OST
Eastern Switzerland
University of Applied Sciences

EASTERN SWITZERLAND UNIVERSITY
OF APPLIED SCIENCES

Project Thesis

Towards Greener Software:

Measuring Performance and Energy Efficiency of Enterprise Applications

Author:
Jan RUCH

Supervisor:
Prof. Dr. Olaf ZIMMERMANN

A project submitted in fulfillment of the requirements
for the degree of Master of Science in Engineering
focusing on Computer Science

Spring Term, 2025

Declaration of Authorship

I, Jan Ruch, declare that this project thesis titled, *Towards Greener Software: Measuring Performance and Energy Efficiency of Enterprise Applications* and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Rapperswil, June 23, 2025

Jan Ruch

Tools Used

- GitHub Copilot [1] has been used as a plugin for my IDE. Copilot has been pre-installed for coding purposes, however, Copilot also supports textual suggestions. Sometimes I have wholly or partially taken over suggestions and adapted them to the context of this report.
- Scribbr [2] has been used to cite all references in an APA 7th style.
- DeepL [3] has been used to translate parts of the chapter Introduction. The initial text has been written in german in collaboration with Prof. Dr. Olaf Zimmermann.
- Elicit [4] has been used to support the research of academic publications.
- ChatGPT [5] has been used to improve the text quality of this report by suggesting improvements, rephrasing sentences, and correcting grammar. The suggestions have not been taken over verbatim, but rather adapted to the context of this report.

Abstract

Performance is an important aspect of software quality in most software applications. In recent years, the topic of energy efficiency has become increasingly important. Enterprise applications running in the cloud receive particular attention. Poor performance and low energy efficiency lead to high operating costs and a negative impact on the environment. Recognizing the significance of performance and energy efficiency as software quality attributes and how to measure them is crucial.

This research project aims to investigate how two types of software quality attributes, performance and resource and energy efficiency, are measured according to the state of the art and the practice today; differences are identified and analysed. Following an empirical approach, an existing application test setup and its measurement results (publicly available in the “Growing Green Software” blog) are first reproduced and then compared with the behaviour of a second sample application. The two respective sample applications are open source-projects leveraging Java and Spring Boot; one of them comes as a set of microservices. Tools such as JMeter and JoularJX, configurations, and metrics across different test environments and enterprise applications were experimented with. Contemporary software engineering practices such as Domain-Driven Design and UML were used to analyse and document the software architectures of the selected applications.

The measurements confirmed that the performance and energy consumption of the application are significantly influenced by external factors such as hardware, operating system, and implementation details. They show that the relative distribution of energy consumption is comparable across different test environments and enterprise applications. Furthermore, the results suggest an inverse correlation between performance and energy efficiency when different hardware is compared. They also indicate a strong correlation when the same hardware is used but implementation details vary.

Future research could explore the energy efficiency of cloud-native applications and cloud infrastructure.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Dr. Olaf Zimmermann for his never-ending support and guidance throughout this project. His expertise and insights have been invaluable for the success of this work. While conducting the measurements for this project thesis, writing the report, and preparing a speech about the topics covered in this thesis, he has always supported me with great effort beyond his responsibilities. I am grateful for the opportunity to work under his guidance and learn from his experience.

I would also like to express my gratitude to Prof. Mirko Stocker who has acted as a subject-matter expert and provided me with valuable feedback and suggestions when it came to the technical aspects of this project. His work inspired the topic of this thesis, and it has inspired me to further explore the field of energy-efficient software engineering.

Furthermore, I would like to thank Prof. Dr. Li Fang and the employees at the College of Computing & Data Science, Nanyang Technological University in Singapore for providing me the technical infrastructure to conduct measurements for this project thesis. Their support allowed me easy and free access to technical means to conduct the measurements and experiments necessary for this project.

I would also like to thank my former fellow students and colleagues Marco, Christian, and Valentin for proofreading this report and providing valuable feedback. Their insights and suggestions have helped me to ensure that the report is understandable and well-structured.

Last but not least, I would like to thank my family, friends, and especially my partner Céline for their unwavering support and encouragement throughout this project and my exchange semester in Singapore. Their belief in me and their emotional support were invaluable during the challenging times of this project.

Table of Contents

Declaration of Authorship . . . 1
Abstract . . . 3
Acknowledgements . . . 5
1. Introduction . . . 9
1.1. Context and Objectives . . . 9
1.2. Target Audience . . . 10
1.3. Results . . . 10
2. Background Information . . . 13
2.1. Enterprise Applications . . . 13
2.2. Software Quality Attributes . . . 25
2.3. Summary and Outlook . . . 35
3. Measurement Techniques and Experiment Design . . . 37
3.1. Measurement Methods and Tools . . . 37
3.2. Architecture of Observed Systems and Tool Deployment . . . 40
3.3. Measurement Challenges . . . 45
3.4. Specification, Tooling and Configuration . . . 47
3.5. Summary and Outlook . . . 55
4. Measurement Results . . . 57
4.1. PetClinic Experiment: Establish a Baseline . . . 58
4.2. PetClinic Experiment: Compare JPA and Spring Data JPA . . . 68
4.3. PetClinic and LakesideMutual Experiments: Compare Master Data APIs . . . 75
4.4. LakesideMutual Experiment: Compare Different Services . . . 82
4.5. LakesideMutual Experiment: Compare Workflow Variants . . . 89
5. Discussion . . . 99
5.1. Analysis and Interpretation of Measurement Results . . . 99
5.2. Generalization of Measurement Results . . . 106
5.3. Related Work . . . 109
5.4. Retrospective . . . 110
5.5. Outlook . . . 114
6. Conclusion . . . 117
7. Appendices . . . 119
Appendix A: Glossary . . . 119
Appendix B: Bibliography . . . 123
Appendix C: List of Figures . . . 131
Appendix D: List of Tables . . . 134
Appendix E: List of Listings . . . 135

1. Introduction

Software quality attributes play a crucial role in the development and maintenance of enterprise applications. This project thesis sets out to investigate the state of the art in measuring selected software quality attributes in enterprise applications. The goal is to contribute to the understanding of how performance and energy efficiency can be effectively measured and analyzed in practice.

1.1. Context and Objectives

Performance and energy efficiency are important quality attributes of software systems [6] [7]. In recent years, the topic of energy efficiency has also become increasingly important, especially for cloud-native applications [8]. Architecture metrics have already been considered in an initial project thesis [9] and in a supplementary paper on observability in software architectures [10]. There is also a new blog on Growing Green Software (GGS) [11], whose posts report on performance and efficiency measurements in a Java-based Spring Boot sample application.

This project thesis aims to investigate whether, how and why the state of the art in the field of performance and efficiency measurements reported in the academic literature differs from practice. The measurement results of selected GGS blog posts will first be reproduced, then compared with the behaviour of another, already known example application, and eventually generalized. The mentioned objective is separated into five sub-objectives and formulated as research questions.

1. How are the two types of software quality attributes performance and resource and energy efficiency defined
   a. in the scientific literature and in official standards (ISO/IEC/IEEE) and
   b. in the gray literature (e.g., Q42, Growing Green Software blog)?
2. How do performance tests and energy efficiency/resource consumption measurements have to be set up so that their results are accurate, meaningful (with respect to the definitions from question 1) and reproducible (e.g., with respect to the FAIR criteria)?
3. Is it possible to reproduce the measurements of the Spring Boot PetClinic sample that are reported in the Growing Green Software blog? Do the interpretations of the data given in the blog posts require clarification and discussion? How could the reported test and measurements be improved (taking the answers to questions 1 and 2 into account)?
4. When measuring selected use cases of the sample application LakesideMutual in the same way as the Spring Boot PetClinic sample, how do the two result sets compare? How can the differences be explained? Does the monolith version of LakesideMutual show a different behavior than the microservices version?
5. How can the results from questions 1 to 4 be generalized so that they can serve as guidelines and examples for future tests and measurements of
   a. other Spring Boot applications
   b. other Web-based applications
   c. any distributed, software-intensive system?

To address these research questions, this project thesis follows a systematic approach. Initially, it researches definitions of performance and resource and energy efficiency from various sources. Next, it examines suitable tools and techniques for measuring these quality attributes in practice. The selected tools and techniques are then applied to reproduce existing measurements and expand the measurements to a second enterprise application. The findings are compared, discussed, and generalized for future measurements of software quality attributes.

1.2. Target Audience

Academic personnel, such as professors, students, and researchers, can refer to the measurement tools and techniques presented in this thesis to further their own research or to apply them in their experiments. They can compare the results of their own measurements with the findings presented in this thesis and gain insights into the effectiveness of different measurement approaches.

Practitioners, such as software engineers or DevOps engineers, can apply the established metrics in their projects to evaluate the quality of their software. The gained insights enable them to understand the implications of their measurements and improve their software. Software architects can refer to the approach presented in this thesis to design and implement continuous measurement strategies in their projects.

Additionally, it may be of interest to business personnel, who are involved in software development projects and want to understand the implications of software quality attributes and the process of measuring them. They can relate to the findings and result discussion of this thesis to assess the impact of software quality on project success and cost efficiency. It can help them make informed decisions about resource allocation and project management.

1.3. Results

This thesis establishes measurable aspects for performance and energy efficiency in the context of enterprise applications, researches best practices for measuring these aspects, applies them in experiments, and evaluates the results.

Performance is not directly measurable, it needs to be characterized with measurable aspects. Performance can be characterized with latency, throughput, and scalability [6]. We specify that scalability should be treated as a separate quality attribute on the same level of abstraction as performance, because scalability requires additional measurable aspects itself. This thesis focuses on latency and throughput as measurable aspects and establishes *round-trip time* as the metric for latency and *average requests* as the metric for throughput.

Resource and energy efficiency is not clearly defined in the context of software engineering. This thesis refers to the terms *useful work* and *energy efficiency factor* to characterize resource and energy efficiency [12]. We propose to leverage the INVEST mnemonic [13] to define useful work in the context of software engineering. We understand this method as a step towards a more structured approach of defining useful work and contributing to the discussion of how to measure resource and energy efficiency in software engineering.

The thesis identifies *RAPL* and *JoularJX* to measure the energy consumption of a Java application on Windows and Linux. JoularJx is combined with *Apache JMeter* as a load testing tool to measure the performance of a Java application. The test setup includes automated test scenarios, setup and cleanup steps, and resource constraints on the JVM to ensure a controlled test environment. The experiments apply these methods and tools to two Spring Boot enterprise applications written in Java, the PetClinic and LakesideMutual. The experiments consist of multiple test scenarios, which build on each other to achieve meaningful results across different test environments and applications. Initial measurements reproduce existing PetClinic measurements from the GGS blog and establish a baseline for this thesis. The thesis then adapts the test plan and applies the same techniques and tools to measure LakesideMutual.

The results confirm that performance and energy consumption are affected by external factors, such as hardware, operating system, and implementation details. The results suggest that the relative distribution of energy consumption is comparable across different systems, different enterprise applications, and across different sets of operations. The findings indicate an inverse correlation between performance and energy efficiency for varying hardware resources. Furthermore, the results indicate a strong correlation when the same hardware is used but implementation details vary.

The research field of performance and energy efficiency measurements is broad and still evolving. Future work could explore different Java, Spring Boot, and database versions, configurations, or even code-related changes. Instead of focusing solely on enterprise applications, future research could also measure other types of applications, such as embedded systems, controllers, or mobile and desktop applications. Such a work could contribute to a better understanding of performance and energy efficiency in a broader context. The topic of performance and energy efficiency is growing and getting increasingly important, especially with cloud-native deployments. We aim to continue this work in the future and investigate on the energy efficiency of cloud-native applications and cloud infrastructure.

2. Background Information

This chapter provides background information on enterprise applications and the software quality attributes that are relevant to their evaluation. It starts with a definition of enterprise applications in terms of their characteristics, complexity, and purpose. The focus then shifts to selected software quality attributes, performance and resource and energy efficiency. Understanding these concepts is essential for measuring the two enterprise applications in subsequent chapters.

2.1. Enterprise Applications

This section focuses on enterprise applications, as opposed to other types of software systems, such as embedded systems, control systems, or desktop and mobile applications. It defines *enterprise applications* in terms of their characteristics and further introduces two example applications, the *PetClinic* and *LakesideMutual*.

2.1.1. Definition of Enterprise Applications

Enterprise applications differ from other software systems in terms of their complexity, size, and purpose. Fowler states that enterprise applications are "about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data" [14]. Enterprise applications face different design challenges, such as user and channel diversity, process and resource integrity, integration needs with other systems, and complex domain models and processing rules [15]. Prominent examples of enterprise applications are customer relationship management (CRM), enterprise resource planning (ERP), supply chain management (SCM) systems, or online shops such as Amazon. These applications are designed to support long-running, complex business processes for multiple users concurrently interacting with the system [16].

User and channel diversity refers to a wide variety of users and points of interaction with the system. Different users can have different roles and responsibilities, and they may interact with the system through different channels, such as web browsers, mobile devices, or APIs. Customers of Amazon would like to access the online shop through their web browser, a mobile app, or the Alexa voice assistant. The challenge is to provide a tailored, but consistent user experience across all channels and users.

Process and resource integrity refers to the process or workflow order and the resources involved in the process. The process order must be followed to ensure that the system behaves correctly and consistently. Amazon needs to ensure that the order process is followed correctly, from selecting items, adding them to the cart, checking out, and processing the payment. The resources involved in the process must remain consistent and available throughout the process.

Integration needs with other systems refers to the need to connect and communicate with other systems, such as databases, third-party services, or legacy systems. Amazon needs to integrate with various systems, such as payment providers, shipping companies, and inventory management systems. The challenge is to ensure that the integration is seamless and does not disrupt the overall system performance or user experience.

Complex domain models and processing rules refer to the business logic and rules that control the system's behaviour. The challenge is to ensure that the domain model remains clear, consistent, and adaptable to changes as the business evolves. All these challenges can be tackled with different established methods or patterns described in Pattern of Enterprise Application Architecture (PoEAA) [14], Domain-Driven Design (DDD) [17], or Enterprise Integration Patterns (EIP) [18].

Non-functional requirements, or software quality attributes, are critical for enterprise applications [16]. The customers of Amazon expect the online shop to be available, responsive, and secure. The online shop must be able to handle hundreds or thousands of concurrent users and transactions without data consistency issues. With inconsistent data, the online shop would not be able to process orders correctly, leading to unsatisfied customers.

Additionally, business processes in enterprise applications are often long-running and complex. Customers of an online shop can select their favourite payment method, pay the goods, select a shipping method, and trigger the delivery process. The entire process can take several days and is updated throughout the process. Such updates may partly depend on third-party companies, such as payment providers or shipping companies. In another example, banking customers may have contracts for loans or mortgages that run for several years up to decades.

Eventually, enterprise applications must comply with regulations and laws in their respective domains. In the banking and finance industry, the Foreign Account Tax Compliance Act (FATCA) [19] or the Sarbanes-Oxley Act (SOX) [20] are examples of regulations that require strict data handling and reporting procedures. A failure to comply with these regulations can result in severe penalties and legal consequences for the organization operating the enterprise application.

Figure 1 shows a mind map including the topics covered and their relationships.

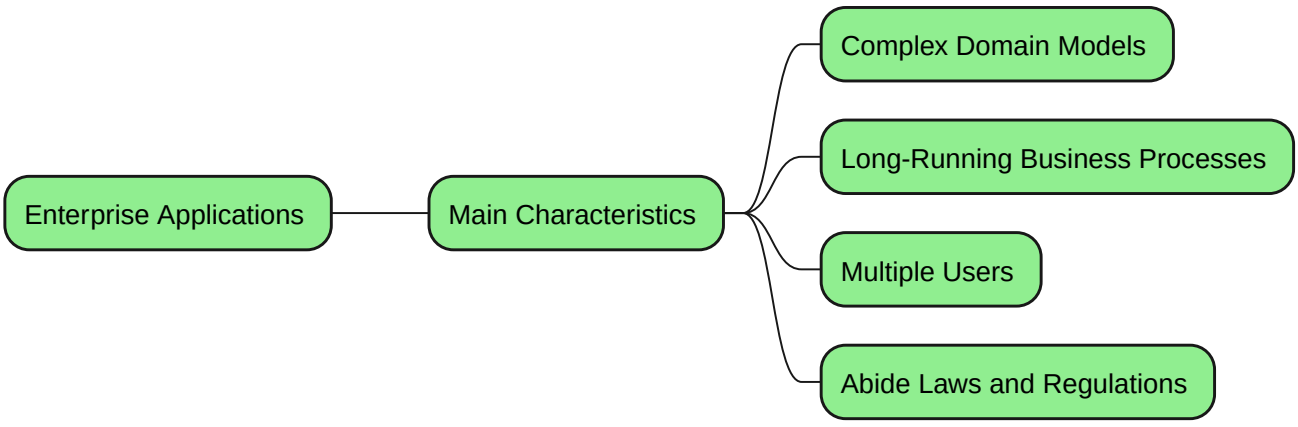

Figure 1. A mind map illustrating the main characteristics of enterprise applications

This list of examples is not exhaustive, but it illustrates the complexity of enterprise applications and the challenges they face. It requires time and effort to design, implement, and maintain such complex software systems. Due to time constraints and the scope of this thesis, it is not feasible to design and implement a new enterprise application. Therefore, two existing enterprise applications are used throughout the experiments to conduct measurements and evaluate the results.

2.1.2. Introduction of the PetClinic Application

The *PetClinic* is a monolithic Spring Boot enterprise application written in Java. The PetClinic is a renowned example application in the Spring community [21]. Its purpose is to demonstrate the capabilities of the Spring Framework.

The Spring framework is used to build production-ready Java enterprise applications [22]. The Spring Boot framework builds upon Spring and eases the development of Spring applications [23]. It streamlines the development process and establishes the concept of convention over configuration [24]. Spring Boot as well as Spring support a variety of configurations, enabling developers to tailor applications to their specific needs.

This thesis aims to establish a baseline for further experiments by reproducing the Growing Green Software (GGS) blog measurements. The GGS blog refers to a specific version of the PetClinic, the *Spring PetClinic REST* project [25]. This specific version solely provides a REST API and no user interface. The PetClinic application could be combined with a user interface like the *Spring PetClinic Angular* project [26]. The focus lies on the HTTP endpoints, therefore a user interface is omitted in the context of this thesis. The application can run with different data sources, such as an H2 or HSQLDB in-memory data store, a MySQL database, or a PostgreSQL database. The database can be accessed with either JDBC, JPA, or Spring Data JPA.

A good understanding of the application, its functionality, its architecture, and its deployment is essential for analyzing and interpreting the experiment results. Figure 2 visualizes the main classes *Owner*, *Pet*, and *Vet* of the PetClinic application and their relationships to other classes.

