Green Coding and Energy Efficiency in REST APIs: Empirical Evaluation

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2025
Teemu Salonen

UNIVERSITY OF TURKU
Department of Computing

Teemu Salonen: Green Coding and Energy Efficiency in REST APIs: Empirical Evaluation

Master of Science (Tech) Thesis, 73 p.
Software Engineering
June 2025

As there is a growing demand for energy in IT systems, energy efficiency in IT and green software are gaining interest and are being research in growing amounts. Green coding and energy-efficient practices in software are not a brand new topic, but lack specificity regarding REST APIs. This thesis explores the applicability of existing methods of building energy-efficient software in the context of REST APIs through a literature review and an empirical analysis. A literature review was conducted on the current practices of developing green software, identifying methods such as overhead minimization, performance optimization and optimal design choices. A selection of these practices were selected and REST API implementations were built to test the effects of the practices. The effects were empirically tested in the form of power measurements on the reference implementations of the REST API. All of the empirically validated green coding practices exhibited at least some level of energy efficiency improvements. The underlying technology implementation had the greatest effect, whereas employing caching had the least significant improvement in energy efficiency. This work contributes both empirical data and practical insights toward the development and design of more energy-efficient and green REST APIs. Additionally, the thesis identifies points of focus for future research.

Keywords: green software, green coding, rest, api, energy efficiency, sustainability

Contents

1 Introduction

1.1 Background

Amid global warming and climate concerns, the software development industry is slowly starting to see the rise of *green coding* and awareness in energy-efficient coding patterns [1]. While the optimizations in IT energy consumption are often approached with hardware features, such as race-to-idle, there has been relatively little attention given to the other angle towards achieving better energy efficiency – the software itself – at least in the web application domain. Mobile applications have been the primary focus when it comes to energy efficiency considerations in software development due to their limited power supply being a battery. As web applications, SaaS products and by extension their APIs (Application Programming Interfaces) are a popular choice nowadays, there is bound to be more interest in the energy efficiency of these APIs.

While direct sources of emissions such as coal-powered factories or cars are an easy point of interest regarding greenhouse gas emission sources, the impact of IT systems or especially software is rarely mentioned in such conversations. Even if it's not immediately apparent for the general public (or even for IT professionals themselves), information and communication technology (ICT) systems produce a lot of emissions; the IT industry currently causes approximately 10% of the total global greenhouse gas emissions [2]. The widespread utilization of APIs makes the

software in web servers potentially be the cause of a substantial part of said emissions. REST (Representational State Transer) APIs are prevalent not only in web applications but also in automated systems and IoT devices, such as medical devices [3]. Consequently, a reduction in energy usage in the software of web servers could potentially result in considerable energy savings across the whole world.

As of 2025, green IT, green coding and energy-efficient software are being researched in growing amounts. Unfortunately, as of yet, when it comes to web applications or web server software, as to the best knowledge of the writer, these topics are relatively unexplored and green coding is not a very mature area of study. Adoption of energy efficiency improvement practices from the mobile domain is applicable to web applications in some cases [4]. Even if there is a lack of existing research for the improvements themselves, it is imperative to know what are the predominant causes for the energy footprint of a REST API in order to even know where to start.

The motivation for this thesis is to broaden the research on what the energy footprint of web server software, REST APIs in particular, looks like. Additionally, the motivation is to aid with the development better tooling for software developers when it comes to evaluating, enhancing and measuring the energy consumption, as the lack of which is, in many cases, one of the major reasons developers do not consider the energy consumption of the software they are developing [5]. Empirically validating which implementation and technical decisions affect the energy consumption should provide better grounds for making educated decisions for creating greener software.

1.2 Problem statement

As is stands, it would seem that an industry-standard catalog of features or characteristics to be evaluated when determining the energy efficiency of server-side API software has not been established. Many existing software energy efficiency and

energy consumption studies focus on improving the energy consumption of mobile software [6] [7] [8] [9]. Moreover, even in situations where the energy usage of other types of software is considered, the practitioners' perspective is, in many cases, to reduce the energy consumption caused while idle [1]. The existing studies for web applications, on the other hand, while focusing on energy-hungry patterns, seem to focus on more abstract definitions, such as "open resources only when necessary" and "release unnecessary resources" [4].

It is, generally speaking, not trivial for a developer to make energy-aware decisions when developing REST APIs. Being mindful of such choices when designing and developing the APIs can make it easier and more straightforward to make choices that support energy efficiency. Being aware of the main factors that impact the energy efficiency of REST APIs helps developed them to be energy-efficient. However, green coding practices for REST APIs specifically are not abundant. The aim of this thesis is to provide useful data on which specific technical characteristics in REST API their developers can try to enhance to achieve better energy efficiency under normal use. This will be elaborated on in section 1.4.

1.3 Research methods and sources

This thesis is divided into two distinct parts. The first part evaluates existing literature and research in green coding in general and green coding practices in the form of a small literature review. The second part empirically evaluates the effects of the main characteristics identified in the first part in the context of REST APIs. Based on the findings in relevant studies, the main factors affecting the energy consumption are empirically validated by measuring the effects of changing the level of utilization of the energy-efficient characteristics.

The platforms for finding the sources for the aforementioned characteristics were ACM Digital Library and IEEE Xplore, since those two contain most of the green

coding and software energy efficiency related papers [10], as well as Google Scholar and SpringerLink. The search terms used for finding the sources were "software", "programming language", "energy", "energy efficiency", "green", "green coding", "sustainable" and "web". The sources are explored more in-depth in Chapter 2.1.

Only highly relevant studies were picked and as mentioned. The selection of the sources was made by judging the title of publication and evaluating whether it addresses relevant topics regarding the thesis topic. This meant that the topic had to clearly address servers, APIs or software in general with the context of energy or power consumption or performance. The sources were only selected if they published in 2010 or after. Many of the relevant sources were also found by reference snowballing, meaning that finding one study on any of the aforementioned platforms lead to finding another relevant study via the references of the initial publication.

1.4 Purpose, scope and research questions

As mentioned, the green coding and its practices are explored from the point of view of the existing literature as well as a more practical approach with empirical tests regarding the points identified in the literature. To guide the research, I will strive to answer the following research questions:

1. **RQ1** What design, implementation and optimization choices have the most significant effect on the energy consumption of software?
2. **RQ2** How can software energy consumption be measured?
3. **RQ3** How effective are the existing green coding practices for REST APIs?
4. **RQ4** How significant is the choice of technology in relation to other green coding practices?

RQ1 is used to establish the main development-level choices and practical solutions that a developer can choose which have an effect on the energy consumption of the software, such as the choice of programming language. **RQ2** aims to assess how the energy consumption of software systems can be determined. The answer to this question will be utilized to conduct the empirical tests. **RQ3** investigates the link and level of applicability between the identified existing green coding practices and REST APIs. This research question will be explored empirically and it is the most important research question in this thesis. And finally, to get a more granular look into green coding practices in the context of REST APIs, **RQ4** assesses whether choosing the correct implementation technology for a REST API is of importance.

Hence, this thesis aims to empirically verify how the research regarding green coding practices applies to the web domain, specifically REST APIs. Additionally, the aim is to provide a base for developers to help them know what to consider when energy efficiency is a concern in their server-side software. It is worth noting that while the energy efficiency of IT systems can undoubtedly be enhanced by hardware optimizations, this thesis focuses purely on the aspect of software efficiency. Furthermore, REST APIs are - at least in principle - an established and a structured way of building server side software.

1.5 What is REST and why it was chosen

REST or Representational State Transfer is a resource-oriented architecture of building APIs. It is commonly a stateless and uniform interface with manipulation methods defined in HTTP, such as GET, POST and DELETE for the resources. The resources are accessible via a disctinct URI, such as `/api/notes/1`. [11]

REST was originally conceptualized as an extension to HTTP rather than an architectural solution by its designer, Roy Fielding. Developers at the time found SOAP to be cumbersome with all its specifications and the switch to REST as

the only perceivable alternative, although only in spirit and misappropriating the dissertation of Fielding, is easy to understand. Given the divergent interpretations in its implementations, the definition of RESTful remains loosely defined as there does not exist as specification for it. Modern RESTful APIs can be seen as JSON-RPCs or even as APIs that merely have a historical link to REST as opposed to a functional and architectural one. [12] These often implement traditional CRUD (Create, Read, Update, Delete) functionalities as well as some hints of RPC style operations depending on the implementation.

In web development, RESTful APIs are a widely-used choice when in comes to API design. While there are a plethora of different alternatives one can choose from when deciding on the approach to implementing APIs for web services, such as SOAP, GraphQL and different RPC solutions to name a few, the RESTful "architecture" was chosen for this thesis due to the overwhelming popularity over its competition [13] [14]. This thesis interprets the aforementioned widely-adopted and loose definition of REST/RESTful APIs as a single type of API differing from, for example, architectures such as GraphQL or the aforementioned SOAP. Hereinafter the term **REST API** will be used to represent the aforementioned architectural style in web server APIs, encompassing what RESTful and the other similar implementations in modern APIs mean.

1.6 Thesis structure

Green coding practices and topics relating to software energy consumption are described and addressed in Chapter 2. Said topics are described in a general manner, providing background information as well as explained more in-depth in the context of this thesis. This chapter also answers research question 1 (RQ1). Chapter 3 goes over how the energy consumption of software systems can be measured and what kind of arrangements other studies have used to achieve their results. Additionally,

different software and hardware-based tools to measure the energy consumption of software are described. This chapter answers RQ2. Chapter 4 describes the test programs and reference implementations used in the empirical section as representatives of different possible choices and characteristics for REST API development. In practice this means that regarding the research questions 3 and 4, we can only answer them in the within the limited scope of the empirical tests. Chapter 5 describes the power measuring setup, which is used in this thesis to determine the energy consumption of the reference implementations. The chapter provides details on the client and host devices as well as the measuring devices and their specifications. Chapter 6 describes and analyzes the data and results of the tests described in Chapter 4 as well as make conclusions and answer the research questions based on the collected data and results. Finally, in chapter 7 the hypotheses, theories, test results and analysis are joined together, analyzed and points in need of further research are identified. The threats to the validity of the results will also be briefly discussed.

2 Green coding and Energy Efficiency in Software

Writing energy-efficient software is not trivial. REST APIs are at the core of the web domain, as they are a widely adopted choice for API development [15]. The total energy consumption resulting from such software comes down to many different things, for instance network infrastructure and data center design [16], the level of the API code itself and even the cooling for the physical machines [17] [18]. For example, a suboptimal, power-inefficient data center can result in an energy overhead [19] for the software even if the software itself is optimized for energy efficiency. Optimizations for energy consumption can thus be done on many levels, for instance by minimizing idle time. This thesis focuses on the choices a developer can make when writing REST APIs regardless of data center efficiency or hosting solution decided upon. In other words, we focus on ways to contribute to improving the energy efficiency of the API itself without changing the functionality or features from the users' point of view.

Applications meant for mobile devices have requirements relating to energy use more often that traditional or data center software and mobile developers are often more concerned about the energy use of their software [1]. The battery-powered nature of mobile devices, such as phones and tablets, forms an inherent limiting factor on their energy use. Hence, the software-bound energy consumption needs

to be as small as possible for the device to remain usable for a prolonged period of time. In some cases, the principles used in mobile software energy optimizations, such as caching, extraneous work avoidance and minimization of transferred data, can be applied to web applications [4]. A study by Rani *et al.* [4] conducted in 2024 found 20 energy-specific design patterns (such as use cache, kill abnormal tasks and suppress logs) from mobile software developing practices, which were at least partially be applicable to the web domain.

In this chapter we will outline the green coding practices under consideration in this thesis. This chapter will also provide an answer to **RQ1**: "What design, implementation and optimization choices have the most significant effect on the energy consumption of software?".

2.1 Green coding in general

As mentioned, improving the energy efficiency of any piece of software, let alone whole entities of client-server architecture web applications, is not exactly straightforward. Amid the environmental concerns regarding emissions caused by IT, measuring the energy consumption of not only web applications but also their APIs could become vital for businesses not only to appease regulations such as the CSRD (Corporate Sustainability Reporting Directive) [20] but also to steer clear of avoidable monetary losses. As software is all but omnipresent in every industry, it's hard to imagine a future on any industry where the energy efficiency of any kind of software is of no concern.

There are many semantic and technical aspects that may have an effect on the energy efficiency of a REST API application. Different technologies, such as programming languages, can have on effect on a system's energy consumption [21].

Software or IT and "greenness" can roughly be divided into two categories; green **by** IT and green **in** IT, which are also known as Green IT and IT for Green, re-

spectively. [22] Green by IT means making, for example, a building complex green by utilizing "traditional" information technology systems. These are in other words systems which are not inherently green. This could be approached by automating energy consumption by methods such as turning off unused devices or using software to intelligently allocate energy between, for example, solar production and consumption from the electric grid and any similar software-related automation in different fields. Green IT, on the other hand, refers to the study and practice of having the IT systems be designed, used and disposed of in a way that is efficient and minimizes the impact on the environment. [22] This stands for the production of as little as possible indirect greenhouse gas emissions by reducing the required energy to run the system, as IT systems inherently require electricity and thus energy to operate. [23]

Green coding on the other hand, is an emerging approach in software development, where the intention is to decrease the energy consumption of software by making smart and educated decisions that have little to no effect as to the functionality of the software, but reduce its carbon footprint. [24] The aforementioned decisions could be, for example, optimizing the functionality of the software and using the latest and most energy-efficient technologies.

Table 2.1: Literature review sources: topics addressed or discussed as a green coding or software energy optimization technique.

Reference	Programming language	Frameworks	Paradigms	Batching	Algorithmic choices	Performance optimizations	Data optimizations	Caching
[21]	x					x	x	
[25]	x		x	x	x	x	x	
[19]	x	x	x		x	x	x	x
[4]	x		x	x	x	x	x	x
[26]	x		x	x	x	x	x	x
[27]	x	x	x		x	x	x	
[28]			x	x		x	x	
[29]	x		x			x		
[30]	x	x				x		
[31]	x				x	x		

As to what is covered in this thesis, the green coding topics in the collected literature are listed in Table 2.1. The table lists the main sources used for the literature review and points out the topics each study covered in some capacity. An x marking under a topic means it was discussed as a way to optimize energy consumption in software.

2.2 Technologies and paradigms

Throughout the existence of software engineering the term *paradigm* has had many interpretations and therefore there exist numerous approaches to delineate its definition [32]. In this thesis we use *paradigm* to refer to well-established or common ways, approaches and procedures of constructing software, mainly in the context of REST API software. An example of a software engineering paradigm could be, for instance, the object-oriented paradigm [32]. As there are a plethora of ways building software can be approached, we focus on the popular REST APIs and consider different paradigms and attributes within that context.

The selection of the implementation *technology*, on the other hand, be it a framework or the programming language used, can have significant implications on the energy consumption of a web application. [21] While developing server-side software for web applications, the developers have to make a choice between different technologies starting from the programming language and whether they prefer, for example, compiled, interpreted or virtual-machine-based languages (such as Java). Having chosen the language, there are numerous frameworks for each language, which give a basis for developing the software further. Not only are there differences for use cases and developer experience between frameworks, but also performance differences. High-quality software, i.e. one that implements for example modularity, can also cause an inherent reduction in energy consumption levels, at least when it comes to object-oriented programming [19].

2.2.1 Programming language

There are many types of different programming languages. One way to categorize the majority of the programming languages could be to divide them into three distinct categories: compiled, interpreted and hybrid[33] (also known as managed), which refers to virtual machine executed languages, such as C#. Compiled languages essentially hide the source code by transforming the human-readable code into machine code for a specific platform via a compiler before the program is run. Errors can be detected in the compilation stage due to the statically-typed nature of the language. As the compiler-generated machine code requires no additional intermediary, such as an interpreter, there is no additional overhead and hence the compiled languages are typically more efficient and faster compared to the other two. [33] In addition, their long and research-rich history has resulted in ever-better compilers, which produce performant code [34]. Managed languages, on the other hand, are usually compiled into byte code, which is then interpreted by the virtual-machine (such as JVM in case of Java). [33] The intermediary could, however, introduce a performance overhead [33], though the generated native code could also prove to make such languages even more efficient due to run time optimizations such as PGO (Profile-Guided Optimizations) [35]. Managed languages perform type checking both before (at compilation stage) and during run time. Interpreted languages are usually dynamically typed. Very early interpreters did not even make a distinction between the human-readable code and the actual executable file. The code was executed line-by-line, which caused a performance overhead due to the interpreter having to translate it each time it is used. [33] However, this is more of a legacy approach. Contemporary interpreters make this process more efficient by, for example, compiling the code into bytecode before execution or alternatively, an AST (Abstract Syntax Tree) interpreter can be used. [36] Another contemporary approach is to use JIT (Just-In-Time) compilation, where the code is compiled dur-

ing execution. This approach enables run time optimizations, as the code can be re-compiled during run time but reverting to the interpreter is also an option. [34]

Reference [9] found compiled programming languages to be mostly outperforming interpreted ones in common programming tasks. The study found different programming languages to be optimal for different tasks. For example, Go prevailed over most implementations in sorting algorithms whereas JavaScript was the better option for regular expressions and Rust had the best results in I/O operations. The choice of programming language for the implementation is a major determinant of energy efficiency - for example, a study showed Rust to be over four times more energy-efficient than the popular scripting language JavaScript [21]. At the other end of the spectrum, some interpreted languages like Perl and Python had energy implications of over seventy times over Rust. Therefore it is justified to examine a representative programming language of each of the three categories in the empirical evaluation.

Table 2.2: Energy values of programming languages commonly found in REST APIs in comparison to C [21] (*outdated).

