SMURF: a Methodology for Energy Profiling Software Systems

Simulate and Measure to Understand Resource Footprints

Masther Thesis

Otto Kaaij

Delft University of Technology

TUDelft

SMURF: a Methodology for Energy Profiling Software Systems

Simulate and Measure to Understand Resource Footprints

by

Otto Kaaij

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday March 7, 2025 at 11:00 AM.

Student number:	4693515	
Project duration:	April 24, 2024 – February 5, 2025	
Thesis committee:	Luís Cruz	TU Delft, daily supervisor
	Arie van Deursen	TU Delft, thesis supervisor
	Anna Lukina	TU Delft, external advisor
	June Sallou	Wageningen University & Research, co-daily supervisor
	José Silva	Solenix, external supervisor

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Icons:	Uniconlabs, Prettycon and Mindworlds via `flaticon.com`
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

Preface

As I sit here, a day before I hand in this thesis, I can't help but think back on the seven and a half years of Computer Science at the TU Delft that are about to come to an end.

I have taken my sweet time, often splitting my focus between this study program and a wild variety of other projects. I am grateful for the support, the encouragement, and the freedom to make my own, inefficient choices that my parents have so seemingly effortlessly provided.

A thousand thank-yous to José, June, and Luís, my all-star team of supervisors, for their unreasonable flexibility, genuine encouragement, and their sharp feedback on this thesis.

Thanks also to Stefan Hugtenburg, who, if only they knew him, would be an inspiration to all teachers everywhere, for his enthusiasm, openness, and availability as a human contact in a bureaucratic ocean, which first set and then kept me on the Computer Science path.

And to my wife, for everything.

Otto Kaaij
Delft, January 2025

Summary

Understanding the energy profile of a complex, multi-faceted software system is difficult. In this thesis, we present a novel methodology, called SMURF, a five-step methodology that gives insights into the energy consumption of a complex system. The methodology is broadly applicable, supports informed decision-making, and closely involves and engages stakeholders. We evaluate the methodology with a case study on MUST, a software system used in spacecraft operations. In the case study, SMURF successfully finds energy hotspots and wasteful components in MUST, and is used effectively to formulate actionable recommendations. Through the case study, we find that the SMURF methodology serves as an effective engagement tool to get developers, users, and product owners interested in sustainable software ideas.

Contents

1 Introduction

It is no secret that computers consume energy: just ask the person next to you on the train, desperately trying to finish an email before their laptop battery runs out. But when the laptop becomes a desktop that is plugged into the wall, or when it becomes a server in a data center far, far away, it becomes easy to forget.

In the midst of the climate crisis, we should not allow ourselves to forget that the ICT industry, the sum of all phones, laptops, datacenters, networking equipment, and all other computing devices, accounts for around 2.1% to 3.9% of global energy consumption [13]. That, as a part of this industry, we generate more than 50 million tons of e-waste per year when we throw away old hardware [22], and that we ruin entire ecosystems by mining for the rare metals needed to produce new hardware. And that we generally remain, willfully or not, ignorant of this.

In recent years there has been more and more academic interest in this topic, sprouting the field of 'Green IT' or 'Sustainable Software Engineering'. This field concerns itself with two questions.

The first is 'how can ICT contribute to a greener society?', also dubbed 'IT for Green' [11]. By increasing the efficiency of logistics systems, by reducing the need to travel by facilitating remote work, through better climate simulations, and in many other ways, ICT systems can be used to limit climate impact or understand certain issues. Though the question is also of particular interest to industry parties attempting to sell 'climate friendly' software systems, ICT systems and digitalization are part of the solution to the climate crisis.

The second question is 'what is the climate impact of ICT?', also dubbed 'Green IT' [11]. In many different ways, and from many different viewpoints, the climate impact of ICT is dissected, evaluated, and mitigated. Through this distinction, it becomes clear that ICT is part of the solution, but also part of the problem.

One particular challenge in mitigating the climate impact of ICT is that just understanding it is difficult – and without understanding it, it is very hard to do something about it. On any scale, there are major challenges. Zooming out all the way to the global scale, the complexity of ICT systems, rapid technology evolution, closed-source systems and company secrets, and interconnected direct and indirect effects make it challenging, if not impossible, to fully understand its climate impact. Zooming in all the way to single functions and data structures, the focus naturally shifts away from climate impact to a more concrete measure of energy consumption, but still, major challenges remain. Measuring energy consumption is difficult: it can vary significantly depending on the hardware and software environments, and isolating single functions for measurements can be almost impossible. The behavior of functions and data structures might change dynamically due to conditions like input size and concurrency, further complicating measurements.

In addition to this global and this very granular one, there are many different scopes that are of interest to the Green IT field. Somewhere in between the two extremes sits the evaluation of the energy consumption of a single, complex software system. This is the scope that this thesis is concerned with.

This scope is of particular interest because it is at this level that a climate-concerned software engineer, or a team of them, will probably start trying to make an impact – it is the scope that naturally arises when you ask yourself the question: “How can I make the system I develop more climate-friendly?”

To make informed decisions, we need an overview of the energy consumption of our system – to understand how much energy different components of the system consume. This energy profile allows us to direct our efforts to the right places, and to evaluate our progress.

Creating such an energy profile of a complex system effectively and accurately is a challenge. Formulating representative test scenarios, performing accurate measurements when access to production servers may be limited, developing the right experiments, and interpreting the results correctly and effectively are all non-trivial steps.

In particular, this thesis places an emphasis on the formulation of representative test scenarios. In earlier work, and as we’ll discuss in much more detail in chapter 3, this process is often based on arbitrarily formulated user stories or on pre-existing test cases. Using these methods, it is still possible to compare different versions of the same software, or of a software component – you run the same test case on different versions and compare the energy consumption. However, using this method, it is rather difficult to compare different components of a software system. For this, it is necessary to understand how much each component is used in practice.

In this thesis, we introduce a new methodology called SMURF to tackle this problem. SMURF is a five-step methodology to evaluate the energy profile of a single, complex, multi-faceted software system based on server logs and other usage data. The SMURF methodology relies on logs that tell us what requests are made to the server. Based on this and some additional usage data, the first step of the methodology is to formulate a ‘typical use case’. This typical use case is then simulated on a test deployment of the system and measured.

This constitutes the first of the two goals of this thesis:

Goal 1: Develop a methodology to evaluate the energy profile of a single, complex software system, based on server logs and other usage data.

The SMURF methodology we introduce in this thesis allows for the creation of energy profiles for complex software systems. While server logs are required, the methodology is agnostic to the exact format. Apart from this requirement, the methodology is applicable to a wide range of different systems and system architectures.

To evaluate the SMURF methodology, we need a software system to test it on. In this thesis, we collaborate with Solenix[1]. Solenix is an international software company specialized in the space domain. Over the past 20 years, the company has contributed to the implementation and development of MUST (Mission Utility and Support Tools) for the European Space Agency (ESA). MUST is a tool for managing, visualizing, and understanding space telemetry data. In chapter 2, we will go into much more detail on what MUST is, what it is used for, and why it was developed.

The space industry, as a whole, has a significant impact on the climate through its carbon emissions. But ESA also plays an important role in measuring and combatting climate change through its meteorological and climate observation missions. In this sense, the ESA fits neatly in the ‘IT for green’ story. But ESA has also committed itself to sustainability goals through the Paris Climate Agreement and the European Green Deal. This sets the stage for development efforts into making ESA, including its IT landscape, more sustainable.

An energy profile of MUST will help guide us to informed decisions about where to focus development efforts. It can help identify which areas of the software might be inefficient or require optimization, and to strategically target these areas for improvement.

This brings us to the second of the two goals of this thesis:

Goal 2: Apply the methodology to MUST to gain an understanding of how the methodology works in practice, and in the process learn about the energy profile of MUST.

[1]`https://solenix.ch/`

Through these goals, we contribute a new methodology for evaluating the energy profile of a complex software system, based on server logs and other usage data, a thorough and practical evaluation of that methodology, as well as insights into the energy consumption of the MUST system.

The rest of this thesis is structured as follows: in *Chapter 2: MUST* we explain the details of the MUST system, and explain why it's a good fit as a case study for our methodology. In *Chapter 3: Related Work* we dive into related work: earlier efforts on climate impact of ICT and on energy profiling. In *Chapter 4: SMURF* we present our methodology, and in *Chapter 5: Applying SMURF* we apply it to MUST and present the results. In *Chapter 6: Discussion* we discuss what we learned when applying the methodology and its limitations, and finally we present our conclusions in *Chapter 7: Conclusion*.

2 MUST

Satellites produce data. Among others, power, thermal control, and communication systems produce telemetry data in various forms. Mission controllers rely on this data to monitor and control the satellite. To keep a satellite flying in space, it is crucial that fast and efficient access to this data is available. But because there is so much of this data – a typical mission has thousands of parameters – this is not a trivial requirement.

At ESA, the European Space Agency, the most common system for this is the ‘Mission Utility and Support Tools (MUST)’. It is used by over 250 engineers at most ESA missions [27], primarily at ESOC, the European Space Operations Center. It is a tool that makes the incredible amount of data that a spacecraft package produces accessible and understandable. Out of 30 active ESA missions, 25 use MUST as a critical part of their process. In collaboration with ESA, MUST is currently developed and maintained by a German-Swiss-Italian company called Solenix.

MUST was created to address inefficiencies in existing packet-based telemetry systems. Before MUST, systems stored raw packets, directly as sent down by the spacecraft. These packets, known as CCSDS packets, can contain multiple parameters, or a single parameter could be spread over multiple packets. In addition, the packets are uncalibrated: for example, the satellite’s system time may need to be synced to UTC time, or some values may need normalization. Before MUST, this work was done on demand, whenever specific data was needed.

This makes data analysis slow, and introduces a lot of redundant work.

MUST, in contrast, stores parameters in parsed and calibrated form in the relational database that lies at its core. In addition to efficient data storage, MUST provides powerful data analysis and flexible data visualization tools. Users can graph data, view it in table view, or configure custom alphanumeric displays. In addition, users can use two data analysis tools, DrMUST and Novelty Detection , to identify correlations and detect anomalies in telemetry data. [17] These encompass the core functionalities of MUST: import data, store data, and visualize data.

Parameters are usually first imported into the Mission Control System (MCS). This system correlates the satellite time with the ground time and calibrates the parameters. The types, units, and calibrations of parameters, as well as synthetic parameters and display configurations, are all mission-specific. The MCS has a Mission Configuration that defines all this and more. The MCS then creates a file with all this data.

Most commonly, the parameters are time series parameters that consist of (timestamp, value) pairs. For example, the spacecraft might record its position or the temperature of a specific component once every second.

MUST can import data from multiple sources and formats, utilizing various importers. The platform supports the importation of diverse data types such as timeseries, housekeeping telemetry, event logs, telecommands, flight dynamics data, and other data. These importers have been developed incrementally over time to meet the demands of numerous European space missions and programmes.

Imported data is accessible via an API, called ‘MUSTLink’ through which the frontend – of which there are two versions: ‘WebMUST’ and the newer ‘ngWebMUST’ – accesses data. The API can also be used to get data from the system programmatically.

Over time, MUST has evolved into a complex system with many different uses and components. In addition, MUST has become a critical part of spacecraft operations at ESA. Because of this, access to production servers is limited, and installing profilers or measuring tools is not possible.

In short, MUST is a complex, multifaceted software system with different components and use cases. This, combined with the limited access to production servers, makes it challenging to formulate an accurate energy profile of the complete system. The second goal of this thesis is to apply the methodology to MUST to gain an understanding of how the methodology works in practice, and in the process learn about the energy profile of MUST.

Stakeholders at ESA and Solenix are particularly interested in the final part of this goal: MUST’s energy profile. In chapter 5 of this thesis, we apply the SMURF methodology to MUST to gain insights into its energy profile, and to formulate actionable recommendations to increase the energy efficiency and sustainability of MUST.

3 Related Work

Attempting to understand, quantify, or learn from the energy consumption of software systems is not a new idea.

There are two distinct areas in which energy measurements and energy profiling are commonly studied. The first is mobile devices, and the reason is clear: mobile devices run on batteries and battery capacity is often one of the main limiting factors. Researchers have studied the problem from many directions. For example, a number of ‘energy patterns’ have been identified, catalogued in works like Cruz and Abreu [6], that often relate to the (mis)use of energy-intensive hardware components like the screen or wireless connections. More closely related to our purposes, Rua and Saraiva [24] present a tool called *E-MANAFA*, an automated energy profiler for Android applications. Gregório et al. [14] take a similar approach, but look for known energy patterns in decompiled code.

The second is artificial intelligence, and again the reason is clear: AI uses a tremendous amount of energy. We refer the interested reader to a review of the subfield by Verdecchia, Sallou, and Cruz [28].

Research in these two areas is highly specific, and hard to generalize to other types of software systems.

On the macro scale, there have been attempts to assess the environmental impact of the ICT industry as a whole. Freitag et al. [13] provide a valuable review and critique of this ever-growing body of work, finding that the ICT industry accounts for 2.1% to 3.9% of global greenhouse gas emissions, and that this number is likely to grow in the future. Notably, their work is from 2021, a year before OpenAI released ChatGPT[1]. Though they take into account growths in AI investments and energy usages, they could not have foreseen the explosion in consumer usage of large language models and other power-hungry generative AI models.

Zooming in slightly, Simon et al. [26] analyse the sustainability of single software systems using a holistic approach that focuses on the whole life cycle of the system, and on a broad range of sustainability aspects. Using a life cycle analysis (LCA), they focus on the environmental impact of software over its entire life cycle, ranging from management to design, and from implementation to deployment.

Porras et al. [23], while providing a less detailed view of the software life cycle, focus more on the different dimensions of sustainability: economic, technical, social, personal, and environmental. Their goal is “to identify potential effects of an IT company’s (software) product and how such identified effects could be linked to the company focus.” They use the ‘Sustainability and Awareness Framework’ (SusAF) [10], a framework that can be used to guide conversations and workshops on the effects of software systems. In such a workshop, Porras et al. [23], discuss the environmental impact of a software system with two product owners of that system, using the questions from SusAF.