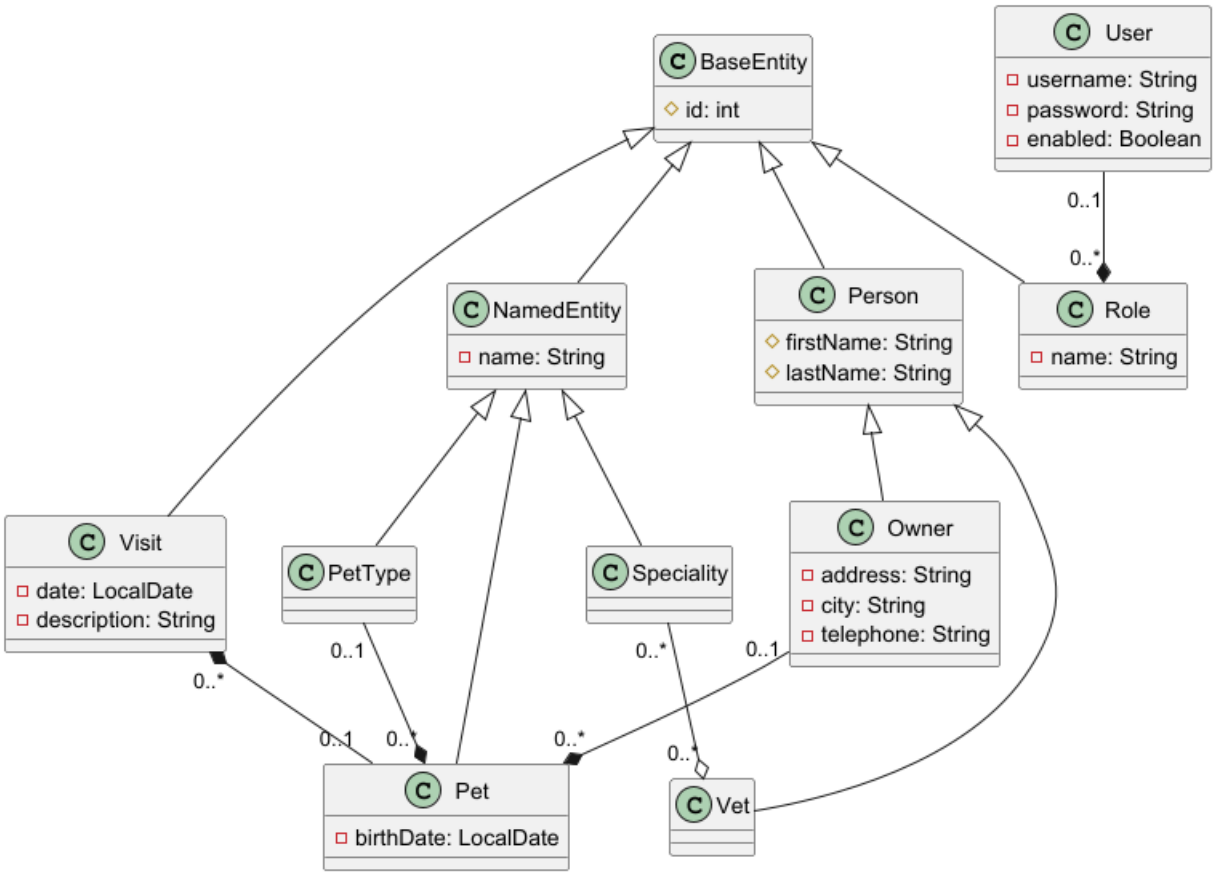

Figure 2. The UML class diagram of the PetClinic application

It appears that the *NamedEntity* and *BaseEntity* classes solely exist for the purpose of providing a name and an id attribute to its subclasses. The concept of inheritance in object-oriented programming is used to extend the behaviour of superclasses [27] while not violating the Liskov substitution principle [28] [29]. Classes like *PetType* or *Speciality* appear to fulfill the role of a database entity while solely storing data without adding behaviour to the object. This design is known as an anemic domain model according to Evans [17].

The purpose of the PetClinic application is to manage owners of pets, their pets, and to schedule visits to veterinarians. The application provides HTTP endpoints for basic create, read, update, and delete (CRUD) operations on owners, pets, veterinarians, pet types, specialities, visits, and users. All seven classes correspond to eight entities, which are mapped to database tables.

Figure 3 shows an entity-relationship diagram. The many-to-many relationship between the *Vet* and *Speciality* entities in Figure 2 is resolved with the *vet_specialities* join table in Figure 3. The 'E' icon corresponds to the Spring Boot @Entity annotation and represents a database table, the 'J' icon refers to the join table.

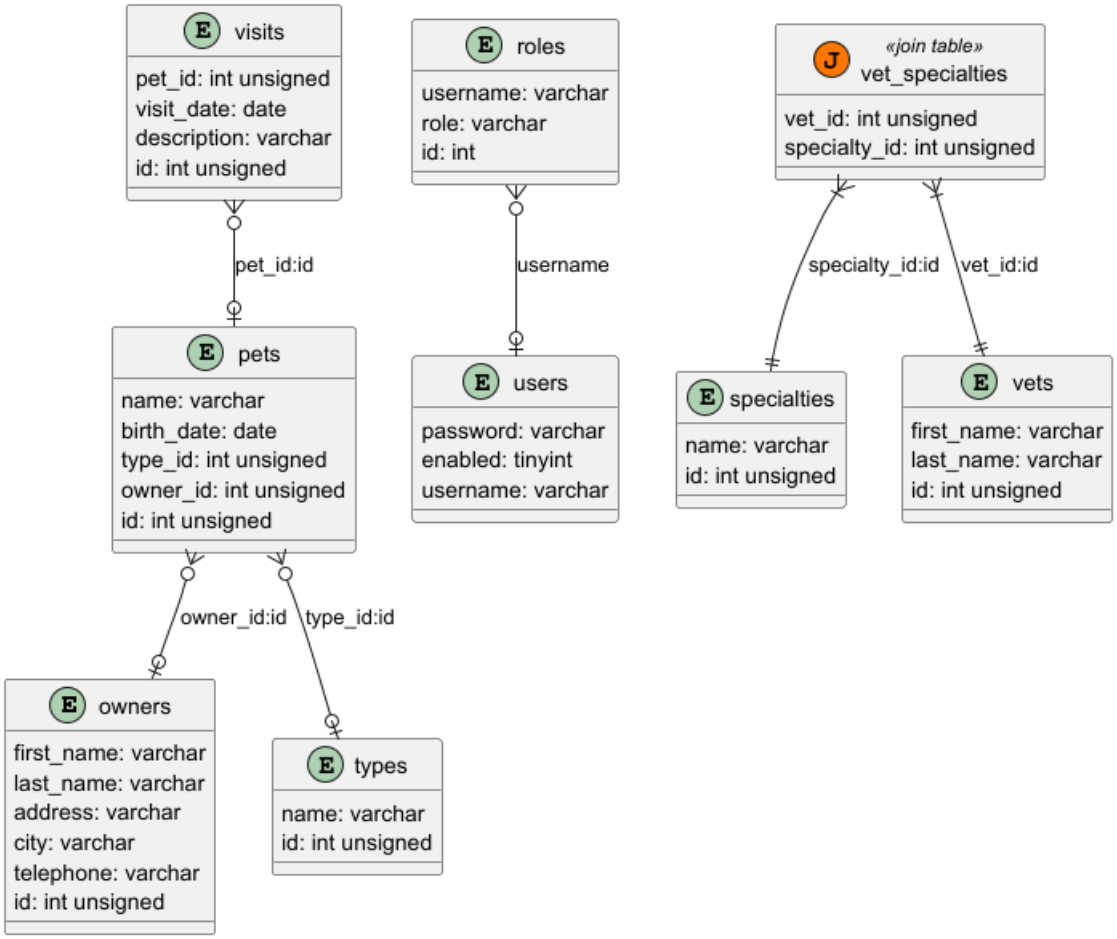

Figure 3. The entity-relationship diagram (UML class diagram) of the PetClinic application

A veterinarian can have expertise in multiple specialities, such as internal medicine, surgery, or radiology. A visit is appointed to a specific pet, which is of a specific pet type, such as a cat or a dog, and owned by an owner. Users of the application are assigned to different roles, either as vets, owners, or admins. Users and roles are solely relevant for a potential user interface.

The existing GGS blog test plan focuses on the create, read, update, and delete (CRUD) operations of the PetClinic application. All data can be accessed via a respective data holder class, which bundles access and provides manipulation operations. Figure 4 shows an example of retrieving all owners via the *OwnerRestController* class, which acts as a master data holder [30] for the Owner entity.

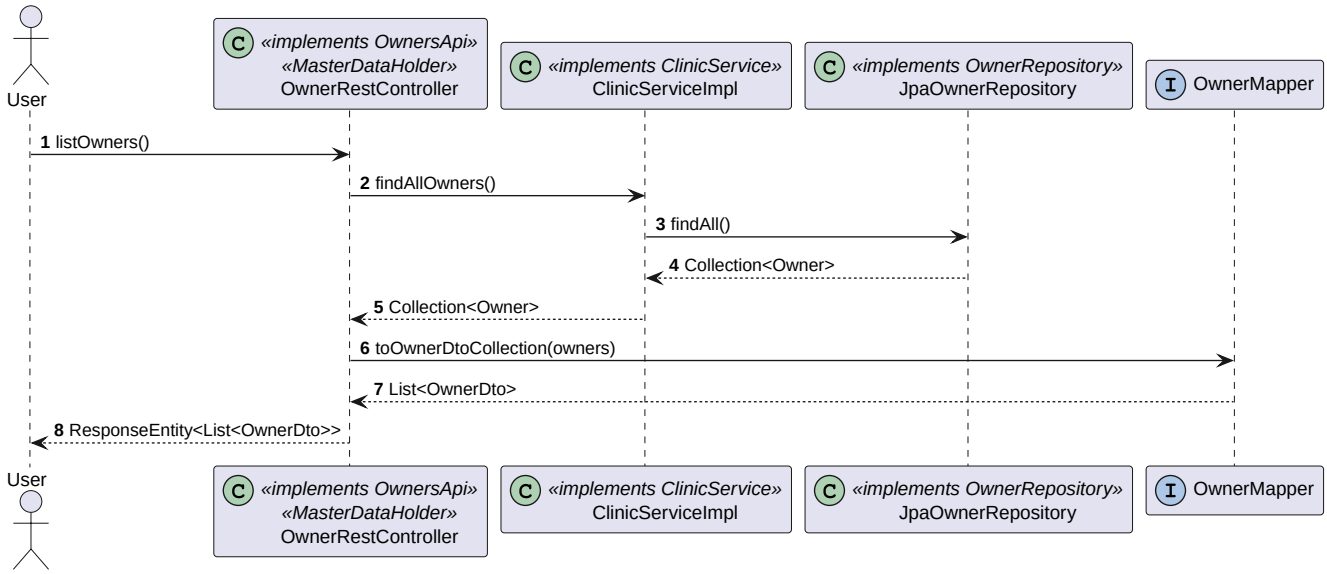

Figure 4. The UML sequence diagram of the PetClinic application for retrieving all owners

The OwnerRestController class provides endpoints for operations such as finding all owners, finding owners by name, finding owners by ID, creating a new owner, updating and deleting an existing owner. Similar operations are available for all other entities, except for the User entity, which only allows for creating a new user.

According to the definition in Subsection 2.1.1, enterprise applications are complex software systems that support long-running business processes and workflows for multiple concurrent users. While the PetClinic represents an enterprise application based on its intended purpose, its complexity is limited compared to other enterprise applications as it does not support complex business processes or workflows. A lack of typical characteristics may result in different results compared to more complex enterprise applications. The LakesideMutual application addresses this problem and allows for a comparison with the results of the PetClinic application.

The PetClinic application is built with a monolithic architecture, which combines all components of an application into a single deployable unit. The application is separated into two tiers, a database tier and a monolithic backend tier. The frontend tier is out of scope for this thesis and replaced with a load testing tool. Figure 5 shows the deployment diagram of the PetClinic application on a localhost.

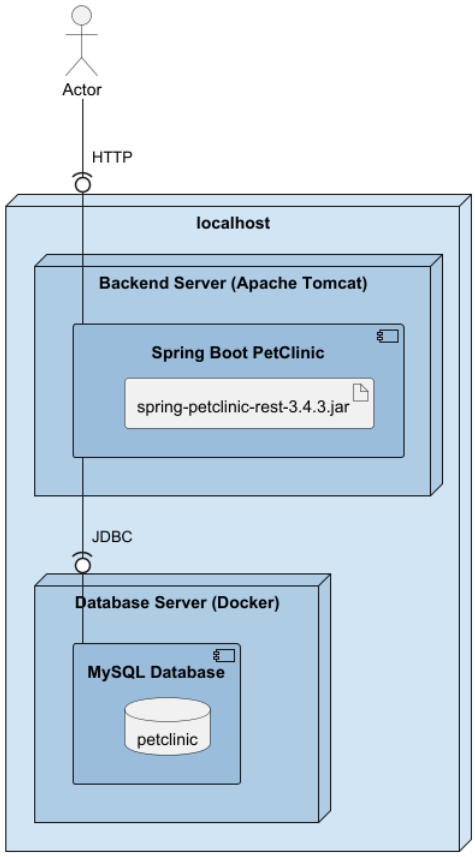

Figure 5. The UML deployment diagram of the PetClinic application

The backend runs as a Spring Boot application on an embedded Tomcat server, while the database runs in a Docker container. The project recommends to use Docker containers for persistent databases. Additionally, the GGS blog refers to a MySQL database in a Docker container for its measurements.

Figure 6 visualizes the main characteristics of the PetClinic application to reinforce the understanding of the application.

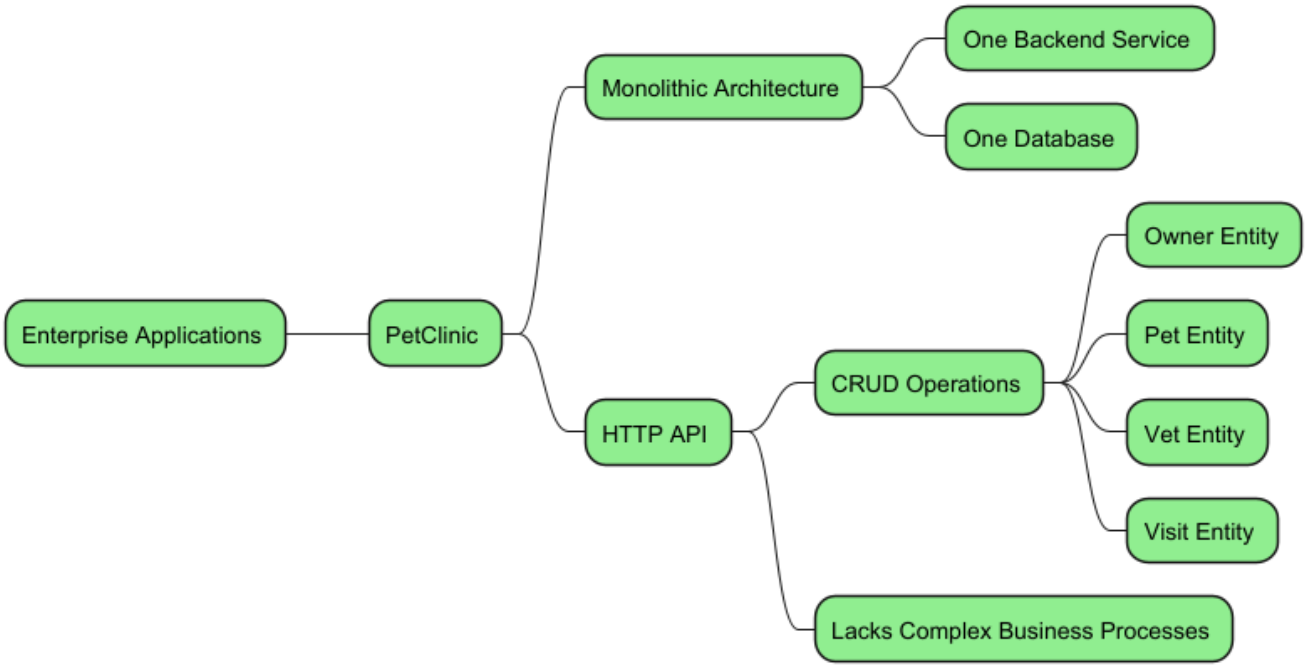

Figure 6. A mind map illustrating the main characteristics of the PetClinic application

2.1.3. Introduction of the LakesideMutual Application

LakesideMutual is a service-oriented Spring Boot enterprise application [31]. It represents the application of a fictitious insurance company called *Lakeside Mutual* and serves as a sample application in the context of Microservice API Patterns (MAP) [30] [32], Domain-driven design (DDD) [17], Patterns of Enterprise Application-Architecture (PoEAA) [14], and Enterprise Integration Patterns (EIP) [18]. This thesis utilizes the LakesideMutual application to adapt the PetClinic measurements to a second enterprise application. The results are compared with each other, interpreted and generalized.

The service-oriented architecture specifies four backend services, three frontend services, two reporting services, and two administrative services. The frontend services are out of scope for this thesis and replaced with a load testing tool. The reporting and administrative services are also out of scope and therefore omitted. The backend services use file-based H2 databases by default, but the application allows for the configuration of arbitrary databases. The databases are accessed via Spring Data JPA.

The backend services fulfill different responsibilities. The customer core service manages the customer master data and provides it to other components via HTTP API. The customer management service provides an HTTP API for the customer management frontend. This API enables employees to manage customer data and interact with customers in case of inquiries.

Figure 7 visualizes the main classes *CustomerAggregateRoot* and *InteractionLogAggregateRoot* of the customer core and customer management services respectively.

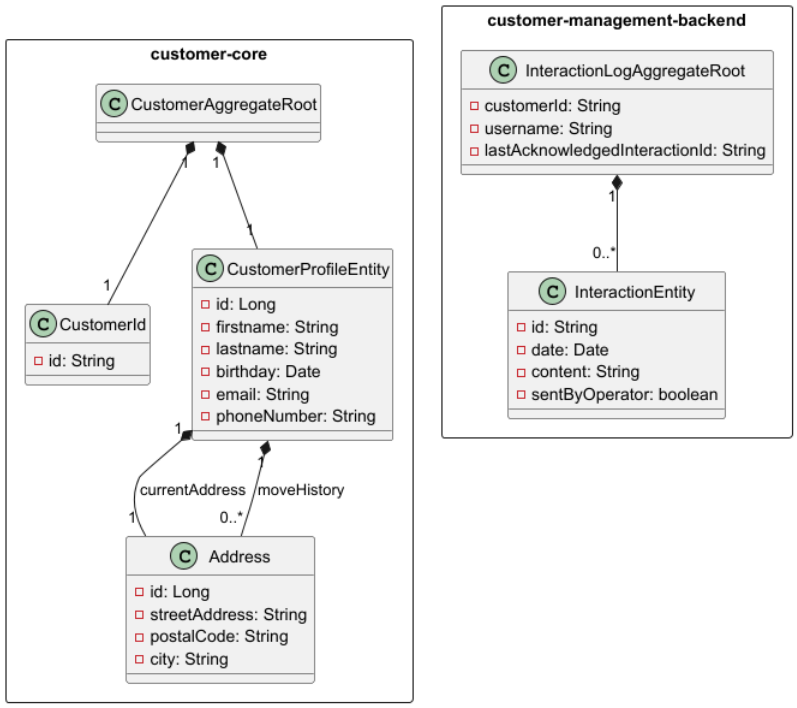

Figure 7. The UML class diagram of the customer core and customer management services

The aggregates mark a conceptual boundary in the DDD principles [17]. Within these boundaries, the composed parts are consistent according to business rules and processes. Figure 8 shows the entity-relationship diagram of both services, in which the one-to-many relationships are resolved using additional join tables.

