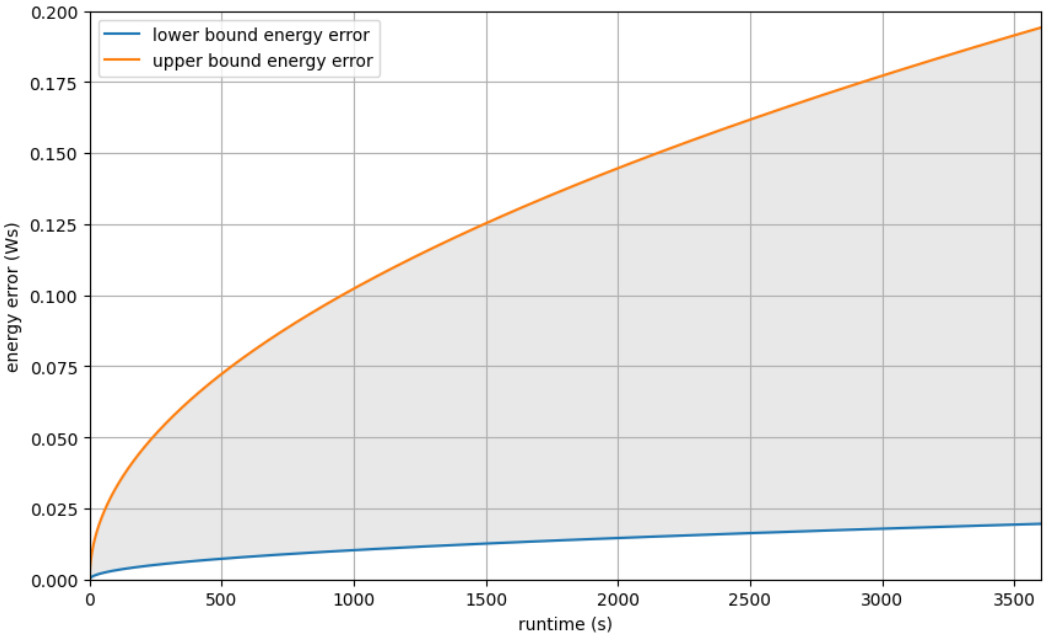

Figure 5.2: Device measuring error bounds for different runtimes.

Equation 5.4 filled in for the upper and lower bounds of ε_P gives us our error bounds shown in Figure 5.2. From this graph, it is immediately clear that the error of both projects with a short and long runtime has an error which is insignificant (< 1Ws)[1]. We also see the growth of both bounds is logarithmic, which means the error grows significantly slower than the runtime and the relative error will only shrink as this runtime increases.

5.3 Maven results

As we observed in Section 4.2, our Maven experiments lead to significantly less consistent results than their Gradle counterparts. When observing the experiment output in real-time, this was already apparent; sometimes a project would stop to a halt and have a delay of minutes when downloading Maven packages, sometimes even resulting in failed builds due to timeouts as seen in Figure 5.3. Even though this meant the results would be less consistent we decided to continue the experiment to gather more data on this inconsistency.

To further investigate the cause of the inconsistencies in the Maven results, we designed and performed an additional experiment on a subset of the dataset. We wanted to know whether the dependency retrieval phase of the CI build shows evidence of being the cause of the inconsistencies in the measurements. The only change in this experiment was that this time we would build a project beforehand and not remove the `.m2` directory containing

[1] The unit Ws is equivalent to J, which is exactly $\frac{1}{3600}$Wh

Figure 5.3: Screenshot of recurring error during Maven experiments.

maven cache files and dependencies. This would ensure the build has no dependencies to fetch, which we hypothesized to cause the delay. The results can be seen in Figure 5.4

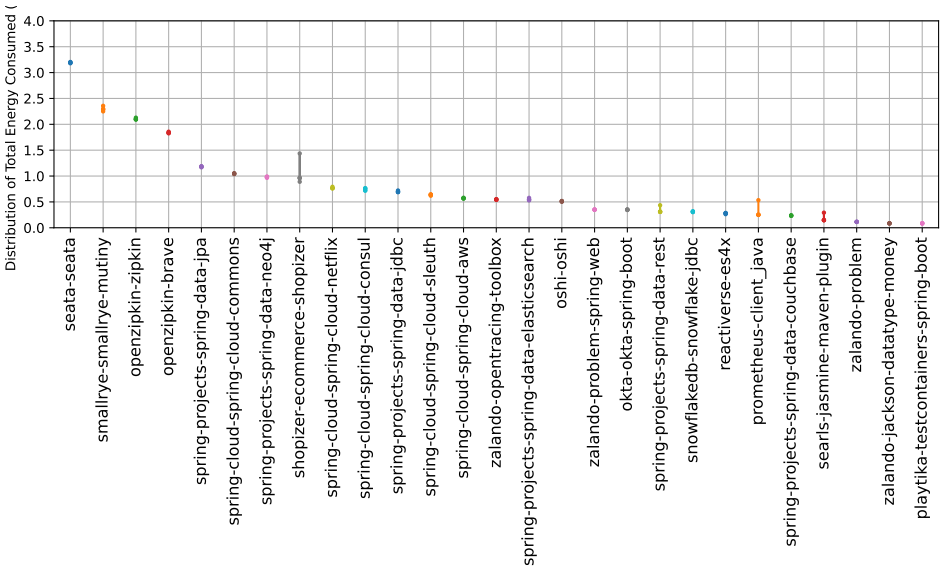

Figure 5.4: Total energy consumption from a subset of Maven projects with caching.

Here we can see that almost all inconsistencies from our Maven results have been reduced to extremely small ranges, in most cases the data points of a single project can not be distinguished from another because of their proximity to each other. There are still some inconsistencies, most obviously the Shopizer project. This time, unlike with Apache Lucene detailed in Section 4.1, there seems to be a near 5-minute delay in the compiling phase with a power draw that is about idle. Again, as we have continuously seen throughout our experiments, the outlier was not the first or last measurement of the project. Unfortunately, we do not have any clear indication of what caused the delay in this instance. For now, we will assume that the inconsistencies in our Maven experiment were caused by the dependency retrieval phase. In Section 5.6 we continue looking into the influences caching has on the runtime and energy performance.