These methods work well to get a broad overview of the effects of a software system, and as a starting point for discussion and raising awareness. However, to get a detailed understanding of the energy

[1]`https://openai.com/index/chatgpt/`

consumption patterns of a single system, and to formulate actionable recommendations, these methods are too broad. Additionally, to get a grip on these higher-level effects, the studies often rely on well-crafted hypotheticals instead of on concrete data. Our work focuses on a detailed study of the energy consumption of a specific system. By basing this investigation on real usage data, our work avoids the hypotheticals and guesstimates that are necessary in these broader approaches.

In the first of a two-part series of articles, Coroama et al. [4] describe flaws in and propose solutions to current methods for assessing the environmental effects of ICT. In this first part, they evaluate single ICT services. Their methodology is based on a ‘comparative Life Cycle Analysis (LCA)’. The idea is to perform an LCA on an ICT service and on a reference activity representing the situation without ICT. By comparing these two, it is possible to assess the footprint of the ICT service.

In the second part, Bergmark et al. [1] turn to multiple services and companies. They take particular care to evaluate indirect effects of ICT systems, such as substitution effects, where a service is replaced by an ICT version, and the rebound effect, where increased efficiency leads to more consumption.

Note, however, that these LCA’s are necessarily based on hypotheticals. One of the two situations, either the one where the activity is replaced by or uses ICT, or the situation without ICT, will always be a hypothetical situation as the two cannot coexist. Often, LCA’s are performed to assess the *potential* of a ICT system, in which case both situations are hypothetical. Both articles in the series recognize this, but cannot solve the problem. In our work, we spend considerable effort to ensure that our analysis is grounded in reality by basing our analysis on real-world usage data.

Zooming in much further, there have been attempts to evaluate software at the level of specific functions, algorithms, or design patterns.

For example, Feitosa et al. [12] evaluate the energy consumption of different architectures and code design patterns – some of which were designed explicitly to tackle energy-related issues like dynamic retry delays, and some of which have energy side effects, like the ‘Template Method’ pattern. Connolly Bree and Ó Cinnéide [3] specifically evaluate a single design pattern, the ’Visitor’ pattern, and find it to have a negative energy impact.

Bruce et al. [2] and Dorn et al. [9] evaluate the energy efficiency gained by code optimizations found by a genetic algorithm. Both find that relaxing the requirements on output quality, by allowing approximations, can dramatically reduce energy consumption. Similarly, Oliveira et al. [19] explore the energy performance of different Java collection implementations, and use the knowledge they gain to build a tool that recommends more energy efficient collections.

While these studies offer valuable insights, their detailed scope and reliance on synthetic test cases limit generalization and applicability to broader and real-world software systems. Our methodology improves on these usability issues by zooming out slightly to a systems level and by basing measurements strictly on real-world usage data.

On the level of complete software systems, the level that this thesis is also concerned with, significant work has been completed.

Jagroep et al. [15] propose an energy profiling methodology to evaluate different versions of the same product. They use both hardware measurements – by attaching a power meter between the device and the wall outlet – and software measurements with an energy profiler. The system they test is a document generator. To test it, Jagroep et al. [15] generate a number of documents on multiple versions of the system and measure the energy consumption. Then they analyze these results, trying to find an explanation for the differences in energy consumption in the source code changes.

In their article, Jagroep et al. [15] only analyze the core functionality of the product they test. This makes sense, but limits their methodology to systems that show a single, clear, core functionality that dominates the energy consumption. In addition, they rely on a sensible, but arbitrary decision of what use case to test. This works because they are comparing the energy consumption of the same use case across different versions. However, this too limits the methodology: getting an overview of the energy consumption of a complex, multifaceted system is not feasible in this way. In contrast, our methodology was designed, from the ground up, to be able to deal with the complexity of these multifaceted systems.

Using a very similar approach, De Macedo et al. [8] compare a number of use cases of browsers (streaming video on YouTube and Twitch, and browsing social media). Instead of comparing two versions of the same software, they compare two browsers: Firefox and Chrome. They use only software measurements, but their approach is otherwise quite similar to Jagroep et al. [15].

After comparing these three use cases between browsers, they attempt to crown an overall most energy-efficient browser. To do this, they write a 'global script' that simulates a typical browsing session: log in, read some news, browse around, watch some videos, etc. They find that Chrome is overall more energy efficient.

The fact that this use case was arbitrarily decided limits the generalisability of this result. It is not possible to be sure that this is how a browser is typically used – in fact, it is likely not. We argue that, while the results convincingly show that Chrome is more energy efficient for this use case, it is impossible to say based on this work that Chrome is more energy efficient overall. To make general claims about the relative energy consumption of complex, multi-use systems (like browsers), much more care needs to be taken in the development of the 'typical' use case under test.

Pereira et al. [20], whose work uses a similar approach but contributes a statistical method of relating increases in energy consumption to specific lines of code, even explicitly argue: "The quality of the tested scenarios is also important because only with tests which stress the components with different inputs replicating real-world scenarios, can one extract reliable information." Nonetheless, they do not elaborate on the method to create such tested scenarios.

Instead of relying on this 'user story approach', in which authors use their own judgement to decide what use cases to test, Danglot, Falleri, and Rouvoy [7] and Pereira et al. [20] rely on existing software tests to achieve a similar goal: compare the energy consumption of software systems across versions. Using these tests eliminates the need to spend time developing other usage scenarios and automating them. This works, but does not mitigate the problems mentioned above.

Our methodology improves on these issues by basing the formulation of the 'typical use-cases' on actual usage data, instead of on developer tests (that were not crafted for this purpose), or arbitrary use-cases formulated by stakeholders or researchers.

4 SMURF

In this chapter, we present the SMURF methodology: a framework for creating energy profiles for complex software systems. The methodology has a simple goal: to understand the relative energy consumption of different components of a complex, real-world software system.

Central to this methodology is the formulation of a 'typical use case'. The methodology uses real-world data to formulate this use case, which ensures an accurate depiction of energy use across various software components. This keeps the methodology relevant even as systems grow more and more complex. This typical use case is based on server logs. Additional usage data may be incorporated, but this is dependent on the specifics of the system being evaluated.

Server logs often offer rich, detailed, and structured information that can be directly used to analyze system behavior and formulate typical use cases, but this does limit the methodology to systems that produce server logs. It's important to note that system logs are commonly available in most modern software environments. Many systems naturally produce logs to monitor performance, diagnose issues, and ensure security, making this restriction not as significant as it might initially appear. And it has major benefits: by basing the typical use case on server logs, we ensure that the tested scenario is grounded in reality and that our methodology stays relevant on complex systems with many different components.

One key challenge in performing energy measurements is that access to production is often – rightfully – limited. Because software is often critical to the processes it supports, changes or disruptions could impact the service or cause unintended effects. Because of this, production environments are often locked down to ensure security, stability, and reliability. Energy measurements tend not to be high enough priority to warrant an exception. Even if access is available, making changes to production environments just to perform energy measurements is often not feasible.

The SMURF methodology avoids needing to make changes to production environments, by leveraging simulations and test deployments to recreate a typical usage scenario, and subsequently measuring energy consumption. This minimizes the risk of interference with live environments while still yielding valuable insights.

Stakeholder participation is integral to the methodology. On the one hand, insights from developers and users are used throughout the methodology to formulate interesting questions, and to explain results. On the other hand, the results serve as an educational tool: encouraging discussions on sustainability in software design and improving understanding of the energy usage of the system. The methodology's step-by-step approach enables detailed analysis of energy use, promoting informed decision-making.

In summary, the methodology has the following four properties:

It gives insights into the energy consumption of a complex system: The methodology produces an energy profile: an overview of the relative energy consumption of the different components of the system. The formulation of a typical use case, grounded in real usage data, ensures the methodology stays relevant and effective even as system complexity increases. This fundamental aspect replaces user stories or arbitrary use cases with empirical data from real-world usage, offering a more accurate and practical depiction of energy consumption across various software components under typical conditions. The tested scenario is mainly based on server logs, incorporating additional data where available and necessary.

It is broadly applicable: The methodology is not dependent on the exact format of the logs. This makes it broadly applicable to a wide range of systems and architectures. Server logs are commonly available across a wide range of systems. Through simulations, test deployments and the formulation of a typical use case, the methodology sidesteps the need to install tools on or make other changes to production environments, which allows the methodology to also be applied to critical systems in which access to production servers is limited.

It supports informed decision-making: The methodology facilitates the analysis of engineering questions, such as the energy costs or benefits of specific optimizations and settings.

It engages stakeholders: Developers and users are involved to leverage their insights and expertise and to amplify the impact by educating them about the findings and enhancing their awareness of software sustainability challenges.

The rest of this chapter first presents an overview of the five steps of the SMURF methodology, and then explores each one in more detail. We use a case study and stakeholder discussions to evaluate the methodology, and in the last section of this chapter, we discuss this evaluation in more detail.

4.1. Summary of the Steps of the Methodology

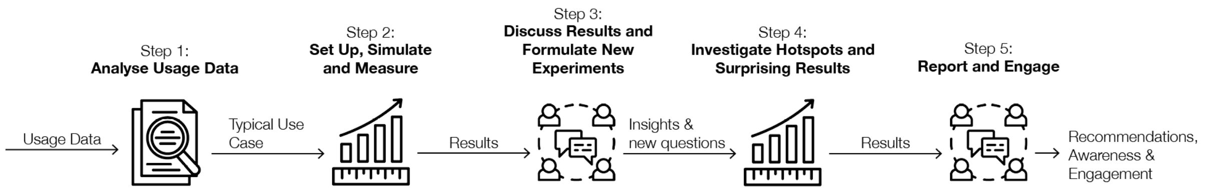
Step 1:
Analyse Usage Data
Step 2:
Set Up, Simulate and Measure
Step 3:
Discuss Results and Formulate New Experiments
Step 4:
Investigate Hotspots and Surprising Results
Step 5:
Report and Engage
Usage Data
Typical Use Case
Results
Insights & new questions
Results
Recommendations, Awareness & Engagement

Figure 4.1: A graphical representation of the SMURF methodology.

The SMURF methodology has the following five steps, which are explained in much more detail in the coming sections. Figure 4.1 shows a graphical representation of the steps of the methodology.

Step one – analyse usage data: Take stock of the available usage data, and using it, formulate a typical use case over a representative period, that describes how much each component of the system is used, under typical use, in that period.

Step two – set up, simulate, and measure: Set up a test deployment of the system, and use it to simulate the typical use case formulated in step one. Measure the energy consumption of the system throughout the simulation.

Step three – discuss results and formulate new experiments: Evaluate the results, closely involving different stakeholders. Use their knowledge to gain an understanding of what results are surprising, and what aspects of the system could benefit from a deeper investigation. Formulate new experiments to answer those questions.

Step four – investigate hotspots and surprising results: Perform the experiments formulated in step three.

Step five – report and engage: Report the results to various stakeholders. Use their insights to deepen the context around the results, and use the results to engage all kinds of different stakeholders with sustainable software ideas.

4.2. Step 1: Analyse Usage Data

In the first step, the goal is to formulate a typical use case. This typical use case gives the answer to the question: which components of the software system are there, and how much is each component used? While the first step is always to talk to experts to get a rough overview, the detailed understanding is based on usage data.

This step starts, then, by taking stock of the available data. This too requires expert input – they will know what data is available and may contain valuable information. Server logs, the outputs of application performance monitoring tools, and similar data files provide excellent input for the formulation of the typical use case. If this data is not sufficient or hard to interpret, expert interviews can be used to figure out the details or the correct interpretation.

With the available data defined, selecting a study period is next. Does the study focus on system usage over a day, a week, or an entire year? The choice hinges on system usage patterns – whether usage is consistent daily or correlated with the work week, or if certain seasons see more activity. Data availability is a factor too; analyzing a year of usage requires a year's data.

Next, the data needs to be categorized into the different components of the system. Defining what components there are, and how small or large to make each component, is a crucial step towards making the results of the research useful. Careful thought, and – we hope you've spotted the trend – expert opinion are a necessity to make the right choices here.

For example, consider a simple grocery list app. The work done by this system may be split into three components: creating new grocery lists, retrieving grocery lists, and 'all the rest of the work'. Alternatively, it could be split into tens of different components – user authentication, editing lists, creating lists with images or without, etc. What choices are right depends on the specifics of the system under test and the specific goals of the research.

This usage data forms the typical use case: it shows, for each of the components, what work is being done in the selected period. If major components are somehow missing from the data, for instance because they don't log to the same log files, the search for usable data goes on. In the worst case, expert input can be used to fill the gaps.

4.3. Step 2: Set Up, Simulate and Measure

With the usage data categorized and a period selected, we need a way to simulate and measure the typical use case.

For this, we need a test deployment of the system, that has two properties:

It is measurable We follow Cruz [5] and Mancebo, García, and Calero [16] for best practices on energy measurements. We need a system that runs Energibridge [25], the software under test and as little as possible otherwise.

It does a reasonable amount of work Imagine, that in our simple grocery list app example, the logs indicate that a user has requested all grocery lists. In our simulation, we perform this request and measure the energy consumption. But it is not so simple: if the database is empty, the request will be much less work than in the real system. Similarly, if we fill the database with hundreds and hundreds of gigabytes of grocery lists, the request will be much more work compared to the real system. To make the system do a reasonable amount of work, we need to create a reasonable amount of data, set reasonable system variables, etc. Exactly what variables, parameters and data needs to be set - and to what values - is completely system dependent, and requires very careful consideration.

With this test deployment, it is now possible to make the system do, for each of its components, an amount of work that is representative for the amount of work the system would do in the representative period defined in step 1. By doing this and measuring the energy consumption, we create an 'energy profile', a plot that shows the relative energy consumption of each component, under typical use. These are not absolute numbers. Energy consumption is dependent on so many factors, from hardware choice to system variables to usage, that it is not possible to use the resulting numbers – measured on our test deployment – to draw any conclusions about the total power consumption of the production environment.

However, the energy profile will give a clear overview of the relative energy consumption of each component.

4.4. Step 3: Discuss Results and Formulate New Experiments

In this step, in which we take the results from step 2 and discuss them with stakeholders, the focus is on two questions:

- What are the hotspots?
- What results are surprising?

The answers to the first question will be fairly clear from the energy profile: just look for the components with the highest relative energy consumption. However, the fact that a result is 'surprising' is necessarily based on existing expectations. The most interesting of these expectations are the expectations of experts, users, and developers. This step, therefore, involves talking to them, presenting the results, and gathering their ideas of what is surprising.

Both the hotspots and the surprising results are likely to raise further questions – conceptual or engineering based.