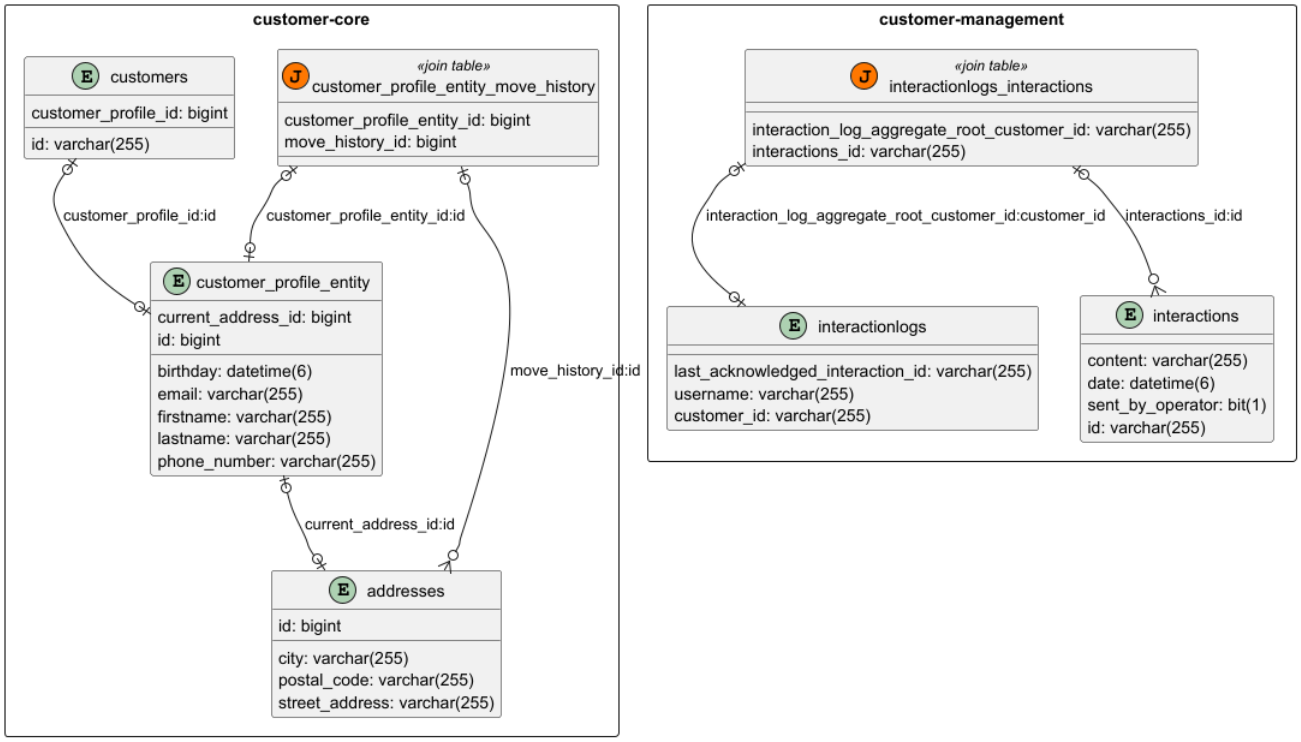

Figure 8. The entity-relationship diagram (UML class diagram) of the customer core and customer management services

The customer self-service service provides an HTTP API for the customer self-service frontend. This API enables customers to manage their personal data and request insurance quotes. The first step of the main business workflow involves customers requesting insurance quotes. Figure 9 shows the main class *InsuranceQuoteRequestAggregateRoot* of the customer self-service service and its related classes.

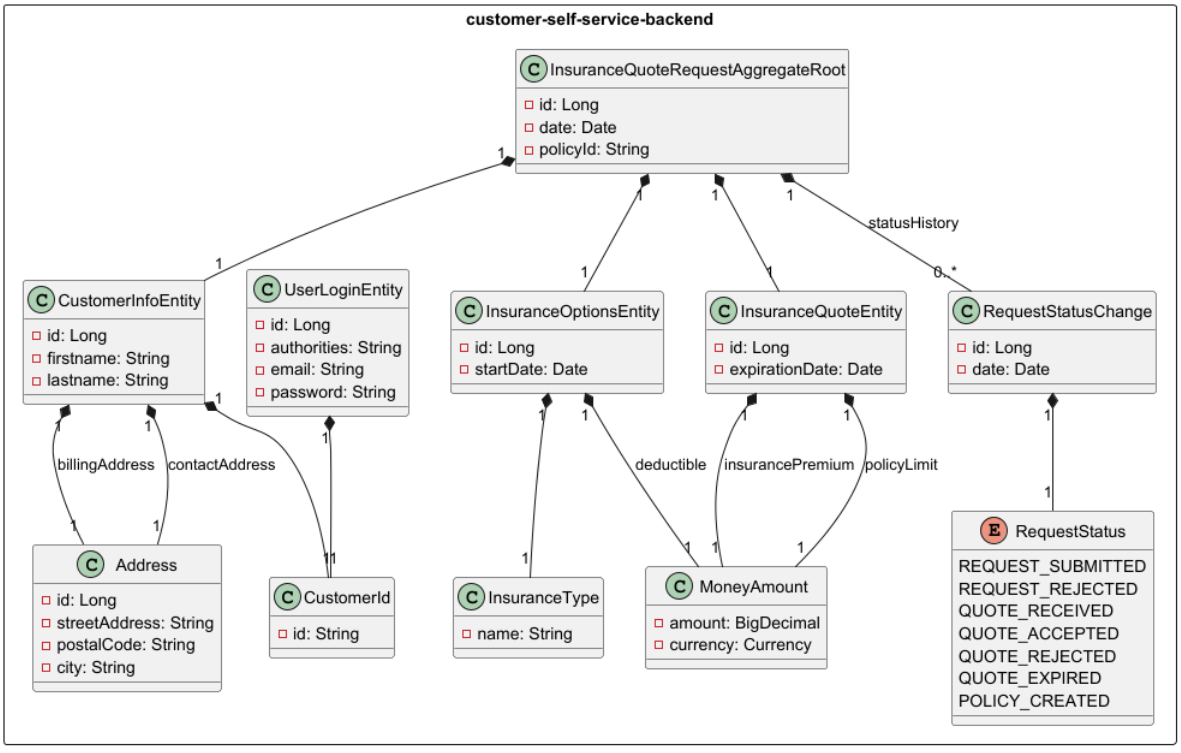

Figure 9. The UML class diagram of the customer self-service service

Classes like the *CustomerId*, the *MoneyAmount*, and the *RequestStatus* are embedded attributes of other classes. This design is known as a value object in DDD [17]. These value objects do not require additional database tables, as they are stored as attributes of other entities. Figure 10 shows the entity-relationship diagram of the customer self-service service and the value objects as embedded attributes.

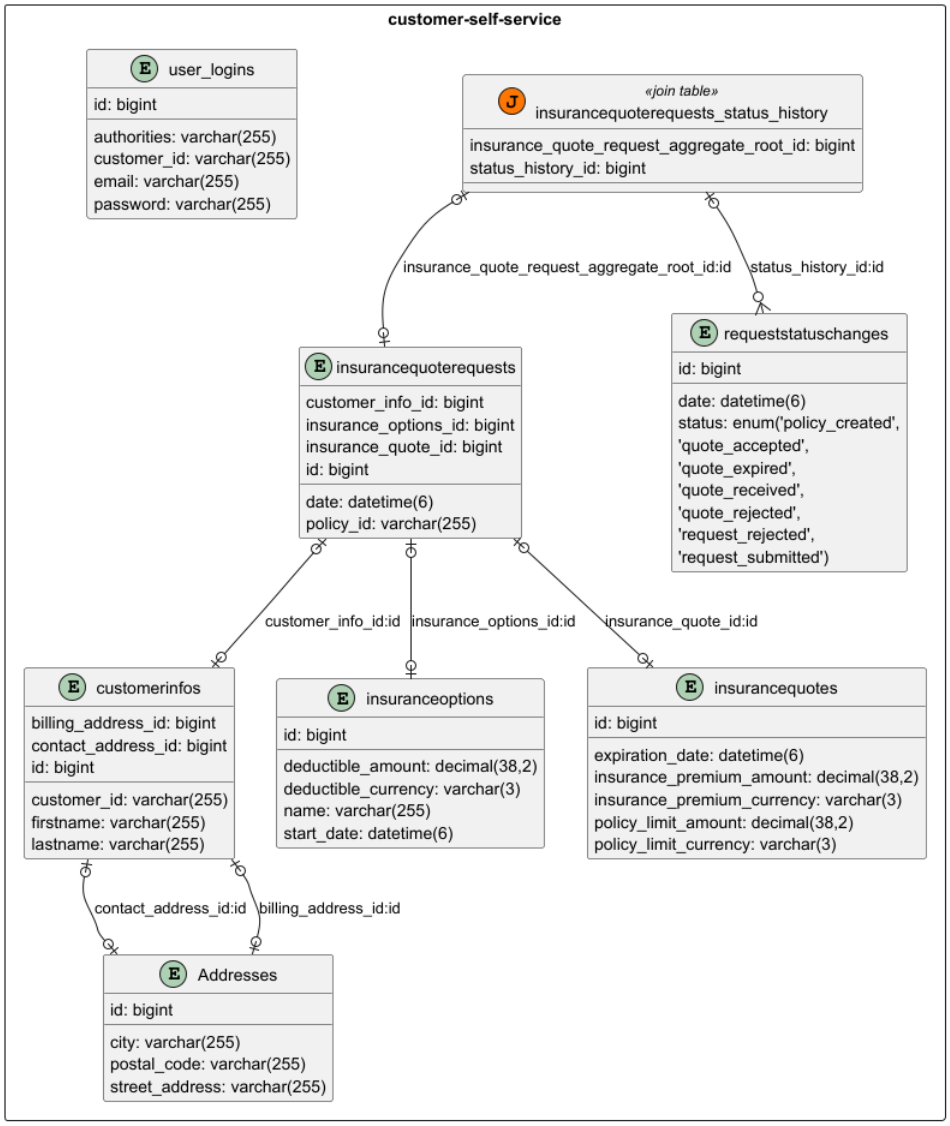

Figure 10. The entity-relationship diagram (UML class diagram) of the customer self-service service

The policy management service provides an HTTP API for the policy management frontend. This API enables employees to manage policies of customers and respond to insurance quote requests. The second step of the main business workflow involves employees of Lakeside Mutual creating insurance offers based on the insurance quote requests of customers. The insurance quote requests are updated and sent back to the customers.

Figure 11 shows the main classes *PolicyAggregateRoot* and *InsuranceQuoteRequestAggregateRoot* of the policy management service as well as their related classes.

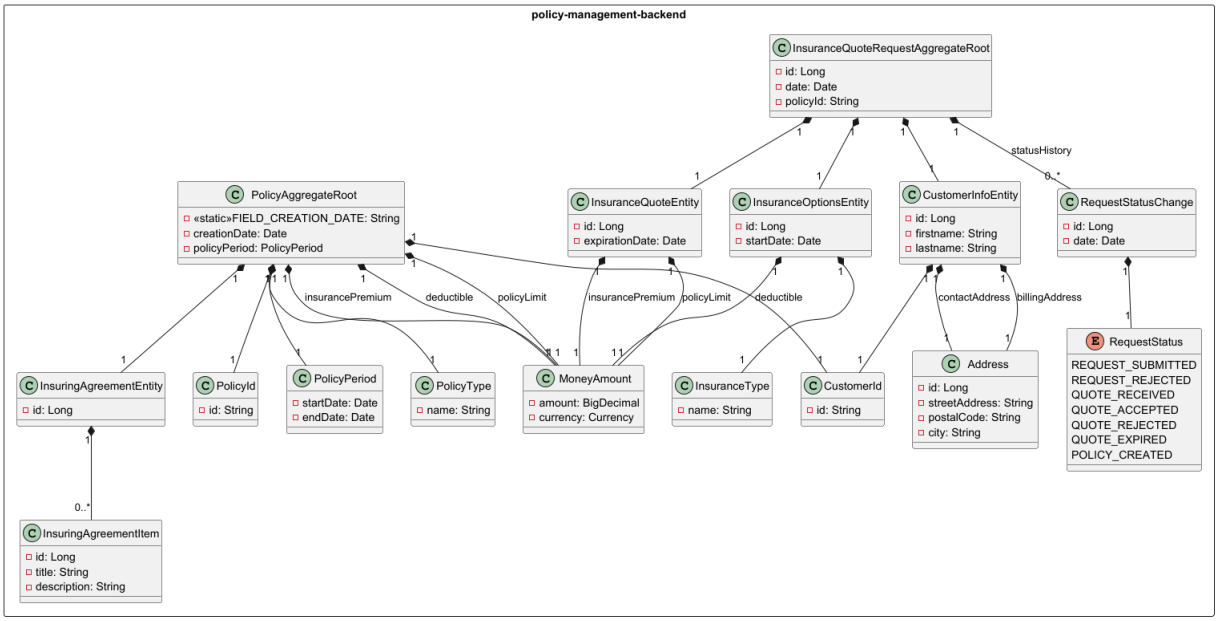

Figure 11. The UML class diagram of the policy management service

The policy management service duplicates the insurance quote request aggregate and introduces the policy aggregate. Two classes *CustomerId* and *MoneyAmount* are shared value objects between the two aggregates. The code duplication is also visible in the entity-relationship diagram, as shown in Figure 12.

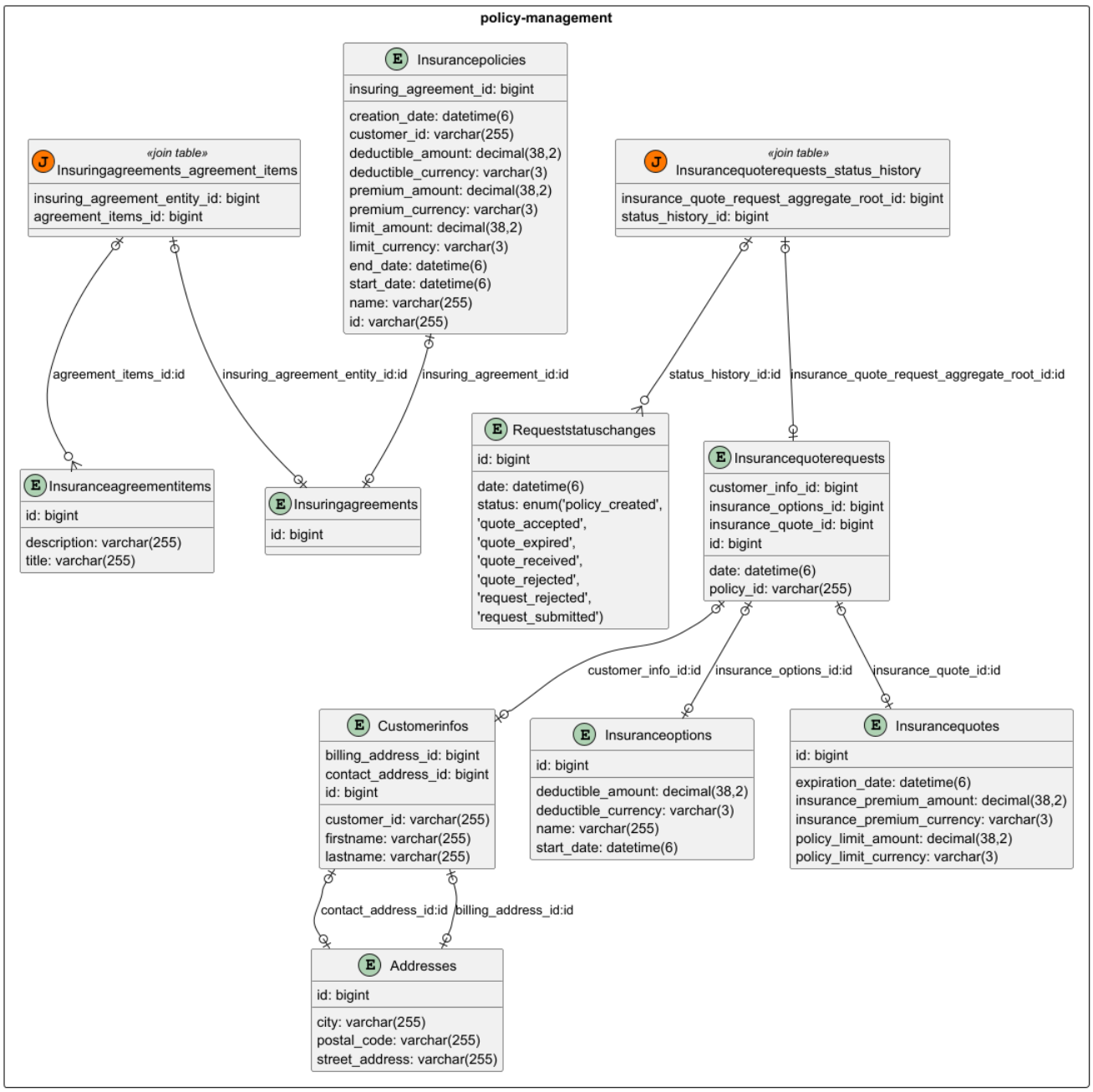

Figure 12. The entity-relationship diagram (UML class diagram) of the policy management service

The third and final step of the business workflow involves customers accepting or declining the received insurance offer in the customer self-service frontend. The acceptance of an offer results in a new insurance policy. In case customers decline an offer, they are free to request a new insurance quote, effectively restarting the main business workflow.

Similar to the PetClinic application, LakesideMutual relies on master data like customers and operational data like insurance quote requests. The *CustomerInformationHolder* class in the LakesideMutual application is a master data holder for the Customer entity [30]. Figure 13 visualizes the sequence diagram for retrieving all customers in the LakesideMutual application.