After searching the internet for any know issues with Maven dependencies, we found a

Stack Overflow thread which discussed what we believe to be the same issue[2]. The thread describes the problem as: "while executing `mvn clean install` the build start hanging every single time" and later appends "**UPDATE** Actually the build does not hang forever (**only** 20 - 25 mins)." These observations seem very much in line with our own delays.

The thread has two accepted responses that explain this occurrence. Both agree that this has to do with specific Maven versions. The first says that there is a bug in 3.5.x versions of Maven and that "if you are building a project with submodules or multiple projects, Maven 3.5.x sporadically locks up when downloading duplicate dependencies." The second answer claims more versions of Maven had this issue and specifies the files that are generated that cause Maven to deadlock. Both provided no solution other than changing the Maven versions. According to the release notes of Maven 3.5.3, the issue "Deadlock in dependency resolution has been fixed."[3]

5.4 Yearly estimates

So now that we are able to tell for all these projects how much energy is used by one CI run, we want to use this information to infer some estimation of the energy use for a project on a yearly basis. We will estimate this for the twenty projects in our list with the highest energy readings. To do this, we only need information on the number of commits a particular project tends to have in a year and some base assumptions. To model the yearly number of commits a project has, we use the total number of commits from 2023, which seems like a fair comparison. We assume that only commits to a project's `HEAD` will call the CI and no other branches are involved—this follows the technical definition of CI, even though in practice other branches are often also connected to a project's CI setup. Besides this, we will assume exactly one build and test phase will occur for every commit. The results are shown in Table 5.4.

As we can see, most projects from this subset of projects had an average of at least one commit a day to the mainline branch as we expect in a proper CI environment. The standout project is *Hibernate ORM* with an average of almost 7 commits per day, which makes it the third highest estimate despite its relatively low measured consumption. These estimates do not show the complete picture however, since many of the projects are built and tested multiple times per commit. While not all projects' workflows are clearly labeled, we can see that almost all of these projects build and run tests multiple times per commit. The only exceptions that we found were Spring Cloud Stream, OpenRewrite, and Shopizer. The reason projects are built and tested more than once per commit has often to do with different versions like different Java versions or operating systems, which are all checked separately.

The number of times a project is built and tested is unfortunately not always clear from looking at the GitHub page, tasks' titles can be unclear and often tests are split over different

[2] *maven hangs for ∼20 mins during the project build (used to work fine)*—last accessed: August 30 2024 URL `https://stackoverflow.com/questions/43792427/maven-hangs-for-20-mins-during-the-project-build-used-to-work-fine`

[3] Maven release notes of version 3.5.3—last accessed: August 30 2024 URL `https://maven.apache.org/docs/3.5.3/release-notes.html`

tasks. This makes it very difficult to give more specific estimates. However, to give some indication we can look at Apache Iceberg since it is the project with our highest estimate already. This project has fifty GitHub Actions checks of which it is hard to determine exactly which build and/or test the project. Looking at the runtimes and titles, we can say that there are **at least fourteen** separate runs of Spark CI using different version combinations that seem to run the complete build and test suites. Factoring this in, we get a lower bound yearly estimate of 442.54kWh which is roughly equivalent to 7.7% of the energy consumption of a person in the EU[4]. This roughly equates to 213kg of CO_2 per year according to data from 2023 [31].

5.5 Highest energy readings

From the results seen in Chapter 4 we have observed which projects from our original list had used the most energy during their combined build and test runtime. To find some characteristics that might explain its relatively high energy consumption we will first go through these projects one by one. The projects we will be looking at are Apache Iceberg, Apache Groovy, and Pravega from the Gradle project list and Apache IoTDB, Spring Cloud Stream, and Apache DolphinScheduler from the Maven list.

To investigate these projects we decided to take a look at their respective resource allocations. We are specifically interested in operations that are notorious for being slow and energy-intensive. For this, we designed an experiment to quantify the resources being used by these six projects throughout their respective build durations. To measure the resources a project utilized, we used the `sar` command from `sysstat`. This allowed us to get data on disk IO and CPU usage for the six projects in question.

Please note that this experiment was less formally executed, in particular, this means that for one, the monitoring and measuring were performed on the same device the build was performed on, which means the resource allocation is influenced slightly by the experimental setup including writing the measurements to files. The measurements were also started and stopped manually which caused short delays and the Spring Cloud Stream project was accidentally cut off because the measurements were set with a limit of two hours. Moreover, since we also wanted to take notes on the output of the build process during its runtime, the MinisForum was connected to a display, mouse, and keyboard in contrast to the experimental setup described in Chapter 3. While this makes the experiment somewhat less exact, the aim is to get a general understanding to be able to compare the resources being allocated by these projects and not to gather any detailed information beyond that.

The drive of our MinisForum uses disk blocks with a size of 512 bytes. This means that when you read 10^6 blocks, this is approximately equal to $512 \cdot 10^6 / 1,000,000 = 512$MB worth of data. The actual data size might be somewhat lower because this calculation does not take into account that some write operations may be smaller than the complete block size. Therefore, we use the actual number of read-and-write operations instead of the approximate size equivalent in Figure 5.6 and Figure 5.7.

[4]Energy data from `https://www.iea.org/regions/europe/electricity`

Table 5.1: Energy and commits of 20 projects with the highest energy consumption during CI.