As a first example, section 4.2 mentioned that splitting the software into its components is a complex, manual task. If a certain component consumes the majority of the energy, does it make sense to split it apart further, and re-investigate different sub-components?

As a second example, a hotspot or a surprising result naturally prompts the question: what measures can we take to reduce energy consumption? Can we implement some measures and see if they work? Or can we dive deeper into the reasons for the hotspot? What factors contribute to the concentration of energy consumption in this component?

The goal of this step is to formulate and prioritize these questions.

4.5. Step 4: Investigate Hotspots and Surprising Results

Step 3 left us with a new set of deeper and more concrete questions. In this fourth step, we use these questions as a basis for formulating new experiments. These experiments can aim to provide a deeper understanding, or to answer engineering trade-off questions.

Based on the existing test infrastructure, this step involves performing these experiments, and analyzing the results. These experiments will have a slightly more concrete goal, and are – more so than the experiments in step one – focused on providing actionable data or engineering recommendations.

4.6. Step 5: Report and Engage

This fifth and final step is a more flexible one. In essence, there are four goals, and the exact method of achieving those goals is dependent on the specific scenario.

The first goal is obvious: to present the results of the experiments both from step two and step four, to developers, users, and product owners.

The second goal is to discuss the meaning of the results with the stakeholders. Steps two and four produce energy measurements. However, these energy measurements often need more context, and only the stakeholders can provide this context. They will understand in much more detail why they made certain choices and why things are the way they are. This data is invaluable in order to formulate sensible recommendations.

The third goal is to use the results and the broader context provided by the stakeholders to formulate recommendations. Based on the deeper context provided by the stakeholders, these recommendations can be formulated in a much more realistic, actionable way.

The final goal is to use the entire process of the SMURF methodology, and the analysis of the system to engage users, product owners, and developers with the ideas of sustainable software. Ultimately, the goal is to create a community that not only adopts these practices but advocates for them in future projects.

Note that these goals don't have to be achieved sequentially. They can often be pursued in parallel or revisited as needed. The flexibility in approach can adapt to changes or new insights gained during the process. In particular, presentations and more interactive meetings can be used, at the same time, to engage with sustainable software ideas, present results and recommendations, but also to place results in a broader context. As each goal is interconnected, progress in one area can help in another, strengthening the overall outcome.

4.7. Evaluation Methodology

To evaluate the effectiveness and applicability of the SMURF methodology, we perform a case study: in the next chapter of this thesis, we apply the SMURF methodology to the MUST system that was introduced in chapter 2. In doing so, we are not only learning about the energy consumption of MUST – in fact, the primary goal of this process is to evaluate the SMURF methodology itself.

Specifically, our evaluation is guided by the following five questions:

Effectiveness of the methodology: How effectively does the SMURF methodology estimate an energy profile of a software system based on empirical usage data and without access to production servers?

Stakeholder influence: How does stakeholder involvement influence the SMURF methodology?

Awareness and engagement: How does the information generated by the SMURF methodology influence awareness of and engagement with energy consumption and sustainability?

Applicability: To what extent can the SMURF methodology be applied to various software systems?

Limitations and challenges: What limitations or challenges arise when implementing the SMURF methodology in a real-world scenario?

The main source of information in answering these questions is the case study itself. In addition, step 5 of the SMURF methodology involves discussing the results with stakeholders. These discussions were co-opted to also include a reflection on the SMURF methodology itself. This allows us to gather valuable feedback on the methodology's applicability and effectiveness. Developers, users, and project managers provided their insights into the methodology, highlighting areas of strength and potential uses and improvements. This collaborative reflection enriches our understanding of the SMURF methodology.

This comprehensive evaluation, based on the case study and the stakeholder discussions and guided by the five questions listed above, will not only validate the SMURF methodology but also inform future research and development efforts.

5 Applying SMURF

Applying the SMURF methodology has two distinct goals. The primary goal is to learn, in practice, what the uses, limitations, and difficulties of applying SMURF are. In the main discussion of this thesis (chapter 6), we discuss this. This chapter is about the secondary goal: to gain a deeper understanding of the energy consumption, hotspots, and patterns of the MUST system by applying SMURF. In this chapter, we apply the five steps of the SMURF methodology to the MUST system and present the results.

5.1. Step 1: Analyse Usage Data

Now that we have a system to test, our first goal is to understand how it is used. As explained in section 4.2, we mainly base our understanding of a typical use case on logs and other usage data. We do use conversations with experts and user stories to get a basic overview. From this, we know roughly that the system is used to store calibrated parameter data, and that users will request this data from the server and visualize it. In addition, users will probably perform a variety of other tasks, like user management and anomaly detection.

But to quantify this image, and get a more detailed overview, we rely on real usage data. Then the rest of this section describes what data is available, and how we use it to understand what a 'typical use case' is.

5.1.1. Available data

We will use data from an Earth Observation Mission at ESA, which agreed to share their instrumentation.

From MUST's production environment, we have access to the following data, which we will use to formulate a typical use case:

Logs There are a variety of systems that create log files, and all of these are available. Most notably, the server produces `localhost_access_log.txt` files, one for each day, that log all requests to the server. It is these logs that we will base the majority of our use case on. The server keeps the last month of these log files.

Crontab The server's crontab file gives insights into all the scheduled work that runs on the server. Most notably, these include the import tasks.

A day of import data Because the imports do not work over network requests, we cannot use our the logs to figure out what a typical use case is for importing. However, we do expect this to be a rather significant part of the total work. The system stores the last day of files imported by the importer in the 'processed' folder. These files are also available, and give a reasonable estimate of the amount of work done in this step.

Glowroot MUST runs an application performance monitoring tool called Glowroot[1]. The tool is a performance instrumentation and profiler web app, that supports investigation once software anomalies

[1]`https://glowroot.org/`

are raised. It also logs requests, as well as more detailed information on things like CPU time used by certain requests. These Glowroot databases are available. However, because Glowroot is not widely available on other systems, the Glowroot database is tricky to parse on a separate machine, and there is not much useful information in there that is not in the logs, we decided against using them.

In the end, we base our typical use case on a week of data, from August 12th, 2024 to August 18th, 2024. We use a week of data because the number and type of requests varies highly from day to day (Sunday tends to be, for example, much quieter than the Monday after), but is relatively constant week to week. Figure 5.1 shows the number of requests per hour, over a period of four weeks (from August 5th to September 1st). This figure shows a clear repeating pattern with a period of a week, with few requests during nighttime and quiet days on the weekends.

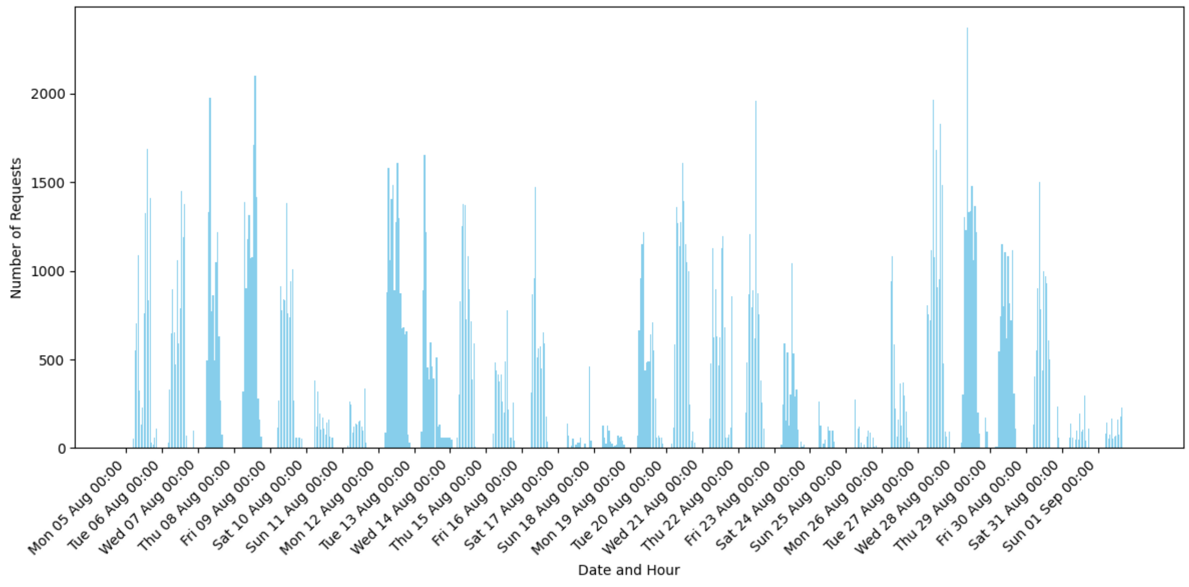

Figure 5.1: The number of requests made to MUST over a four week period.

5.1.2. Categorizing requests

Given that the goal is to get an overview of the relative energy consumption of different components of the MUST system, let us turn to organizing the data we have into these different components. The natural way to do this is by endpoint or shared groups of endpoints. Using simple matching, we assign each request to a group. Table 5.1 shows the 26 categories of requests, and table 5.2 shows an example url for each category.

Each of these categories requires careful individual work. This work comes in three parts: setting up the right system configuration, databases and data in those databases, modifying the URLs to work with the test deployment, and ensuring that the configuration is such that the request does a reasonable and representative amount of work. We will discuss a brief motivating example here, but all details are in section 5.2.

Take for example a request in the 'scripts' group: a get request to `/scripts/568/run`. To measure this request, we need to make sure that there is a script with id 568 in the database, and that running it takes an amount of work that is somehow representative of the amount of work a script usually takes. This would involve discussions with experts and users, or the analysis of a group of existing scripts. In addition, scripts often rely on other data in the database, so we would need to make sure that data exists too.

However, because they are not data intensive and are not used often, as shown by the number of requests in the scripts category, we expect the requests in the 'scripts' group to have only a very minor impact on the total energy consumption. Because of this, we decided to leave the scripts group out of the analysis.

Table 5.1: The categories of requests, with a matching statement, the amount of requests in each category and the purpose of the requests in the category. In each matching statement 'p' is the path of the request. Categories below the horizontal line were left out of the analysis for various reasons explained in section 5.1.2.

Category	Matching statement	Amount	Purpose
drmust	p.endswith("drmust/numresults")	19853	Check if DrMUST has new results
data	re.search(r'/dataproviders/.+ /parameters/data')	4513	Get actual time series data from MUST
tables	re.search(r'/dataproviders/.+ /tables')	1604	Get metadata on available tables
web_tables	/web/tables in p	788	Get table data
parameters	re.search(r'/dataproviders/.+ /parameters')	589	Get parameter metadata
aggregations	re.search(r'/dataproviders/.+ /aggregations')	520	Get aggregation metadata
metadata_treesearch	/web/metadata/treesearch in p	497	Search the parameter tree
dataproviders	p.endswith("dataproviders/") or p.endswith("dataproviders")	255	Get metadata on available dataproviders
metadata_tree	/web/metadata/tree in p	179	Get the parameter tree
category	/web/category in p	150	Get data for a 'Textual Calibrated Plot'
gui	/web/gui in p	123	Get the GUI
tableparams	/web/tables/params in p	121	Get the parameters attached to some table assets
plugins	/web/plugins in p	117	Get metadata on available plugins
not_mustlink	mustlink not in p	2103	Group of requests that do not fall under the mustlink API, usually for static files like stylesheets or icons.
usermanagement	/usermanagement/ in p	285	All requests related to users
profiles	/profiles/ in p	154	Metadata on profiles
login	/auth/login in p	147	Login POST requests
appsettings	/appsettings/ in p	98	Get application settings
novelty	/analysis/novelty/ in p	95	Requests related to novelty detection
scripts	/mustlink/scripts in p	88	Get or create or run scripts
metadata	/metadata/tree/ in p	49	Get the metadata tree
table_data	re.search(r'/dataproviders/[^/]+ /table/[^/]+/data')	31	Get table data
content	/content/ in p	8	Get user content
chart	/mustlink/chart in p	4	Export charts
table_export	re.search(r'/dataproviders/[^/]+ /table/[^/]+/export')	3	Export tables

Table 5.2: An example URL for every request category

Category	Example URL
drmust	`/analysis/drmust/numresults`
data	`/dataproviders/generic/parameters/data?key=name&values=UST01830&from=2024-07-28_05:00:02&to=2024-08-1205:00:02&calibrate=false&mode=SIMPLE`
tables	`/dataproviders/generic/tables`
web_tables	`/web/tables/generic/ev.ob?dateFormat=fromTo&from=2024-08-11_05:54:02&to=2024-08-12_05:54:02&filterKeys=&filterValues=&mode=TIME`
parameters	`/dataproviders/generic/parameters?type=undefined&key=name&value=22962`
aggregations	`/dataproviders/generic/aggregations?key=name&value=484`
metadata_treesearch	`/web/metadata/treesearch?field=name,description,id,unit&text=cjt01871&missions=generic`
dataproviders	`/dataproviders/`
metadata_tree	`/web/metadata/tree?ds=0&id=0`
category	`/web/category/dataproviders/generic/parameters/data?key=name&values=33373&from=2024-08-11_07:32:20&to=2024-08-12_07:32:20&calibrate=false&aggregationFunction=None&aggregation=None&aggregationValue=1&compressionError=0&chunkCount=-1&delta=0`
gui	`/web/gui`
tableparams	`/web/tables/params/generic/ev.ob?elementId=ob.44723&ssc=3523&timestamp=2024-08-11_20:17:20.783`
plugins	`/web/plugins`
not_mustlink	`/styles.css` or `/favicon.ico`
usermanagement	`/usermanagement/userinfo`
profiles	`/profiles/`
login	`POST/auth/login`
appsettings	`/appsettings/`
novelty	`/analysis/novelty/missions`
scripts	`/scripts` or `POST/scripts`
metadata	`/metadata/tree/sen2_deprecated?id=sen2_deprecated`
table_data	`/dataproviders/sen2/table/tc/data?filterKeys=name&filterValues=lcc&from=2024-08-06_22:00:00&to=2024-08-14_01:59:59&representation=SIMPLE&dateFormat=fromTo`
content	`/content/56`
chart	`/chart/export/png?from=2024-08-12_12:00:00&to=2024-08-12_22:00:00&profileJson=OMITTED`
table_export	`/dataproviders/sen2and/table/and.sys0001/export?dateFormat=fromTo&from=2021-08-12_07:15:51&to=2024-08-12_07:15:51&mode=brief&filterKeys=&filterValues=&authorization=OMITTED`

We included all categories that were easy to set up, and all categories that we expected to have a significant impact on the energy consumption in our analysis. The categories in the final group – hard to get working, but with very little impact – were left out. Table 5.1 shows the categories that were left out below the horizontal line. Figure 5.2 shows the number of requests in each category.