Ranking	**Language**	**Execution Type**	**Energy**
1	C	Compiled	1.00
2	Rust	Compiled	1.03
3	C++	Compiled	1.34
5	Java	Virtual-machine	1.98
13	C#	Virtual-machine	3.14
14	Go	Compiled	3.23
17	JavaScript	Interpreted *	4.45
21	PHP	Interpreted	29.30
25	Ruby	Interpreted	69.91
26	Python	Interpreted	75.88

As shown in Table 2.2, if we examine, for example, two fairly new and emerging programming languages, namely Rust and Go, the former prevails over the latter.

Table 2.3: Pareto optimal sets (no one dominant language in each set) for Energy, Time and Memory. [21]

Ranking	**Programming languages**
1	C, Pascal, Go
2	Rust, C++, Fortran
3	Ada
4	Java, Chape, Lisp, Ocaml
5	Swift, Haskell, C#
6	Dart, F#, Racket, Hack, PHP
7	JavaScript, Ruby, Python
8	TypeScript, Erlang
9	Lua, JRuby, Perl

However, when the programming languages are evaluated in relation to energy, time **and** memory, C, Pascal and Go share the first rank while Rust, C++ and Fortran come in at second place, as shown in Table 2.3. We can observe changes in the ranking once again when the evaluation is in relation to energy and time, where Rust is in second place and Go merely seventh, whereas with energy and time being the points of analysis, they share the second rank. [21] This illustrates that evaluating the energy consumption or energy efficiency based on one variable alone might not be sufficient and more in-depth analysis is needed. Hence, in this thesis, the aim is to emulate the real-world implementations of REST APIs as well as possible and compare and analyze the results based on an entire system as opposed to one specific subpart of it.

An important note regarding the results of reference [21] is that the findings are subject to inaccuracies (regarding TypeScript in particular) and they have sparked discussion and criticism [37]. The study is also not exactly the newest in 2025, being 8 years old, and is somewhat outdated. For instance, JavaScript should be categorized as JIT compiled rather than interpreted [38]. Regardless of all the aforementioned, the study goes to show that different programming languages can

have significant differences in their energy consumption. Other studies support the findings as well: reference [29], for example, found similar differences between programming languages in relation to energy consumption.

Managed languages, such as Java, which run on a virtual machine, can also have differing amounts of energy consumption based on the virtual machine platform implementation. For example, there can be as much as 100% difference between different JVM implementations. [39]

Altogether compiled languages seem to be the most energy-efficient. Managed languages come a close second while interpreted languages are clearly the least energy-efficient. The differences in programming languages give a clear indication that choosing the correct one to achieve high levels of energy efficiency is imperative.

2.2.2 Framework

API developers often utilize frameworks to build their APIs. Frameworks can be described as ready-made code or tools which help developers build the logic and communication interface side of their web application. These frameworks are essentially pre-written code which provides functionalities for common features found in REST APIs, such as HTTP request handling. As frameworks provide the aforementioned features (among others) out-of-the-box, developers do not have to re-invent the wheel each time they build a new API. Different framework implementations provide differing amounts of features and support for functionalities commonly found in REST APIs.

A thesis [30] published in 2023 compared web frameworks built with different programming languages and found that there are indeed differences in their energy consumption. For example, the JavaScript-based Express framework consumed less energy on average than the Python-based Django. In effect, when designing the architecture and selecting technologies for, for example, a web application's backend

framework, one should most likely consider using, for example, Rust's Actix [40] or Go's Fiber.

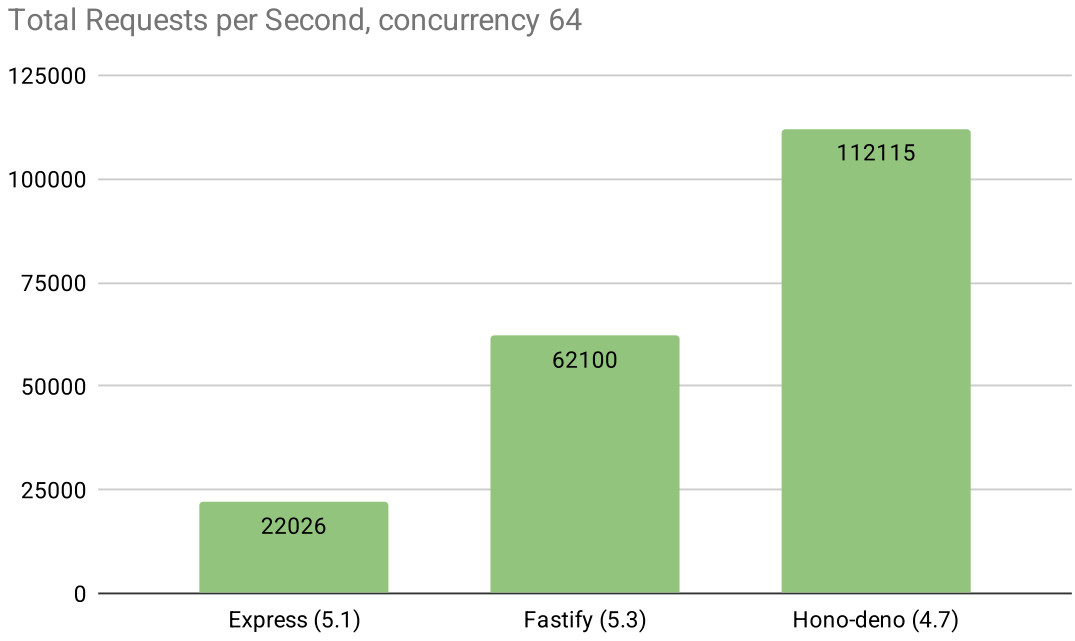

Figure 2.1: Performance differences in frameworks written in JavaScript. [41]

A benchmarking site, which compares many popular web frameworks, indicates great differences in performance between, for example, the following frameworks, all of which are written in JavaScript: Express, Hono and Fastify, as illustrated in Figure 2.1. [41] This implies that there is likely also a difference between the energy efficiency in different frameworks, even if they are built in the same language. A conference paper published in 2014 [42] empirically evaluated the performance differences in various web development technologies, namely Python, Node.js and PHP. They found there to be stark differences topics such as throughput and mean execution time. It is thus probable that the differences apply to the web domain and REST APIs as well.

Reference [27] found that while there are differences in the energy consumption of different frameworks, there was not a clearly most energy-efficient solution. The

scalability behaviour was different with the frameworks, as some reduced performance in order to keep power consumption low, while others increased power usage to maintain their performance and some used a combination of both. There were also differences in the frameworks based on the programming language; for instance, C++ and Rust stood out as the most energy efficient in a single-request context. In the absence of a clear winner, the paper suggested a basing the selection of the framework based on specific needs and use case. [27]

In summary, different frameworks seem to exhibit differences in power use, performance, scalability and energy efficiency. Thus, choosing the correct one to help with developing an energy-efficient REST API is important.

2.2.3 Paradigms

In *parallelism*, multiple processes (instances of a program or a part of a program) can be running at the same time. Assuming only one CPU is being utilized, at any given instant, only a single process can be allocated CPU time. With a single core, a quasi-parallel system can be achieved. That is, from a human's perspective the processes appear to be running in parallel, while in fact the CPU is switching between the processes in a rapid manner. Concurrency means that there are multiple processes in progress, but this can be achieved via a quasi-parallel system that utilizes time slicing, i.e. dividing the available CPU time across multiple processes. [43] The difference between parallelism and concurrency boils down to apparent same time and exact same time [44], respectively. True parallelism cannot be achieved with multi-threading - only multiple physical CPUs can enable said ability. [43]

Parallel programming and multi-threading have been shown to be able to improve the energy efficiency of a software application, provided that it has been configured correctly. [19] That is, in order to achieve better energy efficiency the application has to, for example, utilize the optimal amount of threads among other factors. [26]

Thus, it needs to be taken into account that parallelism is not a one-size-fits-all solution. Thread management constructs such as work stealing, fixed-size thread pooling and explicit thread creation can affect the energy efficiency of software. [26] Explicit thread creation was the most energy-efficient for I/O-bound software [26], which REST APIs commonly are. Different algorithms, such as the one utilized in *thread shuffling*, can also affect the energy consumption. The selection of use-case-specific amount of threads is recommended for maximal energy efficiency. [26]

The Go programming language, for example, can automatically divide the load among the threads of the operating system via the runtime. Concurrency in Go works with "goroutines", an implementation of green threads, which are scheduled by a runtime as opposed to the operating system. Green threads are becoming more popular and they are already in use with for instance Java [45] and the aforementioned Go [46]. The Go runtime automatically multiplexes the green threads as necessary to a configurable amount of CPU cores, which, by default, is the maximum amount of cores available on the device the program is running on [47] [48].

In web development, thread and concurrency management is often abstracted away to be taken care of by the framework used. For example, Go's Echo framework, which utilizes the net/http package, starts a new service goroutine (i.e. green thread) for each listener to divide the load of the HTTP requests [49]. Not all programming languages commonly used in web development fully support application level concurrency. For example, JavaScript programs, which are single-threaded, run on the Node.js runtime, which enables concurrency via the event loop. The event loop in turn offloads the tasks to the system kernel [50].

At least for resource-altering operations, i.e. CRUD operations, it is customary in web development for the framework, such as Gin for Golang, to handle the majority of multi-threading. This means, for instance, that instead of the developer ensuring that each request is performed separately, the framework does it for us.

The empirical section of this thesis will only utilize multi-threading, parallelism and concurrency as they are utilized by the tested frameworks. As highlighted, however, multi-threading and parallel programming are topics that should be considered when it comes to energy efficiency.

2.3 Optimizing performance and minimizing overhead

2.3.1 Batch operations

HTTP is the standard communication protocol in the web domain. When an end-user is using a web application, the user-facing frontend of a the application, i.e. the client application, typically makes multiple HTTP requests to the backend, in this case a REST API. The API in turn commonly processes data or makes calculations and provides said data as a response. The client application then displays said information to the user. These individual requests can (provided that they utilize the HTTP protocol) be grouped into one, forming a single request. For example, Microsoft provides a web API for its Dataverse product, which supports batching requests. [51]

The batch operations' energy efficiency implications were studied in the context of mobile applications in 2019 by Cruz et al. [6]. Rani et al. studied whether said approach could be used in web applications. They found that the HTTP request batching, which they classified as an energy pattern (as opposed to anti-pattern), is indeed applicable to web applications as well [4]. Cruz et al. argue that each request induces superfluous energy consumption caused by the starting and stopping of resources each time a request arrives and is handled. The energy usage curve of these "tail energy consumptions" can be flattened by grouping the individual

requests into one. This effectively reduces the energy losses originating from the unnecessary resource starting and stopping caused by the nonessential requests. [6] As highlighted by Cruz et al., these appear especially in the mobile domain. In the web domain, it is likely that the aforementioned problem could be more prevalent with serverless systems due to an unavoidable cold boot period, but as it is not in the scope of the thesis, an empirical comparison for said architecture will not be conducted. Instead, the effects on a REST API will be observed.

As the goal is to minimize energy consumption, it is feasible to assume request grouping to be beneficial, but utilizing said feature may be very situational as could introduce latency and other problems depending on the usage scenario. Nevertheless, it's worth taking into account as a viable means of optimizing energy use and reducing wasted energy.

2.3.2 Algorithm efficiency

While traditionally CPU-intensive operations, such as sorting, can beat least to some extent ignored as the responsibility of the database engine, I myself have seen many instances where the REST API itself has to perform such operations in the application code. Thus the need for such functionality cannot be ruled out in REST APIs. Different algorithms for tasks such as filtering, deduplication or aggregation can be required in a REST API depending on the features and an optimal algorithm should be chosen.

Different algorithmic choices can have a significant effect on the energy consumption of the application [31], which means choosing the correct algorithm for a specific task is imperative when it comes to minimizing the energy consumption of the software. The algorithm used for, for example, array sorting tasks, can also have significant performance implications. There are great differences in time complexities for, for instance, the algorithms selection sort, quick sort and bucket sort.

[19] There are of course algorithms for many other tasks in addition to sorting and developers do not usually write the algorithms from scratch. Instead, existing solutions are chosen or the choice is implicit based on the technology used, be it the standard library of a programming language or a feature of a specific framework.

2.3.3 Extraneous tasks

As with any software, the less operations performed for a completed task the more optimized the application is. Fewer instructions usually result in a more efficient software [19]. When it comes to REST APIs, performing tasks such as third-party API calls, superfluous data synchronization or frequent checks for an update can be categorized as energy-hungry operations that can be optimized [4]. Acknowledging the issues and reducing these kinds of operations to a minimum can curtail the energy overhead from such superfluous operations. Superfluous in this case could mean, for example, that the request is either made regardless of the fact that the same data is available locally already or that unnecessary data synchronization is performed when there is nothing to update. These requests also inherently increase the network load and thus the energy consumption even if minimal processing is done to the data received.

If performing an operation can be skipped, it has the time complexity of *O(0)*, meaning it should always be preferred to performing some operation, provided it has no effect on the desired end result. There's also an option to store, for instance, calculations into a cache memory, so the expensive CPU operation does not have to be performed, even though fetching the result from a cache does result in some energy consumption.

2.4 Optimizing data use

Due to the abundant nature of web APIs and clients, there is a need to keep the amount of data sent between the client and the web API as small as possible. Web applications transfer differing amounts data between the server and the client in addition to utilizing external, other web APIs. In this thesis we focus on the aforementioned three-part system configuration, excluding, for example, third-party API calls. The transferred data could be the client uploading images, text or other forms of data to the server or the server sending data to the client that they have requested to download. The energy usage could depend on many things, such as the amount of data transmitted (network-bound operations) [26], data format and compression, the latter two of which will be covered in section 2.4.2.

2.4.1 Data amount and network load minimization

In the context of APIs, the energy overhead can be minimized by reducing the amount of data movement [52]. After all, the more packets that have to be sent over the network, the more energy it consumes. If no request are made over a network, the energy consumption overhead from moving data from one machine to another does not exist. For example, retrieving data from an in-memory cache instead of making a database query is most likely a better option when it comes to energy consumption.

Using design techniques such as *pagination* or *lazy loading* could reduce the amount of data transmitted. When lazy loading is utilized, the content or data is requested only when needed. In other words, if, for example, an image on the website is not visible to the user, the client does not request for that specific piece of data before it's deemed relevant to do so, i.e. when the user scrolls to see it. Essentially, parts of the page are loaded and inserted into the DOM only when the user is supposed to be able to see it, making it possible to have reductions in the amount

of data transferred. However, it should be noted that it's not conspicuous whether having the client make multiple smaller requests instead of one large one uses more or less energy. Pagination, on the other, can make the the response fraction of the size, if instead of loading the whole data set a page consists of, for example, only 10% of the total amount of data and the second fraction of the data is sent to the client only when they switch pages. Rani *et al.* [4] discussed the aforementioned features, what they call OOWM (Open Only When Needed), and even though they proved to, in fact, increase the total energy consumption, the measurement was done with the same amount of data loaded for both the pattern (OOWN) and the anti-pattern (not using OOWN). This arguably defeats the purpose of the feature, as the point is to not have to load all the resources. It should also be noted that pagination might even hinder the energy efficiency of the API, if the client uses all of the data but requests it in small parts.

2.4.2 Data compression, structure and format

Using ill-considered data structures can cause considerable increases in energy consumption; in some real-world examples, by choosing the most efficient data structure, energy savings of up to 38% could be achieved [26]. API developers can choose between different supported data formats for text and media data. For text-based content, formats such as XML, JSON and CSV are popular. For media content such as images, formats such as PNG, JPEG and AVIF are popular. There are differences in file sizes in image format. For example, the file sizes can be reduced in browsers by utilizing AVIF and WebP formats in image files. [4] The images can also be compressed [19] [53]. Data compression, if applied correctly, has been found to be able to reduce the energy costs of transmitting data at least in mobile devices [54]. HTTP compression is a technique viable for reducing, for example, the size of JSON and XML files due to the predictable (and thus redundant) nature of lot of

the data, such as repeating characters. Both the image format and data compression can have a big impact on the energy consumption on the energy consumption resulting from data transfer [19]. This can be situational, however, as compression requires serialization and deserialization [55].

2.4.3 Caching

Temporarily storing data on the server can reduce load times by the data being temporarily saved in a location that is faster to access than it's ordinary location, which would usually be a database. In practice this could mean storing frequently accessed data in an in-memory cache as opposed to reading from disk each time. This can also reduce superfluous processing of said data. Utilizing caching on repeatedly used data on both the client application and the server may have significant energy efficiency improvement implications. For example, in the context of web applications, if a needed image is cached in the client, e.g. the browser application, the need for a HTTP request to fetch it from the API is eliminated and the server does not have to use any resources at all. This means that the REST API's energy consumption for that specific "request" is 0. On the other hand, if the API utilizes a cache, it could fetch frequently requested data from the cache instead of the database, which eliminates the need for a database call. A study of performance issues in JavaScript programs in 2016 found that of the 16 open-source projects studied, 13% suffered issues from repeatedly executing the same operations and thus could have benefited (performance-wise) from caching the frequently-used results [56].

Caching can be used to reduce network load [4], and if a newer version of the cached data does not exist, there is no reason to re-fetch it from the server. The application should subsequently ascertain that an update exists before making a request for a piece of cached data [6].

In summary, caching frequently accessed data or results of expensive or performance-

intensive calculations can be beneficial in terms of energy efficiency. The utilization of caching should be considered to reduce the energy consumption of software, should the use case allow for it.

Use of CDNs

Content Delivery Networks (CDN) are a feature many cloud providers offer, where frequently used data is stored geographically close to where it's being used as opposed to having to be fetched from the other side of the world. CDNs can be considered as one form of a caching system. The effects of CDNs are most likely hard to emulate and thus out of scope for this thesis. It should be noted, however, that they may have significant energy consumption implications as the geological location of the server makes each packet pass through many more nodes than if it's close to the client relatively speaking, likely requiring a lot more energy especially if the transferred data amounts are big.