Project name	*Mean energy consumption* (Wh)	*#commits to* `HEAD` *total*[*]	*#commits to* `HEAD` *2023*	*Estimated yearly energy consumption*[**] (kWh)
Apache Iceberg	21.47	5955	1472	**31.61**
Apache Groovy	17.51	21087	671	**11.75**
Apache IoTDB	15.56	11163	1742	**27.11**
Spring Cloud Stream	15.47	4111	287	**4.44**
Pravega	14.10	3295	103	**1.45**
Apache DolphinScheduler	12.10	8498	571	**6.91**
Armeria	11.66	4418	423	**4.93**
Fabric8 Kubernetes Client	10.57	6501	723	**7.64**
Apache ShenYu	9.82	3380	524	**5.15**
Typetools Checker Framework	9.20	18991	887	**8.16**
Picard	9.09	3046	38	**0.35**
Teku	9.07	5550	754	**6.84**
SmallRye Reactive Messaging	8.42	4475	616	**5.19**
OpenRewrite	7.66	6637	1499	**11.48**
Apache Dubbo	7.53	7673	1282	**9.65**
Egeria	7.48	20754	1080	**8.08**
Hibernate ORM	6.90	19260	2483	**17.13**
Enonic XP	6.18	24033	281	**1.74**
Spring Cloud GCP	6.00	2860	493	**2.96**
Shopizer	5.94	29	25	**0.15**

[*] Counted on 4 September 2024.
[**] Assuming the number of commits is equal to 2023 and exactly one CI build per commit to `HEAD`.

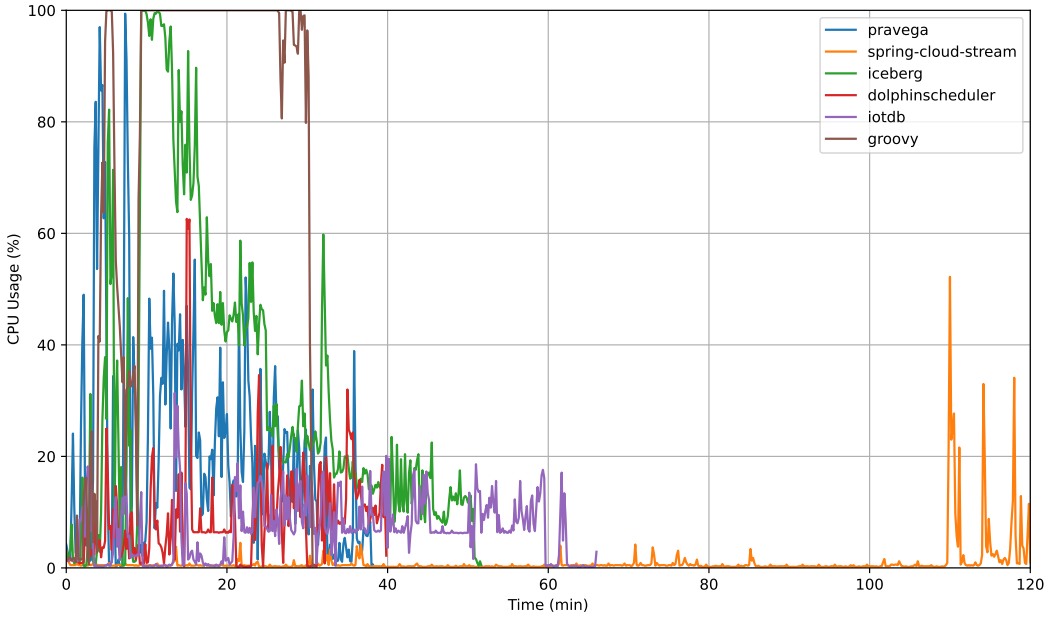

Figure 5.5: CPU usage of highest consuming projects.

Apache Iceberg

> *Iceberg is a high-performance format for huge analytic tables. Iceberg brings the reliability and simplicity of SQL tables to big data, while making it possible for engines like Spark, Trino, Flink, Presto, Hive and Impala to safely work with the same tables, at the same time.*[5]

One thing that stands out during Iceberg's CI is that for the better part of its runtime, we see a decrease in the number of tasks and therefore the number of threads being utilized. We can see this in Figure 5.5 where, after a peak of high CPU utilization, the line crimps to the point that only about 20% of the CPU is used by four tasks that continue for about half of the runtime. This is seemingly because three or four tasks have a relatively long runtime.

Apache Groovy

> *Apache Groovy is a powerful, optionally typed and dynamic language, with static-typing and static compilation capabilities, for the Java platform aimed at improving developer productivity thanks to a concise, familiar and easy to learn syntax. It integrates smoothly with any Java program, and immediately delivers to your application powerful features, including scripting capabilities, Domain-Specific Language authoring, runtime and compile-time meta-programming and functional programming..*[6]

[5]Description directly from `https://iceberg.apache.org`
[6]Description directly from `https://groovy-lang.org`

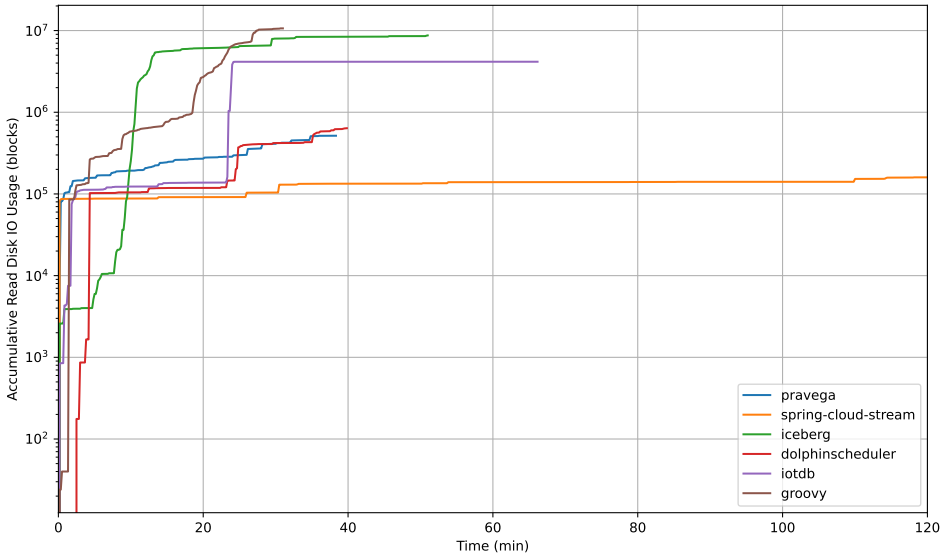

Figure 5.6: Accumulative Disk IO blocks read by highest consuming projects; Y-scale is logarithmic.