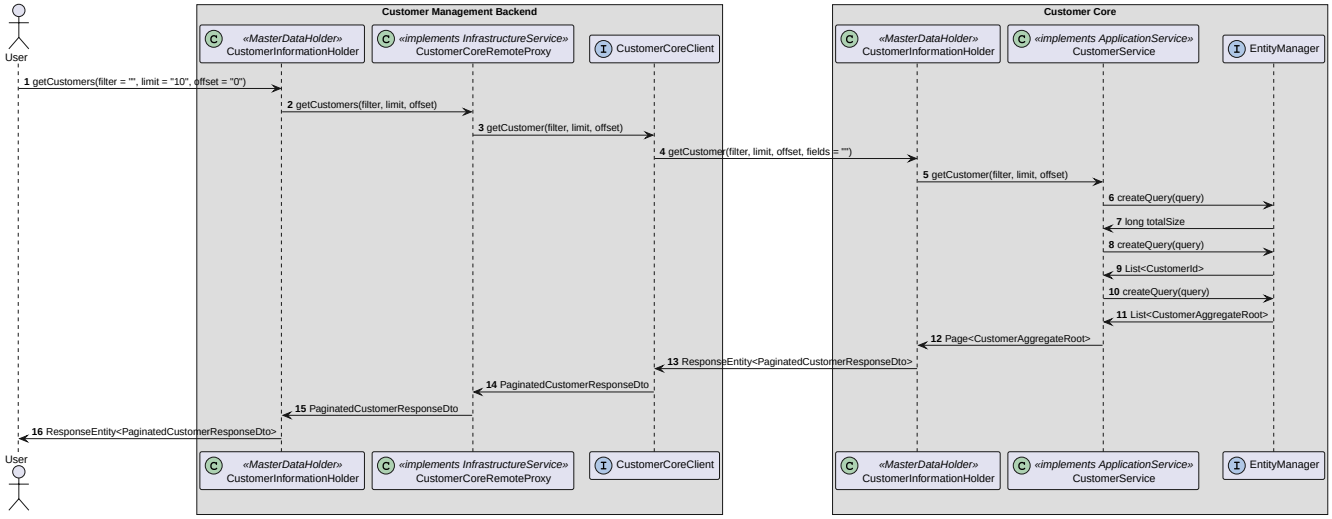

Figure 13. The UML sequence diagram of the LakesideMutual application for retrieving all customers

LakesideMutual is built with a service-oriented architecture, which separates the application into multiple services that are deployed independently. The application is separated into four backend tiers and one database tier. The frontend tiers are omitted for this thesis and replaced with a load testing tool. Figure 14 shows the deployed services of the LakesideMutual application and their communication with each other and the database.

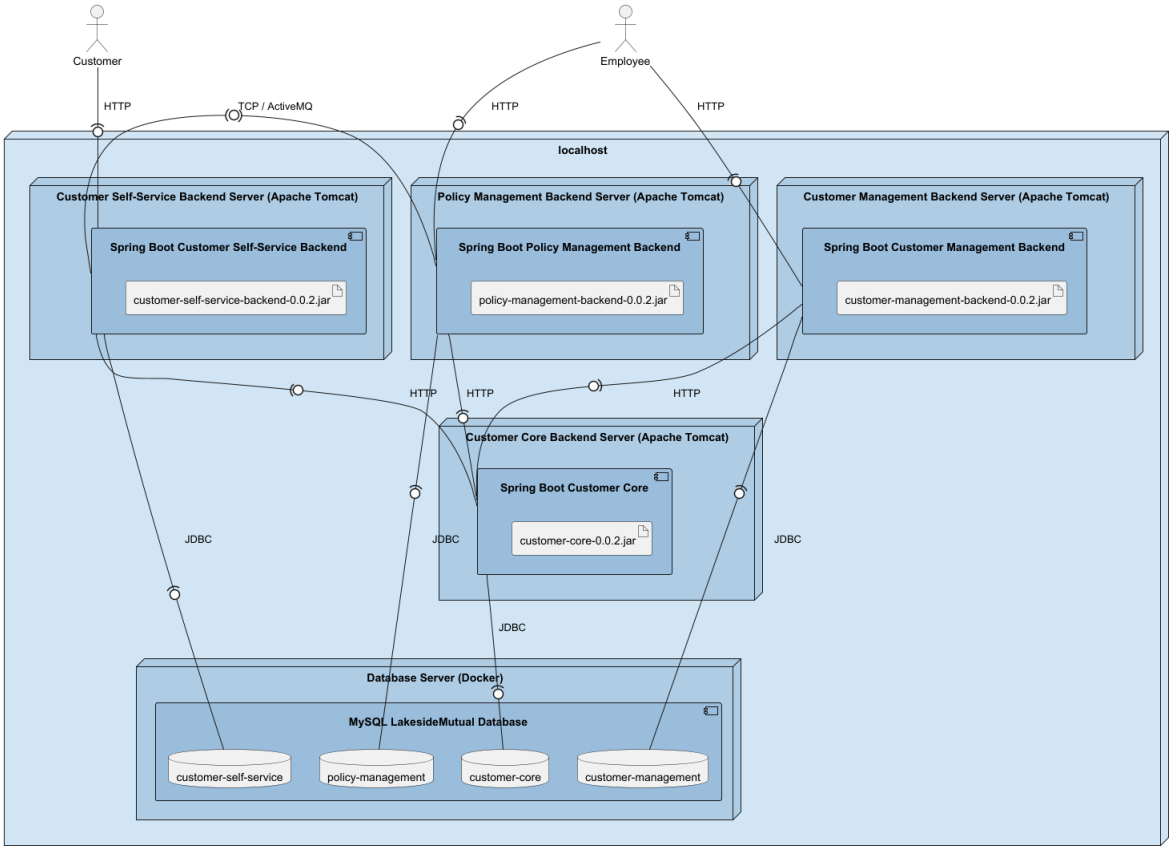

Figure 14. The UML deployment diagram of the service-oriented LakesideMutual application

The diagram illustrates that the backend services communicate with the customer core service, which provides the customer data. The customer self-service and policy management services communicate with each other via message queues to process insurance quote requests. Each service has its own database running in one MySQL Docker container. The experiments in this thesis refer to the illustrated deployment.

This thesis builds on the foundational PetClinic experiment, adapting the test scenarios and applying them to LakesideMutual to broaden the insights into enterprise applications. The adapted measurements focus on the Customer entity and the insurance request workflow. The increased complexity of the business processes and the additional services allow for a more accurate measurement of enterprise applications according to the definition in Subsection 2.1.1. Table 1 provides a comprehensive overview of the two applications, highlighting their characteristics and differences.

Table 1. A comparison of the PetClinic and LakesideMutual enterprise applications in terms of their characteristics

Characteristic	**PetClinic**	**LakesideMutual**
Framework / Programming Language	Spring Boot / Java	Spring Boot / Java
Architecture	Monolithic	Service-oriented
Number of Services	1	4
Number of Databases	1	4
Configured Data Source	MySQL Database	MySQL Database
Configured Database Access	JPA or Spring Data JPA	Spring Data JPA
Data Manipulation Operations	• Create Operations • Read Operations • Update Operations • Delete Operations	• Create Operations • Read Operations • Update Operations • (Delete Operations only on Policies)
Business Processes	None	Insurance Quote Request Workflow

The two enterprise applications differ in their architecture, the number of services, the database access, the data manipulation operations, and the business processes. This thesis considers said differences in the test scenarios to achieve meaningful results and ensure comparability. Section 2.2 provides an overview of the software quality attributes that are relevant for the evaluation of the two enterprise applications.

2.2. Software Quality Attributes

This section elaborates on the software quality attributes *performance* and *resource and energy efficiency* to measure the quality of enterprise applications. It characterizes these two attributes with measurable aspects and metrics considering official standards, academic literature, and grey literature. The additional annex document provides further details on the research methods used to identify the relevant literature.

Software quality attributes should be specific, measurable, achievable, relevant, and time-bound (SMART) [33]. Zimmermann and Stocker mention the Quality Attribute Scenario (QAS), which is an alternative term for SMART [34]. "A quality attribute scenario specifies a measurable quality goal for a particular context" [34]. The Software Engineering Institute (SEI) provides a fact sheet for Quality Attribute Workshops (QAWs) [35], which are used to establish measurable quality attributes. SEI states that the key concept of a QAW is a meeting with stakeholders; "during which scenarios representing the quality attribute requirements are

generated, prioritized, and refined" [35]. These scenarios provide insights into important business objectives and specify measurable quality attributes. Bass, Clements, and Kazman provide further explanations on the different phases and necessary steps of a QAW in [36]. This thesis aims to provide measurable aspects for selected quality attributes.

2.2.1. Performance

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) provide a standard for "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model" [37]. The ISO/IEC 25010:2023 standard contains definitions for performance efficiency and resource utilization. Performance is defined as the "capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions" [37]. The standard mentions an efficient use of resources, which is further elaborated on. Resource utilization is defined as the "capability of a product to use no more than the specified amount of resources to perform its function under specified conditions" [37]. The standard does not provide examples for the terms *resources* and *conditions*, they need to be defined individually for each use case. This standard lacks specific, measurable aspects for performance and Subsection 2.2.2 focuses on resource and energy efficiency in depth.

The Institute of Electrical and Electronics Engineers (IEEE) provides a standard for "Systems and software engineering — Life cycle management — Part 4: Systems engineering planning" [38]. It covers system engineering, not software engineering, but it may help to derive measurable aspects from a different perspective. The ISO/IEC/IEEE 24748-4:2016 standard defines a measure of performance (MOP) as an "engineering parameter that provides critical performance requirements to satisfy a measure of effectiveness (MOE)" [38]. The term MOE is defined as an ""operational" measure of success that is closely related to the achievement of the operational objective being evaluated in the intended operational environment under a specified set of conditions" [38].

The old, replaced IEEE 1220-2005 standard [39] provides a refined definition of performance requirements. It defines a performance requirement as a "measurable criteria that identifies a quality attribute of a function or how well a functional requirement must be accomplished" [39]. This standard emphasizes that performance is not directly measurable; performance needs to be characterised with measurable aspects. According to IEEE, performance measurements should rely on technical performance measures (TPMs), which measure critical MOPs, which in turn satisfy MOEs. In other words, we use TPM to measure performance requirements in order to evaluate the achievement of objectives under specified conditions. In case these MOPs are not met, the project could be at a risk of cost, schedule, or performance problems [39]. The standard neither provides examples or specific aspects for TPMs nor MOPs, it does not provide clear measurable aspects of performance.

Grey literature provides insights into a more practical approach of analysing software quality. The arc42 Quality Model (Q42) [40] is such an approach to analyse product and system quality. Q42 generally follows the ISO/IEC 25010 standard with explanations and examples. Q42 defines *performance* according to the previously established ISO/IEC 25010 standard [37]. It lists an example of a performance requirement [41], which measures performance according to the response time it takes to render an image. This example consists of a context information, a stimulus or trigger, and a metric that specifies the expected performance.

Academic literature provides a comprehensive definition of performance in the context of distributed software systems. According to Denaro et al. [6], **performance can be characterized with latency, throughput, and scalability**.

The definition of latency, throughput, and scalability [6]

> Latency typically describes the delay between request and completion of an operation. Throughput denotes the number of operations that can be completed in a given period of time. Scalability identifies the dependency between the number of distributed system resources that can be used by a distributed application (typically number of hosts or processors) and latency or throughput.
>
> — Denaro et al.

Q42 also provides definitions for latency, throughput, and scalability. Q42 specifies: "Latency in general is a time delay between the cause and the effect of some change in a system" [42]. The definition of throughput depends on the context it is used in, Q42 refers to a definition from Burke: "Throughput is a measure of how many units of information a system can process in a given amount of time" [43] [44]. Q42 refers to the ISO/IEC 25010:2023 standard for the definition of scalability. "Capability of a product to handle growing or shrinking workloads or the ease with which the product's capacity can be adapted to handle variability" [45] [37].

We argue that scalability should be treated as a separate quality attribute on the same level of abstraction as performance. Scalability itself is not measurable, it needs to be characterized with measurable aspects similar to performance. We specify that scaling a software system should not affect other software quality attributes of said system. This requires additional metrics and test cases to ensure that no other quality attribute is affected by scalability. We state that measuring **scalability is out of scope** for this thesis.

Baumgartner measures performance with latency, throughput, and memory consumption [46] [47]. Initial memory consumption and average memory consumption under load are interesting aspects of performance, especially from a server-side perspective. An *Abstracta* blog post [48] suggests metrics, such as response time, system throughput, and concurrent users. The amount of clients or concurrent users is a crucial aspect when performing load or stress tests [49]. The author of the *Growing Green Software* (GGS) blog measures performance by measuring the average execution time for an operation in a Java application [50], He measures performance with a tool named *Apache JMeter* in the subsequent post "Evolution of Energy Usage in Spring Boot" [51]. The author does not further specify the metric he uses, chances are that he refers to the total execution time of a JMeter test plan.

Table 2 presents a breakdown of measurable performance quality attributes on a conceptual level in the context of distributed software systems.

Table 2. Measurable performance quality attributes and metrics (own presentment)

Aspect	Description	Metric	Example
Latency	**Round-trip time (RTT):** The time it takes from sending a request to receiving a responses.	*2 * Propagation Delay + Processing Time*	A login request takes 2 * 100ms propagation delay + 150ms processing time = 350ms. The propagation delay is the time it takes to transmit a signal from a sender to a receiver.
Throughput	**Average requests:** The number of requests that a system can process in a given time frame.	*Total Requests / Time Interval*	The system manages to process 10 login requests per second on average.

Figure 15 summarizes the measurable aspects and metrics of the performance quality attribute.

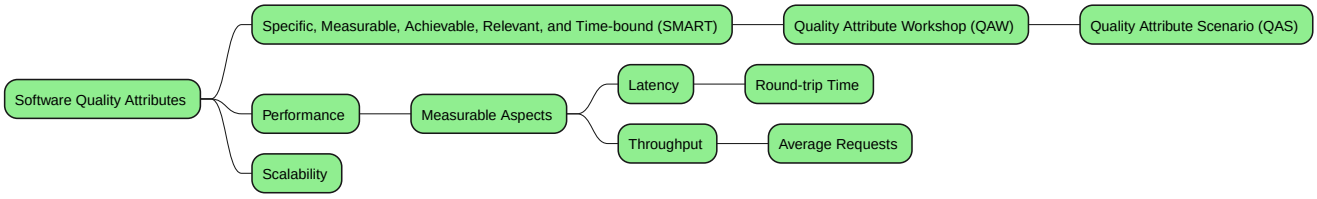

Figure 15. A mind map illustrating the performance software quality attribute

The established metrics are used in the experiments to measure the performance of the applications.

2.2.2. Resource and Energy Efficiency

Like performance, resource and energy efficiency is a software quality attribute that is not directly measurable. According to various sources, resource and energy efficiency are not clearly defined in the context of software engineering [7] [12] [52]. We address this gap by considering definitions from other fields such as hardware engineering and adapt them to software engineering.

The ISO and the IEC provide a standard for "Information technology — Data centres — Key performance indicators". The ISO/IEC 30134:2016 standard contains definitions for data centers and covers their resource usage efficiency. They standardize that information technology (IT) equipment energy consumption has to be measured in kilowatt-hours (kWh) [53]. We can easily convert kWh to other units like joules or watts depending on the use case. Resource usage efficiency is defined as the "ratio of output to the resource used by the device or system when the input and output units are the same" [54]. Even though this definition is associated with servers and data centers, it provides a basis for understanding resource and energy efficiency. However, the terms *input* and *output* are not further specified and need to be defined individually.

The Green Software Foundation (GSF) [55] is a non-profit foundation that aims to reduce the environmental impact of software. The GSF provides standards, tooling and best practices for green software development. The involved people are working on a standard for green software in a GitHub

repository; they have published a "Software Carbon Intensity (SCI) Specification" [56]. The specification has been released, but it appears to be not yet finalized and not yet a generally accepted standard. However, they mention a software sustainability action for energy efficiency, which includes all actions that make software use less electricity to perform the same functionality [56]. This sustainability action, supports the ISO/IEC 30134:2016 standard by considering energy consumption as an input and functionality as an output. We can use this specification to further elaborate on functionality and energy consumption.

Capra et al. state that energy efficiency is not clearly defined [57] and suggest the following definition: "In general, technology is considered efficient when it performs a job with a small amount of extra energy in addition to the theoretical minimum" [52]. The authors adapted this definition to software engineering and stated that software energy efficiency can be measured by describing a task, consisting of functional operations, which the software must perform, and by identifying the theoretical minimum energy required to perform the task [52]. This requires us to define a set of functional operations. We specify a set of functional operations to be the functionality or output in the previously established sustainability action. However, it is still unclear how to define the theoretical minimum energy required to perform a task.

Capra et al. suggest the term *specific energy*, which describes the energy consumption of a system running an application executing a workload compared to the average energy consumption of applications with the same functionality and workload [57]. Figure 16 explains the mathematical definition of specific energy.

The operating definition of specific energy is obtained as follows. Given an application, say i, belonging to a functional area, say A, specific energy SE_i is defined as:

$$SE_i = \frac{Ed_i - \overline{Ed_A}}{\overline{Ed_A}}, \quad (1)$$

where Ed_i is the difference between the power absorbed by the system running application i and the power absorbed by the system in idle, integrated over the time required to complete the workload; $\overline{Ed_A}$ is the mean value of Ed of applications in functional area A.

The lower the *specific energy* required to execute a set of benchmark workloads, the higher the *energy efficiency*. Accordingly, we define the *energy efficiency* of an application i as:

$$EE_i = 1 - SE_{iNORM} \quad (2)$$

where SE_{iNORM} is the value of SE_i normalized to values between 0 and 1 over the sample of applications considered within the same functional area.

Figure 16. The definition of specific energy [57]

It is worth noting that specific energy is normalized and therefore comparable across functional areas [57]. The authors conclude that an "application is considered more energy efficient than another application if it responds to the same request with lower energy consumption on the same hardware" [57]. So far this section established a mathematical definition for specific energy, which can be mapped to the input according to the ISO/IEC 30134:2016 standard. Additionally, it defined a set of functional operations, which can be mapped to the output respectively. However, this approach considers the energy consumption of the entire system, not just the software.

The GGS blog [11] covers topics related to software sustainability. The post "Software Efficiency and Energy Consumption" [58] refers to the definition of *efficiency* from the Cambridge Dictionary: "the relationship between the amount of energy put into a machine or process, and the amount of useful work that it

produces:" [59]. The blog author clarifies that the term *useful work* is not standardized in software engineering. Guldner et al. try to resolve the lack of standardization in the field of green software engineering [12]. They establish a list of relevant metrics for energy efficiency and a glossary [60], which is among other resources based on a framework for energy efficiency testing [62]. Figure 17 shows two relevant metrics to measure energy efficiency, the definition of *useful work* and *energy efficiency factor*.

Useful work	varied	usage scenario	Necessary when calculating energy efficiency (see below), performance metrics (regression error, test accuracy, F1-score, IoU, etc.) can indicate useful work done of ML models
Energy efficiency factor	$\frac{\text{item}}{J}$	Useful work and energy	Especially useful for comparisons, needs additional computation and possibly the recording of additional metrics

Figure 17. Relevant metrics for energy efficiency [12]

Useful work is mentioned as a varying unit, which needs to be defined individually for each software product or use case [12]. This leaves some room for interpretation when defining actual metrics. Eventually, useful work flows into the calculation of the energy efficiency factor as shown in Figure 18.

$$EnergyEfficiency = \frac{UsefulWorkDone}{UsedEnergy}.$$

Figure 18. A mathematical definition of useful work [63]

The energy efficiency factor is defined as "the quotient of the number of processed items (see useful work) and the energy consumed (by the SuT, CPU, GPU, etc.) in the process" [60]. The acronym SuT stands for *System under Test*. When an application has an increased energy efficiency factor, it uses less energy to process the same amount of useful work. This definition aligns with the previously established definitions of energy efficiency. The authors suggest to use meaningful measurement units, such as joules for short energy consumption measurements or when calculating the energy efficiency factor, and kWh for longer, more resource-intensive measurements.