3 Measuring software energy consumption

As discussed, running software induces energy consumption. To monitor and minimize any potential energy overhead, the consumption needs to be measured or estimated to detect energy-hungry sections or modules in the software code at all levels of granularity. There are two main ways the measuring of energy consumption in software can be approached; the **software-based** and **hardware-based** methods. [57] A third, additional approach, namely **hybrid** methods, involves including measures from both of the previously mentioned approaches. [58] This chapter answers to the **RQ2** "How can software energy consumption be measured?".

3.1 Hardware-based measuring

Measuring the energy consumption of software often calls for reliance on attributes which can be measured in a straightforward manner, such as electricity or power consumption. Hardware-based measuring is typically performed by placing an external hardware power meter between the power source and the SUT (System Under Test). A study published in 2013 [59] used a sensor placed in between the system to be measured and it's power source to estimate the energy usage. The instantaneous energy consumption can be inferred from the equation for the calculation of energy,

which is

$$E = Pt$$

which can be used to calculate the used energy from the power consumption (P) of the system as well as how long the operation takes (t). Measuring in practice is, not that simple, of course, as the power consumption is not constant over the duration of the operation.

As software programs run on an operating system and thus have an idle energy consumption, said overhead needs to be accounted for in the measurements when absolute energy consumption values are wanted. An average idle power profile can be measured, which is then subtracted from the total energy consumption to exclude the idle energy consumption overhead as best as possible. This means the energy consumption is determined as the difference between the measured energy consumption and the idle energy consumption. [60]

3.1.1 Alternatives within hardware-based measurement

When it comes to hardware-based measurements, hardware power meters are the typical approach, in which general-purpose measuring equipment is utilized gather the data. As an example, EET or *Energy Efficiency Tester* [61] and Watts Up? [62] are such devices. Additionally, there are specialized computer systems which incorporate energy-sampling sensors, which in turn allow for the monitoring of power consumption of the software program as well as the hardware resources associated with it. [58]

3.2 Software-based measuring

There exist a multitude of different software tools that can be utilized to estimate energy consumption. While software-based measuring is, in general, less accurate

than hardware-based measuring, it often provides cost savings by requiring less effort during the implementation process [57]. A study published in 2019 [58] delved into the different methods for measuring the power and energy consumption of software applications. They point out that while hardware-based approaches are more accurate in comparison to software tools, software tools can be used to measure the power consumption at different levels of granularity for both the hardware and the software. The granularity in this case refers to hardware features such as CPU or DRAM and structural levels of software, such as a line of code, a method or an instruction. [58]

A popular interface utilized in software-based measuring is the RAPL (Running Average Power Limit) interface - many studies that measure the energy or power consumption of software have used said interface [63] [64] [26] [4] [65]. It has been shown to be quite accurate as a software-based power consumption measuring tool [66] [63] and, in addition, it causes a negligible performance overhead [67]. An example of a tool which utilizes the interface is Intel PowerLog [4]. RAPL is a processor feature developed by Intel. It is an interface which can be used for energy consumption reporting. The interface provides a mechanism for tracking the cumulative energy usage across different power domains within an SoC (system-on-chip). [68] RAPL can provide real-time information on both DRAM memory and the CPU package. The following domains can be measured with RAPL: `Package`, `Power Plane 0`, `Power Plane 1`, `DRAM` and `Psys`. It should be noted however, that not all models support all features. `PSys`, for example, was introduced in the Intel Skylake architecture. The energy information is stored into MSRs (Model-specific registers), called energy counters, which update approximately 1000 times in a second. The data is indicated as energy consumed since booting up the processor, in units specific to the processor model in use, such as 61 microjoules in the case of the Skylake architecture. [67]

3.3 Hybrid approach

The hybrid measuring method incorporates elements from both software and hardware-based methods. More information can be provided with this kind of an approach, the aim of which is to collect more precise and broad readings by combining capabilities from both approaches. Examples of hybrid tools include e-Surgeon and PowerScope [58]. The combined results can be used to observe and identify commonalities, patterns and causation within the measurements better than a either of the measurement approaches could by themselves.

3.4 Approach in this thesis

While the measurement setup in use had the possibility to include software-based data in the form of RAPL readings and thus offer the benefits of a hybrid approach, due to time constraints and for the sake of simplicity it was determined that a hardware-based approach would suffice. Thus, the measuring setup utilized in this thesis is based on a hardware setup and the measurements are generated by a power meter based measuring system which is described in detail in Chapter 5.

The energy calculations will be done cumulatively, meaning as the measurement values are discrete, that is they are interpreted to stay the same for the duration of a single data point. This means that as each data point has a time stamp for the run, its duration in the data point is timestamp n + 1 minus timestamp n, resulting in the formula and the instant power consumption at the time of the data point

$$E_n = P_n \cdot \Delta t_n$$

for each individual data point. These data point are then summed to form the total energy consumption.

4 Test programs and test cases

In this thesis we aim to improve existing systems and provide a basis for enhancing future implementations of REST APIs in relation to energy usage. The aim is not to prove a theoretically most energy-efficient system possible. While the empirical test cases are inherently non-exhaustive, for the purposes of limiting the empirical validation to a reasonable subset of what is possible to achieve in the context of energy savings, this thesis focuses only on characteristics deemed relevant for achieving the aforementioned goal. That is, finding features and characteristics in server-side applications that can be either improved up on or utilized in the first place.

It's worth noting that only expected usage, use cases and scenarios in general are included in the evaluation and consideration. Unexpected (and excluded) scenarios could include, for example, temporary unavailability of parts of the system the API is dependent on, such as network communications or the database connection. This thesis focuses only on expected, intended and typical behaviour of the system.

4.1 Selected programming languages and frameworks

The selected programming language categories for this thesis are the three discussed in Chapter 2: compiled, managed and interpreted. For each technology, an alike implementation of a REST API was created.

4.1.1 Interpreted language: PHP with framework Laravel

PHP was selected for testing for its superior popularity among server-side programming languages, as according to W3Techs Web Technology Surveys, 75,9% of websites use PHP in their server-side software. [69] PHP is quite an old but popular programming language first released in 1995. Designed with web development in mind, it is a general-purpose scripting language whose name originally stood for *Personal Home Page*.

The selected framework for PHP was the Laravel framework. Laravel was found to be the most popular PHP framework in the 2024 Stack Overflow Developer Survey [70] and was thus selected for this thesis. The PHP version is 8.2 and Laravel version is 11.9, as it was Laravel's newest version at the time of the starting of the project, which used PHP version 8.2.

4.1.2 Compiled language: Go with framework Gin

Go (also known as Golang) is a popular [71], compiled, garbage-collected and statically typed programming language originally developed and supported by Google. It is an open-source project and it has been designed to have a low learning curve but to still be highly performant. Go has rapid run and compilation times and has built-in support for concurrency with *goroutines* and channels as well as memory management. Go was first publicly announced in 2009 with an official release coming out in 2012. Reference [9] found Go to be the best-suited language for servers. Thus, was selected into testing as a garbage collected representative of the (hypothesized) energy-efficient [21] programming languages.

The framework selected for building the test implementation of the REST API for Go is the Gin framework. It is one of the most high-performing frameworks for Go [72], and was chosen for its popularity; as of the 15th of September 2024, the Gin framework's GitHub repository has the most stars of all the Go web frameworks in

Github. In fact, Gin has more starts than the following two frameworks Fiber end Beego combined. [73] [74] Go version 1.22.2 was used with the version 1.10.0 of Gin.

4.1.3 Managed language: Java with framework Spring Boot

Java was selected as the managed programming language representative, as it is still very popular [75] . Java is an object-oriented general-purpose programming language that is run on the Java Virtual Machine (JVM) platform. Like Go, it is a garbage collected language, but unlike Go it requires the JVM platform to run, although recently it has become possible to compile Java programs to compiled native binaries through, for example, GraalVM [76]. Like PHP, Java was first released in 1995 and is still one of the most popular programming languages. Java version 21 was chosen due to it being the latest long term support (LTS) version. Spring Boot was equivalently chosen for its popularity. As far as web frameworks go, it is the most popular Java framework according to the 2024 Stack Overflow Developer Survey [70]. Spring Boot version 3.3.4 was used.

4.2 Unifying utilities around the software

The decision was made to Dockerize the test implementations and make all of them utilize MongoDB to unify their functionality as much as possible regarding functionality that does not fall under the empirical tests. **Docker** is a software containerization tool, which provides features for packaging the application software in a container effectively separating it from the infrastructure [77], such as the hardware and operating system in use. It should be noted that using Docker as opposed to, for example, "bare-metal" Linux could introduce an energy consumption overhead [78]. However, a study [27] found it to not seem to add any significant additional energy variation, and Docker-based benchmarks even had slightly smaller standard

deviation in comparison to binary versions. Therefore, as all of the concerned test programs in the empirical phase will utilize Docker in an alike manner, the energy overhead caused by Docker is likely to be small and more importantly, similar across the implementations with differing underlying technologies.

MongoDB is a modern, document-oriented, general purpose NoSQL database management system (DBMS). The test implementation programs utilize MongoDB as a database. The effects of different DBMSs are not included in the scope of this thesis, and the choice of the DBMS should not have an effect on the test results, as the all of the different test REST API implementations utilize the same database. The choice of MongoDB was affected by its easy implementation and thus inclusion in the test programs. Furthermore, it is the most popular NoSQL DBMS among developers [79] and the most familiar to myself as well, which inherently helped to keep the development timeline more compact.

An important note regarding MongoDB is that the device the tests were run on did not support AVX instructions, so a special docker image [1] was used to circumvent the issue. The image is not intended for production but was sufficient for testing purposes.

4.3 Technology-related tests

In these tests the intention is to simulate the use of a REST API with some representative features as elaborated on in the table 4.1. The results can then be examined as total energy consumption as well as compare different operations and how different technologies performed with each of the operations.

The simulation of traffic on the API will be performed with the K6 performance testing tool. K6 is a HTTP benchmarking utility, which can generate a heavy load and simulate a considerable amount of demand on the SUT [27], *system-under-test*,

[1] `https://hub.docker.com/r/l33tlamer/mongodb-without-avx`

in this case, the reference implementation of a REST API. A similar method was used in a study [27] which investigated the effects of the web framework stack on the energy consumption of a server. They used a similar tool, WRK [80], for simulating a heavy load on an API. This thesis will be utilizing the same strategy for the empirical tests, with the differentiating factor being the utilization of K6 in place of WRK.

The available endpoints for the REST API implementations seen in Table 4.1 mimic the some common functionality of a contemporary REST API, with the capability to list, create, update and delete resources. All of the aforementioned endpoints make a database query to either get, insert, update or delete a single entry in the MongoDB database. These represent the so-called I/O-bound operations. These functionalities represent operations which primarily alter resources as opposed to making computationally heavy operations. The CPU-bound operations on the other hand, such as the Matrix multiplication calculation endpoint, are meant to simulate computationally demanding endpoint functionalities. Said functionality is intended to highlight differences in especially the performance capabilities of the different technologies, if any.

To highlight the differences as best as possible between the technologies, many tasks that could be outsourced to the database engine were handled in application code instead. In my experience, this also reflects real-life implementations at least in some cases, as not all code optimized to its fullest potential due to differences in developer skill levels. An example of such a scenario is the sorting of the entries in the `GET /notes` endpoint. The update endpoint `PUT /notes/:id` also performs the manipulation of the entry in application code as opposed to letting the database engine handle that by performing an optimal query.

The client script written in JavaScript for controlling the measurement setup (described in Chapter 5) via its API and the actual test runs is shown in Listing

Table 4.1: The available endpoints of the reference REST API implementations.

Method	URL	Request payload	Endpoint type	Description
GET	"/notes"	-	I/O-bound	List all notes in the database ordered by creation time, sorted in application code
POST	"/notes"	{ "name": *string*, "title": *string*, "description": *string* }	I/O-bound	Add a note to the database
PUT	"/notes/:id"	{ "name": *string*, "title": *string*, "description": *string* }	I/O-bound	Update a note
DELETE	"/notes/:id"	-	I/O-bound	Delete a note from the database
POST	"/matrix"	{ "matrix_A": *[][]float*, "matrix_B": *[][]float* }	CPU-bound	Multiply the provided matrices by each other
POST	"/serialization"	*[]*{ "index": *integer*, "guid": *string*, "isActive": *boolean*, "balance": *string*, "picture": *string*, "age": *integer*, "eyeColor": *string*, "name": *string*, "gender": *string*, "company": *string*, "email": *string*, "phone": *string*, "address": *string*, "about": *string*, "registered": *string*, "latitude": *float*, "longitude": *float*, "tags": *[]string*, "friends": *[]object*, "greeting": *string*, "favoriteFruit": *string* }	CPU & IO-bound	Accept a large (3.1MiB) payload of JSON, serialize and store, query from database, deserialize and drop collection, return in response

1. The script invokes methods for controlling PowerGoblin, setting run-specific parameters such as host and port and starting the K6 load test script.

Listing 1 JavaScript client script used for controlling the test runs.

```
const runTest = async () => {
  try {
    console.log("Starting client script on", HOST+":"+PORT);
    setHost();
    await setPort();
    // Empties the Notes collection
    await clearSUTDataBase();

    console.log("Populating database with notes")
    for (let i = 0; i < 150; i++) {
      // Adds a Note to database
      await testPost();
    }

    await startMeasurement();

    for (let i = 0; i < 20; i++) {
      await startRun(i);
      await runLoadTest();
      await stopRun(i);
    }

    await stopMeasurement();
    console.log("Measurement process completed");
  } catch (error) {
    console.error("Error during measurement process:", error.message);
  }
};
```

4.4 Technology-agnostic testing

Technology-agnostic refers to features and semantic attributes of the REST API unrelated to specific technologies. In practice this means, for example, caching or batch requests, which can be implemented regardless of whether the underlying programming language for the REST API is Rust, Perl or any other programming language for that matter, provided that the language has support for such technology in the form of a library. Whether or not such features are implemented to begin with and more importantly how effectively they are leveraged can have a significant

impact on the energy consumption of a REST API, independent of the specific technology used. The feature-specific tests are explained in-detail in this section and summarized in Table 4.2.

4.4.1 Batch requests

The methodology for this test is to run the performance benchmark tool and compare the results when the CRUD operations are performed individually versus in batches on a single CRUD operation set. The REST API implemented for the technology-related tests is used as a reference point and compared against the batch version implemented for this test scenario, with the small difference of the GET requests use the id of the POST request's resource and only fetch one entry.

The tests are conducted by sending requests to perform a set of four operations. The set of operations is performed 5000 times on the REST API during a single test run. The operations on the non-batch endpoints are as follows:

1. Create a new `Note`: `POST` request with `Note` in request body, returns `ID`
2. Retrieve the newly-created `Note`: `GET` request with `ID` as a path parameter, returns `Note`
3. Update the `Note`: `PUT` request with `ID` as a path parameter and new fields for `Note` in request body, returns updated `Note`
4. Delete the `Note`: `DELETE` request with `ID` as a path parameter, returns `ID`

Correspondingly, the same operations are performed on an alike resource (`Note`) using the batch endpoint. The difference is that only a single HTTP request will be used to perform the four operations on the resource. For instance, all the data required for inserting, updating and deleting the `Note` is included in the request body. Having completed the measurements, the resulting power graph from should give an indication of whether there is a non-negligible benefit gained in relation to energy and power use of the REST API from request batching.

It is worth noting that the code does not utilize any bulk queries supported by MongoDB, such as `InsertMany`, `UpdateMany` or `DeleteMany`, as these functionalities can (and arguably should) be implemented even if request batching is not utilized in any manner whatsoever. HTTP request batching, specifically, is the point of focus for this set of tests. Furthermore, while utilizing the features of the database engine in an optimal way would undoubtedly increase, if not the API's energy consumption, at least the its performance, it is out of scope for the thesis and thus such features were intentionally left out of the tests.

The intention of these tests is to highlight possible differing energy requirements between performing multiple operations with one request and traditional one operation per request. Additionally we will try to answer **RQ3** "How effective are the existing green coding practices for REST APIs?" and analyze whether the energy savings achieved are negligible. This test is non-exhaustive in a sense that not all contemporary REST API operations have support for purely CRUD operations. Moreover, as will be explained in Chapter 5, the measurements setup uses local devices and thus cannot represent real-world network conditions and overhead resulting from HTTP requests travelling via the internet, meaning more in-depth research into HTTP request batching will be requires should the test results indicate an improvement in energy efficiency in the API.

4.4.2 Caching

Cache tests will utilize Redis as a caching system. Redis is an in-memory key–value database commonly used as a scalable, high-performance caching solution. These tests will highlight how the energy consumption changes if the operations are performed on Redis instead of the database system. For example, fetching a portion of the data from the in-memory storage as opposed to the database should highlight the impact on the energy consumption of the system. The tests are conducted by

increasing the amount of operations performed on Redis instead of the database. The results are observed for differing amounts of energy savings and the correlation with cache utilization level. In other words, analyze whether a larger hit rate results in more energy savings.

The tests are executed for 100%, 50% and 0% hit rates. For consistence purposes the randomization will be performed on the client as opposed to the SUT. This way the effects of different randomization solutions in the test programs will be eliminated. The hit rate is determined like follows: for each of the endpoint types, namely MongoDB and Redis, a complete test script is executed exclusively. That is, for 0% hit rate (MongoDB-exclusive tests), all read operations are performed on the database and vice versa for the 100% hit rate (Redis-exclusive). As for the 50% hit rate, it is guaranteed by having "flipper" property in the K6 script, as shown in Listing 2.

Listing 2 50% hit rate logic for K6.

```
let flipper = true;

export default function () {
  // 50% hit rate tests
  if (__ENV.HITRATE) { // Environment variable for enabling flipper
    type = flipper ? "mongo" : "redis";
    flipper = !flipper;
  }
  // ...

  // type determines whether Redis is used
  http.get(`${HOST}:${PORT}/${type}/${id}`);

  // ...
}
```

The test flow is the following: POST/create + GET/read a resource. Both the cache and the database are are updated on all POST requests when the cache is utilized. In Mongo-exclusive tests Redis is disabled to highlight the overhead

from said service running. In 50% and 100% hit rate tests the aim is to compare the overhead from using Redis and the possible efficiency improvements regarding energy use, which will show whether utilizing the in-memory cache is indeed worth it if energy use minimization is the goal.