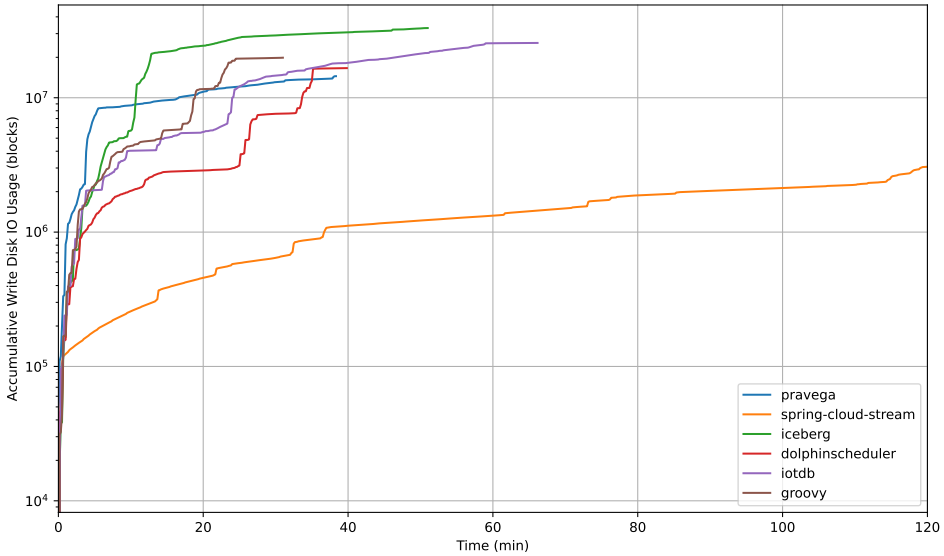

Figure 5.7: Accumulative Disk IO blocks written by highest consuming projects; Y-scale is logarithmic.

As we have observed in Section 4.3, Apache Groovy is the project with the highest average power draw. Similar to Iceberg, the purpose of Apache Groovy is closely aligned with data, in this case, it is a programming language for the Java platform. Compilation is typically a very intensive process because of its use of disk IO and lack of downtime. In this case, we can also see in Figure 5.5 that for most of its runtime, Groovy used the complete 100% of all CPU capabilities, which means Groovy makes efficient use of its resources and has proper parallelization across its CI. Here it seems to be the high intensity of the process that makes for an energy-intensive CI.

Pravega

> *Pravega is about a new storage abstraction – a stream – for continuously generated and unbounded data. A Pravega stream stores unbounded parallel sequences of bytes in a durable, elastic and consistent manner while providing unbeatable performance and automatically tiering data to scale-out storage.*[7]

Although Pravega is again a project that involves storage, when looking at its measured data-block reads this is not as visible as it is for Iceberg and Groovy. We can state that the project does have a substantial number of write-actions. From the data we gathered, this is the only indication of high resource use. We also see in this project that it leaves many of the available threads unutilized.

Apache IoTDB

> *Apache IoTDB (Database for Internet of Things) is an IoT native database with high performance for data management and analysis, deployable on the edge and the cloud. Due to its light-weight architecture, high performance and rich feature set together with its deep integration with Apache Hadoop, Spark and Flink, Apache IoTDB can meet the requirements of massive data storage, high-speed data ingestion and complex data analysis in the IoT industrial fields.*[8]

When we look at the CPU utilization of IoTDB in Figure 5.5, we might not expect it to be the third-highest project in terms of energy consumption. It does not use the many threads available to their potential and while it has a long CI runtime, it is not even close to that of the Spring Cloud Stream project. The main suspect of the high energy intensity is again in this case its high disk IO usage. Like Iceberg, IoTDB reads more than $4 \cdot 10^6$ and writes more than $2 \cdot 10^7$ blocks worth of data to memory. This together with its aforementioned longer runtime most likely leads to its high energy use.

Spring Cloud Stream

> *Spring Cloud Stream is a framework for building highly scalable event-driven microservices connected with shared messaging systems.*[9]

[7] Description directly from `https://cncf.pravega.io`
[8] Description directly from `https://iotdb.apache.org`
[9] Description directly from `https://spring.io/projects/spring-cloud-stream`

Spring Cloud Stream is a project that experiences a lot of downtime during CI. As we mentioned, unfortunately, the experiment was cut off before the build was complete, but we know from our other graphs that this project has a relatively low average power consumption. When looking at Figure 5.6 and 5.7, this project has the lowest disk IO utilization out of the six we tested. It spends a lot of its runtime fetching dependencies, and only after almost two hours did the project start utilizing the CPU.

Apache DolphinScheduler

> *Apache DolphinScheduler is a distributed and extensible open-source workflow orchestration platform with powerful DAG visual interfaces*[10]

Similar to Pravega, this project never reaches extreme values when it comes to CPU usage or disk usage. Its number of write operations seems to be the most significant indication of resource intensity.

Combining the observations on these six projects we see some patterns. For one, we see the projects often reach very high disk read and/or write operations during their build and test phases. This often involves using data from large Docker containers to perform tests. Besides this, we also see that for most of the projects, the system resources—namely the available threads—were not utilized to their full extent, which extends the runtime of these projects. This often involves large test suites, which means that the disparity between tasks could probably be reduced or prevented by separating the largest test suites among multiple tasks.

5.6 Influence of dependency-caching

When building a Gradle project for the first time you are greeted with a message:

Starting a Gradle Daemon (subsequent builds will be faster)

To investigate whether this rings true for our setup and how much enabling caching influences our runtime and energy readings we configured the experimental setup as described in Section 3.7. This enables Gradle and Maven to cache in the home directory. In CI servers the most common form of caching is dependency-caching where the build tool stores artifacts and dependencies for later builds to (drastically) reduce the load in subsequent builds. We tested the ten repositories with the highest energy readings in our first experiment for both Maven and Gradle. We end up with twenty projects which we can see in Figure 5.8.