The term useful work is not defined in sufficient detail and its meaning depends on the domain context it is used in. This thesis suggests to leverage the INVEST mnemonic [13] to work towards a definition of useful work. The INVEST acronym represents a set of criteria to assess the quality of a user story and to help breaking down large work packages into smaller, more manageable tasks. INVEST stands for:
Independent: A story is independent and should not overlap with other stories. Stories should be implementable in any order.
Negotiable: A story is not a contract, it is a placeholder for a conversation. The story should contain the essence and motivate the team to discuss the details.
Valuable: A good story is valuable to the customer. Each story should contain a vertical slice of functionality, which increments the product and provides value.
Estimable: Each story should be estimable, which requires the team to understand the story, and allows them to estimate overall effort. This criterion is affected by the size of the story, the complexity of the task, and the teams experience.
Small: Smaller stories tend to be easier to understand, easier to estimate, and easier to implement.
Testable: Each story should be tested to ensure it meets the acceptance criteria.

The INVEST mnemonic is applicable to large work packages such as epics as well as too large user stories. Table 3 shows examples of PetClinic and LakesideMutual Epics that are too large and how they can be refined according to the INVEST criteria.

Table 3. An example of refined user stories according to INVEST

Application	Epic	Refined User Story
PetClinic	I as a pet owner want to manage my pets, so that they are taken care of.	I as a pet owner want to create a visit appointment for my pet at the pet clinic, so that a veterinarian treats my pet, and I am informed about the treatment.
LakesideMutual	I as a customer of LakesideMutual want to manage my insurance policies, so that I can keep track of my insurance coverage.	I as a customer of LakesideMutual want to request a new insurance quote by providing my personal information and insurance requirements, so that I can receive a tailored insurance offer.

We understand these criteria as guidelines and not as strict rules, therefore we adapt these guidelines to our needs. We utilize parts of the INVEST acronym to define useful work in the context of software engineering and energy consumption measurements. Table 4 presents a definition of useful work according to the INVEST acronym.

Table 4. A definition of useful work according to the INVEST acronym

Criterion	Description	PetClinic Example	LakesideMutual Example
Independent	Useful work should consider a set of functional operations that can be executed independently. In some more complex scenarios this might even be an entire workflow.	The creation of a visit appointment can be executed independently of other operations given that the pet and the clinic are known.	The insurance quote request can be executed independently of other operations.
Valuable	Useful work should be valuable to the users and consider entire vertical slices, such as a user-facing workflow. A workflow might consist of different strategies according to the Gang of Four (GoF) Strategy design pattern [64].	The visit appointment is valuable to the pet owner, as it ensures that their pet receives medical treatment.	The process of requesting and receiving an insurance offer is valuable to the customer as it allows them to compare different insurance options.

Criterion	Description	PetClinic Example	LakesideMutual Example
Small	Useful work should be as small or slim as possible. This allows for better understanding of the functionality, easier testing, and easier localization of potential issues.	The vet visit scheduling process is small, as it only requires the pet and clinic information. Different outcomes, such as successful scheduling, cancellation, or error cases, should be handled separately.	The insurance request process is small. Submitted and cancelled requests, as well as error cases should be handled separately.
Testable	Useful work can be derived from existing test cases as important functionality should be covered by tests.	The scheduling process can be tested by creating unit tests, integration tests, and user acceptance tests to ensure that the appointment is created correctly and that the pet owner is informed about the treatment.	The process can be tested by creating unit tests, integration tests, and user acceptance tests to ensure that the request is processed correctly.

The two criteria *Negotiable* and *Estimable* do not seem to fit into the context of this thesis and are therefore omitted. We propose to define useful work according to the use cases and requirements of the software product, under the assumption that they are documented. These usually contain the essential functionality that the software product must provide. In case there are no documents available, we suggest to analyze the source code, especially the test cases, and conduct interviews with experienced personnel to identify the essential functionality. This practical approach addresses the gap in defining useful work.

Figure 19 shows an updated mind map including the measurable aspects and metrics of the energy and resource efficiency software quality attributes.

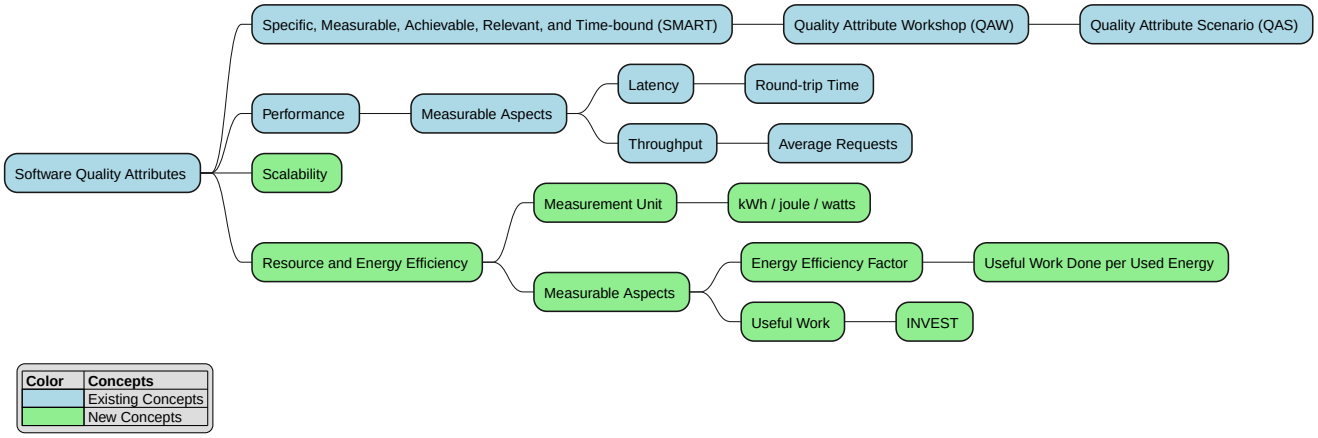

Figure 19. A mind map illustrating the energy and resource efficiency software quality attribute

The established metrics are used in the experiments to measure the energy efficiency of the two enterprise applications.

2.2.3. Correlation Between Performance and Energy Efficiency

The research revealed somewhat contradicting statements regarding the correlation between performance and resource and energy efficiency. This subsection aims to clarify this correlation.

Capra et al. suggest that the quality attributes energy efficiency and performance are not necessarily correlated, even worse, they are conflicting with each other [57]. As opposed to Brunnert, who states that "efficiency has always been at the core of software performance engineering research" [7]. The author mentions that while it is common for servers or data centers to derive their efficiency from power consumption and CO2 emissions, there is no established metric for software efficiency.

Naumann et al. state that energy efficiency and performance not necessarily correlate [65]. While they can closely relate to each other for a software running on a single computer, they can differ significantly for distributed systems. Additionally, the authors state that there is no tool available to measure energy efficiency, they recommend to use a performance-based approach to estimate energy efficiency.

Q42 cites a Wikipedia article to define *resource efficiency* and refer to their previously established performance definition [37] as a special case of resource efficiency. "Resource efficiency is the maximising of the supply of money, materials, staff, and other assets that can be drawn on by a person or organization in order to function effectively, with minimum wasted (natural) resource expenses" [66]. This quality attribute can be seen in the context of sustainability and environmental impact.

Q42 cites a ChatGPT prompt to define *energy efficiency*. "In the context of software engineering, “energy efficiency” refers to the ability of a software system to optimize its energy consumption while performing its intended tasks effectively" [67]. Energy efficiency can be interpreted as a sub-category of resource efficiency, as it specifically focuses on energy as an asset.

Q42 defines *efficiency* according to the Merriam-Webster dictionary: "capable of producing desired results with little or no waste (as of time or materials)" [68]. They provide similar examples for efficiency requirements as for performance requirements [69] [70]. The terms *producing desired results* and *no waste* are abstract; Table 5 further specifies them with examples and fictitious values.

Table 5. Specific definitions for efficiency terms (own presentment)

Term	Definition	Metric	Example
Producing desired results	Performing an intended task or a set of operations.	Useful work according to INVEST	Send an insurance quote request to a server, process the request, and return a response.

Term	Definition	Metric	Example
No waste	Performing effectively or performant while optimizing energy consumption.	Latency	The round-trip time of the request is below 500ms.
		Throughput	The system can process 10 requests per second.
		Energy efficiency in requests per kWh	The system consumes 12kWh for 10 requests per second with 1 node. The efficiency factor is 10 requests per second / 12kWh = 0.833 requests per kWh.

It is uncertain to what extent performance and energy efficiency correlate with each other. ChatGPT states that they are closely related, but sometimes conflicting, depending on the context. Apparently, optimized algorithms and efficient hardware can lead to an increase in performance and energy efficiency. They conflict when it comes to aggressive performance optimizations, such as overclocking CPUs, or when balancing execution speeds versus power savings. The additional annex document provides the prompt and the full response.

Based on the literature review, Table 6 formulates three hypotheses for the correlation between performance and resource and energy efficiency.

Table 6. Hypotheses for the correlation between performance and resource and energy efficiency

Hypothesis	Description	Example
H1 — No correlation	Actions that affect performance do not affect resource and energy efficiency.	When we have a distributed enterprise application with high redundancy or backups, and we scale up the primary system, our action does not affect the secondary system, even though they belong to the same distributed enterprise application.
H2 — Inverse correlation	Actions that increase performance decrease resource and energy efficiency, and vice versa.	When we scale up a system to increase performance, the additional hardware resources decrease resource and energy efficiency. When we activate power-saving modes to increase energy efficiency, the fewer hardware resources decrease performance.
H3 — Strong correlation	Actions that increase performance also increase resource and energy efficiency, and vice versa.	When we improve the codebase to utilize suitable data structures and algorithms, the resource and energy efficiency increases due to fewer resources needed. Performing a few large requests instead of many small requests increases performance and energy efficiency due to fewer overhead.

The subsequent experiments aim to test these hypotheses by analyzing and evaluating the results.

Figure 15 shows an updated mind map including the potential correlation.

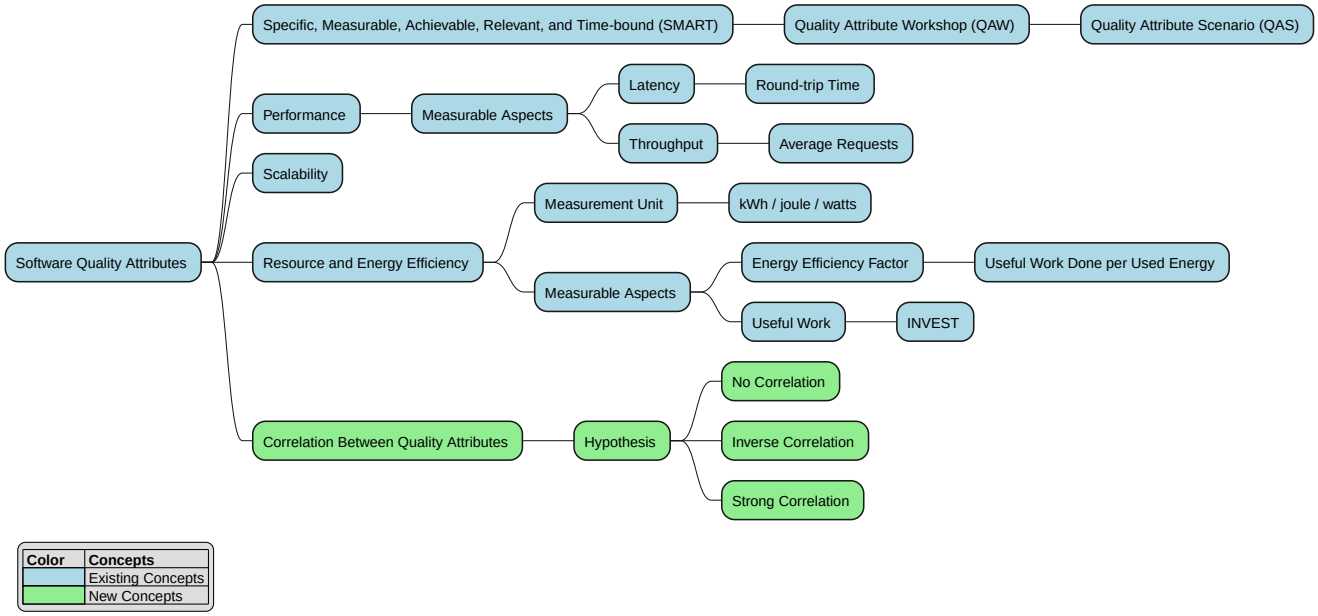

Figure 20. A mind map illustrating the correlation between performance and resource and energy efficiency

This section established measurable aspects and metrics for both quality attributes. It identified a potential correlation and proposed three hypotheses for further investigation. Section 2.3 summarizes the key findings of this chapter.

2.3. Summary and Outlook

This chapter defined enterprise applications as complex software systems characterized by long-running business processes and high data consistency requirements. It introduced two representative applications as case studies for the experiments in this thesis: *PetClinic* and *LakesideMutual*. The PetClinic is a monolithic Spring Boot application that provides HTTP endpoints for managing pet clinic data. LakesideMutual is a distributed, service-oriented Spring Boot application including complex domain models and business processes to manage insurance contracts. The experiments conduct similar measurements on both applications, allowing a comparative analysis of the results.

This chapter defined the two software quality attributes *performance* and *resource and energy efficiency*. Denaro et al. refined the definition of performance in the context of distributed software systems and identified three measurable quality attributes: latency, throughput, and scalability [6]. This thesis measures latency with the metric *round-trip time* and throughput with the metric *average requests*. Scalability should be treated as a separate quality attribute on the same level of abstraction as performance.

The literature does not provide a clear definition on resource and energy efficiency in the context of software engineering. A GGS blog post on software efficiency and energy consumption [58] refers to a definition of efficiency from the Cambridge Dictionary [59] and mentions the interesting term *useful work*. Guldner et al. introduce a definition for useful work and state that it has a varying unit of measurement [12]. We propose to leverage the INVEST mnemonic [13] to derive useful work, serving as a slightly structured approach and a small improvement to the varying unit of measurement.

The literature review yielded contradicting statements on the correlation between performance and resource and energy efficiency. This thesis formulates three hypotheses to investigate this correlation in the context of distributed software systems. It applies a deductive research approach, conducting measurements on different applications and test environments.

Chapter 3 builds on the foundational knowledge of this chapter. It introduces measurement methods and tools, describes the computer system architecture and tool interaction, and specifies the configurations used in the experiments.

3. Measurement Techniques and Experiment Design

This chapter builds on the enterprise applications and software quality attributes defined in Chapter 2. It introduces key measurement methods and tools, describes the hardware and software interaction, and explains measurement challenges. It then provides a detailed description of the measurement setup and configuration for the two test environments. These environments provide the foundation for the experiments and discussions in the subsequent chapters.

3.1. Measurement Methods and Tools

This section focuses on methods and tools to measure the established software quality attributes. It selects two specific tools, which are used in the experiments to measure the performance and energy consumption of enterprise applications.

3.1.1. Performance Measurements

There are multiple performance testing methods and tools available to measure the performance of software systems. Which method or tool is appropriate depends on the system under test and the specific requirements.

A known performance testing method is *microbenchmarking*, which is the process of measuring the performance of code units. *Java Microbenchmarking Harness (JMH)* supports microbenchmarking for Java applications [50] [71]. This tool allows to specify warm-up iterations, which resolves Just-In-Time (JIT) compilation issues to a certain extent. Listing 1 shows an example of a JMH test execution by the Growing Green Software (GGS) blog [50].

Listing 1. Example of a JMH test execution [50]

```
java -jar target/java-collection-impls-benchmark.jar
# JMH version: 1.37
...
Benchmark                                (collectionSize)  Mode  Cnt   Score   Error  Units
CollectionAdd.addToJavaArrayList                   100000  avgt    5   0.298 ± 0.004  ms/op
CollectionAdd.addToJavaArrayList                   200000  avgt    5   0.606 ± 0.004  ms/op
CollectionAdd.addToJavaArrayList                   500000  avgt    5   4.360 ± 0.151  ms/op
CollectionAdd.addToJavaArrayList                  1000000  avgt    5  15.290 ± 1.407  ms/op
CollectionAdd.addToJavaLinkedList                  100000  avgt    5   0.342 ± 0.009  ms/op
CollectionAdd.addToJavaLinkedList                  200000  avgt    5   0.689 ± 0.015  ms/op
CollectionAdd.addToJavaLinkedList                  500000  avgt    5   1.712 ± 0.068  ms/op
CollectionAdd.addToJavaLinkedList                 1000000  avgt    5   3.533 ± 0.282  ms/op
```

Laaber et al. state that JMH or microbenchmarking in general takes a considerable amount of time to execute [72]. The authors suggest an approach to reduce execution times while maintaining a high level of accuracy. They utilize dynamic configurations to stop the execution once the results are stable. The modified version of JMH is available on GitHub [73]. JMH misconfigurations can affect the actual performance measurements; common pitfalls should be avoided when using JMH [74].

Performance testing evaluates a software systems robustness and includes load testing and stress testing [49]. Load testing is an aspect of performance testing, which simulates expected load and user traffic on a software system. Load testing can be further divided into volume testing (extensive data loads), peak load testing (extensive user activity) and endurance testing (extended periods of load). Stress testing, on the other hand, evaluates the system its behaviour under extreme conditions and tries to identify its limits.

Apache JMeter is a tool used for performance testing in terms of load and stress testing [75] [76]. It simulates user behaviour and generates load on a system under test by accessing its interface, such as an HTTP API. The tool can be used for volume, peak load, endurance, or stress testing depending on the configuration of the test plan. JMeter allows a wide range of configuration options, making it a versatile tool for performance testing. Figure 21 shows an example of a JMeter test execution for reference.

Label ↑	# Samples	Average	Throughput	Error %
Accept Insureance Quotes HTTP Request	1	24	41.7/sec	0.00%
Auth HTTP Request	1	343	2.9/sec	0.00%
Create Insureance Quote HTTP Request	1	62	16.1/sec	0.00%
Get All Insureance Quotes HTTP Request	4	12	12.0/sec	0.00%
Get All Policies HTTP Request	3	51	9.1/sec	0.00%
Get Customer HTTP Request	2	160	3.2/sec	0.00%
Get Insureance Quote HTTP Request	4	8	29.2/sec	0.00%
Get User HTTP Request	2	13	3.2/sec	0.00%
Update Insureance Quotes HTTP Request	1	36	27.8/sec	0.00%
TOTAL	19	55	18.1/sec	0.00%

Figure 21. Example of a JMeter test execution [9]

Huerta-Guevara et al. argue that JMeter runs static pre-configured workloads, which require sufficient knowledge of the system under test in order to create appropriate test plans [77]. They propose *DYNAMOJM*, a tool built on top of JMeter, which enables the creation of dynamic workloads for performance testing. It appears that the plugin is not yet publicly available and can therefore not be considered for this thesis. However, this might be an interesting tool in the future, as it could help to create more realistic test plans and workloads. Other viable tools for performance testing may include *Gatling* [78] and *Locust* [79].