Two categories are worth mentioning quickly already: by far the largest of these categories is the 'DrMUST' category, consisting of 19853 requests to the `/analysis/drmust/numresults` endpoint. DrMUST is a tool for anomaly investigation and pattern detection in data. It runs in the background, and all these requests are polls to see if there are new results available. The largest category that was left out is 'not_mustlink', which consists of 2103 requests, mainly related to retrieving static materials like stylesheets and icons.

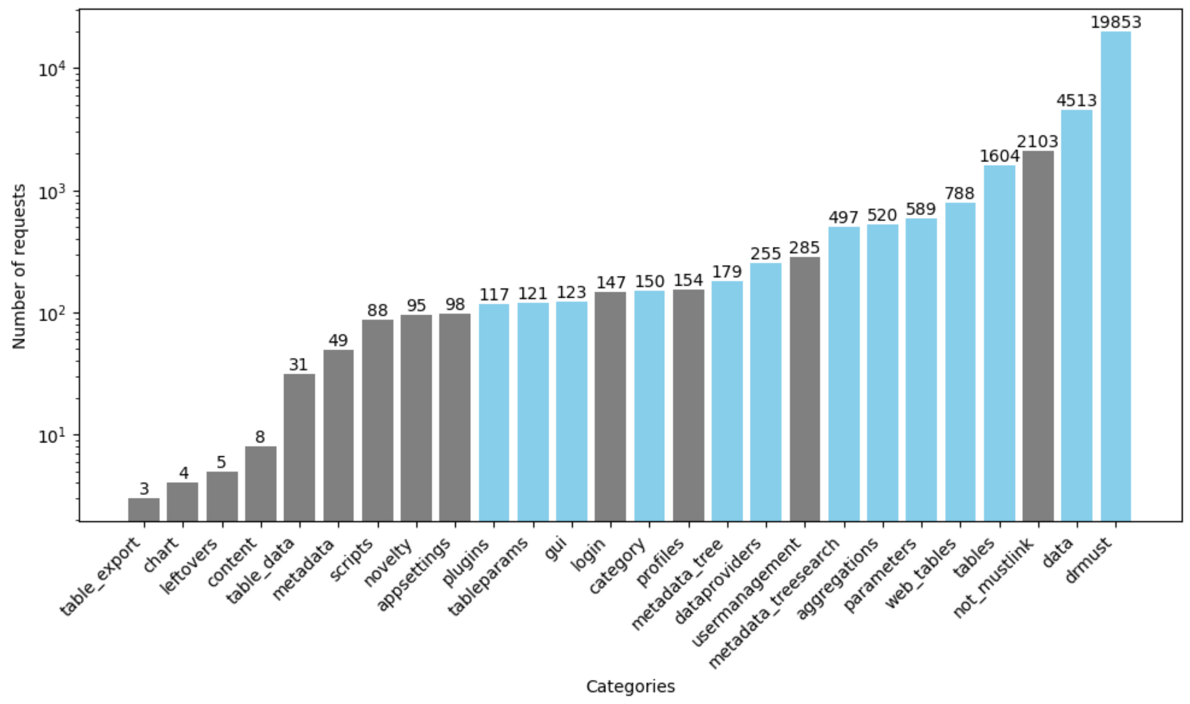

Figure 5.2: The number of requests in each category. Blue categories were included in the typical use case, grey categories were excluded. Note the logarithmic scale.

5.1.3. Import analysis

The one main component of MUST that is impossible to simulate based on API requests is the importations, because it is not triggered through the API. Instead, the importer runs at set intervals. These are defined in the server's crontab[2] file.

The system is scheduled to run the importer every ten minutes, at 3, 13, 23, 33, 43, and 53 minutes past the hour.

Low Earth Orbits last about 100 minutes[3]. In order to download data from the satellite, ground stations need to be able to physically see the satellite. While satellites pass over numerous ground stations during an orbit, they typically expect only a single download pass per orbit.

This means that, roughly every 100 minutes, the last 100 minutes of data from the satellite becomes available. Meanwhile, the importer runs without data 9 times every 100 minutes. There is some nuance

[2]Cron is a Linux utility for scheduling jobs. The crontable (crontab) defines scheduled jobs. `https://man7.org/linux/man-pages/man5/crontab.5.html`

[3]There is an incredible fact here that is too interesting not to share. If you dig a hole all the way through the earth, and drop a coin through it, it will accelerate to the middle of the Earth, and come out on the other side, reaching the exact same height on the other side of the Earth as where you released it (assuming no air resistance and other nuisances like the impossibility of drilling a perfect hole through the earth). The time it takes for that coin to fall through the Earth, come out the other end, fall back down and return to its starting position is exactly the same amount of time an orbit at that same height takes. Incredible.

here: schedules might change, systems might have failures, or other anomalies might occur. Because of this, the 100 minute cycle is not entirely predictable. Still, it is a good approximation.

We have access to one day of data. Given the consistent and predictable nature of telemetry data generated by the satellite, we can extrapolate from one day of data to a week. Hence, we will measure the energy consumption of importation in one day and multiply it by seven.

5.1.4. Typical use case

Combining this import cycle and the categorized requests creates our typical use case. For the period from August 12th to August 18th, 2024, we analyzed the logged requests, which were categorized into the system's various components. Additionally, we dissected the importation process, utilizing a day of data that was extrapolated to represent a week. Combining these data points provides a comprehensive overview of the system's activity in the specified time frame.

5.2. Step 2: Set up, Simulate and Measure

This step of the SMURF methodology involves two major sub-steps: setting up a deployment of MUST that is capable of simulating the typical use case scenario, and then simulating and measuring that use case.

Let us start with the second step: simulating and measuring the use case that we've distilled in the previous section. MUST runs on a specialized server known as the 'Greenserver'. This is a dedicated server that is kept as clean as possible from background tasks, to ensure accurate energy measurements. With MUST running, we take each category of request that appears in the typical use case, perform all requests in that category, and measure the energy consumption using a tool called Energibridge[4]. Similarly, we perform all the imports in the typical use case, and measure in the same way.

To get accurate results, each request needs to trigger a reasonable amount of work. As an example, if parameter 'A' is requested, from November 1st 00:00 to November 2nd 23:59, the system must have data for parameter 'A' for those two days - or it will crash. In addition, it should have a representative *amount* of data. Based on expert input, we decided to give parameters a sample every second. In our example, this gives 172740 data points for this request. These choices, their justification and the alternatives considered will be discussed further in section 5.2.1.

This goal of making requests trigger a representative amount of work forms the basis for the choices made in setting up a deployment of MUST. MUST runs in a containerized setup in a number of Docker containers that are networked together. After setting up the necessary database structures, we analyze the requests in our typical use case to figure out what data is necessary and import it into the databases. Then, we make some changes to the request URLs in our typical use case to ensure they work with the test deployment.

The following two subsections explain this process – first the setup, and the simulation – in much more detail. The details are interesting for reproduction and further study, or someone just generally interested in all the details, but the casual reader is advised to skip straight to section 5.2.3, in which we describe the results of measuring this simulation.

5.2.1. Test deployment setup

MUST runs as a composition of docker containers, split into two groups. In the `generic-data-store` group, the most significant container is the `mariadb` container, which contains the main MUST database. Also in that group are the `archiver` and `rabbitMQ` containers. RabbitMQ is a message broker, which makes it easy to programmatically push data into the system through the `archiver`. In the `client` group, the `ng-webmust` container provides the front-end, the `mustlink` container the API. the `user_content_mariadb` container the user and authentication databases, and the `ng-webmust-docs` container the documentation.

Table 5.3 shows these containers.

[4]Energibridge is a multi-platform energy measurement utility, created by Sallou, Cruz, and Durieux [25]

Container	Description
generic-data-store	
mariadb	The main MUST database
rabbitmq	The RabbitMQ frontend
archiver	The RabbitMQ <-> database interface
client	
ngwebmust	The main MUST frontend
mustlink	The main MUST API
user_content_mariadb	A secondary MUST database with user data
doc	The MUST documentation

Table 5.3: The docker containers for MUST

General setup

First, we need to create a mission configuration. We use a copy of the configuration of the Earth observation mission, for which we also have data. To do this, we import an SQL dump of the configuration. For the same reasons, we also import the matching version of the `tchistory` table.

Lastly, we need to add aggregation options. The data request has a parameter `aggregation`, specifying the time period for data aggregation (e.g., daily, weekly, monthly). Mustlink's `spring-assembly.xml` defines these aggregations like so:

```
<bean
    class="esa.esoc.ops.hsc.dataprovider.aggregation.SimpleAggregation">
    <constructor-arg name="name" value="Minutes" />
    <constructor-arg name="periodMillis" value="60000" />
</bean>
```

If a request has a value for `aggregation` that is not defined in this XML file, the request will crash. All aggregations that appeared in the logs were added to this file. In our case, the missing aggregations were 'minutes', as seen above, as well as 'days', with a `periodMillis` of 86400000.

Creating data

There are two possible methods to import data into the MUST database.

The first of these is through the RabbitMQ interface. This method is the easiest of the two. The queue accepts a JSON-like message that specifies (parameter, value) pairs at a specific moment in time, of the following form:

```
    datetime: <timestamp>,
    <P1>: <P1Val>,
    <P2>: <P1Val>,
    ...
    <Py>: <PyVal>,
```

The `archiver` then handles correctly adding these pairs to the database. While this works, it is too slow to create the millions of necessary parameters.

The second method is to directly add the data to the database through MariaDB's `LOAD DATA INFILE` instruction. It can read data from a file and insert it into a table much faster than individual INSERT statements because it minimizes the overhead associated with each insert operation, as well as reducing the number of INSERT statements by reading data in bulk.

We spin up a separate `setup` container, which creates, for each parameter we want to create data for, a CSV file with (datetime, value) pairs.

Before importing, the `setup` container needs to perform one additional step to ensure database integrity. The `parameter` table in the database contains metadata about the parameter. Using RabbitMQ, the `archiver` automatically handles this, but using this method we need to take care of it ourselves. The following SQL statement creates that metadata ({p} is the name of the parameter).

```
INSERT INTO parameter (PNAME, PDESCR, SSNAME, UNITS, DBTYPE, PTYPEID) VALUES ("{p}", "",
"general", NULL, "double", 1)
```

MariaDB now automatically assigns the parameter an ID, which is used in the import.

The container then outputs a csv file to a docker volume, which is shared with the `mariadb` container. This shared volume removes the need to send data over the (local) network. Then the container executes the following `LAOD DATA INFILE` instruction on that file, passing in the new ID.

```
LOAD DATA INFILE '/setup/{p}.csv' INTO TABLE doubleparamvalues FIELDS TERMINATED BY ','
(datetime, value) SET PID={pid}
```

This second method is orders of magnitude faster. For our initial, exploratory experiments, we used the first method because of its ease of use. However, for larger scale experiments - and all the experiment mentioned in this thesis - we exclusively use the second method.

What data should we create?

Now that we know how we can efficiently create data in our database, a new question arises: what data should we create?

To address this question, we can subdivide it into four key subquestions:

For what parameters should we create data? We should create data for all parameters that appear in the logs. Specifically, parameters appear in requests in the categories *data*, *category*, and *parameter*.

For what periods should we create data? For parameters that are requested in the *data* or *category* groups, we need to figure out the period for which we should create data. If only one request requests data for a specific parameter, then we just use the period from that request. If multiple requests request a parameter, all overlapping periods are merged, and non overlapping periods are kept separate. For example, if the logs contain two requests for parameter A, one from March 1st to March 5th, and one March 10th to March 15th, we create data in two blocks. However, for parameter B, requested from March 1st to March 5th and March 3rd to March 8th, a single block of data, from March 1st to March 8th is created. The *parameter* group of request only gets parameter metadata; therefore, for parameters that are exclusively requested in this group, creating a single data point is enough.

How many datapoints should we create? Now that we know the periods, this question turns into a question of frequency. In practice, these parameters usually occur at a set frequency. Some parameters have a higher frequency than others: a satellite may record its position every second, but its temperature only once every minute. The frequency of the parameter - and consequently the amount of data - has a large influence on energy consumption, as we will explore further in chapter 5.3. Based on expert opinion, we determined that a frequency of 1 Hz is a reasonable estimate for most Earth Observation (EO) missions, while parameters for Planetary missions may be recorded at a lower frequency of 0.25 Hz. This choice is primarily driven by the spacecraft's capacity to store and downlink information. Higher data rates would require more onboard storage and faster downlink capabilities, which may not be feasible for all missions. Therefore, adopting a frequency of 1 Hz allows us to balance data granularity with the operational constraints of the spacecraft.

What should the values be? Because only the amount of data matters, we can use random values.

In summary, we will consider all parameters requested in the logs, determine the appropriate periods for data creation, and generate data for those periods, with one sample every second and random values.

Constructing experiment URLs

Directly executing URLs from the logs does not work, for a variety of reasons. Instead, we transform each URL into an 'experiment-url': a URL that represents the same request but that works on our test setup.

We need to point the URL at the MUST test deployment, by making our requests to `localhost:8599`, the port where mustlink is running.

Similarly, in the production environment, the mustlink API is hosted under `ng-webmust/mustlink` or `webclient-must/mustlink`. The test deployment hosts mustlink directly under `/mustlink`, so the `ng-webmust` and `webclient-must` prefixes are dropped.

Next, as a consequence of our data creation process, parameters are created with the 'name' specified in the logged URL, even when the logged URL requests a parameter by its 'ID'. Therefore, all instances of `key=id` are changed to `key=name`.

The production environment provides a number of different dataproviders. On the test deployment, the 'generic' dataprovider provides all the data, so all dataproviders in the requests are changed to the 'generic' data provider.

Lastly, some parameters can be calibrated. Usually, these are textual calibration, specifying, for example, that 1 means on and 0 means off, or that values between 100 and 1000 are good, and others are bad. These calibrations require extensive setup, spread over many different tables in MUST's databases. However, these calibrations aren't expected to contribute significantly to energy consumption. In the interest of time, we decided to turn off calibration in the requests.