In some of the projects, we saw a dramatic reduction in both runtime and total energy consumption. What you could see in these projects is that some of the Gradle tasks read `FROM CACHE` which indicates that Gradle skipped test execution and reused the outcomes of tests that were performed in earlier iterations. This is an important factor when we look at the differences between Groovy with and without caching. Do note that in this experiment we used the same commit and therefore same code for each iteration; under normal circumstances, there would be some changes to the codebase which would force the CI to rerun (some of) these tests which would increase the energy use.

Interestingly, our highest-consuming project out of all has no measured decrease with caching enabled. This can be because it is disabled in the Gradle scripts in the project or because its caching (mostly) takes place in the project-specific build location which we removed each iteration.

[10] Description directly from `https://dolphinscheduler.apache.org`

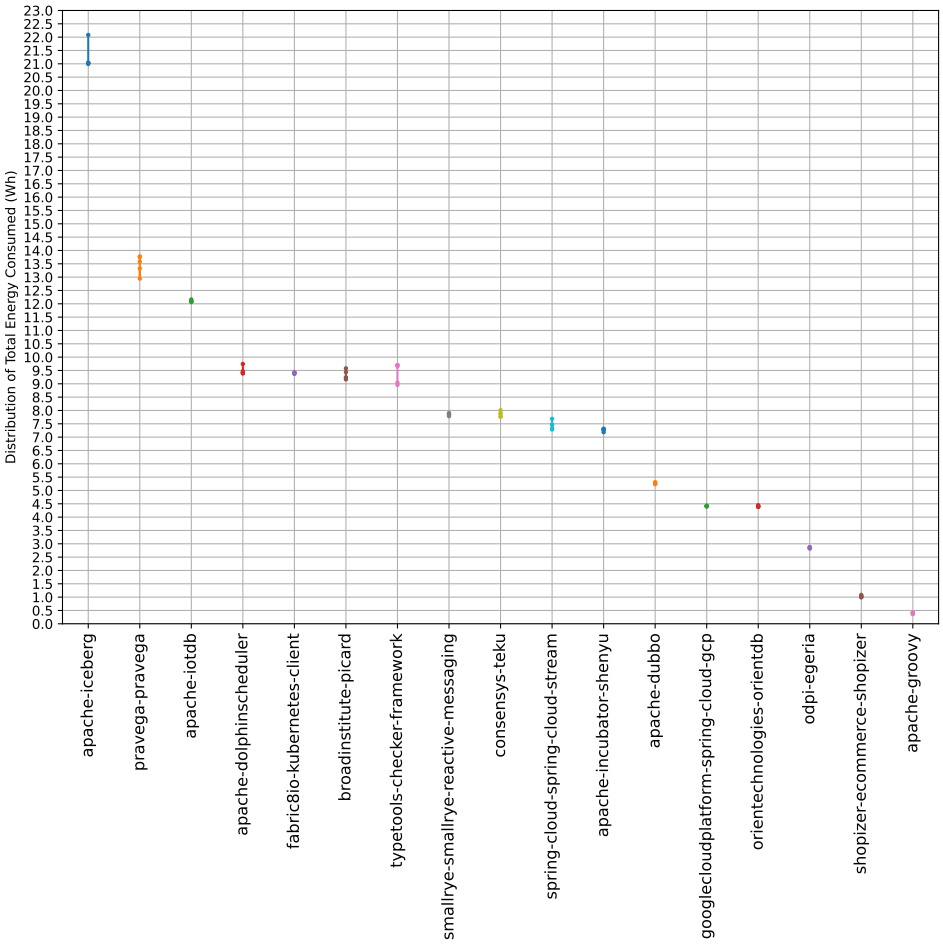

Figure 5.8: Total energy consumption from projects with caching.

Chapter 6

Related Work

This research is of course not the first to tackle software industry practices from an environmental perspective. Just like in many other industries, the subject of energy consumption and the environmental impact of software technology and development is getting more attention. To the point that some universities are providing courses on the topic of sustainable software engineering. The Green Software Foundation[1], the Green Web Foundation[2] and Green Coding Solutions[3] are some examples of the part of the software development industry trying to provide resources for software developers to develop sustainable practices [5, 6, 14, 21].

6.1 Short-paper

This research is based on a short-paper called *An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects* [39]. In it, Zaidman explores ten large open-source Java projects and measures their energy usage over the runtime of their respective CI, similar to our experiments. The main difference is that the experiments from this short-paper were not automated, which introduces some variability. Experiments had to be manually started and stopped, which drastically reduced the scale at which they could be performed. The main finding of Zaidman was that individual build and test simulations did not use that much electricity, but looking at the number of commits from these projects, they reached notable estimates of up to 161kWh per year. Note that the experiments were executed on the same device used in this research (the MinisForum EM680).

When comparing the results, we see that the individual energy readings of the short-paper mostly fall within the range of projects we found in our research except for the ElasticSearch project which places itself quite far above Apache Iceberg, taking 32Wh worth of energy to build. This high energy reading combined with the incredibly high frequency of commits (5025 commits in the year 2022) made the project account for the equivalent of about 9.7% of the average household energy consumption of a citizen of the European Union. Fortunately, our results suggest that this combination of high energy consumption and high commit frequency might not be very common. To know for certain how common it is for a project to fall into both these categories we need to do more research.

[1] `https://greensoftware.foundation/`
[2] `https://www.thegreenwebfoundation.org/`
[3] `https://www.green-coding.io/`

6.2 E-Compare

In another research called *E-Compare: Automated energy regression testing for software applications*, the energy consumption of software projects over time was investigated through the use of a regression testing tool[4] designed by Hagen [18]. The regression tool faced the same limitations as we have seen in this research, namely that the hardware components of the servers are an unknown factor. The tool was ultimately designed to use energy profilers from energy models based on accessible information, like the CPU load, which unfortunately makes the results hard to compare to our data. The main focus of this research was the increase or decrease in energy use over time. To measure this, the tool was used on about 50 commits for thirteen software projects spanning different languages including TypeScript, Python, and JavaScript. This functioned as a proof of concept for the tool to provide insight into the relative energy cost of a project over its versions.