3.1.2. Resource and Energy Efficiency Measurements

Guldner et al. conclude that there is no existing, generally accepted, measurement model for resource and energy consumption for software [12]. Together, they developed the *Green Software Measurement Model* (GSMM). The GSMM consists of measurement models, setups, and methods from multiple research groups. However, according to the authors, the GSMM approach has limitations when it comes to complex architectures or distributed systems. Brunnert states that the GSMM approach does not consider interrelationships of software components [7]. The article proposes "the use of resource demand measurements at the level of individual components and transactions as a basis for measuring how green a software is" [7]. Resource demands are the consumption of CPU, memory, storage, or network.

Jay et al. conducted experiments with multiple software-based power meters and concluded that the results correlate [80]. Deviations between hardware-based and software-based power meters are significant and not constant. Apart from the inconsistent results, it remains unclear whether hardware-based tools are suitable for measuring the energy consumption of distributed systems.

Castor [81] and De Souza [82] compare hardware-based power meters with software-based power meters in their respective articles. They conclude that hardware-based power meters do not affect the system under test, but only measure the entire system, whereas software-based power meters can measure individual components, but may affect the system under test. Castor defines an ideal approach to measure the energy footprint of an application.

An ideal approach to measure the energy footprint of an application [81]

> An ideal approach is noninvasive, i.e., it does not affect what is being evaluated, accurate, i.e., the values it reports perfectly match what is being observed, and supports a wide range of levels of granularity, from whole system all the way to individual lines of code.
>
> — Castor F.

We require a method or technology that allows us to measure individual components in a distributed system, as opposed to measuring the entire system. *Running Average Power Limit* (RAPL) is a suitable technology [83] [84]. RAPL is a feature of modern Intel processors, which allows for measuring the power consumption of various power domains, such as the CPU or memory. "RAPL readings are highly correlated with plug power, promisingly accurate enough and have negligible performance overhead" [83]. This technology can be used as an alternative to hardware-based power meters when utilizing Intel processors.

JoularJX [85] is a Java-based agent, which utilizes RAPL and provides real-time power data, which allows the analysis of energy consumption over time and the detection of hotspots. The term *real-time* is not further specified, but it appears that the power data is collected at runtime, aggregated, and logged at the end of the execution. JoularJX hooks into the JVM to measure the energy consumption of entire Java applications down to single methods. This technology in combination with performance testing tools can provide a detailed insight into the energy consumption of Java applications.

Other viable options that may be used to measure the energy consumption of applications include:
PinPoint supports various platforms, among them RAPL on Linux, FreeBSD, and macOS [86]. It is a command-line tool that takes different configuration parameters to measure arbitrary processes passed as an argument. The tool then calculates the power consumption and prints the results to the console.
PowerLetrics is available for Linux and provides the energy footprint in real-time for each process based on RAPL [87]. The project is still in an early phase and not available on other operating systems.
SmartWatts is a software-based power meter that uses RAPL to estimate the power consumption of containers, such as Docker containers or Kubernetes pods [88]. The entire tool or at least parts of it are solely available for Linux.
Windows Energy Estimation Engine is a built-in tool on Windows to measure consumed power [89]. It has two major constraints, it requires a device with a battery, and it measures all processes running on the system. Processes can be filtered, but it is not always clear which process belongs to which application.
MacPowerMonitor is available on GitHub for macOS and reads the power consumption through the built-in *Powermetrics* utility [90].

Table 7 lists the main methods and tools discussed in this section.

Table 7. An overview of methods and tools to measure performance and energy consumption

Software Quality Attribute	Test Method	Tool
Performance	Microbenchmarking	Java Microbenchmarking Harness (JMH)
	Load Testing	Apache JMeter
		Gatling
		Locust
Resource and Energy Efficiency	Hardware-based Power Meter	Various models and manufacturers
	Software-based Power Meter	JoularJX
		PinPoint
		PowerLetrics
		SmartWatts
		Windows Energy Estimation Engine
		MacPowerMonitor
	Green Software Measurement Model (GSMM)	Multiple measurement models, setups, and methods combined

This section established methods and tools to measure the performance and energy consumption of enterprise applications. Section 3.2 refers to the selected tools and explains how they interact with the hardware and software of the test environments.

3.2. Architecture of Observed Systems and Tool Deployment

This section focuses on the architecture and layers of the two test environments. It explains their similarities and differences, and it describes how the tools work and interact with the application under test. Understanding the architecture of the computer system helps to set up the test environments and to troubleshoot potential issues.

Figure 22 shows a black box overview of the test environment and its components.

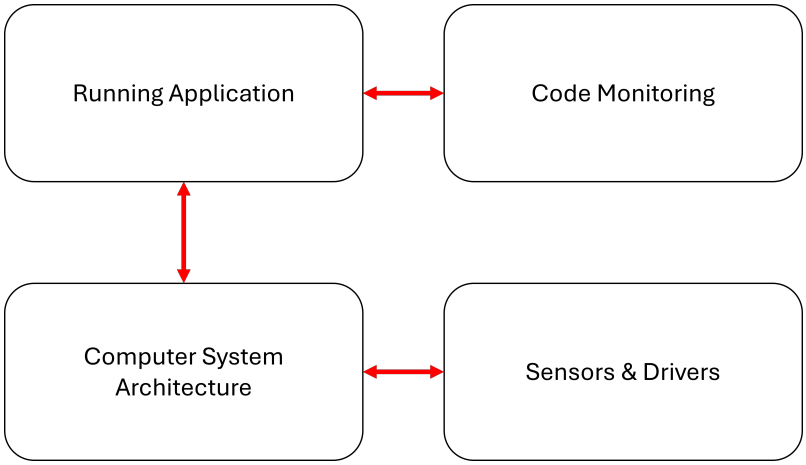

Figure 22. Overview of the test environment and its components

The test environment consists of a computer system that relies on hardware sensors and software drivers to measure the power consumption of the system. This computer system runs the application under test, the load and performance testing tool, and the monitoring tool. The monitoring tool then measures the energy consumption of the application under test. This setup allows for measuring the power consumption of specific processes, as opposed to measuring the entire system. Measuring specific processes isolates the application under test from other services and daemons running on the system.

The two test environments are based on Windows and Linux. Different operating systems result in slightly different system architectures. Figure 23 zooms in on the system architecture and illustrates the differences between Windows and Linux.

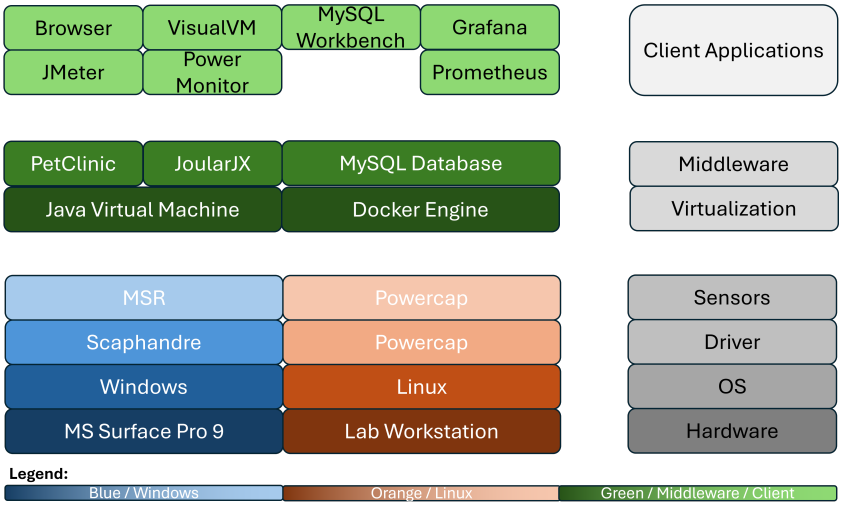

Figure 23. The computer system architecture for Windows and Linux

The lower part of the system architecture coloured in blue refers to Windows, orange refers to Linux. JoularJX on Windows requires an additional driver installation to interact with the RAPL interface [91] [92]. Linux comes with Powercap already pre-installed to read data from the available sensors. Both environments run the same virtualization layer and middleware coloured in green. The upper part illustrates potential client applications interacting with the middleware.

An important client application is the PowerMonitor tool on Windows. JoularJX requires the PowerMonitor tool to read the power consumption of the CPU through the RAPL interface and integrated components

like sensors and model-specific registers (MSR) [93]. Figure 24 from the official repository [94] shows the interaction between the sensors, the Scaphandre driver, and the MSRs.

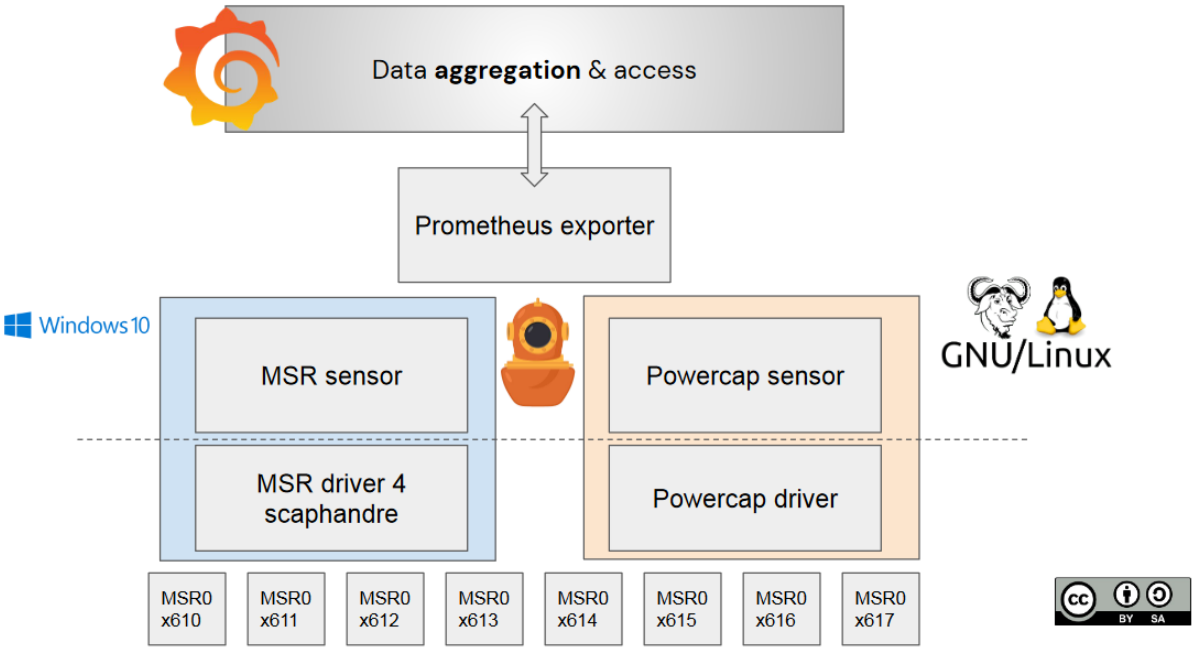

Figure 24. The interaction between the sensors, the Scaphandre driver, and the MSRs [94]

Grafana and Prometheus are used as a means to visualize the data collected by the monitoring tools and to test if the setup works correctly. They are not part of the test environment in this thesis. Instead of pushing the data to Prometheus, the data is consumed by PowerMonitor (on Windows) and JoularJX, which are not visualized in this figure.

JoularJX is a Java agent and runs in a separate thread alongside the monitored application. It calculates the power consumption of the application under test based on the CPU usage of the JVM. Figure 25 shows how a thread in a Java application is monitored by JoularJX. The JoularJX thread itself is not specifically mentioned but is one of the N threads in the JVM.

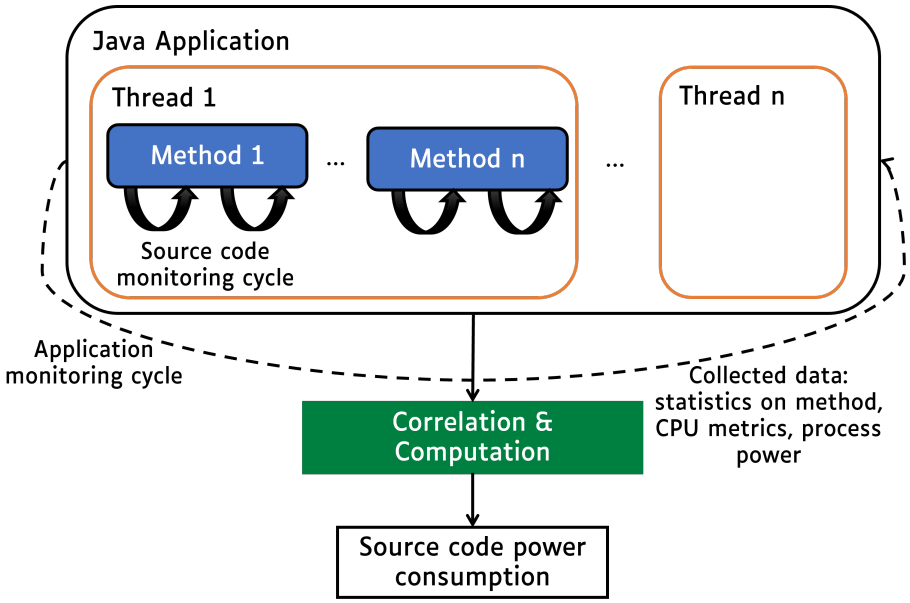

Figure 25. The architecture of JoularJX [95]

JoularJX calculates the power consumption of each thread in the JVM and checks the stacktrace to identify the method being executed. "JoularJX statistically analyzes the ratio of each method observed in the stacktrace, and allocate the power consumption accordingly" [95]. As the application is monitored in cycles, JoularJX can provide evolving power consumption data over time for different execution branches and methods. The data collection over time allows for a detailed analysis of the power consumption of an application. Figure 26 and Figure 27 show the application monitoring cycles and the statistical analysis of methods.

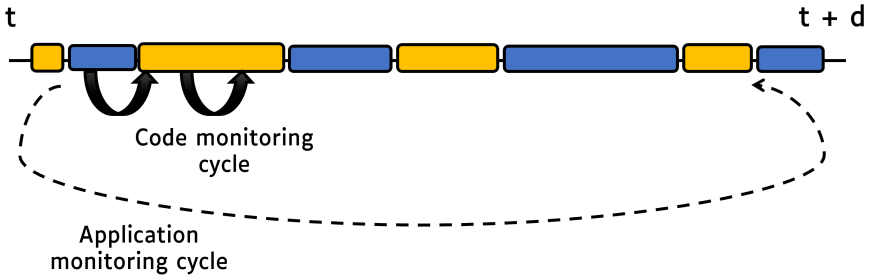

Figure 26. The application monitoring cycles by JoularJX [95]

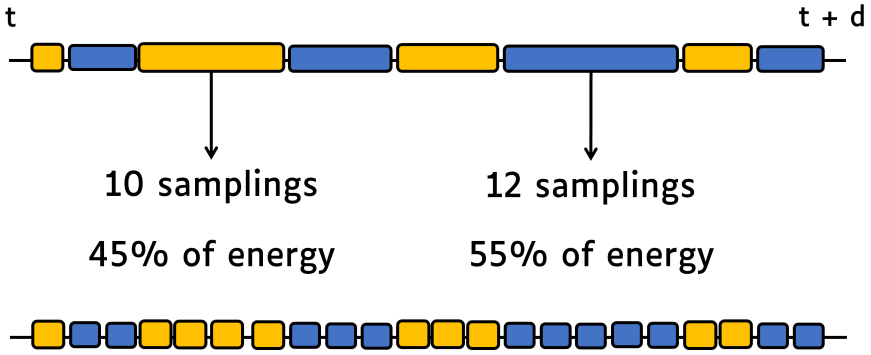

Figure 27. The statistical analysis of methods by JoularJX [95]

This thesis refers to the total energy consumption of the application under test instead of the evolution of power consumption over time. JoularJX calculates the total energy consumption by summing up the power consumption of execution branches. JoularJX allows for method-level filtering to support the analysis of the energy consumption for specific methods. This feature is of great importance for the interpretation of the results, as it allows to identify hotspots in the application.

Figure 28 summarizes the system stack, the interaction with the software drivers and hardware sensors, the running application under test, and the monitoring cycles with JoularJX.

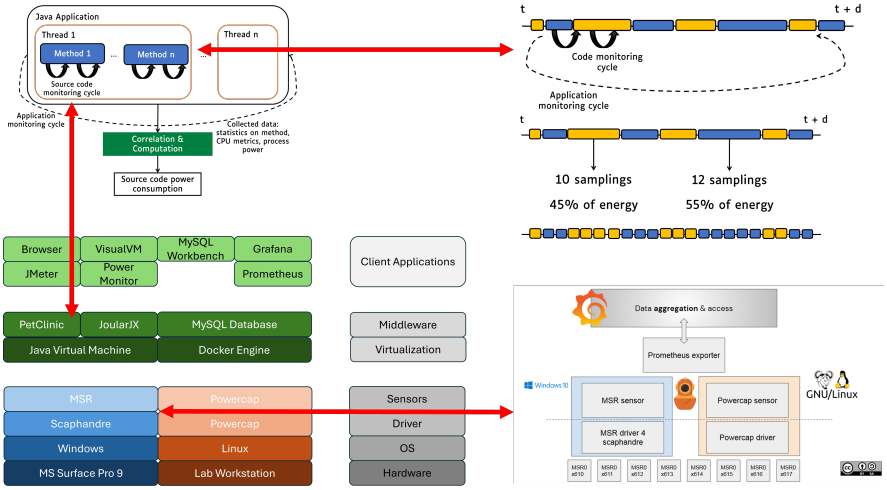

Figure 28. Summary of the system stack and the interaction of its components

This illustration replaces the black box overview in Figure 22 with a white box summary.

Figure 29 shows a mind map covering the test environment, the system stack, and the interaction of the components.