By default MongoDB's WiredTiger storage engine uses an internal caching system to store data in RAM. To minimize the use of this internal in-memory caching system and thus highlight the differences between the in-memory cache (Redis) and a disk-reading database system (MongoDB in the context of the thesis), all requests in the load tests will perform operations on a new entry, that is a `Note` with new field values. This means that the flow for a single test run is the following: create a resource in both MongoDB and Redis, query said entry from one or the other depending on the hit rate.

4.4.3 Algorithms

As with any algorithm, sorting tasks can have an effect on the energy consumption of the API. The aim of these tests is to observe whether there is a great difference between algorithms and their energy consumption. CPU-bound operations are in focus in these tests, as no resource-manipulation is performed.

The effects of algorithmic efficiency are tested by running tests for three different sorting algorithms. The algorithms that were chosen for evaluation in this testing round were **selection** sort, **insertion** sort and **quick** sort, for which the best-case time complexities are $O(n^2)$, O(n) and O(nlogn), respectively. The average case time complexities are $O(n^2)$ for both selection and insertion sort, whereas for quick sort it is O(nlogn). The worst case time complexity is same for all algorithms, $O(n^2)$. Selection sort is efficient for small sorting amounts of data but ill-suited for large inputs, whereas quick sort is well-suited for handling large input data amount while being inefficient with input data being already sorted. Insertion sort is simple and

works efficiently with small sets of data, while as with quick sort, it is inefficient with large data sets. [81]

The test will be performed by querying a randomized (yet same across different tests) amount of documents from the database and sorting them with each of the algorithms on different test runs. The sorting will be based on the *created_at* Unix timestamp. Analyze the results for significant and determinable energy savings caused by algorithm change. The arrays of unsorted resources are always unsorted in the beginning. These tests should highlight the difference the design and implementation choices of the developers can make on the energy consumption of the API.

Table 4.2: Technology-agnostic tests.

Feature	**Hypothesis on energy savings**	**Test type**
Batch requests	Reduced energy use from less network traffic and fewer requests	Individual requests vs. batches of 4
Caching	Reduced energy use from queries to an in-memory storage as opposed to a disk-reading DBMS	Database only vs. cache only vs. 50% hit rate
Algorithms	Reduced energy use from more efficient CPU utilization	Comparison of energy efficiency between algorithms

4.4.4 REST API implementation

Since the technology of the implementation is not of importance due to the technology-agnostic nature of the tests, the decision was made to implement said tests on one the most familiar technology to the author, namely Go. While JavaScript, for example, would have been even more of a familiar technology to myself, it lacks multithreading due to it being a single-threaded language and might not have been an optimal candidate for the implementation of the base of the REST API for the aforemen-

tioned reasons. Go has great community support and libraries for features such as caching for, for instance, the aforementioned Redis. Additionally, it is not uncommon to see said programming language used on the server side code of contemporary web applications nor in REST APIs, which makes it an ideal candidate to building the REST API implementation required in the tests. Furthermore, a base for the implementation was implemented in for the technology-related tests, which considerably shortened the development time for the base functionalities of the API, such as setting up the database and dockerizing the solution to help with using it on the measurement setup. Each of the non-technology-specific features are in the same REST API under their own routes (or endpoints).

5 Measurement Setup for Empirical Testing

When conducting empirical tests on the energy consumption of software, one should not overlook the possibility of the test execution environment having an impact on the results. Thus, the aim is to minimize the effect of the environment on the data that is gathered from the tests.

Benchmark tests should meet the following criteria to be successful: the test must be reproducible, results must be accurate (i.e. findings should be roughly the same each time the benchmark is run) and the tests should represent reality. [27] The measuring setup used in this thesis was similar to the one used in [59], where there is a sensor between the SUT and the power source and multiple runs for each scenario. The equation for the calculation of the energy consumption differs in slightly, as in this thesis the idle power is not subtracted. This is since absolute energy values are not the key point of interest. Instead, the relative performance between test cases is evaluated.

5.1 Test setup overview

This thesis utilizes a power measurement tool called PowerGoblin [82] [83] originally developed by the University of Turku under the Visiiri [84] project. The tool is designed to measure the power consumption of a system and collect, organize and

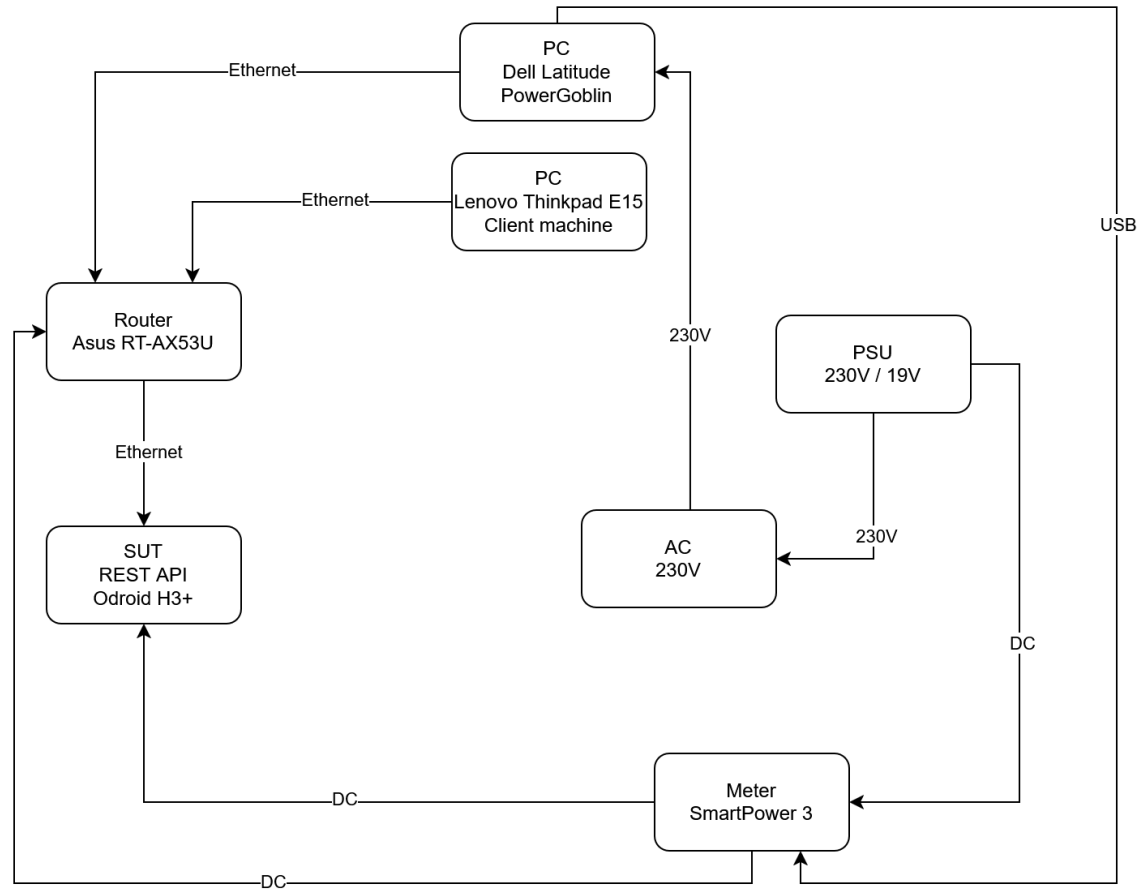

Figure 5.1: Physical connections of the test setup based on an example setup [85].

manage data from said measurements.

In addition to the measurement setup there is the server machine (SUT) as well as the client machine, which is responsible for providing the necessary traffic for the duration of the tests. As shown in Figure 5.1, the Lenovo PC acts as the client which sends the requests and causes the demand for the SUT. The SUT, on which the REST API is hosted, is connected to both the PowerGoblin hosting PC and the client PC with ethernet cables via a router. The power meter is in between the SUT and the PSU (Power Supply Unit), so that the power consumption of the system can accurately be measured.

5.2 Measurements process

Having set up the hardware and designed the test script, the measurement process can begin by first starting all the necessary elements, namely the REST API in the SUT and PowerGoblin to read measurement data. The measurement script can then be started. It controls the PowerGoblin via, for instance the HTTP API, as well as causes the demand on the REST API itself with HTTP requests. [86]

In the PowerGoblin software, a *session* is the starting point when measuring. Sessions, each with a distinct id, consist of data such as start and (optional) stop time stamps, configuration data and session-related aggregated data. Sessions can be either open (without stop time) or closed (with stop time), and the idea behind them is to have re-reviewable data of a session, in other words, that a session can be restored. [87]

Each session can contain multiple *measurements*, which represent a task under review and they are identified by trigger messages when starting and stopping the measurement. Each measurement can have multiple *runs*, which in turn represent a single test run for a single task. Runs are meant to provide a way to repetitions of a single measurement to, for example, rule out outlier scenarios. [87]

5.3 Features of PowerGoblin

PowerGoblin (version 2.0.0) has features for data acquisition, exportation, communication and deployment, which will be explored in this section.

5.3.1 Importing Measurement Data

The tool includes support for a power meter called *HardKernel SmartPower*, connected via USB for data acquisition as well as a USB UART (Universal Asynchronous Receiver-Transmitter) meter *HardKernel SmartPower 3*. Additionally, there is a

functionality for simulating dummy meters, which enables testing and simulation without the need for actual physical hardware. Furthermore, collectd can be utilized to provide data for a more thorough and comprehensive analysis. [88]

5.3.2 Exporting Measurement Data

Measurement data can be exported in multiple formats. These include the exportation of the entire measurement setup data as JSON, as well as exportation of event-specific data in CSV and OpenDocument formats. Additionally, for visual representation, PNG image plots can be generated using the *ggplot2* library, if graphical analysis is required. [88]

5.3.3 Communication & APIs

The protocol for communicating with PowerGoblin in this thesis was the HTTP API provided by the tool, via which the whole measuring process can be managed. The REST-like API allows programmatic control of measurements. For instance, a `GET` request to `/api/v1/run/start/:unit` would start a run within a session and measurement and `/api/v1/run/stop/:unit`, correspondingly, would stop the run. The API supports configuration for measurement-related parameters such as `session`, `measurement`, `run`, but also for system configuration such as `meter` and `cmd` which is for executing commands. Additionally, the HTTP POST API allows for uploading resource data for collectd. [88]

The web UI offers a user-friendly platform for visually inspecting the measurement process. The UI has six distinct components through which the user can observe and interact with the system. The *Config* section displays information relating to the configuration of the software, including enabled APIs as well as the meters present. The configuration parameters are not modifiable through the UI. The *Meters* section displays the available meters, which can be either simulated

or real meters. This section displays the instantaneous power, current and voltage values, which are updated periodically to view how the session in proceeding. The *Measure* section provides functionality to monitor and manage sessions, measurements and runs. One can, for instance, create a new one or manually control an active one. Within a sessions, measurements can be started or stopped. In addition, within measurements new runs can be started and stopped. The section *Logs* lists prior saved sessions. This view has links to files relating to their respective session. The *System* section contains useful information for troubleshooting, such as the latest MQTT messages and system logs. One can also access the HTTP GET API from this view. [89]

There is a support for MQTT communication, via which indirect monitoring of the measurement process is possible. This functionality provides real-time updates and integration support for MQTT-compatible systems. The Telnet API, on the other hand, implements a minimal protocol for controlling measurements. It utilizes a high performance `io_uring` backend to handle data. [88]

5.4 Server machine (SUT)

The server machine, in other words the system-under-test was an X86 architecture Odroid H3+ [1] by Hardkernel. The device has Intel Pentium Silver N6005 as its CPU, which has 4 cores and 4 threads. Memory-wise, two of Kingston's KF3200C20S4/8G DDR4 Synchronous 3200 MHz RAM (*Random-access memory*) sticks were installed. Graphics-wise, the device employed Intel UHD Graphics (Jasper Lake). As for storage, there is a 500GiB Kingston SNV2S500G NVMe SSD. The system has a Linux operating system with the *Arch Linux* distribution. The system was powered with the SmartPower 3, which supplied a voltage of 17.85V.

[1] `https://www.hardkernel.com/shop/odroid-h3-plus/`

5.5 Client machine

The client device utilized was a Lenovo Thinkpad E15 (2022 model), with a Ryzen 7 5700U CPU and 16GB of DDR4 RAM. The operating system on the device was Fedora Linux 41 with Node.js version 20, which was used for making HTTP requests to the measurement system to initiate, configure and stop the measurement process as well as to test the actual SUT. The client machine was connected to the SUT via a CAT5 RJ45 ethernet cable.

6 Results, Analysis and Discussion

This chapter goes over the results of the empirical tests. Two types of graphs were selected to visualize the performance of each practice. The first is power over time graphs to visualize the runtime behaviour, power-hungriness and differences in duration. The second are box plots of cumulative energy consumption. These graphs allow for easier comparison of median performance, variance and possible outliers. Additionally, bar charts of energy consumption were added for the technology tests to better highlight differences in the technologies.

6.1 Technology-related tests

Each test case for both the technological and the technology-agnostic cases had a session of its own on the measurement setup. Each session consisted of 20 test runs, which were controlled programmatically by using the HTTP API of the measurement system PowerGoblin. To ensure each session would be as alike to the others as possible, the `Notes` collection was cleared and populated with 150 entries before starting the measurement.

For the technology tests every run lasted the amount it took for each API implementation to run the specified amount of iterations, which was 50. This amounted in 300 requests for each test run, which consisted of the K6 load test period. The duration of a single run varied depending on the technology used, ranging from approximately 6 seconds to almost 20 seconds. K6 was configured to use 10 virtual

users during the test runs. A more detailed analysis of said runs and the variations is provided later in this section.

6.1.1 Test results

The results of the test runs were exported as CSV from the measurement setup. The data was plotted per-run to visualize the power consumption over time. Each run was plotted with a line of its own and a unique color help with distinguishing the runs from each other. Since a specific amount of iterations was configured for the performance testing tool, the duration of each run could differ based on how fast the API was able to process the requests. These differences between the API implementations and the technologies can be clearly seen in Figures 6.1, 6.3 and 6.5. In general the measurement data for Spring Boot and Gin look quite similar, whereas Laravel is quite conspicuously less performant in comparison to the former.

The cumulative energy consumption for each run can be seen in Figures 6.2, 6.4 and 6.6. All the aforementioned figures have the warm up run marked in gray to differentiate it from the rest, while still providing data on the differences of the technologies.

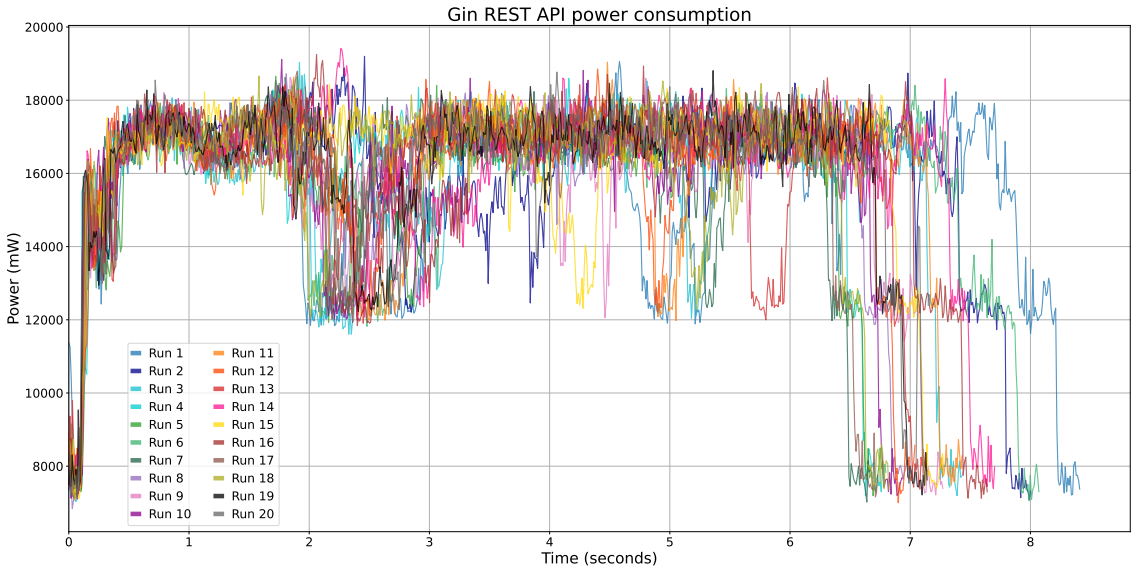

Figure 6.1: Power consumption of the REST API with Gin framework.