In the paper, Hagen found two main influential factors that affect energy consumption in a software project: the addition/removal of tests and package updates. In the experiments, the majority of commits lead to minor changes that mostly seem to fall within the margin of error.

6.3 Green Mining

In *Green mining: a methodology of relating software change and configuration to power consumption*, Hindle presents "a general methodology for investigating the impact of software change on power consumption" [20]. The methodology aims to alleviate the need for expensive testing regarding energy consumption. The study investigates how software changes and object-oriented metrics influence power consumption, with case studies using the Firefox browser and Azureus/Vuze BitTorrent client. Additionally, it explores how library versioning affects power consumption in rTorrent, offering insights into the relationship between software metrics and energy efficiency.

This is a very different approach to energy measurements relating to software energy efficiency where the actual measurements by energy profilers or power monitors would become redundant. The stateless regression model Hindle made from data on the Mozilla Firefox project based itself on three variables found to be statistically significant: % user-time per second, transactions per second (disk hits), and kB of active memory. The model was trained on 500 versions of Firefox and predicted the mean power draw with an R^2 value of 0.38.

The study also showed how small changes on a large scale can impact global energy savings. In this case, a lower consumption branch of Mozilla Firefox showed a 0.25W saving on the main branch. If applied at a broad scale of 4 million users, Hindle argues this could save 1.0 Mega-Watt of power worldwide. This is roughly equivalent to saving an American household's monthly power use every hour.

[4]The tool can be found at `https://koenhagen.github.io/E-Compare/`

Chapter 7

Threats to Validity and Limitations

In this chapter, before we draw conclusions about our research, we will take some time to go over the limitations of the research. Each section discusses a shortcoming of the research or experiments and describes how it came to be and its potential threats to the validity of the research.

7.1 Scope

Because of the simplicity of the way we approached compiling and testing the repositories, trying to make the implementation as broad as possible, we introduced a form of survivorship bias. Our set of projects was in a large way determined and shaped by whether the repository in question would be able to build properly without any modifications to the system or the commands being executed. This could influence the types of projects that could be measured and probably lead to larger and more complex projects not being tested as often as smaller simpler projects. Examples of some specific projects that we could not build in this design were ElasticSearch and Flink. These are two projects which had very high readings in the short-paper this thesis is a continuation of [39]. Especially ElasticSearch would have been an interesting addition since it has so many commits per year[1].

Another limiting factor for the scope of this research was the time. Even though the time was used economically, we still from the start decided to go for a quantitative analysis in regards to the number of projects that were being researched, which came at the cost of the sample sizes for individual projects. The thought process was that the combined total would still make for reliable data about the distribution between the projects. It did allow us to measure our total of over 200 projects which was the bigger priority, but larger sample sizes could have given more credibility to the results.

7.2 Crashes and build fails

One aspect of the experiments that we did not measure or keep track of was the occurrence of crashes and errors. They did sporadically occur throughout the experiments as briefly mentioned in Section 4.2. When builds failed we most often just repeated the experiment until we had five measurements to make sure the different data points of a project had the same result. The occurrence

[1] In the last year there were a total of 6181 commits in the ElasticSearch project. Counted on 10 September 2024

of these build fails does beg the question of why some projects would build one time and fail another. For this, we have two explanations.

One explanation is the inclusion of timed tests in a project. When a test with a tight timeout window exists within a project's CI, this can introduce inconsistent results since these timeouts have windows that are designed with large servers in mind. While our MinisForum is quite powerful for a mini-PC, this is one area it can be restricted. Many errors that occurred during the formation of our project list seemed to be because of timeouts.

The other explanation has to do with external factors such as dependency retrieval. This can be somewhat inconsistent as we have seen with our Maven results. In our Maven cases, as we showed in Section 5.3, this sometimes leads to crashes due to timeouts after long waits without responses.

7.3 Environmental factors

In any experiment, some environmental aspects cannot reasonably be controlled. While we have made an effort to reduce the unknowns and unstable factors in our experiments, there are still some environmental factors we want to acknowledge in this section.

Temperature The experiments in this thesis have been performed over the span of multiple months including over the spring and summer. The weather and therefore temperatures have not been consistent throughout this time. We acknowledge that this can influence the measurements to some degree. Results from individual experiments are never measured further apart than a little over one month (Our longest-running experiment was the main Gradle experiment which lasted from May 23rd to June 28th).

Network Stability Another environmental factor was the network on which the experiments were run. This is a home network with three different people which met all types of differences in terms of network usage. Just like with temperature, we acknowledge this might influence the measurements to some degree, but this is sadly a factor that could not reasonably be reduced.

Ordering In many types of experiments, it is a good idea to randomly order the experiments to make the results of a single experiment independent. In our setup, it was significantly easier to arrange the projects once and then run all measurements for a single project at once as it is already set up on the server system. It would have been better to adjust the setup to one with stochastic ordering.

Chapter 8

Conclusions and Future Work

This final chapter gives an overview of the contributions this research provides. After that, we will reflect on the results we have seen and discussed to draw conclusions. Finally, we will discuss some directions future work could take.

8.1 Contributions

This thesis aimed to contribute to research on the sustainability of software. One way this thesis contributes is the experimental setup along with its replication package [3]. This allows researchers to use a similar setup to the one used in this thesis. This automated setup is especially useful because of its use of hardware power monitoring and its division of labour between the systems which allows for isolated measurements on the specific system you need to measure with very little overhead. The replication package is also useful in many other scenarios where a researcher wants to measure energy consumption with a hardware power monitor since the various scripts can all easily be adapted to different tasks on the server system.

Besides this, this thesis serves as a continuation of exploratory research into the costs that come with Continuous Integration in software development, but also the cost of building and testing software in general. We provided a thorough analysis of the data we gathered to explain the extremes in our findings. We also explored some simple but effective ways for projects that are energy-intensive, which could impact their footprint dramatically.