Following this process, the url:

```
/webclient-must/mustlink/dataproviders/SEN2B/parameters
?type=undefined&key=id&value=10055
```

transforms into:

```
http://localhost:8599/mustlink/dataproviders/generic/parameters
?type=undefined&key=name&value=10055
```

And similarly,

```
/webclient-must/mustlink/dataproviders/SEN2B/parameters/data
?key=id&values=29851
&from=2024-08-11 15:01:47&to=2024-08-12 15:01:47&calibrate=true
&aggregationFunction=None&aggregation=None&aggregationValue=0
&compressionError=0&chunkCount=1191
```

transforms into:

```
http://localhost:8599/mustlink/dataproviders/generic/parameters/data
?key=name&values=29851
&from=2024-08-11 15:01:47&to=2024-08-12 15:01:47&calibrate=false
&aggregationFunction=None&aggregation=None&aggregationValue=0
&compressionError=0&chunkCount=1191
```

5.2.2. Measuring a typical use case

The test deployment is now set up to simulate our typical use case.

To perform energy measurements, we use a specialized server, colloquially known as the 'Greenserver'. The specifications of the Greenserver are shown in table 5.4. The Greenserver is purposefully kept as clean as possible from anything that could introduce noise or inaccuracies in the energy measurements: background tasks, external devices, external processes, etc.

We use energibridge, a cross-platform energy measurement utility, to perform measurements [25]. For each category of requests and for each import job, an 'orchestrator' python script spins up a docker container that performs all requests or imports in that task, and attaches an energibridge measurement.

We run all experiments five times, and the results are averaged. Before each experiment, there is a 30 second 'warm up' period to make sure that the system is in a stable state before starting measurements. The order of all experiments is also randomized to minimize biases.

To measure the energy consumption of data importation, we measure the empty import, and each of the 14 100-minute batches of data. In a day of usage, we thus have $14 \times 9 = 126$ empty runs, and 14 runs with different 100-minute batches of data. To extrapolate to a week of usage, we multiply the results by 7. As mentioned before, the amount of data created is consistent over time, and so this will give a valid approximation.

To measure the energy consumption of the requests, we perform all requests in each bin of requests.

Component	Specification
CPU	AMD Ryzen 9 7900X
GPU	MSI Geforce RTX 4090 VENTUS 3X 24G OC
RAM	Corsair DDR5 Vengeance 2×32GB 5600

Table 5.4: The specifications of the Greenserver

5.2.3. Results

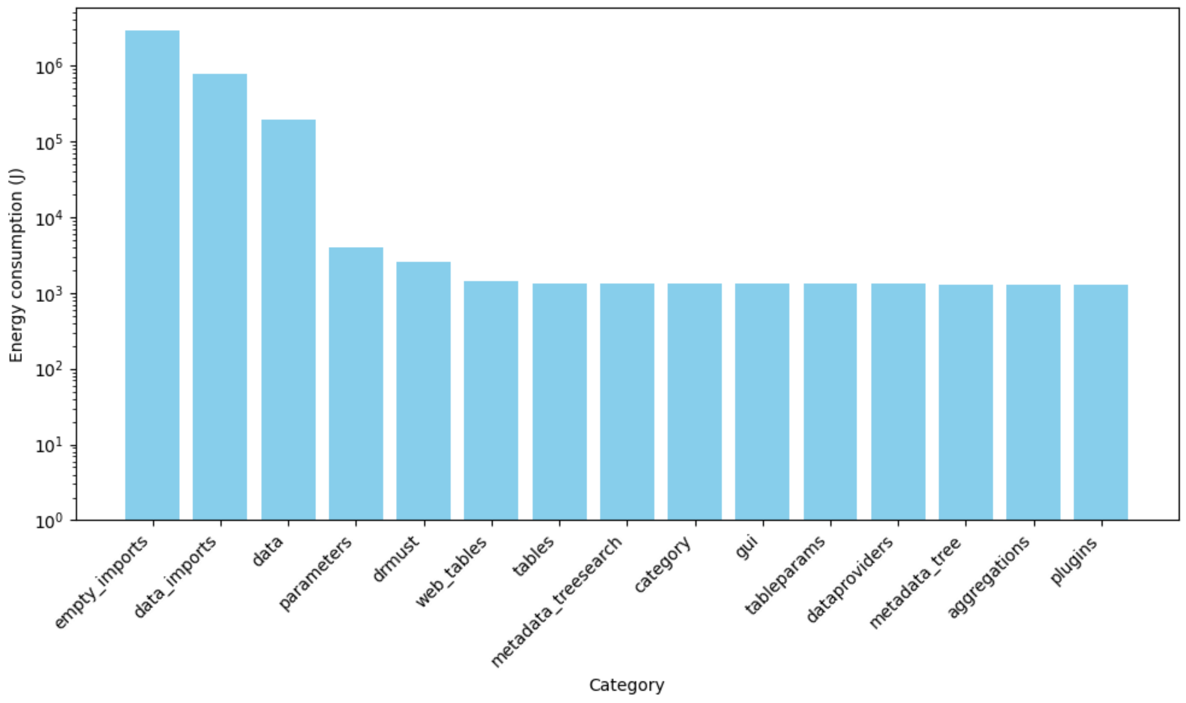

Figure 5.3: The average energy consumption of the different tested components of MUST.

Figure 5.3 shows the average energy consumption of each of the components of MUST: all tested categories of requests, as well as the total energy consumption of the importation. The categories with the highest energy consumption are the imports and the data requests.

Figure 5.4 shows the average energy consumption of nine empty imports and the average energy consumption of importing each of the thirteen 100-minute data chunks. Remember that these thirteen chunks represent one day of data, and that for each chunk of data, the importer runs without data nine times. The energy consumptions of importing the thirteen different data chunks are similar to each other. While doing a single empty import consumes less energy than importing data, doing nine empty imports consumes significantly more energy than a single data import.

So using the following formula, we can get the approximate energy consumption of a week of importation:

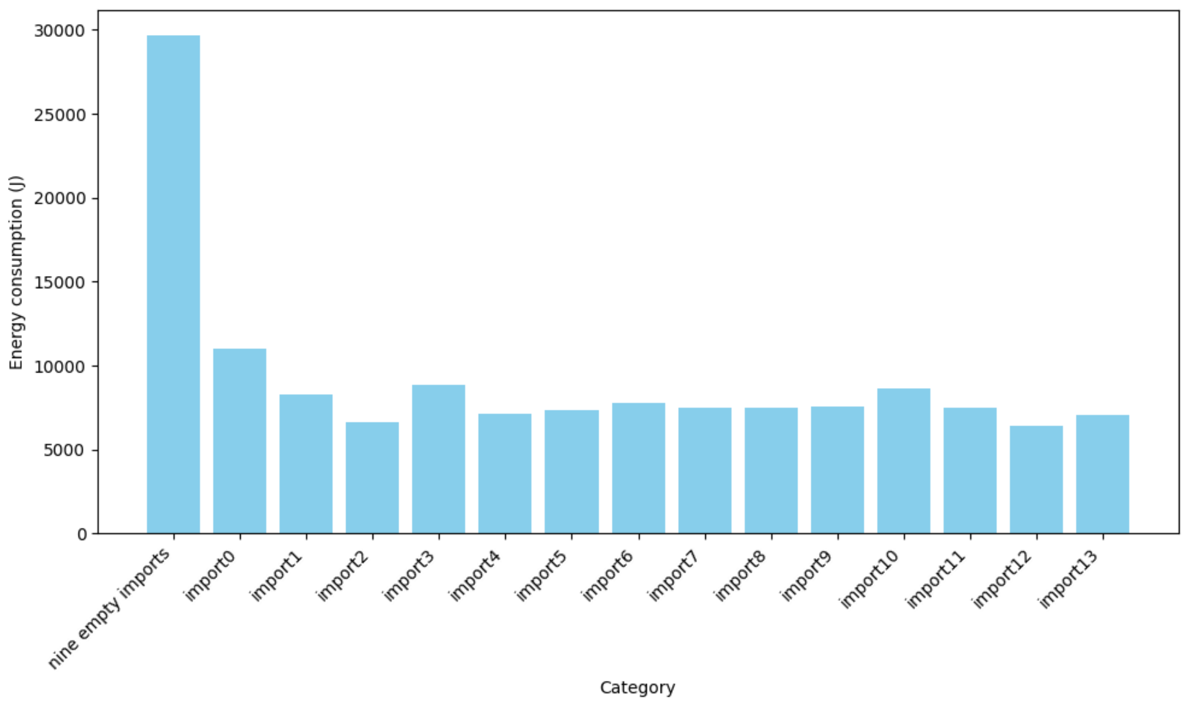

Figure 5.4: The average energy consumption of the importing the 14 100-minute data chunks, as well as the average energy consumption of nine empty imports.

$$E_{week} = 7 \times \sum_i (E_i + 9 \times E_{empty})$$

where E_i is the average energy consumption of importing a certain data chunk and E_{empty} is the average energy consumption of an empty import.

This gives an approximate total energy consumption, for importation of around 3.6 megajoules.

5.3. Step 3: Discuss Results and Formulate New Experiments

Figure 5.3 shows two clear hotspots: imports and data requests.

Before we discuss them, however, we need to discuss the *drmust* request category. As a refresher[5], DrMUST is a tool that helps users diagnose potential anomalies. Instead of using a websocket to push results to the user when there are new results, the frontend polls the server for results. In our week of data, the system polled DrMUST for a result 19853 times. However, because DrMUST is not used often, and even then rarely finds anomalies, none of these resulted in a result.

So, even though the contribution to the total energy consumption of these DrMUST requests is relatively low, practically all of it is unnecessary. A websocket approach would likely save energy.

Compared to imports and data requests, all other categories represent an insignificant amount of work: summed together, all other work represents only around 0.5% percent of the total energy consumption.

First, let us consider the importing of data. The fact *that* this is a hotspot makes sense, and was expected. MUST handles large amounts of data, and importing and calibrating this takes time and work. However, the amount of work wasted by running the importer with no data is staggering. Every 100-minute cycle, the amount of energy used to actually import data is less than the amount of energy used by running the importer with no data.

[5]there are more details in Chapter **??** and in Martinez [17]

Ingesting data from space systems involves managing extensive metadata, including packet structures, parameter locations, and calibration methods. Spacecraft missions generate numerous packets and parameter definitions, along with command sequences, alarms, and calibrations that must be accessible for accurate telemetry processing. Before checking to see if there is data to be imported, the importer loads these assets into memory and sets up Java structures for database connections.

Due to the extensive energy consumption of this step, we recommend that:

- The importer checks if there are new to import files before starting its initialization phase. This would be easy to implement and would save energy, in the order of megajoules per day.
- An on-demand approach for fetching metadata could be investigated. The importer always initializes the metadata for all parameters, even if not all of them are present in the parameter files currently in the processed folder.

Because of time constraints, we decided not to implement and validate these recommendations.

The other significant hotspot is the data requests. Again, the fact *that* it is a hotspot makes sense. MUST's main use case is to get data from the database and plot it. Because of this, data requests represent the vast majority of work that is triggered by users. This is true both in terms of the number of requests (recall from section 5.1 that the data requests represent the largest portion of the total requests) and in terms of the amount of work per request. Figure 5.5 shows the average energy consumption of a single request in each category, which clearly shows that the data category is most expensive. Also of note here is that a single DrMUST request barely consumes any energy ($0.13J$ on average, vs 42.44 for the data request). The sheer volume of requests in this category still results in a quite significant contribution to the total energy consumption.

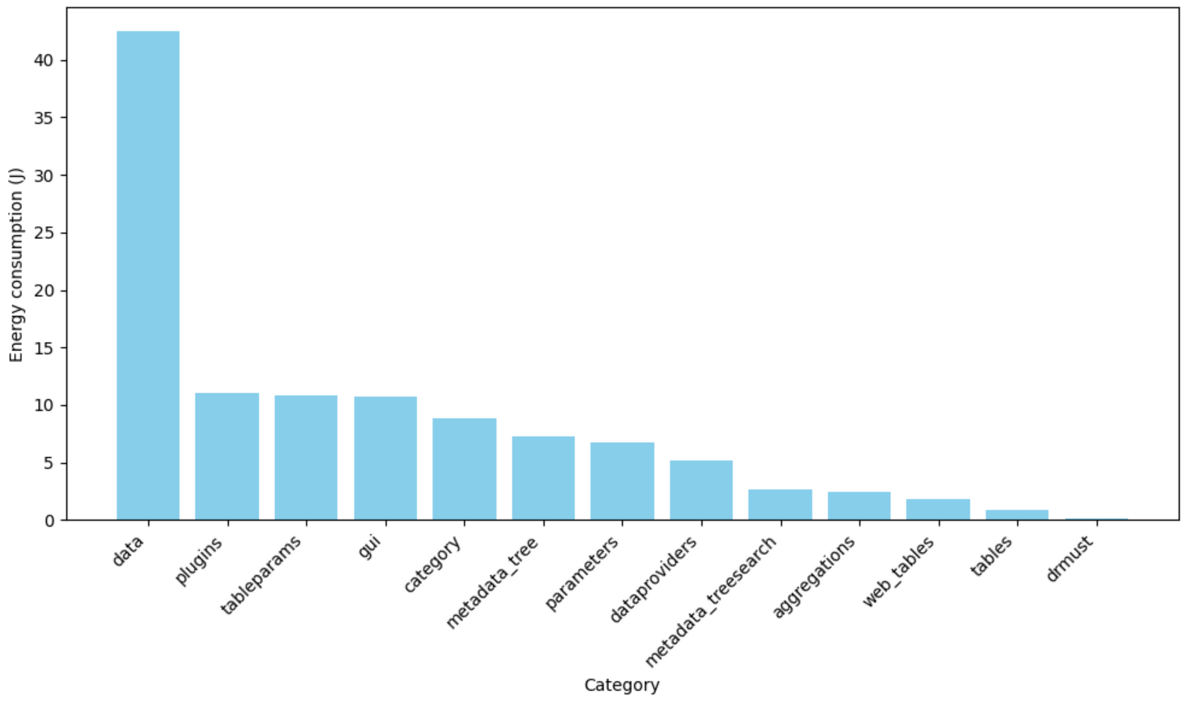

Figure 5.5: The average energy consumption per request for each category

It makes sense, then, to dive deeper into these data requests, to investigate further what factors play a role in the energy consumption of these data requests and what we can do about that.