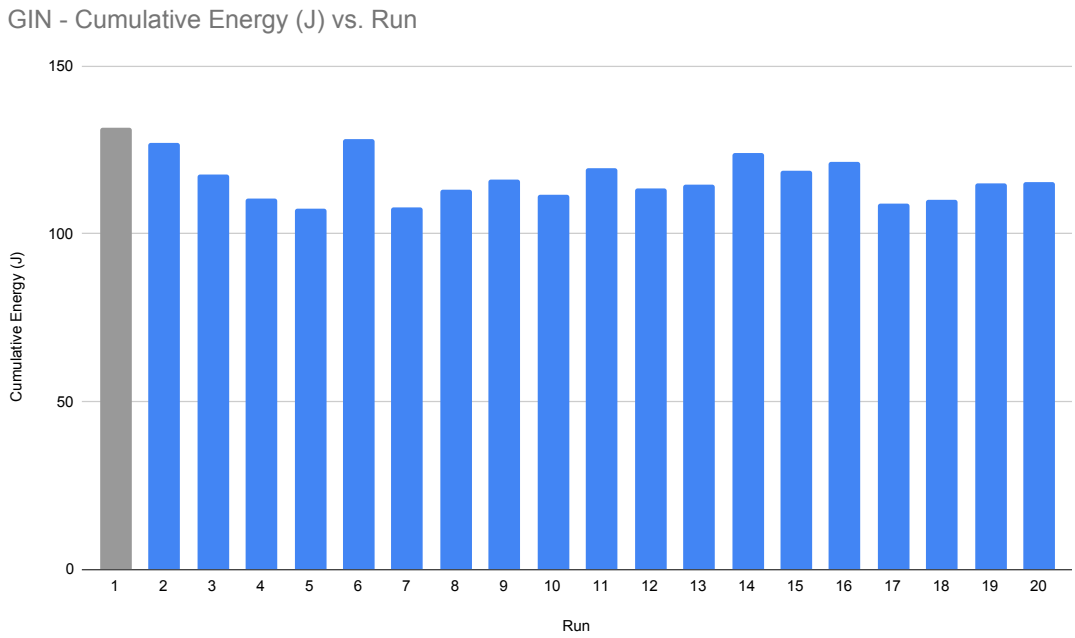

Figure 6.2: Gin REST API's energy consumption for each run (gray indicates warm up run).

6.1.2 Gin

The resulting power consumption over time of the test runs is shown in Figure 6.1. The duration of the first run was marginally larger than that of the subsequent ones. This may be due to some caching features in the Gin framework itself or CPU register optimizations.

The decrease in the duration is not linear, however, as for instance the last two runs were closer to the median duration-wise as opposed to the fastest. While some of the tests have small anomalies at approximately 4 to 6 seconds, the runs seem consistent and there are no significant anomalies in any of the plots of the runs. The fastest run for Gin was run number 7.

The energy consumption was on average 116.67 joules per run. The least energy-efficient run was number 1 with a cumulative energy consumption of 131.62 joules, whereas the most energy-efficient run, number 5, had a cumulative energy consump-

tion of 107.65. The total energy consumption for all of the runs was 2,333.44 joules.

6.1.3 Spring Boot

The resulting power consumption over time of the test runs for the Spring Boot REST API can be seen in Figure 6.3. The first observation that can be immediately drawn from the plotted data is that the first test run is a conspicuous outlier. This is likely due to the JIT-compiled nature of Java. Said run differs enough from the other runs that it can be ruled out as a statistical outlier, as the cause of it is quite obvious and cold start performance is not taken into account in these tests. Hence it is categorized as a warm-up period and will be ignored from the calculations later on.

Otherwise the data resembles that of Gin's quite well. The runs are of similar duration and the power consumption is roughly the same as well. The fastest run for Spring Boot was run number 12. The power consumption mostly hovers around 19 000 mW during the run time of the test, as was the case with Gin.

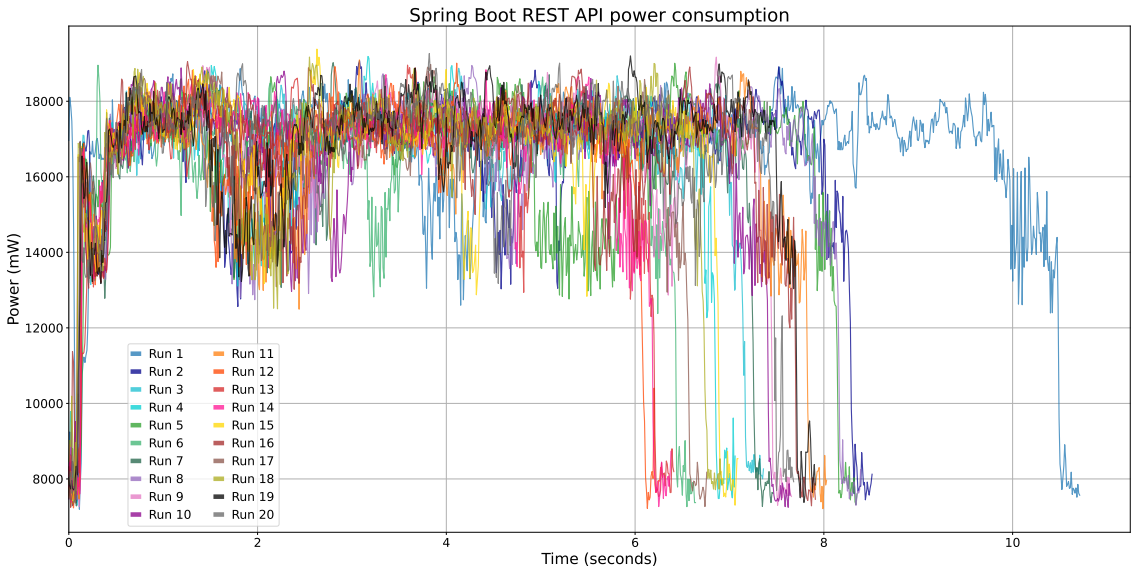

Figure 6.3: Power consumption of the REST API with Spring Boot framework.

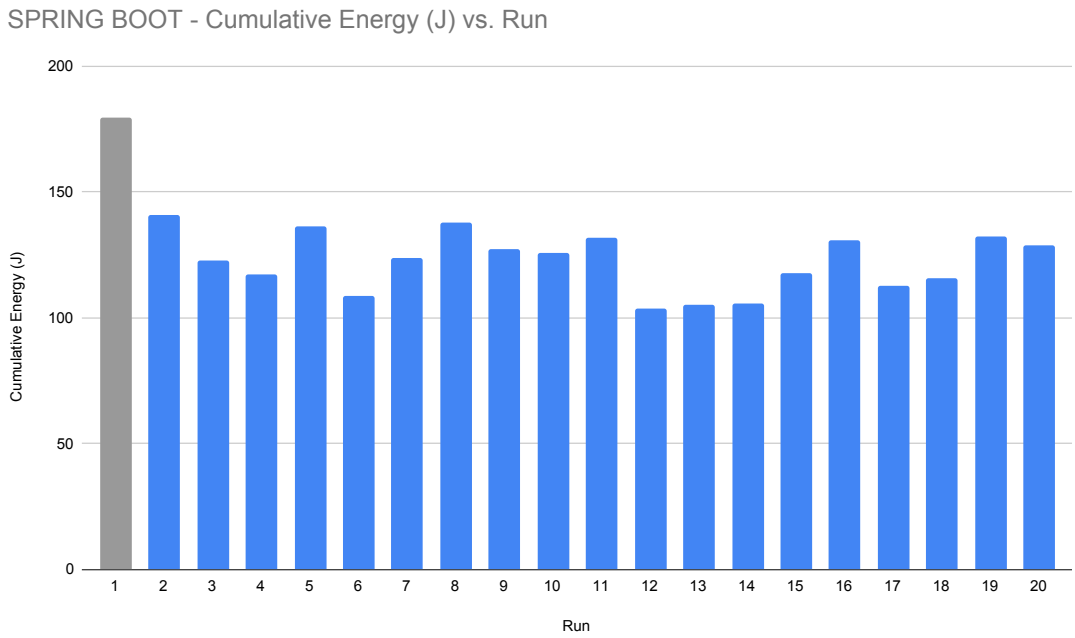

Figure 6.4: Spring Boot REST API's energy consumption for each run (gray indicates warm up run).

6.1.4 Laravel

The power consumption over time of the Laravel Boot REST API is shown in Figure 6.5. What is immediately evident from the plotted data is that the test runs are almost twice as long in comparison to both Gin and Spring Boot. This makes Laravel clearly the least efficient of the three, which is also highlighted in Table 6.1, since even the most efficient run of Laravel required more than double the energy than the least-efficient runs of both Spring Boot and Gin.

Distinctly, the power consumption stays more stable in Laravel when compared to the others. While Spring Boot and Gin seem to consistently have a drop in power consumption at roughly the two second mark in each run, no such observation can made in case of Laravel and the consumption stays both stable and high until the run is over. This affects the energy efficiency negatively, as the power consumption does not lower for the non-intensive parts in the same way it does for Spring Boot

and Gin.

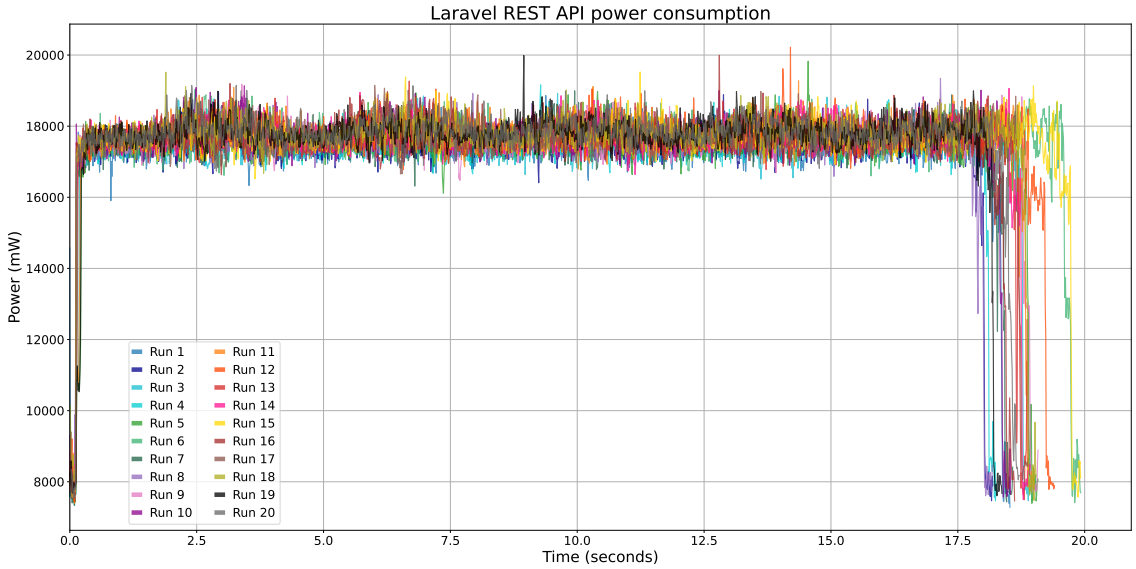

Figure 6.5: Power consumption of the REST API with Laravel framework.

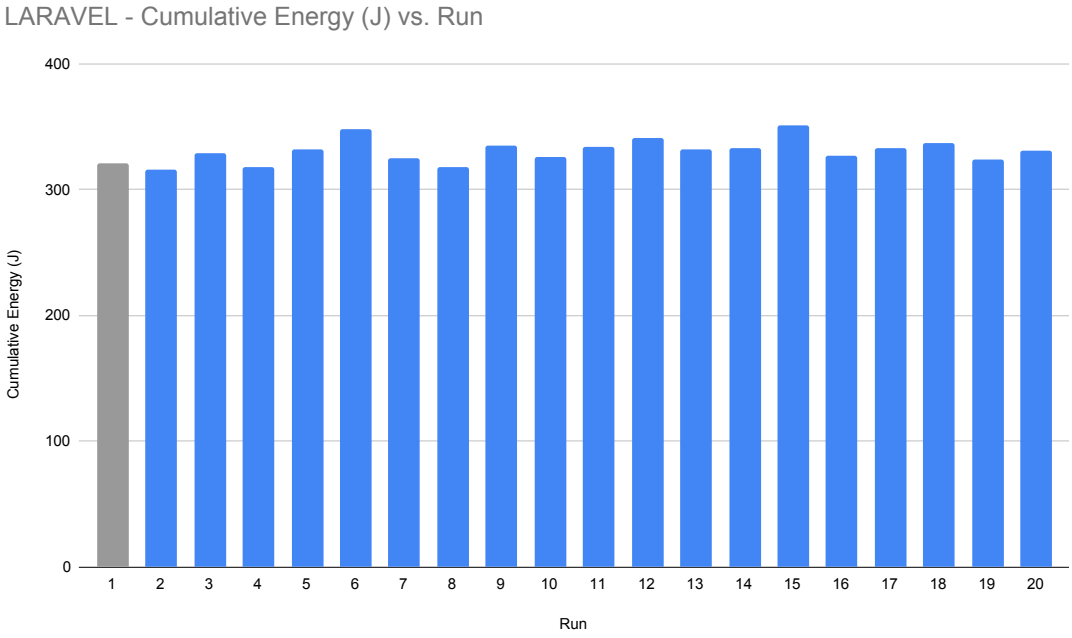

Figure 6.6: Laravel REST API's energy consumption for each run (gray indicates warm up run).

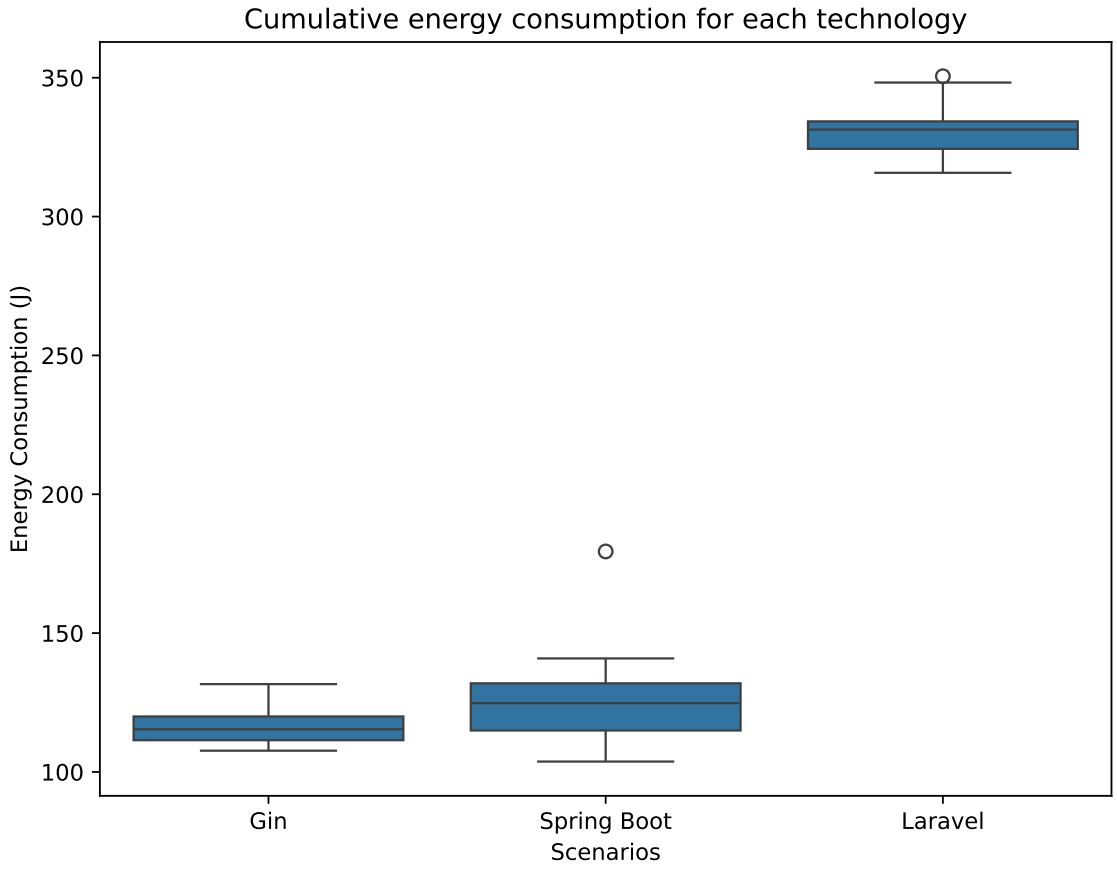

Figure 6.7: Energy consumption of each of the three REST API implementations.

6.1.5 Analysis

As mentioned earlier, the first run is a clear outlier and is considered a warm up run. As this thesis explores the energy consumption in a real-world setting, cold boot behaviour is no in the scope of these tests and is thus ignored from all the calculations hereinafter. This applies for all the technology-related tests, including Laravel and Gin, not just Spring Boot. For instance, Table 6.1 only considers runs 2-20 for each of the technologies. Figure 6.7 includes and highlights the outliers well; Gin has the most consistent performance overall with no explicit outliers, while the single outlier for Spring Boot is the clearly the most extreme. The only outlier of Laravel is very close to the maximum data value.

The results are quite clear and give a great answer to **RQ4** - there is a significant

difference between certain technologies when it comes to energy consumption. In fact, based on the results, the energy required can be reduced to almost a third of the original if one were to switch from Laravel to Gin. These results prove a point, but a more in-depth examination of real-world applications and use cases would be required to confirm the results. Nevertheless, they indicate a stark difference between the energy requirements of different technologies. On average the most energy-efficient technology, Gin, was 185.6% more energy-efficient than the least efficient technology Laravel. In addition, it was also 161.4% faster on average.

Predictably, the implementations written in Java and Go were quite close to each other in regards to energy requirements during the runs, as hypothesized in Chapter 2. What differs, though, in regards to the expectations from said chapter, is that with the empirical tests conducted in this thesis Gin outperformed Spring Boot. This, however, may be due to differences in the REST API frameworks themselves as opposed to the underlying programming language. Since the difference between the two are quite small, an argument could be made for another slightly different test scenario to provide reciprocal results for the two technologies.

Spring Boot had the most variation between its runs even excluding the warm up run. It had a 35.8% difference between its highest and lowest runs, whereas for Gin the respective percentage was 19.1% and for Laravel it was only 11.0%. With the absolute joule values, Gin had the least variation with 20.52 joules whereas the values for Spring Boot and Laravel were 37.14 and 34.77, respectively. The lower variation provides grounds to argue that even if Gin was not significantly more energy-efficient in comparison to Spring Boot, having less variation over time and a more consistent energy consumption makes it a better candidate for building green REST APIs, as its behaviour is more predictable.