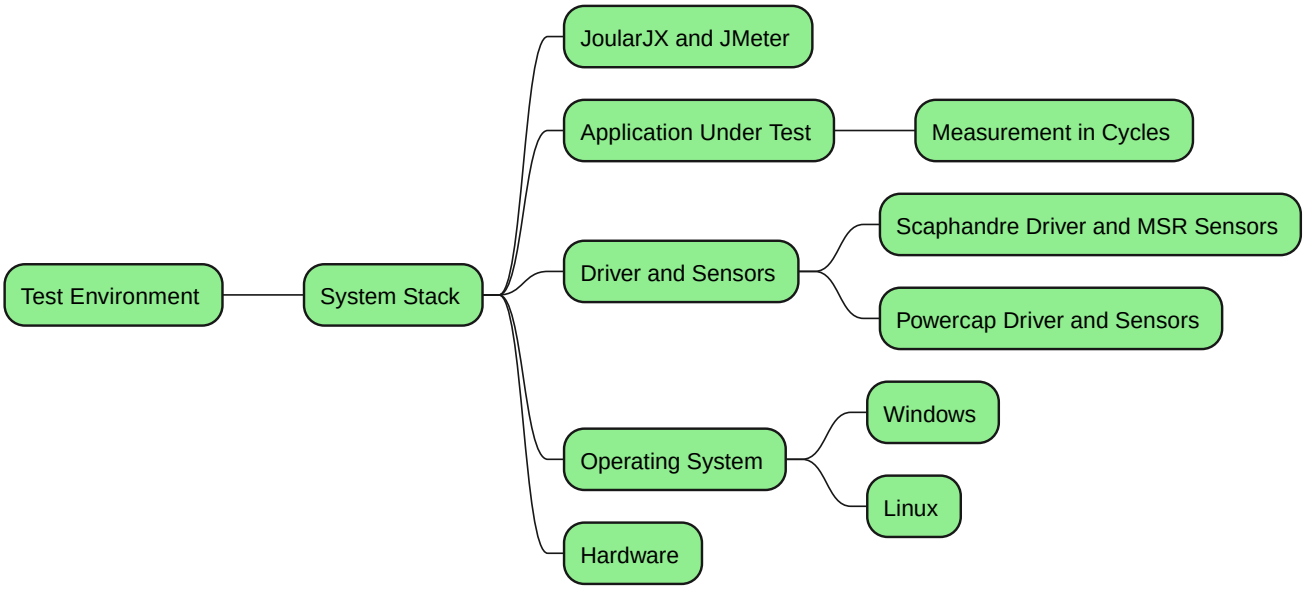

Figure 29. A mind map illustrating the test environment, the system stack, and the interaction of the components

This section provided insights into the test environment and the tools used to measure the power consumption of the application under test. Section 3.3 describes certain challenges and solutions that arise when measuring applications.

3.3. Measurement Challenges

This section focuses on the challenges of measuring Java applications and how to establish a controlled test environment. This thesis aims to apply techniques and best practices from academic and grey literature to achieve reproducible and comparable measurements.

Measuring Java applications can be challenging due to the complexity of the Java ecosystem and the various factors that can influence performance. Java requires the Java Virtual Machine (JVM) as a part of the Java Runtime Environment (JRE) to execute Java bytecode. The JVM uses Just-In-Time (JIT) compilation to optimize the execution of Java bytecode according to the available hardware. This optimization can lead to performance deviations between different test environments. Additionally, the JVM uses the Garbage Collector (GC) to manage memory and clean up unused objects. The GC runs unpredictably leading to performance deviations between different test runs. Measuring Java applications accurately can be challenging due to the impact of JVM, JIT and GC [96] [97].

Such problems require a controlled test environment and applicable best practices to reduce the impact of uncertain factors on the measurements. Guldner et al. state that "there is no consensus on measurement setups, methods, or techniques for data analysis" [12] in the context of software systems. They refer to sustainability labels such as the *Energy Star* label or the *Blue Angel* label. The Energy Star label specifies requirements for energy measurement test setups, such as the input power, ambient temperature, relative humidity, light measuring devices, and power meters [98] [99]. While this specification applies for computers, it is a good starting point for setting up and documenting a controlled test environment.

A controlled test environment includes a controlled test scenario to ensure that the measurements are reproducible and comparable. Figure 30 presented by Guldner et al. visualizes the flow of a test scenario including a setup before the test scenario, the test actions, and a cleanup after the test scenario [12]. The entire run can be iterated multiple times, potentially automated.

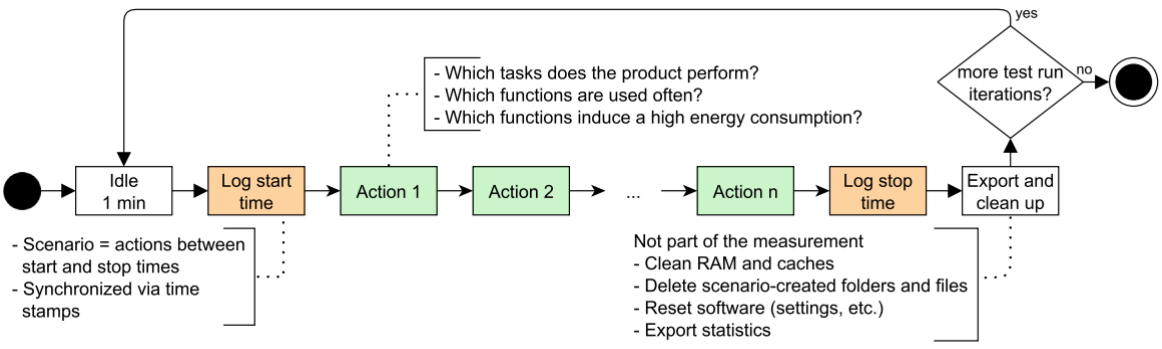

Figure 30. Test setup and cleanup [12]

In order to achieve meaningful results close to a real world scenario, Baumgartner runs tests multiple times, usually seven times, removes the peaks and calculates the average [46] [47]. Castor comes to a similar conclusion as Baumgartner that multiple measurements are important. However, he states that a long test execution does not require multiple test runs as transient factors get irrelevant over time and additional executions are a waste of time [81]. The GGS blog executes measurements multiple times and provides a shell script to run the tests in a loop [51] [100].

Results and findings are easily digestible when presented in a structured or visualized way. Guldner et al. support the use of graphs to display deviations and outliers in their measurements [12]. Castor also includes deviations and outliers in his graphs to provide a more realistic view of the test results [81]. The GGS blog uses diagrams and graphs to illustrate the results [50] [51].

The *Abstracta* blog post on "How to Do Performance Testing for Web Application?" [48] suggests to mirror the testing environment as closely to the productive environment as possible. Details, such as bandwidth limitations, resource utilization, and potential third-party services are important to consider. Equal environments help to uncover real challenges that users might face when interacting with an application.

Replicating a productive environment or setting up a controlled test environment can be challenging. Brunnert and Krcmar agree that test environments are not always available due to expenses and intensive setup procedures [101]. JoularJX enables measurements on a regular system, without the need for a dedicated and hardened test environment, as it measures specific processes instead of the entire system.

Baumgartner uses a dedicated machine with guaranteed resources as a controlled test environment to measure the performance of applications [46] [47]. This thesis utilizes resource constraints on the JVM to ensure equal resource allocation across system boundaries. The JVM can be parameterized by using the `-Xmx`, `-Xms`, and `-XX:ActiveProcessorCount` flags [102].

Baumgartner suggests to explore VM optimizations with the VM Options Explorer [103]. He mentions a few other optimizations in his speech [46] such as a tool named *buildpacks* [104] to create optimized container images. Optimizations would introduce additional variables and complexity, which would make it increasingly difficult to reproduce existing measurements. Optimizing the JVM, containers, or the application itself is out of scope for this thesis. However, such optimizations might be valuable for applications in a production environment.

Table 8 summarizes the techniques and best practices to achieve a controlled test setup and which sources cover them. It states if and how the technique is applicable in the project thesis.

Table 8. Techniques to establish a controlled test environment and their applicability in the project thesis

Technique / Best Practice	**Academic Literature**	**GGS Blog**	**Other Grey Literature**	**Applicability**
Detailed Test Scenario	Yes	Yes	No	**Yes** with an Apache JMeter test plan in a .jmx file containing all steps.
Automated Test Execution with Setup and Cleanup Steps	Yes	Yes	No	**Yes** with a shell script to set up, run, and clean up all tests.
Multiple Measurements	Yes	Yes	Yes	**Yes** with a shell script to run the test scenario multiple times and with JMeter its built-in support for multiple iterations.
Dedicated Test Environment	Yes	No	Yes	**No** because a dedicated machine, mirroring the productive environment, is not applicable in this thesis.

Technique / Best Practice	Academic Literature	GGS Blog	Other Grey Literature	Applicability
Guaranteed Resources	No	No	Yes	**Yes** with resource constraints that are configured on the JVM.
Optimizations	No	No	Yes	**No** because optimizations of the JVM, containers, or the application itself are out of scope for this thesis.
Result Illustration	Yes	Yes	Yes	**Yes** with diagrams, graphs, tables, and screenshots to illustrate the results.

This table solely refers to the sources cited in this thesis. Additional sources likely exist that cover said techniques, which are not included in this table.

This section discussed the challenges of measuring Java applications and how to establish a controlled test environment. Section 3.4 refers to these techniques and covers tooling and test environment configurations for the subsequent experiments.

3.4. Specification, Tooling and Configuration

This section focuses on the environment specifications, tools, and configurations used for the experiments. It aims to enable reproducibility and comparability of the results.

3.4.1. System Specifications

The experiments in this thesis are conducted on two different systems, a Microsoft Surface Pro 9 and a Linux remote server. All experiments refer to the same tools and configurations to ensure meaningful results. Table 9 lists the system specifications of the two test environments.

Table 9. System specification of the Microsoft Surface Pro 9 and the Linux remote server

Specification	Windows	Linux
Device	Microsoft Surface Pro 9	Linux Remote Server
Operating System	Windows 11 Home 24H2	Ubuntu 22.04.5 LTS
Processor	Intel i7-1255U, 2.60 GHz	Intel i9-10940X, 3.30 GHz
Memory	16 GB	128 GB

The two systems mainly differ in terms of available resources, such as CPU and memory. The Microsoft Surface Pro 9 is a mobile device with limited resources, while the Linux remote server has a powerful Intel i9 processor and 128 GB of memory. The connection to the remote server is established via SSH.

This thesis utilizes resource constraints to account for the different system specifications. The initial experiments are conducted both with and without resource constraints on the Java Virtual Machine (JVM) to analyze their impact on the performance and energy consumption of the applications under test. Further experiments are solely conducted with resource constraints to increase the comparability of the

results while reducing the complexity of the setup and the analysis. Listing 2 shows an example command to specify resource constraints on the JVM.

Listing 2. Example command to specify resource constraints on the JVM

```
java -Xmx8g -Xms8g -XX:ActiveProcessorCount=8
```

The `-Xmx8g` parameter configures the maximum heap size to 8GB of memory. The `-Xms8g` parameters configures the minimum heap size respectively. The `-XX:ActiveProcessorCount=8` parameter configures the number of active processors the JVM can use. The PetClinic application is configured with 8GB of memory and 8 active processors. The LakesideMutual application receives the same amount of resources, shared between the four backend services. Therefore, each service is configured with 2GB of memory and 2 active processors.

3.4.2. Tools and Versions

The experiments are conducted with the same tools and versions on both systems to guarantee meaningful and reproducible results. Table 10 lists the tools and versions used in this thesis.

Table 10. Tools and versions used for the experiments

Tool	Version
Java Development Kit	17.0.14-tem
Spring Boot	3.4.3
MySQL Database	8.4.4
JMeter	5.6.3
JoularJX	3.0.1

The Java Development Kit is separately installed on both systems. The Spring Boot version is configured in the Maven pom.xml file of the application. The MySQL database version is configured in the Docker run command or the docker-compose.yml file respectively. JMeter and JoularJX have their own specific configurations and command line parameters.

3.4.3. JMeter

JMeter can be started as a regular Desktop application with a user interface. This user interface allows users to configure the test plan or to analyse test results with tool support. While it is also possible to execute the test plan in GUI mode, Apache recommends running load tests in non-GUI mode. Listing 3 provides an example command to run JMeter.

Listing 3. Example command to run JMeter in non-GUI mode for load testing

```
jmeter -n \
-t /path/to/test-plan.jmx \
-l /path/to/test-result.jtl \
-j /path/to/jmeter.log
```

The `-n` parameter triggers the non-GUI mode, the `-t` parameter specifies the path to the test plan file, the

`-l` parameter specifies the path to the test results file, and the `-j` parameter specifies the path to the log file. The test plan file contains the test configurations and specifies which requests are sent to the application under test. The test results file contains important metrics such as the response time, throughput, and error rate. JMeter can open the results file in a separate listener in the GUI to analyze it. The log file contains information about the test execution, such as the amount of requests sent, the number of errors, and the test duration.

The PetClinic test plan consists of multiple variable configurations, thread groups, requests, and variable extractors. Figure 31 displays the JMeter GUI with the test plan for the PetClinic application including the global variables, which are visible in the center panel.

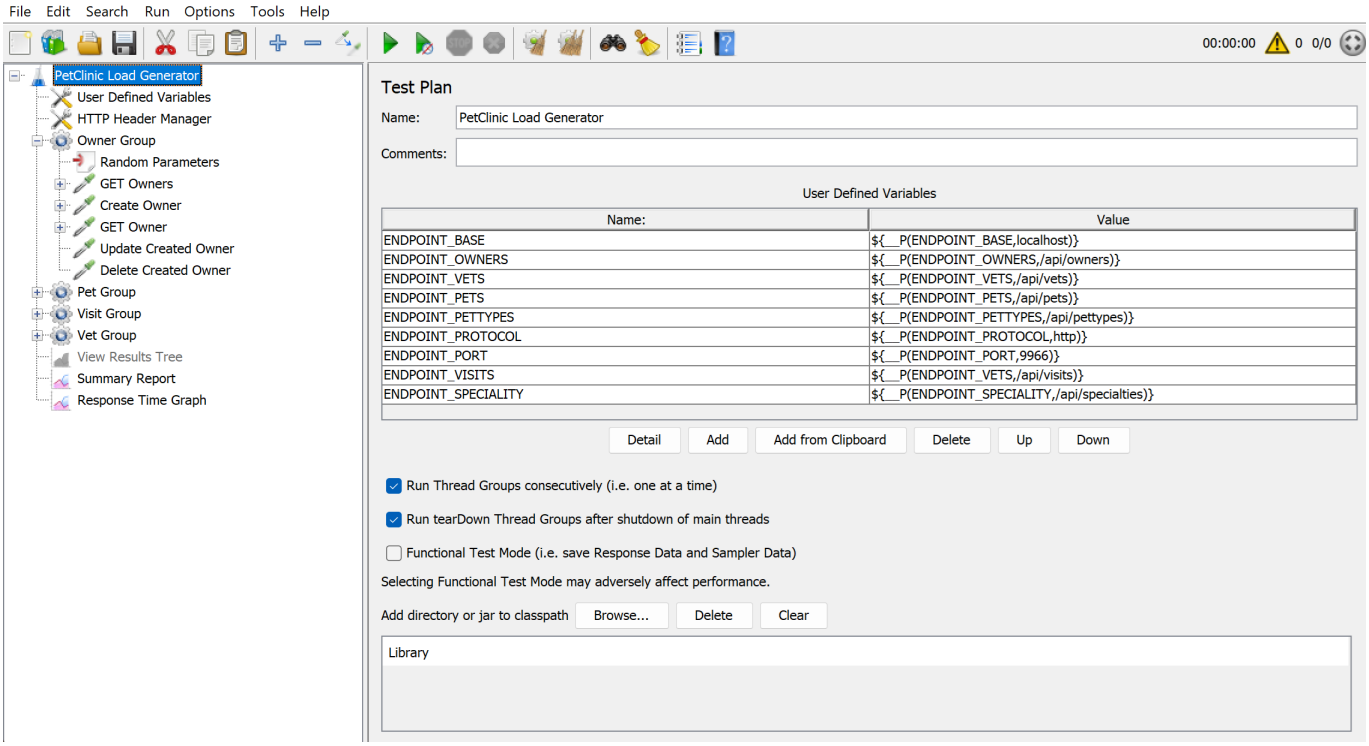

Figure 31. The PetClinic test plan with global variables in JMeter

The global variables configure the protocol, host, port, and context path of the application under test. These variables are used in the thread groups and requests to avoid hardcoding the values in multiple places.

The test plan defines thread groups for owners, pets, visits, and vets in the left panel. Each thread group consists of multiple requests, such as *GET Owners*, or *Create Owner*. Figure 32 shows the *Owner Group* configuration in the center panel.

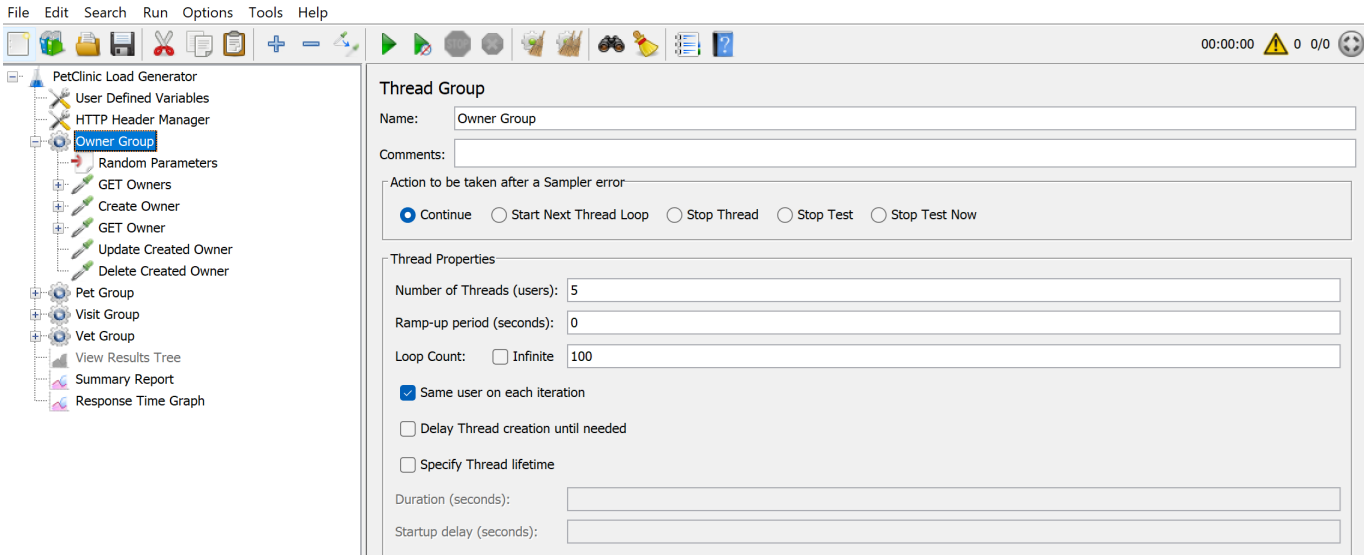

Figure 32. The PetClinic test plan with a thread group for the owner endpoint in JMeter

The configuration in the center panel allows, among other options, to configure the number of threads, the ramp-up period, and the loop count. This configuration defines how many users (threads) are simulated, how long it takes to start all threads (ramp-up period), and how often each thread executes the requests (loop count). All respective requests within the thread group are executed in series once the thread is executed, while multiple users (threads) are executed in parallel.

Each thread group executes requests to fetch all entities, create a new entity, get a specific entity, update an entity, and delete an entity. Figure 33 shows the *GET Owners* request configuration in the center panel.