Based on expert opinion, we identified two key questions:

1. What is the influence of the quantity of data on the energy consumption of the data requests?
2. What is the influence of the various paramaters of the data requests on energy consumption?

For question 1., we vary the amount of data that a data request has to process by varying the duration of the requests. At a set frequency, a request for a two-week period of data will contain twice as much data as one for a one-week period of data.

For question 2., we identified three interesting parameters of the data requests:

VRA VRA stands for Viewport Resolution Aggregation [18]. The underlying idea is simple: if a user requests data to create a graph, and the graph is only 1000 pixels wide, it is not necessary to send hundreds of thousands of datapoints over the network. Instead, given a viewport that is N pixels wide, the algorithm divides all datapoints into N chunks, and returns the minimum and maximum value in each chunk. This reduces the amount of datapoints from (possibly) hundreds of thousands to just 2N. Even a full screen image on a 4K display this is only ± 16000. However, this approach comes with (some) overhead of retrieving all the data and finding the minimum and maximum values. MUST can be configured to engage the VRA algorithm once a threshold of a certain number of datapoints is reached. By default, MUST engages VRA when more the response would otherwise contain more than 150000 datapoints.

Streaming This is a boolean parameter. Without streaming, the whole result set from the database is transformed into a java object, then into a json object, which is then sent over the network. With streaming, the database result set is streamed directly into a JSON buffer, which is then sent over the network. This saves memory and a conversion step, which can, for large datasets, have a significant impact.

Compact MUST sends datapoints as a list of json objects of the form `data: [{"datetime": <datetime>, "value": <value>}, {...}, ...]` when `compact` is false. When `compact` s true, the format changes to `data: [{"d": <datetime>, "v": <value>}, {...}, ...]`. When sending a large number of datapoints, this small optimization can have a major impact. Due to a bug in MUST, in the current implementation, `compact` only works when `streaming` is enabled.

5.4. Step 4: Investigate Hotspots and Surprising Results

To answer the questions formulated in the previous step, we formulate and perform a number of additional experiments.

5.4.1. Request period

Starting from an empty database, we create a single test parameter with datapoints in a period of two months, with a frequency of one datapoint every second, with the VRA disabled. Then we perform requests for various periods. We perform requests with periods ranging from 1 to 1000 hours, in steps of 50 hours each. Each experiment is performed 5 times, with a 30-second warm-up period, in a random order, and the results are averaged.

As we are performing the same request over and over again, the system may be able to cache the results, which would invalidate the measurements. To counteract this, we slide the period of a week for which we request data by one second every time we do a request. In this way, every request is different, and the system cannot cache it.

We perform each experiment 5 times, with a 30-second warm-up period each time, and average the results.

5.4.2. VRA

To measure the impact of VRA, we perform the same experiment as described in the previous section, but this time with VRA enabled. We use a set chunkcount – the width of the viewport for the VRA algorithm – of one thousand[6].

As explained in section 5.3, the VRA algorithm kicks in once a request requests more than 15000 samples. At 1 Hz, we reach the 15000 sample threshold at a request period of approximately 4.1 hours. Skipping from a 1 to a 50 hour request period, as in the previous experiment, does not give enough resolution around this point.

Therefore, we also perform requests with periods ranging from 1 to 100 hours in steps of 1 hour, with

[6]The mean chunkcount in the requests made in our period of data is 1196. 1000 is a nicer number, and close enough.

three different settings: VRA enabled from the default 15000 sample threshold, VRA disabled, and VRA always enabled.

We perform each request 1000 times and each experiment 5 times, making sure to slide the request period by 1 second to invalidate the cache and to include a 30-second warm-up each time.

5.4.3. Streaming

To evaluate the impact of the streaming parameter, we repeat the previous experiment, with request periods ranging from 1-100 hours in steps of 1 hour again, with VRA disabled, but this time with streaming enabled.

5.4.4. Compact

Due to a bug in MUST, the compact parameter only activates when streaming is also on. To measure its impact, we repeat the same experiment, this time with both streaming and compact enabled, and with VRA still disabled.

5.4.5. Results

Figure 5.6 shows the results of the request period experiment from section 5.4.1 and the first of the VRA experiments from section 5.4.2. These are the two experiments that have request periods ranging from one to one thousand hours in steps of 50 hours. The figure reveals that as request periods increase, energy consumption also rises. However, for these long request periods, activating VRA decreases the rate at which energy consumption increases.

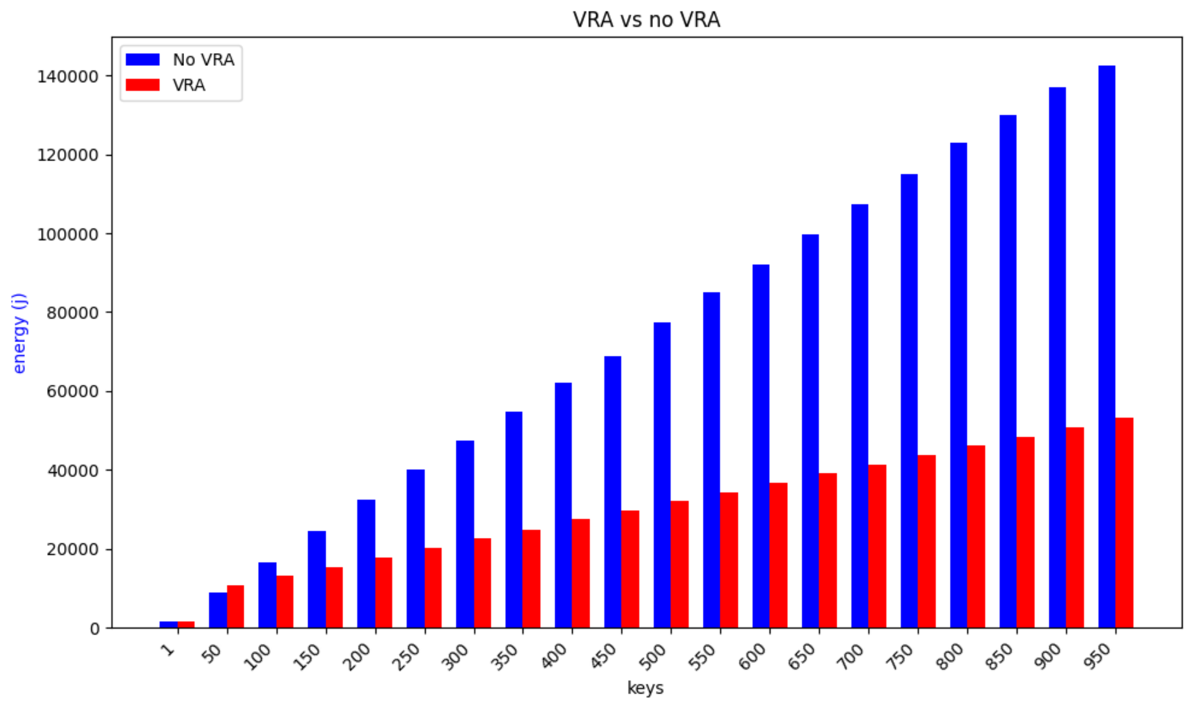

Figure 5.6: Shows the energy consumption of a request for a parameter with a 1 second frequency, ranging from a 1 hour request period to a 950 hour period in 50 hour steps. For the red bars, VRA was enabled and for the blue bars, it was not

5.7 shows the result of the 'zoomed-in' VRA experiments, with request periods ranging from one to one hundred hours in steps of 1 hour. The plot shows the results of when VRA is always enabled, VRA is enabled from the 15000 hour threshold (which occurs between the four and five hour mark), and with VRA disabled. We see a clear jump, right at this threshold, in the energy consumption when VRA is turned on. Furthermore, before the 65 hour mark (which corresponds to 234000 samples, much more than the default 15000 sample threshold), enabling VRA actually consumes more energy. After the 65 hour mark, these lines cross and VRA improves energy efficiency.

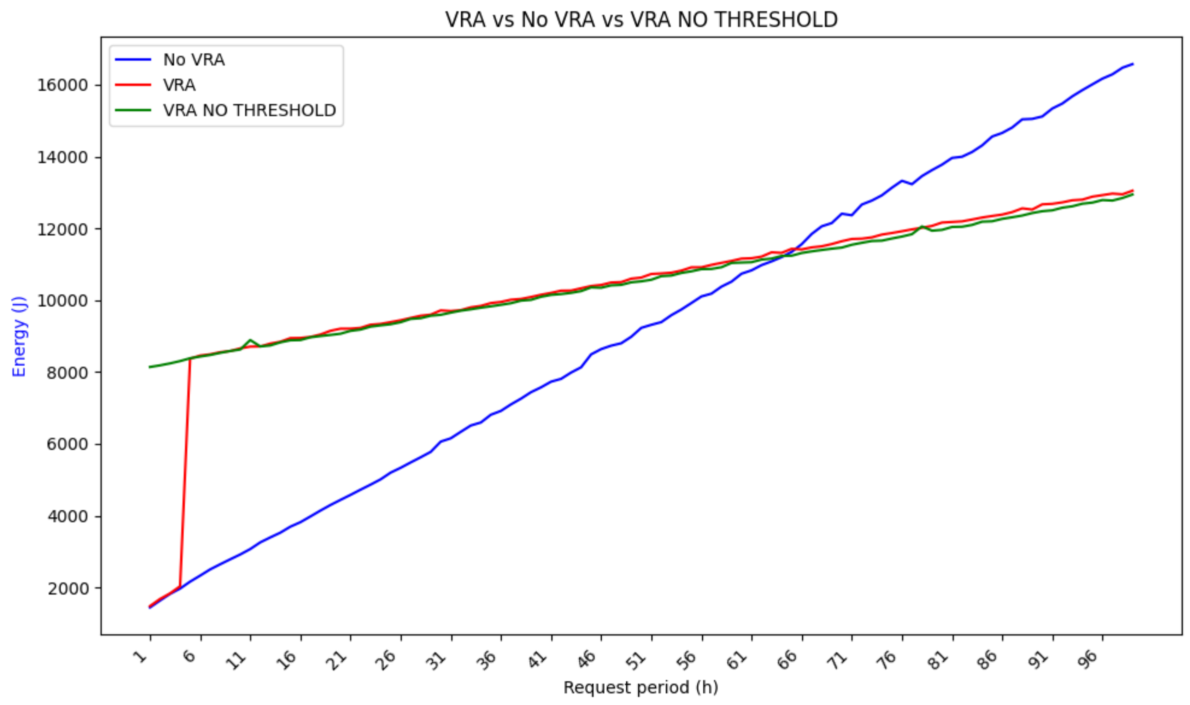

Figure 5.7: The energy consumption of a request for a parameter with a 1 second frequency, ranging from a 1 hour request period to a 100 hour period in 1 hour steps. The red line has VRA enabled as normally configured, from the threshold of 15000 samples. At 1hz, this threshold is reached between a 4 hour request period (14400 samples) and a 5 hour request period (18000 samples). The green line has VRA enabled always, and the blue line has VRA disabled.

Figure 5.8 shows the results of the investigations into the `streaming` and `compact` parameters. Again, these are requests with a period ranging from 1 hour to 100 hours, in 1 hour steps. The plot shows that streaming is more efficient than not streaming, but that the `compact` parameter barely makes an impact when enabled in conjunction with `streaming`. We were unable to test `compact` separately from `compact`, due to a bug in MUST.

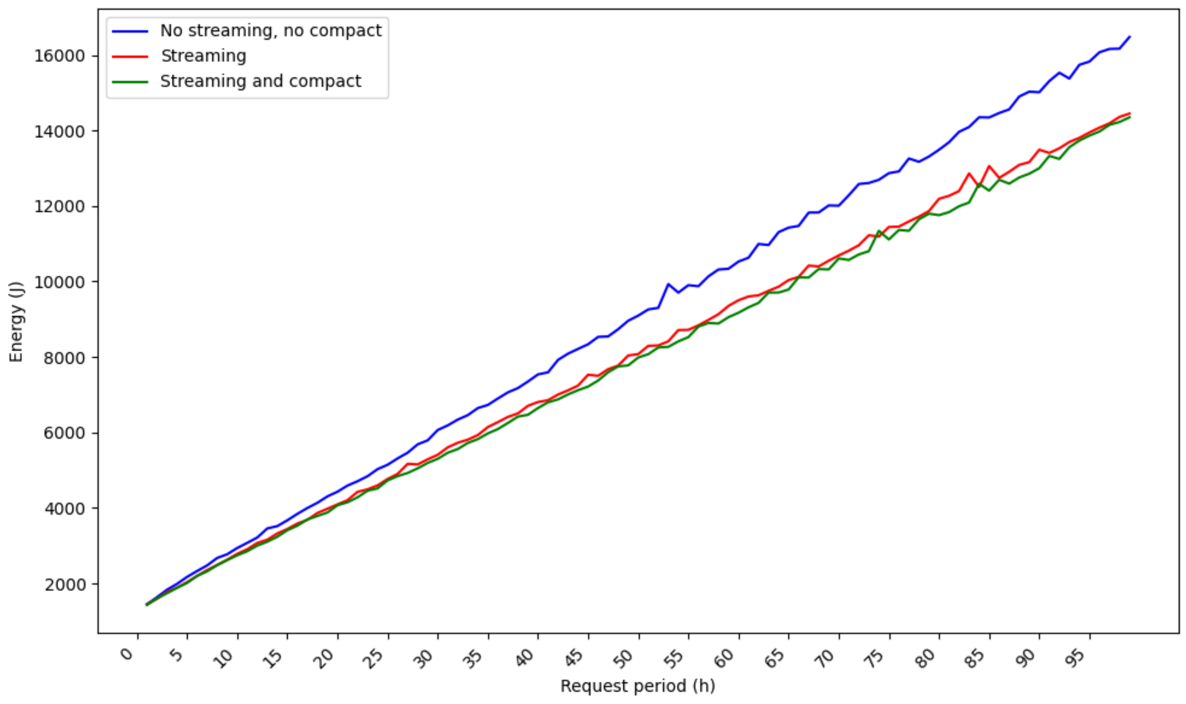

Figure 5.8: The energy consumption of a request for a parameter with a 1 second frequency, ranging from a 1 hour request period to a 100 hour period in 1 hour steps. For the blue line, requests were done with streaming and compact disabled. With the red line streaming is enabled, and with the green line, both streaming and compact are enabled.