While the results align with the hypotheses made in Chapter 2, more granular and targeted testing would be required to isolate and identify the key factors within

the different technologies that cause the divergence in the performance. A more granular test palette would likely explain the differences between the Java and Go implementations, as they were quite close to each other.

Table 6.1: Cumulative energy consumption results for each technology (runs 2-20).

Technology	Avg energy (J)	Highest run (J)	Lowest run (J)	Total (J)	Avg duration (s)
Gin	115.89	128.18	107.65	2201.82	7.216
Spring Boot	122.39	140.88	103.73	2325.42	7.395
Laravel	330.99	350.53	315.76	6288.74	18.860

6.2 Technology-agnostic tests

In contrast to Section 6.1, the cases discussed in this section do not consider the underlying technology, and a warm up run was performed before the tests were conducted. This means that all runs from 1 to 20 are considered hereinafter.

6.2.1 Caching

The power consumption over time for all three test cases, Redis only, MongoDB only and 50% hit rate can be seen in Figure 6.8. An observation that can be pointed out is that the figures look moderately uniform across all runs and that there is a clear difference in execution time in the database-only and cache-only tests. The 50% hit rate one, on the other hand, exhibits more variation between its runs while having a slight gravitation towards 100% hit rate in later runs.

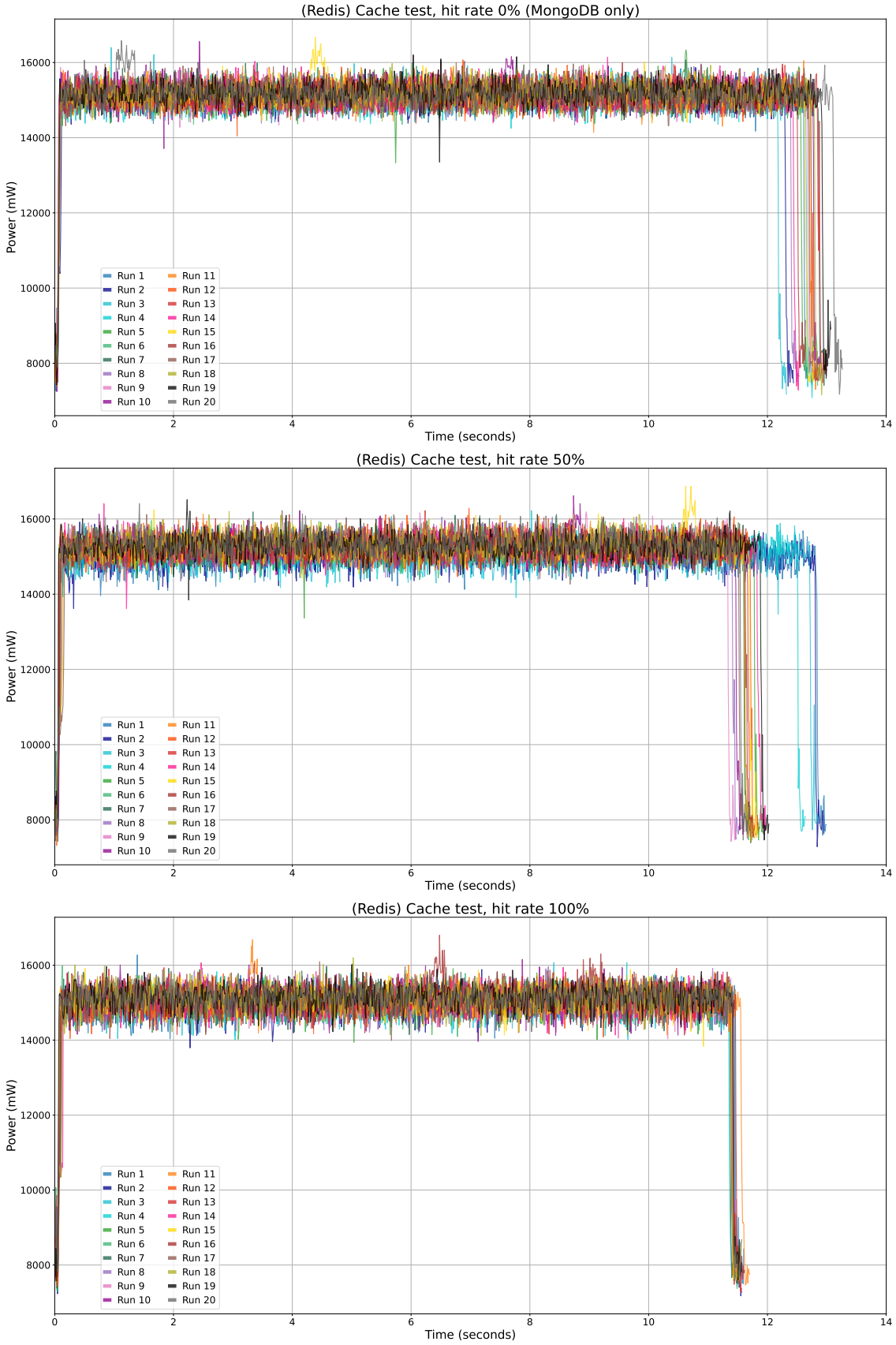

Figure 6.8: Power consumption across all three cache hit rates (0%, 50%, 100%).

From Figure 6.9 the observation that can be made is that the more the hit rate the lower the energy consumption. From Table 6.2 the conclusion can be drawn that the initial 50% seems to result in more energy savings than the latter half when observing the total amount of energy consumed over all 20 runs. That is, having a 50% hit rate decreased the energy consumption a total of 5.93% in comparison to 0% hit rate, while 100% hit rate lowered the energy consumption 4.55% in relation to the 50% hit rate. Overall, regarding total energy consumed, a total of 10.21% energy savings could be achieved between 0 to 100% across all 20 runs, which expressed in joules was 392.59.

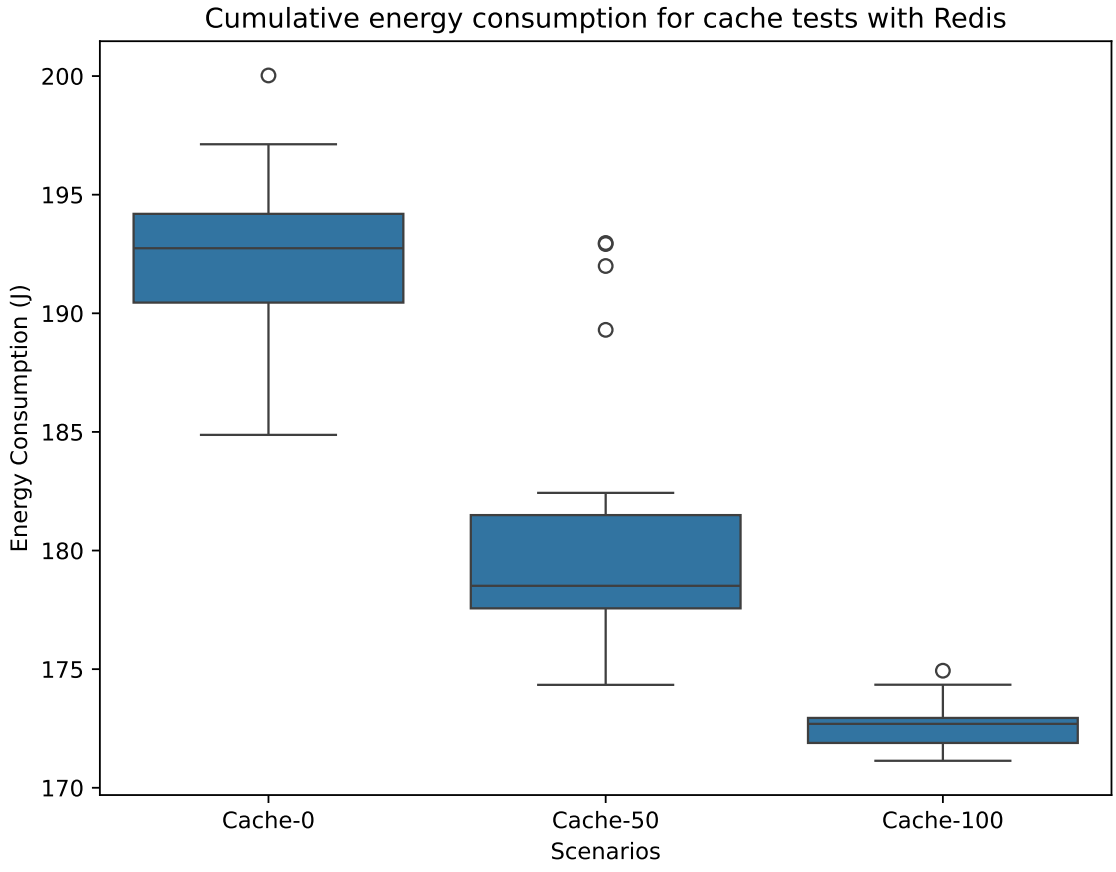

Figure 6.9: Energy consumption across all three hit rates (0%, 50%, 100%) for the Redis cache tests.

Achieving such a hit rate is of course purely theoretical or at least not sustainable

over long periods of time, but nevertheless the results indicate that the energy consumption can indeed be lowered by using a caching system in a REST API. What is encouraging in regards to real-world systems is that the greater proportion of energy savings occurred comparing 0% to 50% hit rate, in contrast to 50% to 100%. This is meaningful, since as discussed, it is highly unlikely for any practical deployment of a REST API to achieve such high hit rates consistently.

As was hypothesized in Chapter 2, both the execution time and the energy consumption were lower the higher the hit rate (and consequently cache utilization). The empirical tests thus correlate with the theoretical background quite well. The relative time and energy savings between the best and the worst were not as significant as with, for example, technologies - only the aforementioned 10.12%. Nevertheless, there is a clear indication that having an in-memory cache between the API and the database can provide energy savings. The savings are likely caused by the less expensive operation of reading from an in-memory cache as opposed to from the disk.

Table 6.2: Energy consumption results for each hit rate in cache tests.

Hit rate	Avg energy (J)	Highest run (J)	Lowest run (J)	Total (J)	Avg duration (s)
0%	192.29	200.03	184.88	3845.85	12.802
50%	180.89	192.96	174.34	3617.87	11.983
100%	172.66	174.93	171.14	3453.26	11.568

6.2.2 Batch requests

Batching seems to have quite a significant effect in the energy consumption, as is evident from the power graphs in Figure 6.10. On average the duration for each run is almost halved and furthermore, there is a noticeable reduction in power consumption on average with each run, as it hovers under 14 watts. With no batching employed, however, the average power consumption is clearly closer to 15 watts.

This is confirmed by Figure 6.11, where a significant reduction in the energy consumption can be seen when employing batches. In fact, on average, there is a 50.84% reduction in energy consumption when comparing the batch-utilizing version with the regular one. The reduced average power level means that the energy consumption reductions are not purely the result of having a shorter run duration, although that is likely the primary reason.

Table 6.3: Energy consumption results for batch tests.

Batch type	**Avg energy (J)**	**Highest run (J)**	**Lowest run (J)**	**Total (J)**	**Avg duration (s)**
No batching	391.88	399.59	384.84	7837.52	26.532
Batches of 4	192.64	195.59	189.32	3852.86	13.988

Table 6.3 highlights the differences between the batch sizes (0 vs. 4). Executing the requests in batches clearly reduces the overhead in both energy consumption and time required. Both the duration and the energy consumption are roughly halved, with the aforementioned 50.84% reduction in energy use and 47.28% reduction in time required on average. The variation is similar, though, with no batching the difference between the highest and the lowest run being 3.83% whereas for batches of four the corresponding difference was 3.31%.

As was hypothesized in Chapter 2, the much lower energy consumption and execution time is most likely due to the overhead removal between the API having to send the respond, the client receiving the response and then using the value of the response as part of the next request. Removing this back and forth network traffic and superfluous processing had a significant effect on the energy efficiency and performance of the REST API.

A noteworthy point is that the while the results indicate that request batching can provide significant benefits in terms of energy consumption, it is not applicable in all scenarios and is highly dependent on the situation. If, for instance, in our test

scenario the update operation required user input as opposed to merely updating the field with a pre-determined string, that specific scenario would not be able to utilize request batching in the same manner. So while the test scenario's use case is not exactly the most realistic, it shows that HTTP request batching should be implemented and utilized wherever possible, as it provides great energy savings as well as performance enhancements.

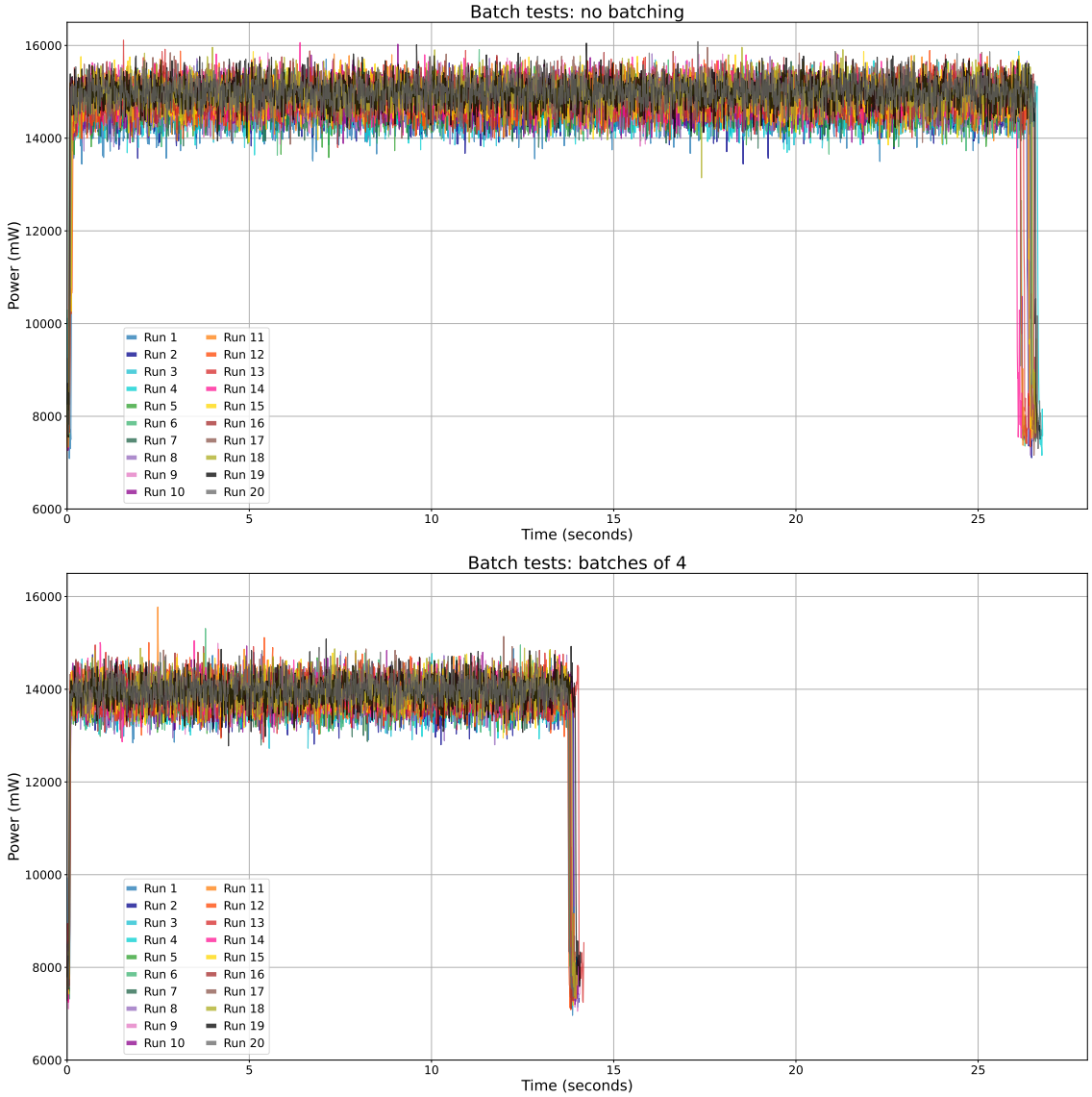

Figure 6.10: The power consumption graphs of no batching employed and with batches of 4.

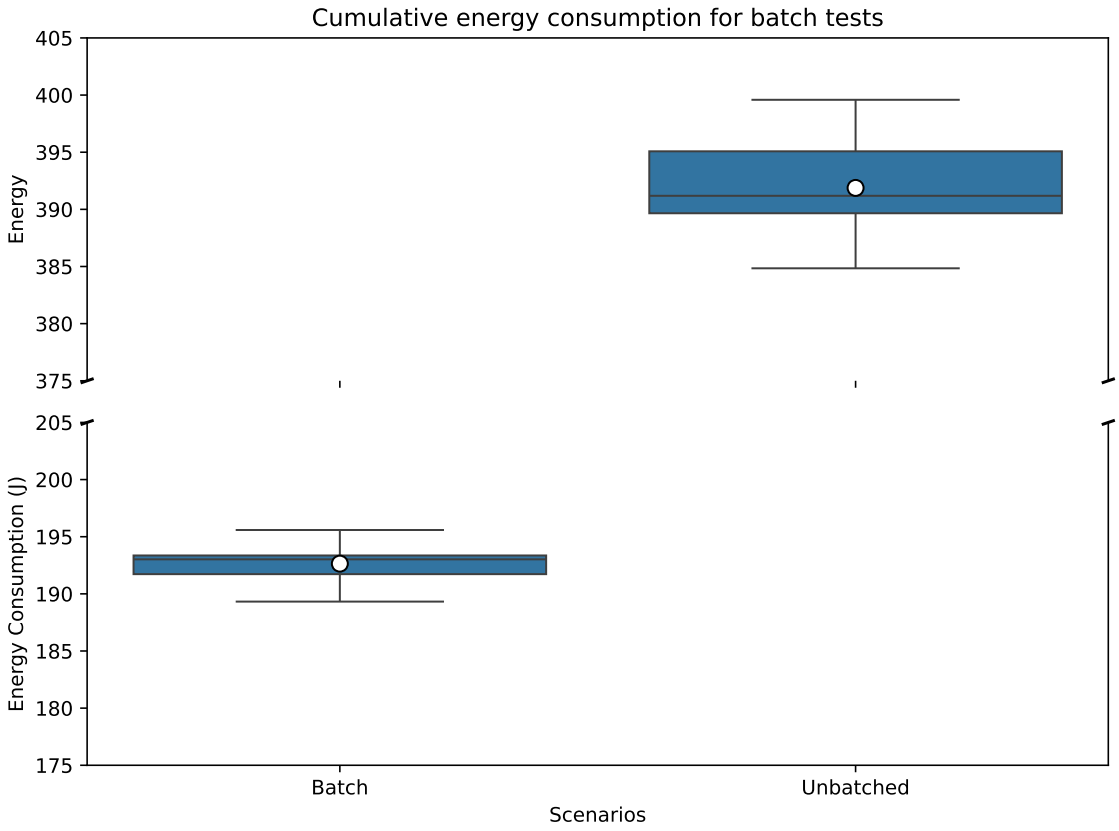

Figure 6.11: The cumulative energy consumption for the batch tests with no batches and batches of 4.