8.2 Conclusions

In this thesis we have explored the topic of CI, in particular, we looked at the benefits it provides to the developer and to the end product, the way CI is particularly useful in open-source development, but also how the way CI is performed makes it a resource-intensive part of the modern software development methodology. This comes down to its short development cycles and thorough testing. Developers tend to be aware of the benefits CI brings, but not so much of the costs that come with it. We have designed our experimental setup and measured over 200 Java projects to learn about their energy uses. We were able to separate these measurements between two major phases—compiling and testing—to show the relation these phases have to the total energy consumption of these projects. Finally, we made observations about our data and discussed the patterns we have seen. Here we will go over the research questions one by one.

RQ1. How much energy is used by an open-source Java project during the build and test phases of CI?

As we have discussed extensively in Chapter 5, our measurements fortunately showed an overwhelming majority of the projects built and tested with no cause for concern even in frequent commit development environments like those that use CI. Individual builds and test runs from Gradle and Maven projects alike most often consumed under 10Wh of energy. Some projects, like Apache Iceberg, showed significantly higher readings, measuring up to $21\frac{1}{2}$Wh for a single cycle. When such intensive builds are combined with high-frequency building development environments like the ones that use CI, this results in a high yearly energy consumption. While it is hard to determine exactly how many times some projects build and test the project per commit, for Iceberg we note at least fourteen separate build and test cycles occur per commit. This makes this single project account for at least 442kWh of energy on a yearly basis used for building and testing alone. This roughly equates to 213kg of CO_2 which is more than the emissions of a return flight between Amsterdam and Dublin[1].

RQ2. Which phases of the CI pipeline use more and which use less energy?

According to our findings, the ratios of energy use between the compiling and testing phases of CI can differ a lot between projects. Projects ranged from $> 98\%$ of time spent compiling to as little as 7%. We did find that there was a correlation between this ratio and a project's total energy consumption. Projects with high total energy consumption during compiling and testing typically spent more of their energy during testing whereas projects with low combined energy consumption spent more time compiling.

RQ3. Which characteristics of a Java project impact its energy use the most?

Looking at the projects with the highest energy readings, we saw a couple of recurring characteristics. Many of the projects had purposes relating to data such as storage abstractions, databases, or programming languages. Projects that are data-intensive can take a relatively long time to compile and test, which increases energy use. They also often use large Docker containers to test their projects. We also saw that some projects did not use the system's capabilities to its full potential. Leaving many threads unused, which leads to unnecessary long runtimes. Dividing large test suites into multiple smaller test suites might be able to reduce these runtimes and with them, the energy spent by these projects.

RQ4. How does dependency caching affect the runtime and energy usage of large Java projects?

We have also seen that often, for some of these projects that do have energy-intensive compiling and testing phases during CI, the energy consumption can be reduced dramatically by using caching to its full extent. For Maven projects especially, we have seen that fetching dependencies can take a lot of runtime, and thus removing the need to gather these dependencies can drastically reduce the total energy consumption of the build. Besides this, test result caching can, when possible, also heavily impact the test execution time. Our experiment modeled the extreme case where no changes were added to the codebase; this allowed all test-result caches to be used, which would be admittedly rare under normal circumstances. This does not change the fact that these caches can dramatically impact energy consumption.

[1] According to Carbon Calculator; URL `https://www.carbonfootprint.com/calculator.aspx`

8.3 Future work

This research still leaves us with many questions regarding the energy consumption involved in CI. One way in which this research can still be improved is the level at which the data was segmented. In our experiments, we only made the distinction between compiling and testing as phases in CI. Segmenting any further would quickly become complicated and less precise. One segmentation we did not have that can easily be made is fetching the repository. This would form a small segment which we included in the compiling phases. Finding a way to distinguish more phases or separate elements like static analysis, gathering dependencies and integration testing would allow even more analysis on the parts of a project that use the most energy.

Another thing to explore in future work is to look at the progression of energy consumption for projects over time, similar to Hagen [18]. In this research, we looked at a single commit and extrapolated that data to make conclusions about a longer timespan, but if we were to look at the same project over time we would be able to see regressions and to more accurately compare the influences of caching.

Lastly, we propose to expand this research into different programming languages. In this research, we only looked at Java projects, specifically Gradle and Maven projects and it would be very interesting if similar conclusions could be formed for projects written in other programming languages.

Bibliography

[1] Muhammad Ovais Ahmad, Jouni Markkula, and Markku Oivo. Kanban in software development: A systematic literature review. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 9–16, 2013. doi: 10.1109/SEAA.2013.28.

[2] Samar Al-Saqqa, Samer Sawalha, and Heba Abdelnabi. Agile software development: Methodologies and trends. *Int. J. Interact. Mob. Technol.*, 14:246–270, 2020. URL https://api.semanticscholar.org/CorpusID:225548331.

[3] Robert Arntzenius. Measuring Energy Consumption during Continuous Integration of Open-Source Java Projects - Replication package. 9 2024. doi: 10.6084/m9.figshare.27103042.v2. URL https://figshare.com/articles/thesis/Measuring_Energy_Consumption_during_Continuous_Integration_of_Open-Source_Java_Projects_-_Replication_package/27103042.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company, 1999.

[5] Sara Bergman. How to measure the energy consumption of your backend service. *Green Software Foundation*, October 2021.

[6] Sara Bergman. How to measure the energy consumption of your frontend application. *Green Software Foundation*, September 2021.

[7] Amit Bhanushali. Ensuring software quality through effective quality assurance testing: Best practices and case studies. *International Journal of Advances in Scientific Research and Engineering*, 26(1), 2023.

[8] Mat Brown. Digging into data center efficiency, pue and the impact of hci. *Nutanix*, May 2023.

[9] Stephen Cass. The top programming languages 2024 > typescript and rust are among the rising stars, August 2024. URL https://spectrum.ieee.org/top-programming-languages-2024. Online graph; last accessed: September 8 2024.

[10] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering*, 24(4):1649–1692, July 2018. doi: 10.1007/s10664-018-9640-7.

[11] Luís Cruz. Tools to measure software energy consumption from your computer, July 2021. URL https://luiscruz.github.io/2021/07/20/measuring-energy.html. Online; last accessed: July 3 2024.