5.5. Step 5: Report and Engage

Let us start by recalling from section 4.6 that this step has four sub-goals:

1. Present the results
2. Discuss the meaning of these results
3. Formulate recommendations
4. Engage with sustainable software

We targeted these goals in three interactive sessions. Each session had a different focus, depending on the audience and their relation to MUST. Before the sessions, we formulated three claims about MUST.

Firstly, based on our analysis, we conclude that the DrMUST endpoint is a (relatively small) waste of energy. We argue that there are a number of possible solutions, such as websockets[7] or server-sent events[8], that would improve the situation and be fairly simple to implement.

Secondly, we conclude that the importer wastes a really significant amount of energy, and that in this specific case the solution is almost trivial: by just checking to see if there are files before running the importer, the importer would cut down on energy waste tremendously.

Thirdly, the data endpoint, in contrast to the importer and the DrMUST endpoint, has received valid and useful energy optimizations in the Streaming and VRA parameters. These parameters, especially for larger requests, drastically reduce the amount of work necessary. We argue that the reason these changes were implemented, while the importer and DrMUST were not optimized, is that the users notice a lack of efficiency in the data endpoint anytime they make a graph in MUST. In contrast, the import and DrMUST run in the background, so users do not notice the inefficiencies, and thus, optimizations are not a priority.

5.5.1. Session 1: Solenix engineering division

The first session was a ‘developer meetup’ with the engineering division at Solenix. Around 25 software engineers, some working on MUST and some working on other projects, joined the session. The session mainly targeted the fourth goal: to engage stakeholders with sustainable software ideas.

In the session, it was clear that the practical results from the SMURF analysis helped relate broader sustainable software engineering ideas to concrete and practical issues in a familiar software system. It also became clear that for many, this was the first time even thinking about these ideas. The MUST team’s leader described the presentation as an “eye-opener”. Multiple software engineers asked for more resources on sustainable software engineering, and one stated that he expected the phrase “let’s not do this, it’s not energy efficient” to come up in technical meetings because of heightened awareness.

5.5.2. Session 2: ESOC

This session was attended by six interested employees at ESOC. These all had practical experience using MUST to manage spacecraft. They took on a double role: one as users, and one as the product owners: these were, in part, the people who decide what gets priority in MUSTs development.

The session specifically targeted goals one, three, and four. After the presentation, there were two main topics of discussion. The first point was that the low-hanging fruits, the things that waste a lot of energy but are easy to fix, definitely deserve priority. MUSTs technical officer at ESOC even remarked that he was ‘going to put this on the server this afternoon’.

While they were interested in these specific low-hanging fruits, they also remarked that MUST was already quite a reasonable system, and that there are many much worse examples of blatant energy waste in the ESOC operations. They expressed hope that the work in this thesis could help bring attention and priority to these other systems.

Then, the discussion changed into a discussion on how to implement these ideas. Ideas like hardware or other sustainability requirements, energy-saving bonuses, other financial incentives, and general awareness campaigns were proposed and discussed.

[7] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[8] https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

5.5.3. Session 3: MUST developers

In this session, goals one and two were specifically targeted. Five MUST developers were present at the meeting.

They were able to provide insights on why the importer works as it does. As stated in chapter 2, there are many different types of importers, based on a shared interface. While some of these importers are file-based, many are not. For the ones that are not, all the preparation work that the importer does is necessary to figure out, in the first place, if there is data available. Implementing a general solution that more efficiently checks for available data would take considerable effort. However, everyone agreed that for the specific case of these file-based importers, a fix was easy and useful.

The developers also provided more context for the results of the VRA experiments. They noted that pushing the threshold to 65 hours, the point where the VRA and no-VRA lines cross, is infeasible: the client would not be able to handle that many datapoints and there would be increased network costs, which aren't measured in this study. However, they were surprised by the results that, below this threshold, the energy consumption is so much worse. They also argued that the vast majority of requests made to the must system will fall below this threshold, in which case, VRA is actually hurting performance. They suspected that there must be an inefficiency in the implementation and intended to investigate.

The MUST team was already aware of the wasted work in the DrMUST and the importer case, but stated that this energy-consumption perspective provided a much stronger motivation to start work on them.

Additionally, they mentioned the work that is currently happening on moving MUST to Kubernetes and the cloud. They discussed the potential of applying SMURF, as well as general sustainable software ideas in the design process. In particular, they discussed the potential of MUST deploying on-demand, shutting it back down again when nobody is using it, and envisioned a role for the SMURF methodology in the decision-making process.

Lastly, the developers expressed the intention of applying SMURF to other projects they were developing. In addition to finding similar low-hanging fruits as in MUST and thus improving their software, they hoped that this would also allow them to win better contracts by including the sustainability angle as a selling point.

5.5.4. Conclusions

Taking all that into consideration, we draw the following conclusions on MUST and Solenix.

The importer and DrMUST waste energy. In the case of the importer, a general fix for all importer types will take considerable effort, but the fix for file-based importers is simple and should be implemented. For DrMUST, the fix is also relatively simple but will take some effort.

We should take care of the low-hanging fruits first. Within MUST these have been identified by applying SMURF, but within the Solenix and ESOC ecosystem of systems, there are many more that have not been identified. Further application of SMURF to other systems can help identify them. In more difficult issues, such as optimizing VRA or all importers in MUST, more complex trade-offs will present themselves, in terms of priority, effort required or man-hours and budget available. It is important to realize that we are not yet at the stage where this is important, because there are so many major issues with simple, even trivial, fixes left.

However, if we do want to dive deeper and fix more complex issues, we need to approach this from a policy and procurement perspective. It is difficult to create the right incentives to promote sustainable software, and while options have been suggested, these issues need to be explored in much more detail.

Overall, two things came to light in all conversations and sessions: these results provide a strong motivation to start tackling the uncovered issues, and promote a much-needed awareness of sustainable software issues. This heightened awareness can help normalize sustainability discussions in technical meetings, can help give priority and free up budget to fix sustainability issues, and can cascade into even more awareness and care.

6 Discussion

In this thesis, we introduced the SMURF methodology to evaluate the energy consumption of complex software systems. The goal of this section is to evaluate that methodology. To see how well it works, how generalizable it is, what the role of stakeholders is, how useful its results are, and what issues we ran into.

Recall the five questions we formulated previously in section 4.6:

Effectiveness of the methodology: How effectively does the SMURF methodology estimate an energy profile of a software system based on empirical usage data and without access to production servers?

Stakeholder influence: How does stakeholder involvement influence the SMURF methodology?

Awareness and engagement: How does the information generated by the SMURF methodology influence awareness of and engagement with energy consumption and sustainability?

Applicability: To what extent can the SMURF methodology be applied to various software systems?

Limitations and challenges: What limitations or challenges arise when implementing the SMURF methodology in a real-world scenario?

In this section, we answer each of these questions in turn. At the end of the chapter, we discuss potential directions for future work.

6.1. Effectiveness of the Methodology

While there has been previous work studying the environmental impact of software systems, many of these works tend to either zoom out or zoom quite far.

When they zoom out, works such as the works by Freitag et al. [13] and Simon et al. [26] focus on a broad spectrum of sustainability dimensions or environmental aspects. In doing so, they tend to become speculative, and often – at least partly – based on guesses and hypotheticals. That is not to say they have no scientific value. The guesses and hypotheticals are formed carefully and as accurately as possible. They are necessary because of the complex nature of the problem.

When they zoom in, as in the works by Feitosa et al. [12], Dorn et al. [9], and Bruce et al. [2], they focus on single functions, specific data structures, or design patterns. In doing so, they may lose track of the broader context that is needed to make sense of a complete software system.

By limiting the scope to a single software system and to energy consumption, but not zooming in further, our approach allows us to identify specific components that contribute most to energy usage, and ensures that our findings are relevant and applicable, bridging the gap between theoretical assessments and practical implementations in software engineering.

In the MUST case study, the methodology clearly and convincingly showed these hotspot components. In addition, the methodology gave valuable, evidence-based, concrete insights into components of the system that used more energy than one might expect. Both of these factors facilitate informed decision-making for developers and stakeholders, guiding them to the right components of the software to focus energy-saving efforts on.

Previous work that has a similar scope – a single, complex software system and a focus on energy consumption ([15, 8, 21, 7]) – tends to base these investigations on existing test cases (which were not designed for the purpose) or on use cases formulated by developers or stakeholders, often rather arbitrarily.

Using this approach, it is still possible to compare different versions of the same system. No matter how much a certain component is used in a scenario, as long as that scenario is kept identical over different versions, it is possible to say if a certain version uses more or less energy.

But with this approach it is not possible to compare different components of a system. In the first step of the SMURF methodology, we formulate a typical use case, based on server logs and other usage data, to make sure that the tested scenario is grounded in reality. This gives the necessary insight into how much different components of a software system are used under typical use, and allows us to make these kinds of comparisons.

From our case study, we find that formulating this typical use case, based on server logs and some auxiliary usage data, is possible and contributes to a much more grounded understanding of the energy profile of the system. The data that is necessary for this is available on a wide range of systems, and the methodology is format-agnostic – its applicability is independent of the exact format of the logs.

One explicit goal of the methodology was to allow for a non-intrusive measurement approach. In many environments, direct access to production systems is restricted due to security protocols, operational risks, or potential disruptions to service. By leveraging simulations and test deployments, our methodology enables researchers and developers to study energy patterns in these types of environments. Furthermore, the non-intrusive nature of the methodology allows for repeated measurements and iterative testing, which allows us to dive deeper into specific questions and to perform targeted experiments. This can contribute to better, evidence-based decision making.

In conclusion, the methodology can be used to effectively create an energy profile. The methodology does not suffer from the lack of production access – in fact, in some ways, it is an advantage. The use of server logs to formulate a 'typical use-case' allows the SMURF methodology to accurately compare the energy consumption of different components within a single version of the system, and to evaluate the energy consumption of systems with more than one core functionality, something that earlier works struggle to do.

6.2. Stakeholder Influence

Another goal of the methodology was to involve stakeholders at multiple points in the process. While the typical use-case is mainly formulated based on server logs, in the case study, expert input was essential in gaining a basic understanding of the system, and identifying components – the importer in our case study – that might not show up in the logs.

The case study showed clearly that involving stakeholders aids in formulating relevant questions and validating assumptions. In the case study, it was these conversations that allowed us to make sense of the results and to formulate the follow-up experiments in step 4 that gave a much deeper understanding of the energy consumption of the data endpoint.

The context that these stakeholders can give to energy measurements increases the value of the results, adding nuance and deeper insights to otherwise plain energy measurements that might not tell the complete story. An example of this is how in the case study, these discussions added nuance to the VRA measurements, incorporating front-end performance considerations that might otherwise have been missed and highlighting that there might be an implementation issue.

In summary, the stakeholder involvement is an important element of the methodology. It is crucial for validating the typical use case, for placing results in the right context, and for formulating interesting additional experiments.

6.3. Awareness and Engagement

The methodology also serves as an educational tool, raising awareness about energy consumption and sustainability in software design. In the case study, all stakeholders (users, developers, and product owners) indicated that they had not yet thought much about the sustainability of software systems, and that this study raised their interest. The fact that stakeholders are already intimately familiar with the system under test changes the ideas of sustainable software engineering from something abstract and far away to something more concrete and closer to home.

These ideas can serve as a catalyst for discussions around sustainability in technical meetings. This awareness can contribute to shifting organizational culture towards valuing energy efficiency and sustainability, and can empower people to raise these kinds of sustainability concerns.

Overall, the results of the SMURF methodology act as a powerful tool for raising awareness and fostering engagement with sustainability issues.

6.4. Applicability

At its core, the SMURF methodology is applicable to a wide range of systems: it is agnostic to specific architectures and specific log formats. All five steps are flexible enough to apply to a wide range of systems. It can be tailored to meet the specific needs of different organizations and projects. Teams can tweak the steps of the SMURF methodology to focus on particular aspects of energy consumption that are most relevant to their context. Because of its use of a typical use-case grounded in empirical usage data, it remains relevant even as systems grow into complex, multi-faceted systems with multiple core components.

There are two nuances to be made, however.

Firstly, in this thesis, we have evaluated the SMURF methodology on only one system: MUST. While the aforementioned attributes suggest a broad applicability, it is important to acknowledge that the insights and recommendations derived from this single case study may not universally translate to other systems without further validation. Each software system has its unique characteristics, operational contexts, and user behaviors that could influence the effectiveness of the SMURF methodology.

Secondly, while there is a broad range of systems that SMURF could be applied to, there are some qualities that system must have. Systems that lack these qualities may not benefit from the SMURF methodology or may face significant challenges in its implementation.

The SMURF methodology requires access to server logs, and in some cases, systems may not produce these. However, server logs are commonly available in many modern software systems, as they are integral to monitoring and troubleshooting applications.

Successful implementation of SMURF additionally requires buy-in from various stakeholders. Gaining consensus on sustainability goals and prioritizing energy efficiency can be challenging, especially if stakeholders have differing priorities. Organizations may face limitations in terms of time, budget, and personnel when attempting to implement the SMURF methodology or when supporting its implementation. Conducting thorough analyses and staying involved can require resources and time that may not be available.

In conclusion, many attributes of SMURF strongly suggest that it can be applied to a wide variety of systems. However, because our case study is on a single system, further verification is needed. In addition, SMURF places a number of requirements on the systems that it can be applied to. While many systems will meet these requirements, there will inevitably be some that don't. In these cases, it will be impossible to apply the SMURF methodology, or only possible in a modified form.

6.5. Limitations and Challenges

The second step of the methodology involves setting up the test deployment of the system. In practice, it turns out that this is where the main practical and conceptual difficulties appear.

Configuring the test environment to closely mimic the production environment is a major challenge, both on the hardware and on the software side.

On the hardware side, hardware configuration has a significant impact on energy measurement results. On some systems, it may be possible to run the test environment on a system that has a very similar hardware configuration to the production environment. In our case study, this was not feasible. Because of this, the methodology is limited to only providing relative numbers. However, we've seen in our case study on MUST that these relative numbers still hold valuable insights – and the fact that the SMURF methodology achieves this without relying on specific hardware configurations makes it much easier to implement in practice.

On the software side, the difficulty of matching the configuration is also highly dependent on the specific software system being configured. MUST is an enormously configurable software system, and the correct setup of the system was one of the more time-consuming steps of applying the methodology. Some of the configuration is crucial to ensure that measurements are representative, and on complex systems there can be many different parameters to get right. In the MUST case study, we relied on expert input to make the correct choices.