6.2.3 Algorithms

Figure 6.12 shows the difference in power consumption between the selected sorting algorithms. The immediate observation that can be made is that the quick sort algorithm is not only the most power-efficient but also takes the least time by quite a margin. Contrary to the best-case time complexities discussed in 4.4.3, the actual run times of insertion sort and selection sort do not seem to correlate to those exactly in the setting of these tests, but quick sort is the clearly the fastest.

Insertion sort exhibits the most variation in both power consumption and disparity of durations between runs. While the duration scale is different, both selection sort and quick sort have a more consistent performance across different runs. The

same effect can be seen in Figure 6.13, where the energy consumption disparity between runs is quite conspicuous for insertion sort.

In relation to the energy consumption, both Table 6.4 and Figure 6.13 not only confirm quick sort as the most optimal choice but also highlight the answer to the question posed for the algorithm-related tests: there is indeed a significant difference in energy efficiency for different algorithms aimed at performing the same task. What should be noted, of course, is that there might be different use cases for different algorithms. When it comes to energy consumption in, quick sort seems like the obvious choice based on the test results.

Table 6.4: Energy consumption results for each sorting algorithm.

Algorithm	**Avg energy (J)**	**Highest run (J)**	**Lowest run (J)**	**Total (J)**	**Avg duration (s)**
Quick sort	121.74	122.65	120.86	2434.88	8.791
Selection sort	233.77	237.20	231.64	4675.38	15.843
Insertion sort	258.02	284.01	246.94	5160.42	16.898

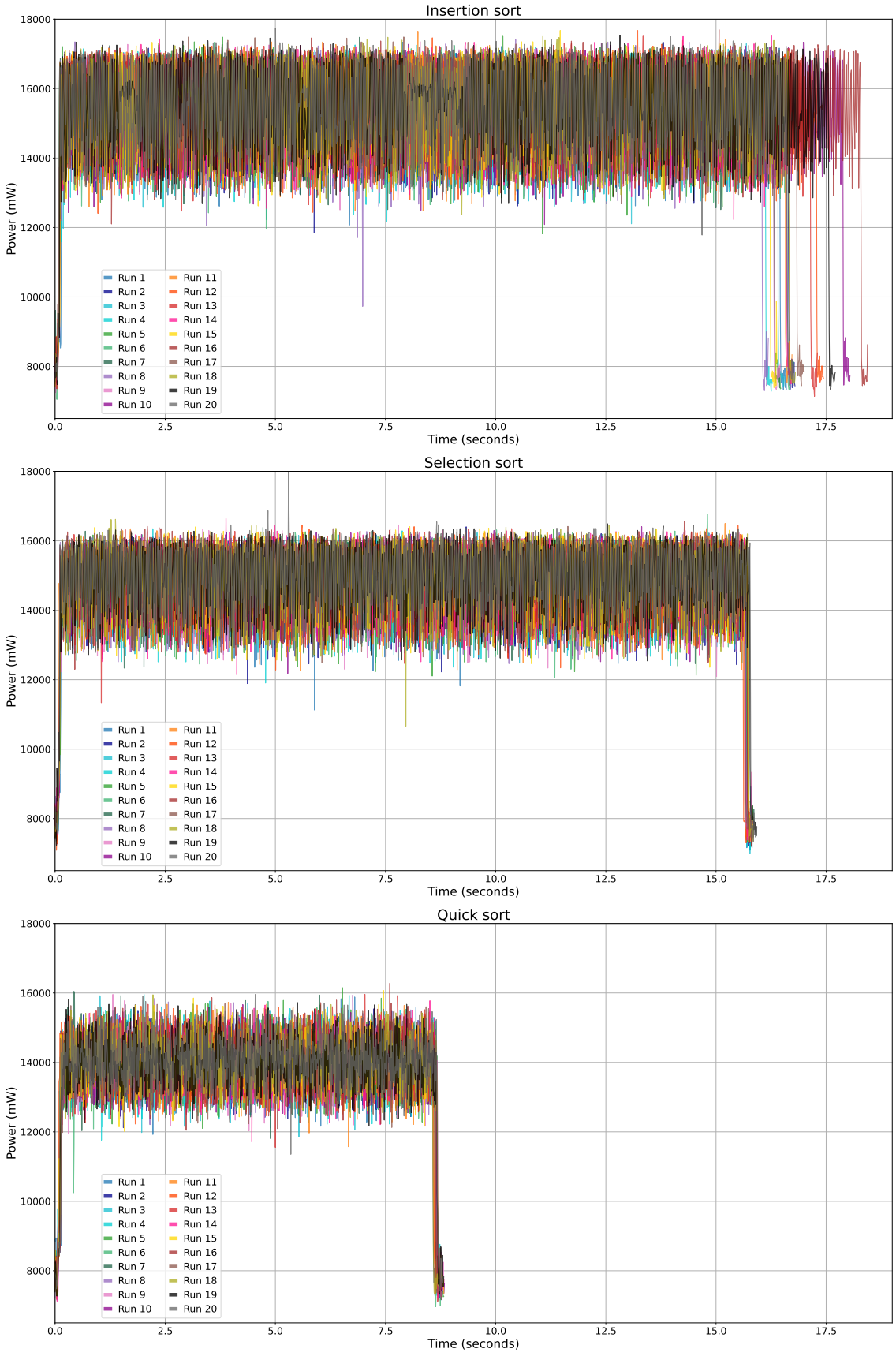

Figure 6.12: The power consumption of the sorting algorithms.

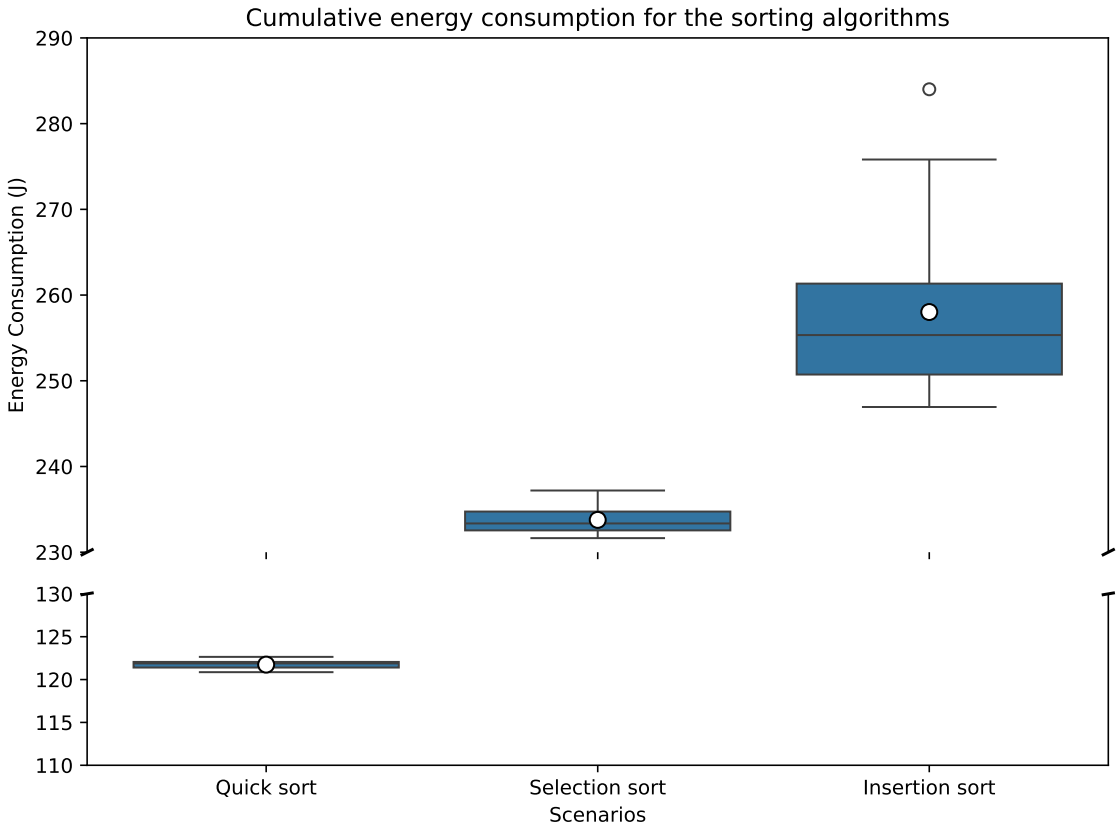

Figure 6.13: The cumulative energy consumption of the sorting algorithms.

6.2.4 Analysis

All of the tests demonstrated that the "improved" versions provided energy savings on some level. The disparity in the levels of savings achieved were not entirely consistent with the expectations, as I anticipated the savings to be of similar proportions between all the technology-agnostic tests.

In section 2.3.1, the hypothesis of reducing energy consumption by minimizing the superfluous consumption of receiving a request and responding to it seems to be correct in this case. The energy consumption was significantly lowered even though the same operations were performed, due to both a decreased run time and lower power consumption in the test runs.

What is surprising is that utilizing a cache resulted with relatively low energy

savings. This, however, may be due to the testing setup. In real-world scenarios such as more complex caching solutions utilization, for instance generating preview images, or simply having significantly larger data sets could greatly increase the energy savings provided by a caching system. Nevertheless a cache solution undoubtedly has earned its place in the effective green coding practices in REST APIs. The vast difference in energy consumption between the sorting algorithms was unexpected to an extent, as the expectation was for them to be closer to the other technology-agnostic tests in relative divergence between test cases, as opposed to the levels of energy savings achieved by technology changes. The results are not directly comparable, though.

What is noteworthy about the batch tests is that it is not applicable to all operations or use scenarios. For instance, while it is technically possible to perform the CRUD operation containing 4 individual requests that was utilized in the tests, if the update operation, for example, required user input that can only be added once they know the result of the read operation, the entirety of the CRUD operation flow cannot be batched. This does not mean, however, that batching should not be utilized where it is possible to do so, since it proved effective in providing energy savings.

As was briefly discussed in Chapter 4, the time complexities of the sorting algorithm differ from each other. Quick sort was best in that regard, and is well-suited for modern computer architectures [81]. It is thus no surprise that it performed the best.

6.3 Summary and Discussion

To assess the impact of each green coding practice, the worst- and best-performing test cases were compared in each category. The average reduction in energy consumption is shown as a percentage relative to the worst-performing test in each

category, providing a normalized view of the achievable energy savings for each practice. For instance, on average Gin required only 35,3% of the energy required by Laravel, so 64,7% energy savings were achieved. The tests are not directly comparable due to differing workloads and the nature of the tests, but they do provide an insight into the relative energy efficiency improvement provided by each category in isolation. As is shown in Figure 6.14, the most energy savings were provided by changing the underlying implementation technology, while the least energy savings were provided by cache utilization. Changing the algorithm provided quite substantial energy savings as well, whereas batching also cut the energy consumption in half.

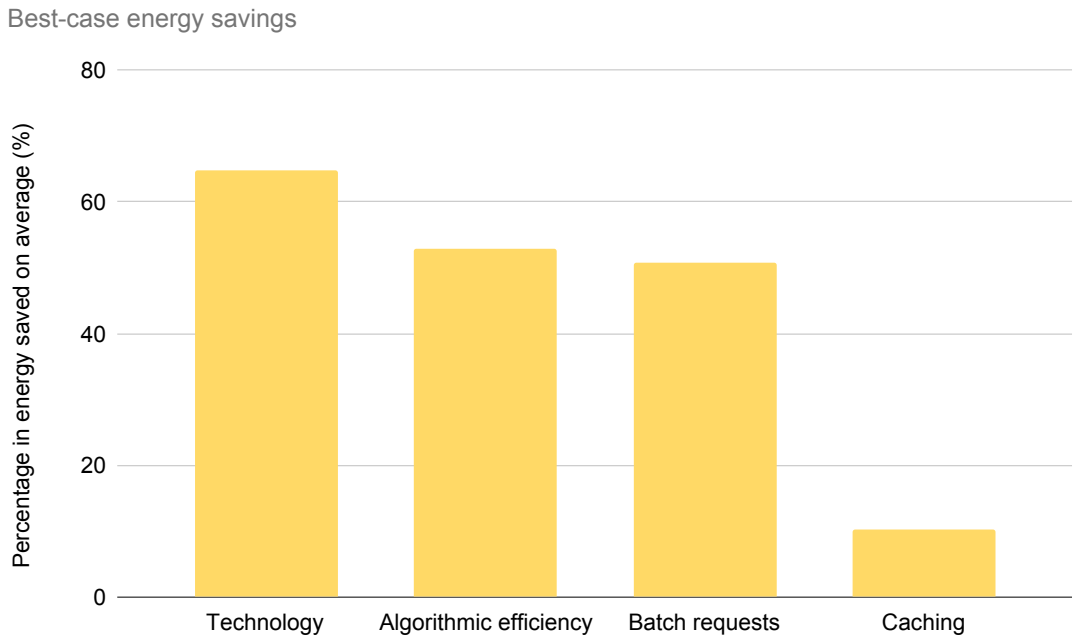

Figure 6.14: Relative average energy savings for each category as a percentage of energy saved. The values represent the reduction in energy consumption relative to the worst case in each category.

6.3.1 Answers to Research Questions

RQ1 "What design, implementation and optimization choices have the most significant effect on the energy consumption of software?" was already answered thor-

oughly in Chapter 2, but to sum it up here, the identified technologies, meaning programming languages and frameworks, paradigms, performance and data optimizations, batch operations, correct algorithmic choices and caching utilization.

RQ2 "How can software energy consumption be measured?" was answered in Chapter 3. The three main ways of measuring the energy consumption of software were found to be software-based and hardware-based methods as well as hybrids of the two.

To answer **RQ3** "How effective are the existing green coding practices for REST APIs", the results are quite apparent. The energy savings from the correct choice of technology likely affect all operations on the API and thus lower its energy consumption altogether, but as seen, individual modules or parts of the API can be optimized further. For instance, as both the choice of the correct algorithm and the utilization of request batching can roughly half the energy consumption individually, combining the effect of all the green coding practices for a cumulative effect are likely to provide quite significant gains in energy efficiency. Some of the green coding practices are not as good as the others, however. As discussed in 6.2.1, the 5.93% reduction in energy consumption from the more reasonable 50% hit rate means in some cases, the cost of implementing the caching system might not be worth it. This would mean that in specific cases, a single practice might prove to provide negligible energy savings in relation to the cost of implementing such feature. For all the other categories besides caching, based on the empirical tests it is safe to assume implementing them should be a priority, should maximal energy efficiency be the end goal.

The answer to **RQ4** "How significant is the choice of technology in relation to other green coding practices?" is clear. Of all the tested categories, most energy savings were achieved with technology improvements. In addition, the technology has an effect on all activity and tasks in the REST API, so any energy efficiency

gains resulting from that likely provide overall benefits the best. Based on the results, some technologies are far behind the others, namely PHP in relation to Java and Go. In summary, based on the results of the empirical section, the correct choice of technology is the most important green coding technique that can be utilized when developing a REST API.

7 Conclusion

In this thesis the main problem was to elicit the current landscape of green coding and evaluate whether the existing energy efficient practices apply to REST APIs. In the format of a literature review in Chapter 2, this thesis first explored the features and characteristics of a REST API that affect its energy consumption and what makes software energy-efficient in general. The literature review identified technologies, paradigms, performance and data optimizations, batch operations, algorithmic choices and caching as the most significant practices. Chapter 3 then explored how the energy consumption of software can be measured. The conducted analysis resulted in three main methods; software-based, hardware-based and hybrid methods. A test software palette was then assembled in Chapter 4, where the tests related to the practices of technology, batching, algorithmic choices and caching were built and introduced. In Chapter 5 the power and energy consumption measurement setup was introduced. Then, in Chapter 6 the topics were explored empirically - i.e. the empirical tests were performed. The results were then analysed and compared to the findings and hypotheses made in the literature review. All of the empirically tested topics, technology, batching, algorithmic choices and caching had an effect on the REST API's energy consumption, as hypothesized. Thus, the empirical section showed that by adhering to green coding practices, the gains in energy efficiency can be significant. The findings in this thesis contribute to the mapping out the landscape of energy efficient practices tailored for REST APIs.

7.1 Threats to validity

7.1.1 Internal validity

In some tests, to better grasp whether the energy savings resulted from what the hypothesis was, a more in-depth test palette might be required. For instance, with the batching tests, the implementation of the batched version might not totally correlate with a real-world example. In addition, the test case was by no means exhaustive, and could even be described as rudimentary.

This thesis utilized a performance testing tool in the empirical tests. This means that the results could have been skewed by the fact that under normal operating conditions, a REST API is usually not under such loads. Hence, the results could change with a different demand.