[12] Luís Cruz. Green software engineering done right: a scientific guide to set up energy efficiency experiments, October 2021. URL https://luiscruz.github.io/2021/10/10/scientific-guide.html. Online; last accessed: August 31 2024.

[13] Statista Research Department. Electricity consumption worldwide from 2000 to 2022, with a forecast for 2030 and 2050, by scenario (in 1,000 terawatt-hours). Technical report, McKinsey & Company, November 2023. URL https://www-statista-com.tudelft.idm.oclc.org/statistics/1426308/electricity-consumption-worldwide-forecast-by-scenario/. Online graph; last accessed August 7 2024.

[14] Green Software Foundation. Building green software through standards and collaboration. *Green Software Foundation*, July 2024.

[15] M. Fowler and M. Foemmel. Continuous integration, 2005. URL http://www.martinfowler.com/articles/continuousIntegration.html. Online; last accessed: August 30 2024.

[16] GitHub Docs. About workflows, 2024. URL https://docs.github.com/en/actions/writing-workflows/about-workflows. Online; last accessed: September 15 2024.

[17] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of ci services in github. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672, 2022.

[18] K.A.P. Hagen. E-compare: Automated energy regression testing for software applications. Master's thesis, TU Delft, 2024. URL https://resolver.tudelft.nl/uuid:28d8b827-5d50-4365-bff0-57a858658e91.

[19] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970358. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2970276.2970358.

[20] Abram Hindle. Green mining: A methodology of relating software change to power consumption. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 78–87, 2012. doi: 10.1109/MSR.2012.6224303.

[21] Asim Hussain. What is green software? *Green Software Foundation*, August 2021.

[22] Madhumitha Jaganmohan. Energy consumption worldwide from 2000 to 2019, with a forecast until 2050, by energy source (in exajoules). Technical report, BP, January 2023. URL https://www-statista-com.tudelft.idm.oclc.org/statistics/222066/projected-global-energy-consumption-by-source/. Online graph; last accessed August 7 2024.

[23] Erik Jagroep, Jan Martijn E. M. van der Werf, Slinger Jansen, Miguel Ferreira, and Joost Visser. Profiling energy profilers. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 2198–2203, New York, NY, USA, 2015. Association for

Computing Machinery. ISBN 9781450331968. doi: 10.1145/2695664.2695825. URL https://doi.org/10.1145/2695664.2695825.

[24] Timo Johann, Markus Dick, Stefan Naumann, and Eva Kern. How to measure energy-efficiency of software: Metrics and measurement results. *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*, 06 2012. doi: 10.1109/GREENS.2012.6224256.

[25] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in java-based open source software development. *Software: Practice and Experience*, mar. 2024. doi: 10.1002/spe.3321.

[26] Xiang Li, Dorsan Lepour, Fabian Heymann, and François Maréchal. Electrification and digitalization effects on sectoral energy demand and consumption: A prospective study towards 2050. *Energy*, 279:127992, 2023. ISSN 0360-5442. doi: https://doi.org/10.1016/j.energy.2023.127992. URL https://www.sciencedirect.com/science/article/pii/S0360544223013865.

[27] Dominik Maximini. *The Scrum Culture*. Springer International Publishing, Cham, Januari 2018. doi: 10.1007/978-3-319-73842-0.

[28] NOS Nieuws. Wereldwijd problemen door computerstoring: onder meer luchthavens en ziekenhuizen getroffen. *NOS*, July 2024. URL https://nos.nl/artikel/2529464-wereldwijd-problemen-door-computerstoring-onder-meer-luchthavens-en-ziekenhuizen-getroffen. Online; last accessed: August 1 2024.

[29] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016. doi: 10.1109/MS.2015.83.

[30] Sean Previl. Crowdstrike outage, cyberattacks a 'wake-up call' to dangers of big tech reliance. *GlobalNews*, July 2024. URL https://globalnews.ca/news/10644838/tech-reliance-crowdstrike-outage-cybersecurity/. Online; last accessed: July 31 2024.

[31] Hannah Ritchie and Pablo Rosado. Electricity mix. *Our World in Data*, 2020. URL https://ourworldindata.org/electricity-mix. Last revised in January 2024.

[32] Eddie Antonio Santos, Carson McLean, Christopher Solinas, and Abram Hindle. How does docker affect energy consumption? evaluating workloads in and out of docker containers. *Journal of Systems and Software*, 146:14–25, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.07.077. URL https://www.sciencedirect.com/science/article/pii/S0164121218301456.

[33] S. S. SHAPIRO and M. B. WILK. An analysis of variance test for normality (complete samples)†. *Biometrika*, 52(3-4):591–611, 12 1965. ISSN 0006-3444. doi: 10.1093/biomet/52.3-4.591. URL https://doi.org/10.1093/biomet/52.3-4.591.

[34] Naveen Tavva and Phuong-Ha Nguyen. Software: market data & analysis. Technical report, Statista Market Insights, October 2023. URL https://www-statista-com.tudelft.idm.oclc.org/study/102689/software-report/. Online; last accessed September 15 2024.

[35] Serhii Uspenskyi. How many software engineers are there in 2024? Technical report, Springs, July 2024. URL https://springsapps.com/knowledge/how-many-software-engineers-are-there-in-2024. Online; last accessed: September 3 2024.

[36] Roberto Verdecchia, Fabio Ricchiuti, Albert Hankel, Patricia Lago, and Giuseppe Procaccianti. *Green ICT Research and Challenges*, pages 37–48. Springer International Publishing, 01 2017. ISBN 978-3-319-44710-0. doi: 10.1007/978-3-319-44711-7_4.

[37] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd edition, 2008. ISBN 0470031468.

[38] Max Wei, Colin A. McMillan, and Stephane de la Rue du Can. Electrification of industry: Potential, challenges and outlook. *Current Sustainable/Renewable Energy Reports*, 6(4): 140–148, November 2019. doi: 10.1007/s40518-019-00136-1.

[39] Andy Zaidman. An inconvenient truth in software engineering? the environmental impact of testing open source java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, AST '24, page 214–218, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705885. doi: 10.1145/3644032.3644461. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3644032.3644461.