Conceptually, the main difficulty in this second step of the methodology, especially in our case study, was creating a representative dataset. While we argue that our methodology is already much more grounded in reality compared to earlier approaches that use user stories or test cases to create similar energy profiles, in this aspect we do rely on estimates and approximations. This introduces a level of uncertainty that can affect the accuracy of our energy consumption profiles. In the MUST case study, this is especially apparent in the data request category, where the amount of data that a request collects directly influences the energy consumption of that request. To mitigate these issues, we engaged with stakeholders to gather insights and validate our assumptions, ensuring that our dataset reflects a nuanced understanding of actual usage.

In addition, in any energy measurement, there are inaccuracies. Many different factors, from external processes to the temperature of the room the server is in, can influence the energy consumption of a machine, and this could influence the results. Any methodology that relies on energy measurements will have to deal with this. To mitigate this, we perform our experiments on a specialized server that is kept clear of other processes and is kept in a constant state throughout the experiments. In addition, following Cruz [5], we establish a warm-up time before running any experiments, we repeat each experiment multiple times, and we shuffle the experiments.

Given those uncertainties, the findings should be interpreted with caution: numbers cannot be interpreted absolutely, and it is not feasible to dive too far into the details or make claims that are too specific. However, as we've seen in the previous chapter, acknowledging these limitations does not stop us from providing valuable insights into energy consumption patterns.

In conclusion, the main challenges and limitations in applying the SMURF methodology are the practical difficulty of setting up a test deployment that reflects the production system, both in terms of hardware and software, the difficulty of performing accurate energy measurements, and the fact that the results can be used to spot general trends and hotspots, but that they will always remain relative numbers.

6.6. Future Work

The results paint a number of opportunities for future work. The first and most concrete of these is to pick the low-hanging fruits that the SMURF methodology identified in MUST, and implement the simple changes to the DrMUST endpoint and to the importer. This requires buy-in from the necessary stakeholders. All relevant stakeholders have indicated in meetings that they are interested in fixing these issues, so this should be the first order of business.

Furthermore, in step five of the methodology, MUST developers expressed surprise about the (lack of) performance of the VRA implementation for smaller requests. Especially because these smaller requests are the most common type of request, this issue should be investigated further.

To further advance and verify the SMURF methodology, the SMURF methodology should be applied to more systems. This would have numerous benefits. It would verify whether the methodology is indeed easily applicable to a broad range of systems, and allow for modifications to the methodology where necessary. It would allow for the identification of common pain points in the process of applying the methodology, and would enable more targeted improvements alleviating those particular points.

In particular, it would be interesting to apply SMURF to a diverse set of systems. The case study has shown that the SMURF methodology works well for a system like MUST, with a single server that does the majority of its work through API calls. The steps of the SMURF methodology are formulated in such a way that they are general enough, and it's simple to envision applying the SMURF methodology to systems with a different topology. However, whether this is possible, and how straightforward and effective it is, should be verified in future research.

Another interesting opportunity arises from more longitudinal studies. Coming to the end of this thesis, we've left MUST, Solenix, and ESOC with a number of concrete recommendations and a boost in sustainable software engineering awareness and engagement. It would be incredibly interesting to study the influence of these results over a longer period of time. It remains to be seen which, if any, of the recommendations will be implemented, and if these results are the starting point for a larger shift in sustainability awareness, or if the results will be forgotten. Studying this longitudinally, especially over multiple projects, could give insights into the long-term effects of the SMURF methodology.

The main concern that all stakeholders raised during the meetings in step 5 of the case study was that it is difficult to create the right incentive structure to encourage environmentally sustainable software development. And this is crucial: without it, economic and technical concerns tend to take priority. This is a major challenge to the field of sustainable software engineering, and we call for more research into appropriate incentive structures and effective policy to stimulate sustainable software development.

7 Conclusion

This thesis set out to achieve two goals.

The first goal was to develop a methodology to evaluate the energy profile of a single, complex software system, based on server logs and other usage data. In this thesis, we present the novel SMURF methodology. With this methodology, following a systematic five-step approach, it is possible to evaluate the energy profile of a single complex, multi-faceted software system. By utilizing server logs and other usage data to formulate a typical use case, the methodology stays grounded in reality, even when system complexity increases.

The second goal was to apply the methodology to MUST to gain an understanding of how the methodology works in practice, and in the process learn about the energy profile of MUST.

The SMURF methodology was able to uncover a number of hotspots and energy-wasting components within the MUST system. The results allowed for the formulation of actionable recommendations and served as new motivation to implement these recommendations. Through the SMURF methodology, we were able to uncover wasteful implementations of the importation process and the DrMUST functionality in MUST.

In addition, the methodology allowed for a deeper dive into the energy consequences of specific implementation details. Through our investigation into MUST's data endpoint, we were able to verify that the optimization strategies that have been implemented are effective, especially for large requests. The investigation also highlighted a possible implementation flaw in the implementation of the Viewport Resolution Aggregation algorithm that warrants further investigation.

It is worth noting that of the three highlighted components (DrMUST, imports and data requests), the data endpoint is the only one where users would directly be impacted by poor performance, and that it is also the only one that has seen serious optimization efforts. When these practical considerations align with sustainability concerns, budget and time become available to fix the issues. To make meaningful progress on sustainability issues, in areas where these considerations do not necessarily align, engagement with and awareness of sustainability is crucial.

The process and results of applying the SMURF methodology served as an effective engagement and awareness tool on sustainable software principles.

In meetings with developers, with users, and with product owners, a few common trends appeared. All stakeholders agreed that the most significant issues with the simplest fixes – the low-hanging fruits – should be fixed. They expressed that they had not thought about sustainable software engineering before, and meaningfully engaged with this material for the first time through the process of applying SMURF. They all raised concerns about creating the right incentive structure to ensure that sustainability issues get the priority they deserve, and yet, agreed that the sustainable software engineering perspective also provided new and much-needed motivation to fix issues.

The case study on MUST showed that the methodology achieves its goals of giving insights into the energy consumption of a complex system, being broadly applicable, supporting informed decision-making, and engaging stakeholders.

References

[1] Pernilla Bergmark et al. “A Methodology for Assessing the Environmental Effects Induced by ICT Services: Part II: Multiple Services and Companies”. In: *ICT4S 2020: 7th International Conference on ICT for Sustainability, Bristol, United Kingdom, June 21-27, 2020*. Ed. by Ruzanna Chitchyan et al. ACM, 2020, pp. 46–55. DOI: 10.1145/3401335.3401711.

[2] Bobby R. Bruce et al. “Approximate Oracles and Synergy in Software Energy Search Spaces”. In: *IEEE Transactions on Software Engineering* 45.11 (Nov. 2019), pp. 1150–1169. ISSN: 1939-3520. DOI: 10.1109/TSE.2018.2827066. (Visited on 12/16/2024).

[3] Déaglán Connolly Bree and Mel Ó Cinnéide. “Energy Efficiency of the Visitor Pattern: Contrasting Java and C++ Implementations”. In: *Empirical Software Engineering* 28.6 (Oct. 2023), p. 145. ISSN: 1573-7616. DOI: 10.1007/s10664-023-10387-8. (Visited on 12/16/2024).

[4] Vlad C. Coroama et al. “A Methodology for Assessing the Environmental Effects Induced by ICT Services: Part I: Single Services”. In: *ICT4S 2020: 7th International Conference on ICT for Sustainability, Bristol, United Kingdom, June 21-27, 2020*. Ed. by Ruzanna Chitchyan et al. ACM, 2020, pp. 36–45. DOI: 10.1145/3401335.3401716.

[5] Luís Cruz. *Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments*. http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post. 2021. DOI: 10.6084/m9.figshare.22067846.v1.

[6] Luis Cruz and Rui Abreu. “Catalog of Energy Patterns for Mobile Applications”. In: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2209–2235. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-019-09682-0. (Visited on 12/16/2024).

[7] Benjamin Danglot, Jean-Rémy Falleri, and Romain Rouvoy. “Can We Spot Energy Regressions Using Developers Tests?” In: *Empirical Software Engineering* 29.5 (July 2024), p. 121. ISSN: 1573-7616. DOI: 10.1007/s10664-023-10429-1. (Visited on 12/16/2024).

[8] João De Macedo et al. “Energy Wars - Chrome vs. Firefox: Which Browser Is More Energy Efficient?” In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*. Virtual Event Australia: ACM, Sept. 2020, pp. 159–165. ISBN: 978-1-4503-8128-4. DOI: 10.1145/3417113.3423000. (Visited on 12/16/2024).

[9] Jonathan Dorn et al. “Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs”. In: *IEEE Transactions on Software Engineering* 45.3 (Mar. 2019), pp. 219–236. ISSN: 1939-3520. DOI: 10.1109/TSE.2017.2775634. (Visited on 12/16/2024).

[10] Leticia Duboc et al. “Do We Really Know What We Are Building? Raising Awareness of Potential Sustainability Effects of Software Systems in Requirements Engineering”. In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 2019, pp. 6–16. DOI: 10.1109/RE.2019.00013. (Visited on 12/16/2024).

[11] S. Faucheux and I. Nicolaï. “IT for green and green IT: A proposed typology of eco-innovation”. In: *Ecological Economics* 70.11 (2011). Special Section - Earth System Governance: Accountability and Legitimacy, pp. 2020–2027. ISSN: 0921-8009. DOI: https://doi.org/10.1016/j.ecolecon.2011.05.019. URL: https://www.sciencedirect.com/science/article/pii/S0921800911002084.

[12] Daniel Feitosa et al. “Patterns and Energy Consumption: Design, Implementation, Studies, and Stories”. In: *Software Sustainability*. Ed. by Coral Calero, Mª Ángeles Moraga, and Mario Piattini. Cham: Springer International Publishing, 2021, pp. 89–121. ISBN: 978-3-030-69969-7 978-3-030-69970-3. DOI: 10.1007/978-3-030-69970-3_5. (Visited on 12/16/2024).

[13] Charlotte Freitag et al. “The Real Climate and Transformative Impact of ICT: A Critique of Estimates, Trends, and Regulations”. In: *Patterns* 2.9 (2021). DOI: 10.1016/j.patter.2021.100340. (Visited on 12/20/2023).

[14] Nelson Gregório et al. “E-APK: Energy Pattern Detection in Decompiled Android Applications”. In: *Proceedings of the XXVI Brazilian Symposium on Programming Languages*. Virtual Event Brazil: ACM, Oct. 2022, pp. 50–58. ISBN: 978-1-4503-9744-5. DOI: 10.1145/3561320.3561328. (Visited on 12/16/2024).

[15] Erik A. Jagroep et al. “Software Energy Profiling: Comparing Releases of a Software Product”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. Austin Texas: ACM, May 2016, pp. 523–532. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889216. (Visited on 12/16/2024).

[16] Javier Mancebo, Félix García, and Coral Calero. “A Process for Analysing the Energy Efficiency of Software”. In: *Information and Software Technology* 134 (June 2021), p. 106560. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106560. (Visited on 12/16/2024).

[17] Jose Martinez. “DrMUST - a Data Mining Approach for Anomaly Investigation”. In: *SpaceOps 2012 Conference*. Stockholm, Sweden: American Institute of Aeronautics and Astronautics, June 2012. DOI: 10.2514/6.2012-1275109. (Visited on 12/18/2024).

[18] Henrique Oliveira et al. “Enabling Visualization of Large Telemetry Datasets”. In: *12th International Conference on Space Operations (SpaceOps 2012), Stockholm, Sweden*. 2012, pp. 11–15. (Visited on 01/28/2025).

[19] Wellington Oliveira et al. “Improving Energy-Efficiency by Recommending Java Collections”. In: *Empirical Software Engineering* 26.3 (May 2021), p. 55. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-09950-y. (Visited on 12/16/2024).

[20] Rui Pereira et al. “SPELLing out Energy Leaks: Aiding Developers Locate Energy Inefficient Code”. In: *Journal of Systems and Software* 161 (2020), p. 110463. DOI: 10.1016/j.jss.2019.110463. (Visited on 12/16/2024).

[21] Rui Pereira et al. “The Influence of the Java Collection Framework on Overall Energy Consumption”. In: *Proceedings of the 5th International Workshop on Green and Sustainable Software*. Austin Texas: ACM, May 2016, pp. 15–21. ISBN: 978-1-4503-4161-5. DOI: 10.1145/2896967.2896968. (Visited on 12/16/2024).

[22] Ana Pont, Antonio Robles, and José A. Gil. “e-WASTE: Everything an ICT Scientist and Developer Should Know”. In: *IEEE Access* 7 (2019), pp. 169614–169635. DOI: 10.1109/ACCESS.2019.2955008.

[23] Jari Porras et al. “How Could We Have Known? Anticipating Sustainability Effects of a Software Product”. In: *Software Business*. Ed. by Xiaofeng Wang et al. Vol. 434. Cham: Springer International Publishing, 2021, pp. 10–17. ISBN: 978-3-030-91982-5 978-3-030-91983-2. DOI: 10.1007/978-3-030-91983-2_2. (Visited on 12/16/2024).

[24] Rui Rua and João Saraiva. “E-MANAFA: Energy Monitoring and ANAlysis Tool For Android”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–4. ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3561342. (Visited on 12/16/2024).

[25] June Sallou, Luís Cruz, and Thomas Durieux. *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*. Dec. 2023. DOI: 10.48550/arXiv.2312.13897. arXiv: 2312.13897 [cs]. (Visited on 12/18/2024).

[26] Thibault Simon et al. “Uncovering the Environmental Impact of Software Life Cycle”. In: *International Conference on ICT for Sustainability, ICT4S 2023, Rennes, France, June 5-9, 2023*. IEEE, 2023, pp. 176–187. DOI: 10.1109/ICT4S58814.2023.00026.

[27] Solenix. *Revolutionizing Space Operations: The Evolution of MUST | Solenix*. https://solenix.ch/blog/2023-11-01/revolutionizing-space-operations-evolution-must. (Visited on 01/28/2025).

[28] Roberto Verdecchia, June Sallou, and Luís Cruz. *A Systematic Review of Green AI*. May 2023. DOI: 10.48550/arXiv.2301.11047. arXiv: 2301.11047 [cs]. (Visited on 12/04/2023).