7.1.2 External validity

In regards to external validity, the thesis' scope might be too large. A more in-depth analysis for each test case is required to get a better understanding of what causes the energy savings. Additionally, while the results do provide a way to guide engineering decisions, a more in-depth study would be required to find the most optimal solution.

Additionally, the setting of the tests could have been more realistic. By taking a real-world application and applying green coding practices to that, one could get a better picture of how an energy-optimized REST API would behave in real use cases. Larger data sets and greater variation in the test palette could also change the results.

7.2 Further research and limitations

This thesis does not provide a big picture view on what would be the difference in energy consumption if all the sustainable practices were combined and compared to a non-optimized REST API. This is a topic that would be great for a next step after this thesis to find out, how much the energy consumption can be lowered by combining different practices. Additionally, the empirical tests did not cover all the green coding practices, such as network and data set size related topics.

Topics for further research could be a more realistic testing setup with heavier, real-world applications and a longer time span with lower (not performance test scaled) demand. While this thesis shed light on whether the existing sustainable practices do indeed apply to REST APIs and more importantly have the ability to provide a non-negligible decrease in the API's energy consumption, the next step could include more thorough yet similar tests to those conducted in this thesis to not only affirm the results but also look into the theoretical maximum increase a system could get in its energy efficiency.

Additionally, all of the test cases could be examined in detail. For example, comparing different frameworks built with the same programming language would provide more insight into how developers can choose the correct REST API development technologies for energy efficiency. Moreover, the more abstract practices such a avoiding extraneous tasks could be explored with case studies of real REST API implementations to emphasize the more non-tangible energy efficiency improvement techniques.

References

[1] I. Manotas, C. Bird, R. Zhang, *et al.*, "An empirical study of practitioners' perspectives on green software engineering", in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 237–248.

[2] E. Gelenbe, "Electricity consumption by ict: Facts, trends, and measurements", *Ubiquity*, vol. 2023, no. August, pp. 1–15, 2023.

[3] H. Sartaj, S. Ali, and J. M. Gjøby, "Rest api testing in devops: A study on an evolving healthcare iot application", *arXiv preprint arXiv:2410.12547*, 2024.

[4] P. Rani, J. Zellweger, V. Kousadianos, L. Cruz, T. Kehrer, and A. Bacchelli, "Energy patterns for web: An exploratory study", in *Software Engineering in Society (ICSE-SEIS'24)*, Lisbon, Portugal, 2024. DOI: 10.1145/3639475.3640110.

[5] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "On reducing the energy consumption of software: From hurdles to requirements", in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.

[6] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications", *Empirical Software Engineering*, vol. 24, pp. 2209–2235, 2019.

[7] C. Caiazza, V. Luconi, and A. Vecchio, "Energy consumption of smartphones and iot devices when using different versions of the http protocol", *Pervasive*

and Mobile Computing, vol. 97, p. 101 871, 2024, ISSN: 1574-1192. DOI: https://doi.org/10.1016/j.pmcj.2023.101871. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574119223001293.

[8] Y.-W. Kwon and E. Tilevich, "Reducing the energy consumption of mobile applications behind the scenes", in *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 170–179.

[9] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis, "What are your programming language's energy-delay implications?", in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 303–313, ISBN: 9781450357166. DOI: 10.1145/3196398.3196414. [Online]. Available: https://doi.org/10.1145/3196398.3196414.

[10] C. Marimuthu and K. Chandrasekaran, "Software engineering aspects of green and sustainable software: A systematic mapping study", in *Proceedings of the 10th Innovations in Software Engineering Conference*, ser. ISEC '17, Jaipur, India: Association for Computing Machinery, 2017, pp. 34–44, ISBN: 9781450348560. DOI: 10.1145/3021460.3021464. [Online]. Available: https://doi.org/10.1145/3021460.3021464.

[11] C. Rodriguez, M. Baez, F. Daniel, *et al.*, "Rest apis: A large-scale analysis of compliance with principles and best practices", Jun. 2016, pp. 21–39, ISBN: 978-3-319-38790-1. DOI: 10.1007/978-3-319-38791-8_2.

[12] S. Target, *Roy fielding's misappropriated rest dissertation*, Accessed 2024-10-28. [Online]. Available: https://twobithistory.org/2020/06/28/rest.html.

[13] RapidAPI, *2022 state of apis*, Accessed 2024-10-01. [Online]. Available: https://stateofapis.com/.

[14] P. Inc., *2023 state of the api report*, Accessed 2024-10-01. [Online]. Available: https://www.postman.com/state-of-api/api-technologies/.

[15] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing restful apis: A survey", *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–41, 2023.

[16] L. Ardito and M. Morisio, "Green it – available data and guidelines for reducing energy consumption in it systems", *Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 24–32, 2014, ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2013.09.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537913000504.

[17] Q. Zhang, Z. Meng, X. Hong, *et al.*, "A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization", *Journal of Systems Architecture*, vol. 119, p. 102 253, 2021.

[18] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey", *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 732–794, 2015.

[19] V. Salmikuukka, "Guidelines for sustainable software", Master's Thesis, Tampere University, 2024.

[20] *Directive 2022/2464 - en - csrd directive - eur/lex*, Accessed 2025-05-16. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32022L2464.

[21] R. Pereira, M. Couto, F. Ribeiro, *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?", in *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 2017, pp. 256–267.

[22] C. Guthrie, "How green is green it? a multidisciplinary bibliometric study and research agenda", *Procedia Computer Science*, vol. 239, pp. 701–709, 2024.

[23] S. Nurmivaara *et al.*, "Green in software engineering: A systematic literature review", Master's Thesis, University of Helsinki, 2023.

[24] S. Mahmudova, "Green coding in programming and practices", *Journal of Engineering and Technology (JET)*, vol. 15, no. 2, 2024.

[25] R. Manimegalai, S. Sandhanam, A. Nandhini, and P. Pandia, "Energy efficient coding practices for sustainable software development", in *Proceedings of the First International Conference on Science, Engineering and Technology Practices for Sustainable Development, ICSETPSD 2023, 17th-18th November 2023, Coimbatore, Tamilnadu, India*, 2024.

[26] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: Techniques and tools", *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–33, 2019.

[27] M. C. Belgaid, "Green coding: An empirical approach to harness the energy consumption of software services", Ph.D. Thesis, Université de Lille, 2022.

[28] G. Pinto and F. Castor, "Energy efficiency: A new concern for application software developers", *Commun. ACM*, vol. 60, no. 12, pp. 68–75, Nov. 2017, ISSN: 0001-0782. [Online]. Available: `https://doi.org/10.1145/3154384`.

[29] A. Gordillo, C. Calero, M. Á. Moraga, *et al.*, "Programming languages ranking based on energy measurements", *Software Quality Journal*, pp. 1–42, 2024.

[30] F. de Mander and W. Gren, *Comparison of energy usage and response time for web frameworks*, 2023.

[31] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "A preliminary study of the impact of software engineering on greenit", in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 21–27. DOI: 10.1109/GREENS.2012.6224251.

[32] C. Cares Gallardo, J. Franch Gutiérrez, and E. Mayol Sarroca, "Perspectives about paradigms in software engineering", in *Proceedings of the CAISE* 06 Workshop on Philosophical Foundations on Information Systems Engineering, PhiSE'06: Luxemburg, June 5-9, 2006*, CEUR-WS. org, 2006, pp. 737–744.

[33] A. E. Kwame, E. M. Martey, and A. G. Chris, "Qualitative assessment of compiled, interpreted and hybrid programming languages", *Communications on Applied Electronics*, vol. 7, no. 7, pp. 8–13, 2017.

[34] C. Pichler, P. Li, R. Schatz, and H. Mössenböck, "Hybrid execution: Combining ahead-of-time and just-in-time compilation", in *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2023, pp. 39–49.

[35] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "Aot vs. jit: Impact of profile data on code quality", in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2017, pp. 1–10.

[36] P. Körner, D. Schneider, and M. Leuschel, "On the performance of bytecode interpreters in prolog", in *International Workshop on Functional and Constraint Logic Programming*, Springer, 2020, pp. 41–56.

[37] *Typescript fannkuch-redux implementation skews results in paper*, Accessed 2025-05-10. [Online]. Available: https://github.com/greensoftwarelab/Energy-Languages/issues/34.

[38] MDN web docs, *Just-in-time compilation (jit)*, Accessed 2025-05-10. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation.

[39] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating the impact of java virtual machines on energy consumption", in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21, Bari, Italy: Association for Computing Machinery, 2021, ISBN: 9781450386654. DOI: 10.1145/3475716.3475774. [Online]. Available: https://doi.org/10.1145/3475716.3475774.

[40] Actix, *Actix*, Accessed 2024-03-09. [Online]. Available: https://actix.rs/.

[41] T. Benchmarker, *Web frameworks benchmark*, Accessed 2025-05-20. [Online]. Available: https://web-frameworks-benchmark.netlify.app/compare?f=express,fastify,hono-deno.

[42] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in php, python, and node. js", in *2014 IEEE 17th international conference on computational science and engineering*, IEEE, 2014, pp. 661–668.

[43] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 5th ed. Harlow, United Kingdom: Pearson Education Limited, 2024, Global Edition.

[44] T. Hunter II and B. English, *Multithreaded JavaScript*. " O'Reilly Media, Inc.", 2021.

[45] Ron Pressler and Alan Bateman, *Jep 444: Virtual threads*. [Online]. Available: https://openjdk.org/jeps/444 (visited on 05/20/2025).

[46] *Goroutines*. [Online]. Available: https://go.dev/tour/concurrency/1 (visited on 05/20/2025).

[47] The Go Authors, *Runtime package*, Accessed 2024-09-19. [Online]. Available: https://pkg.go.dev/runtime.

[48] K. Rosendahl, "Green threads in rust", Ph.D. dissertation, Master's thesis, Stanford University, Computer Science Department, 2017.

[49] The Go Authors, *Http package - net/http*, Accessed 2025-05-10. [Online]. Available: https://pkg.go.dev/net/http.

[50] I. Vilhelmsson, "A performance comparison of an event-driven node. js web server and multi-threaded web servers", Master's Thesis, KTH, School of Electrical Engineering and Computer Science, 2021.

[51] Microsoft, *Execute batch operations using the web api*, Accessed 2024-03-26. [Online]. Available: https://learn.microsoft.com/en-us/power-apps/developer/data-platform/webapi/execute-batch-operations-using-web-api.

[52] K. Eder and J. P. Gallagher, "Energy-aware software engineering", in *ICT - Energy Concepts for Energy Efficiency and Sustainability*, G. Fagas, L. Gammaitoni, J. P. Gallagher, and D. J. Paul, Eds., Rijeka: IntechOpen, 2017, ch. 5. DOI: 10.5772/65985. [Online]. Available: https://doi.org/10.5772/65985.

[53] F. O. Chete and A. Omorokunwa, "A fast image compression web service using a rest api", *Journal of Electrical Engineering, Electronics, Control and Computer Science*, vol. 8, no. 2, pp. 1–6, 2021.

[54] R. Xu, Z. Li, C. Wang, and P. Ni, "Impact of data compression on energy consumption of wireless-networked handheld devices", in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, 2003, pp. 302–311. DOI: 10.1109/ICDCS.2003.1203479.

[55] L. H. Nunes, L. H. Nakamura, H. d. F. Vieira, *et al.*, "Performance and energy evaluation of restful web services in raspberry pi", in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2014, pp. 1–9.

[56] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study", in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 61–72, ISBN: 9781450339001. DOI: 10.1145/2884781.2884829. [Online]. Available: https://doi.org/10.1145/2884781.2884829.

[57] J. Mancebo, F. García, and C. Calero, "A process for analysing the energy efficiency of software", *Information and Software Technology*, vol. 134, p. 106 560, 2021, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106560. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000446.

[58] T. A. Ghaleb, "Software energy measurement at different levels of granularity", in *2019 International Conference on Computer and Information Sciences (ICCIS)*, IEEE, 2019, pp. 1–6.

[59] A. E. Trefethen and J. Thiyagalingam, "Energy-aware software: Challenges, opportunities and strategies", *Journal of Computational Science*, vol. 4, no. 6, pp. 444–449, 2013, Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011, ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2013.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750313000173.

[60] A. Rajan, A. Noureddine, and P. Stratis, "A study on the influence of software and hardware features on program energy", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and*

Measurement, ser. ESEM '16, Ciudad Real, Spain: Association for Computing Machinery, 2016, ISBN: 9781450344272. DOI: 10.1145/2961111.2962593. [Online]. Available: https://doi.org/10.1145/2961111.2962593.

[61] J. Mancebo, H. O. Arriaga, F. García, M. Á. Moraga, I. G.-R. de Guzmán, and C. Calero, "Eet: A device to support the measurement of software consumption", in *Proceedings of the 6th International Workshop on Green and Sustainable Software*, ser. GREENS '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 16–22, ISBN: 9781450357326. DOI: 10.1145/3194078.3194081. [Online]. Available: https://doi.org/10.1145/3194078.3194081.

[62] E. A. Jagroep, J. M. van der Werf, S. Brinkkemper, *et al.*, "Software energy profiling: Comparing releases of a software product", in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 523–532, ISBN: 9781450342056. DOI: 10.1145/2889160.2889216. [Online]. Available: https://doi.org/10.1145/2889160.2889216.

[63] T. Carçao, "Measuring and visualizing energy consumption within software code", in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2014, pp. 181–182.

[64] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement", in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 1327–1334.

[65] A. Katsenou, X. Wang, D. Schien, and D. Bull, "Comparative study of hardware and software power measurements in video compression", in *2024 Picture Coding Symposium (PCS)*, 2024, pp. 1–5. DOI: 10.1109/PCS60826.2024.10566286.

[66] H. Zhang and H. Hoffman, "A quantitative evaluation of the rapl power control system", *Feedback Computing*, vol. 6, 2015.

[67] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements", vol. 3, no. 2, Mar. 2018, ISSN: 2376-3639. DOI: 10.1145/3177754. [Online]. Available: https://doi.org/10.1145/3177754.

[68] Intel Corporation, *Running average power limit energy reporting*, Accessed 2025-01-26. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[69] W3Techs, *Usage statistics of php for websites*, Accessed 2024-09-06. [Online]. Available: https://w3techs.com/technologies/overview/programming_language.

[70] Stackoverflow, *Web frameworks and technologies | 2024 stack overflow developer survey*, Accessed 2025-01-26. [Online]. Available: https://survey.stackoverflow.co/2024/technology#2-web-frameworks-and-technologies.

[71] P. Jansen, *Tiobe index for september 2024*, Accessed 2024-09-27. [Online]. Available: https://www.tiobe.com/tiobe-index/.

[72] M. A. Domingues, "Performance testing of open-source http web frameworks in an api", *DSIE'17*, p. 8, 2017.

[73] mingrammer, *Top go web frameworks*, Accessed 2024-09-20. [Online]. Available: https://github.com/mingrammer/go-web-framework-stars.

[74] M. Martínez-Almeida, *Gin web framework*, Accessed 2024-09-15. [Online]. Available: https://github.com/gin-gonic/gin.

[75] Stackoverflow, *Technology | 2024 stack overflow developer survey*, Accessed 2025-05-31. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-programming-scripting-and-markup-languages.

[76] Oracle, *Native image*, Accessed 2025-04-04. [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/.

[77] Docker Inc., *What is docker?*, Accessed 2024-09-16. [Online]. Available: https://docs.docker.com/get-started/docker-overview/.

[78] E. A. Santos, C. McLean, C. Solinas, and A. Hindle, "How does docker affect energy consumption? evaluating workloads in and out of docker containers", *Journal of Systems and Software*, vol. 146, pp. 14–25, 2018.

[79] Stackoverflow, *Technology | 2024 stack overflow developer survey*, Accessed 2024-10-30. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-databases.

[80] Docker Inc., *Wrk - a http benchmarking tool*, Accessed 2024-09-16. [Online]. Available: https://github.com/wg/wrk.

[81] P. Prajapati, N. Bhatt, and N. Bhatt, "Performance comparison of different sorting algorithms", *vol. VI, no. Vi*, pp. 39–41, 2017.

[82] University of Turku, *Powergoblin*. [Online]. Available: https://gitlab.utu.fi/tech/soft/tools/power/powergoblin (visited on 01/18/2025).

[83] Jari-Matti Mäkelä and Katariina Moilanen, *Mittaamisella kohti vihreämpiä prosesseja ja ohjelmistoja*. [Online]. Available: https://greenict.fi/mittaamisella-kohti-vihreampia-prosesseja-ja-ohjelmistoja/ (visited on 05/21/2025).

[84] Tieke, *Visiiri – vihreän siirtymän ict-ekosysteemi*. [Online]. Available: https://tieke.fi/hankkeet/green-ict-visiiri/ (visited on 05/21/2025).

[85] University of Turku, *Powergoblin user manual - example configuration: Energy efficiency of sorting algorithms.* [Online]. Available: `https://tech.utugit.fi/soft/tools/power/doc/manual/usage/ex-algo/index.html#setup` (visited on 06/03/2025).

[86] University of Turku, *Powergoblin user manual - example configuration: Energy measurement of back-end software.* [Online]. Available: `https://tech.utugit.fi/soft/tools/power/doc/manual/usage/ex-backend/#high-level-description` (visited on 06/03/2025).

[87] University of Turku, *Powergoblin user manual - introduction, terminology.* [Online]. Available: `https://tech.utugit.fi/soft/tools/power/doc/manual/usage/intro/index.html` (visited on 06/01/2025).

[88] University of Turku, *Powergoblin technical manual.* [Online]. Available: `https://tech.utugit.fi/soft/tools/power/doc/technical/index.html` (visited on 06/01/2025).

[89] University of Turku, *Powergoblin user manual - documentation for the user interface.* [Online]. Available: `https://tech.utugit.fi/soft/tools/power/doc/manual/webui/ui/` (visited on 06/01/2025).