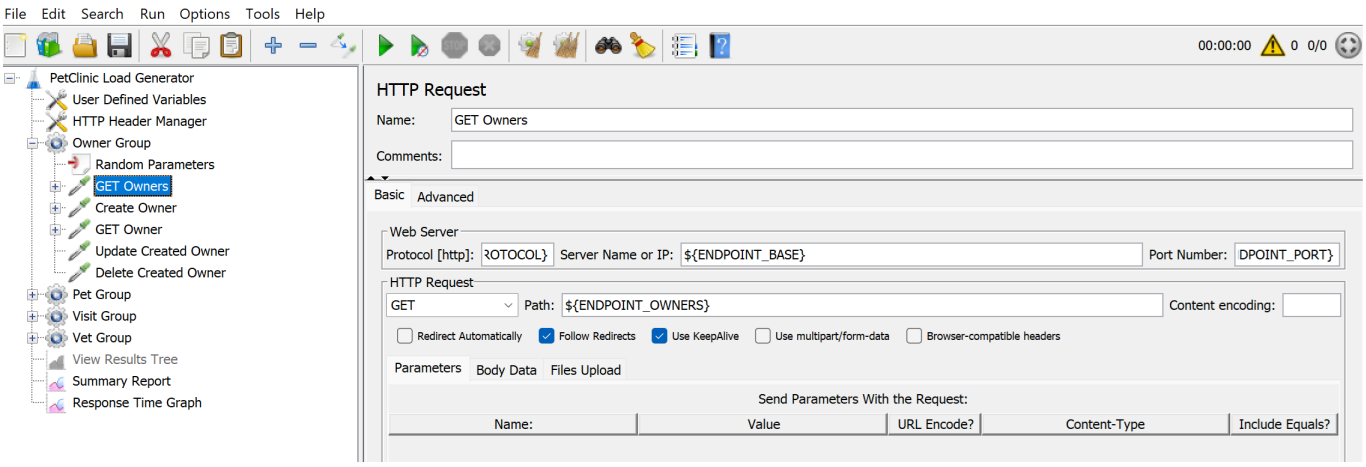

Figure 33. The PetClinic test plan with request configurations to fetch all owners in JMeter

The screenshot shows that the global variables, presented in Figure 31, are used to configure the protocol, host, port and context path. The request method is selected via dropdown, in this case *GET*. Additional parameters or body data can be configured in the respective tabs at the bottom of the center panel.

The test plan for the PetClinic application executes a total of 11'500 requests. These 11'500 requests are split into 5'500 GET requests and 6'000 POST, PUT, and DELETE requests. The LakesideMutual test plan is adapted from the PetClinic test plan and focuses on customer master data and workflow executions. Figure 34 illustrates the LakesideMutual test plan with all thread groups in the left panel.

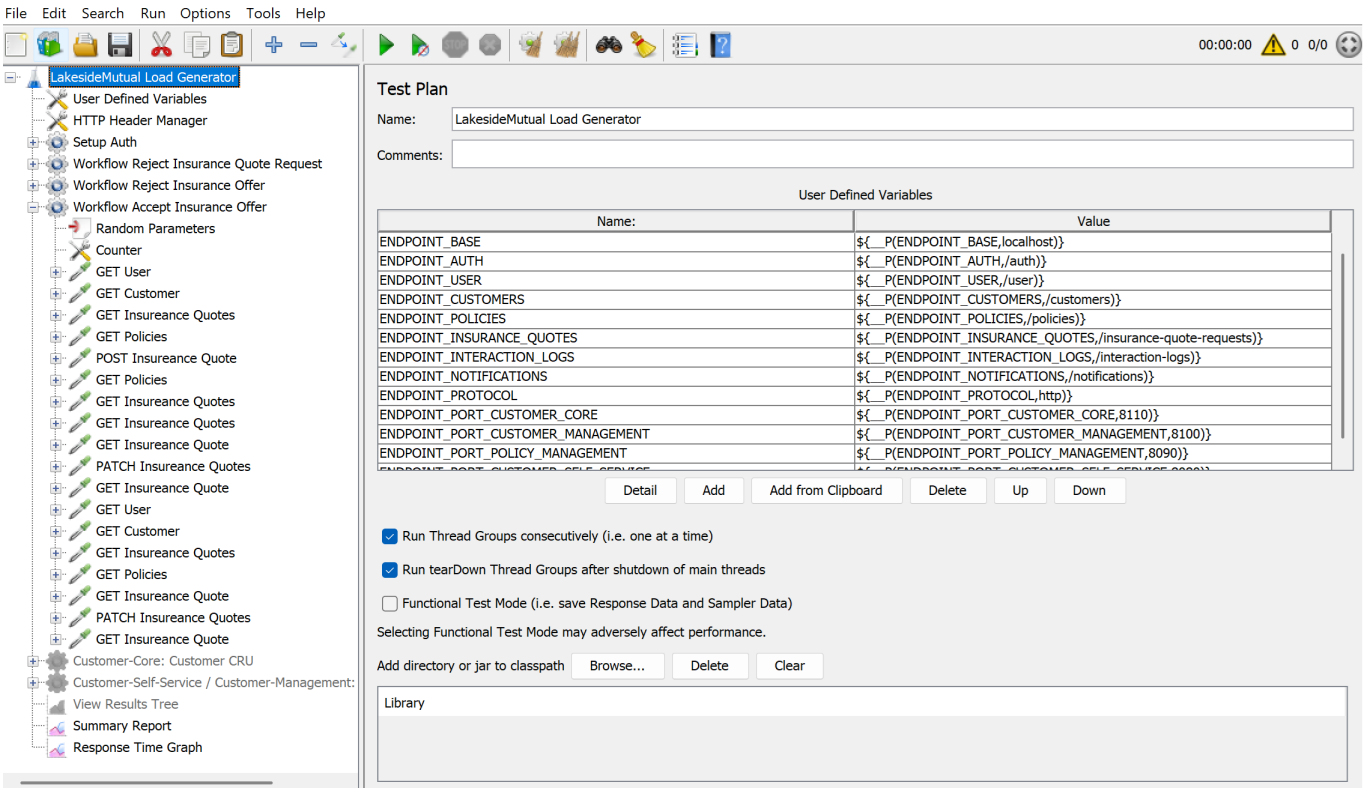

Figure 34. LakesideMutual test plan with thread groups for workflows and customers

The left panel lists all thread groups including the insurance request workflow with three different outcomes. Each outcome is represented by a separate thread group and consists of multiple requests to simulate the workflow. Each thread group starts with a customer requesting an insurance quote, which is processed by a LakesideMutual employee. The first group resembles an employee rejecting the insurance quote request. The second group resembles an employee accepting the insurance quote request and sending an offer to the customer, which is then rejected by the customer. The third group resembles a customer accepting the received insurance offer and effectively creating a new policy. The remaining two thread groups execute create, read, and update requests for customer master data.

JMeter and its load test scenarios are combined with JoularJX to measure the energy consumption of the applications under test.

3.4.4. JoularJX

JoularJX runs as a Java agent attached to the JVM running the application under test. JoularJX requires a separate configuration file to configure the measurement intervals, filter method names, and other parameters. The configuration file is passed to the JVM as a system property. Listing 4 provides an example command to run JoularJX.

Listing 4. Example command to run JoularJX as a Java agent attached to the JVM

```
java -Xmx8g -Xms8g -XX:ActiveProcessorCount=8 \
-javaagent:/path/to/joularjx.jar \
-Djoularjx.config=/path/to/config.properties \
-jar /path/to/application.jar
```

The `-javaagent` parameter specifies the path to the JoularJX jar file. The `-Djoularjx.config` parameter specifies the path to the configuration file. The `-jar` parameter specifies the path to the jar file of the application under test. The config.properties file for the PetClinic application is configured according to the GGS blog measurement setup. Listing 5 shows the config.properties file for the PetClinic application.

Listing 5. JoularJX config.properties file for the PetClinic application

```
filter-method-names
=org.springframework.samples.petclinic.rest.controller.OwnerRestController,org.springframework.sa
mples.petclinic.rest.controller.PetRestController,...
save-runtime-data=false
overwrite-runtime-data=true
logger-level=INFO
track-consumption-evolution=false
evolution-data-path=evolution
hide-agent-consumption=true
enable-call-trees-consumption=false
save-call-trees-runtime-data=false
overwrite-call-trees-runtime-data=true
stack-monitoring-sample-rate=1
application-server=true
powermonitor-path=path//to//spring-petclinic-energy-benchmarking//PowerMonitor.exe
```

The most important configurations are the *filter-method-names*, the *application-server*, and the *powermonitor-path* under Windows. The filter-method-names parameter specifies the classes that JoularJX logs into a separate log file. The application server specifies whether the application under test runs on top of an application server or framework. This is true for Spring Boot applications as they run on top of an embedded Tomcat server. JoularJX requires the PowerMonitor executable under Windows to read the CPU power consumption.

The config.properties file for the LakesideMutual application is adapted accordingly. Listing 6 reveals different filter-method-names for LakesideMutual.

Listing 6. JoularJX config.properties file for the LakesideMutual application

```
filter-method-names
=com.lakesidemutual.customercore.interfaces.CustomerInformationHolder,com.lakesidemutual.customer
core.interfaces.CityReferenceDataHolder,...
```

The list of filter-method-names in both listings are just examples and not exhaustive. Their sole purpose is to illustrate the different package names and classes.

3.4.5. Test Automation Script

The setup is designed to be reproducible and allows for meaningful comparisons between the two applications. The test automation scripts and instructions are available on GitHub. The benchmark setup for the PetClinic application is available in the *spring-petclinic-energy-benchmarking* repository [100]. The benchmark setup for the LakesideMutual application is available in the *LakesideMutual-energy-benchmarking* repository [105].

Setup and cleanup steps are part of the script and ensure that the environment is ready for the test execution and that no leftover data is present. These steps include starting and stopping the database Docker container, loading test data, starting and stopping the application under test, and copying the JoularJX logs to a specific location. On Linux, the test execution is automated with a shell script that performs all setup steps, executes the tests, and cleans up the environment.

On Windows, the test automation is not possible at the time of writing this thesis. The reason is that JoularJX is attached to the JVM as an agent and immediately terminates when the application under test is stopped. The process can not be terminated gracefully enough to allow JoularJX to finish writing the logs. This problem only occurs when the process runs in the background, not when it runs in the foreground. Therefore, the test execution requires manual interaction via the command line. A user needs to start the database Docker container, start the application under test, start the JMeter test plan, wait for the test plan to finish, stop the application under test, copy the JoularJX logs and store them in a file, and stop the database Docker container.

3.4.6. Database Configuration on Windows

The database runs in a Docker container and is initialized with test data from separate sql scripts. All files are located in the spring-petclinic-energy-benchmarking and LakesideMutual-energy-benchmarking repositories respectively.

The PetClinic application uses a MySQL database that is started with Docker. This thesis refers to commands tested on Windows with Git Bash in order to run Linux-like commands. Listing 7 lists the Docker run command to start the MySQL database.

Listing 7. Docker run command to start the MySQL database

```
docker run --name mysql -d --rm -e MYSQL_ROOT_PASSWORD=petclinic -e MYSQL_DATABASE=petclinic -p
3306:3306 mysql:8
```

The container is started with the name *mysql*, in detached mode, and is removed automatically once stopped. The docker container spins up an empty MySQL database with the name *petclinic* and the root password *petclinic*. Listing 8 shows the command to initialize the database with test data.

Listing 8. Docker command to initialize the MySQL database with test data

```
docker exec -i mysql mysql -u root -ppetclinic petclinic < ./benchmark-ddl-and-data.sql
```

The *docker exec* command executes the *mysql* command inside the running container named *mysql*. The input redirection operator < specifies the file to be executed inside the container.

The LakesideMutual application uses multiple MySQL databases, each with its own schema, running in one Docker container. The Docker container and the volume are created and started with Docker Compose. Listing 9 shows the Docker Compose command to start the MySQL databases.

Listing 9. Docker Compose command to start the MySQL databases

```
docker-compose -f docker-compose.yml up -d
```

The Docker Compose file is located in the LakesideMutual-energy-benchmarking repository and configures the database and the database initialization. The sql scripts are copied into a volume that is mounted to the container. MySQL automatically executes the scripts when the container is started.

3.4.7. Application Configuration

All experiments refer to the same application configurations with slight changes to the database access. Both applications under test, the PetClinic and LakesideMutual, are configured with application properties files and external parameters.

All applications and configurations are available on GitHub. The PetClinic application is available in the *spring-petclinic-rest* repository in the *spring-petclinic* community [25]. The LakesideMutual application is available in the *LakesideMutual* repository in the *Microservice-API-Patterns* community [31].

When it comes to the PetClinic experiments, all test executions are performed on the main branch of the spring-petclinic-rest repository. The application is started with the available *mysql* and *jpa* profiles. The mysql profile configures a mysql database connection, while the jpa profile configures the database access with JPA. The first experiments use the same configuration parameters as the GGS blog to replicate the results. The subsequent experiments adapt the configuration to use Spring Data JPA instead of JPA. This change is necessary because LakesideMutual relies on Spring Data JPA to access the database. This intermediate step allows to compare the results within the same application and across different applications.

Listing 10 presents the command to start the PetClinic application with the correct profiles and datasource properties.

Listing 10. The command to start the PetClinic application with the correct system properties

```
java -Xmx8g -Xms8g -XX:ActiveProcessorCount=8 \
  -javaagent:/path/to/joularjx/target/joularjx-3.0.1.jar \
  -Djoularjx.config=/path/to/spring-petclinic-energy-benchmarking/config.properties \
  -Dspring.sql.init.mode=never \
  -Dspring.profiles.active=mysql,jpa \
  -Dspring.datasource.username=root \
  -Dspring.datasource.password=petclinic \
  -Dspring.threads.virtual.enabled=false \
  -jar /path/to/spring-petclinic-rest/target/*.jar
```

The *-Dspring* parameters configure system properties that add or overwrite existing values in the application properties files. The *-Dspring.sql.init.mode=never* parameter prevents the application from automatically initializing the database, as this is done separately. The *-Dspring.profiles.active=mysql,jpa* parameter activates the mysql and jpa profiles. The *-Dspring.datasource.username* and

-Dspring.datasource.password parameters configure the database connection. The *-Dspring.threads.virtual.enabled=false* parameter disables the use of virtual threads and is configured according to the GGS blog.

When it comes to the LakesideMutual application, the experiments are conducted on the spring-boot-3.4.3-benchmark branch. This branch is based on the main branch and contains the necessary changes to run the benchmarks. The necessary changes include the update to Spring Boot 3.4.3, the configuration of the database connection, updates to the test data files, and a removal of the circuit breaker configuration. Each service of the LakesideMutual application is started separately and requires its own configuration and command to start. Listing 11 starts the customer core service of LakesideMutual with the correct system properties.

Listing 11. The command to start the LakesideMutual customer core service with the correct system properties

```
java -Xmx2g -Xms2g -XX:ActiveProcessorCount=2 \
  -javaagent:/path/to/joularjx/target/joularjx-3.0.1.jar \
  -Djoularjx.config=/path/to/LakesideMutual-energy-benchmarking/joularjx_LakesideMutual-
SOA_config.properties \
  -Dspring.threads.virtual.enabled=false \
  -Dspring.profiles.active=default,test \
  -jar /path/to/LakesideMutual/customer-core/target/*.jar
```

The main difference to the PetClinic application lies in the activated profiles. The LakesideMutual application would generally use the *default* profile if no other profile is specified. By default, the customer core service loads the test data file and initializes the database through repositories. This mechanism affects the energy consumption of the service. To prevent external influences on the measurements, an external sql script loads the test data is into the database on startup. The *test* profile is activated to prevent the customer core service from loading the test data file and initializing the database through repositories.

This section established the system specifications, tools, and configurations used for the experiments. Section 3.5 summarizes the key findings of this chapter.

3.5. Summary and Outlook

This chapter established methods and tools to measure the performance and resource and energy efficiency of enterprise applications. It illustrated the test environments, their system architecture, and the interaction of the components running on the test systems. Furthermore, it provided an overview of challenges faced when preparing the test environments and the test plans. Eventually, it presented the test configurations that are used to conduct the experiments.

When it comes to performance testing, we identify different testing methods like load and stress testing, as well as two tools: JMH and JMeter. We decide to leverage JMeter for our performance testing, as it is a versatile tool, which we already used in a previous thesis to measure the performance of LakesideMutual [9]. Additionally, the same tool is used on the GGS blog to measure the performance of the PetClinic application [51]. Furthermore, JMH can run for a considerable amount of time, which might not be suitable for our project thesis since we are limited in time and hardware resources.

When it comes to measuring resource and energy efficiency, we identify the GSMM approach, which appears to be a good starting point in general, but not suitable for complex architectures or distributed systems. We decide to utilize software-based power meters over hardware-based power meters, because they enable us to measure specific processes on a running system. We choose JoularJX to conduct resource and energy efficiency measurements as it is based on RAPL, which is available on our test systems, and because it supports fine granular measurements down to single methods. JoularJX is used on the GGS blog to measure the energy consumption, which makes it a suitable tool to reproduce the test results.

The selected tools are applied on the two test environments running on Windows and Linux. The two test environments differ in the used hardware resources, sensors and software drivers. Both applications run in the JVM with JoularJX attached to the JVM as an agent. JoularJX then measures the application under test in cycles and calculates the energy consumption based on CPU power statistics.

The test scenarios include setup and cleanup phases, automated with a test script on Linux. Each scenario is executed multiple times to account for deviations and outliers. The JVM is parameterized with resource constraints to ensure equal resource allocation across system boundaries.

We rely on the test plan provided by the GGS blog to reproduce the measurements. This test plan is based on JMeter and contains a set of requests to measure the performance of the PetClinic. The test plan for LakesideMutual is adapted from the existing PetClinic test plan. The test plans are stored in .jmx files and the results are stored in .jtl files, which enables easy sharing with the research community. This structured approach allows others to comprehend and trace our test results.

The configurations and setup procedures are documented in detail, including the tool versions, tool configurations, and application startup procedures. The respective resource files are publicly available in their GitHub repositories. These actions should enable other researchers or practitioners to reproduce our results and to build upon our work.

This chapter established measurement methods, tools, and configurations that provide a foundation for the experiments. Chapter 4 presents the experiment results of the local measurements using the PetClinic and LakesideMutual applications.

4. Measurement Results

This chapter summarizes the experiment results of the local measurements using the PetClinic and LakesideMutual applications. The experiments are conducted on two different operating systems, Windows and Linux, to compare the performance and energy consumption in different environments. The results are compared with the Growing Green Software (GGS) blog results, with each other across the different systems, and with each other between the two enterprise applications. The author of the GGS blog acted as a subject-matter expert for this thesis. He shared his knowledge and stated his personal opinions in personal conversations with us.

The experiments include multiple test scenarios, each scenario changes one test parameter at the time to achieve comparable results. Figure 35 illustrates the experiments, their test scenarios, and how they build on each other.

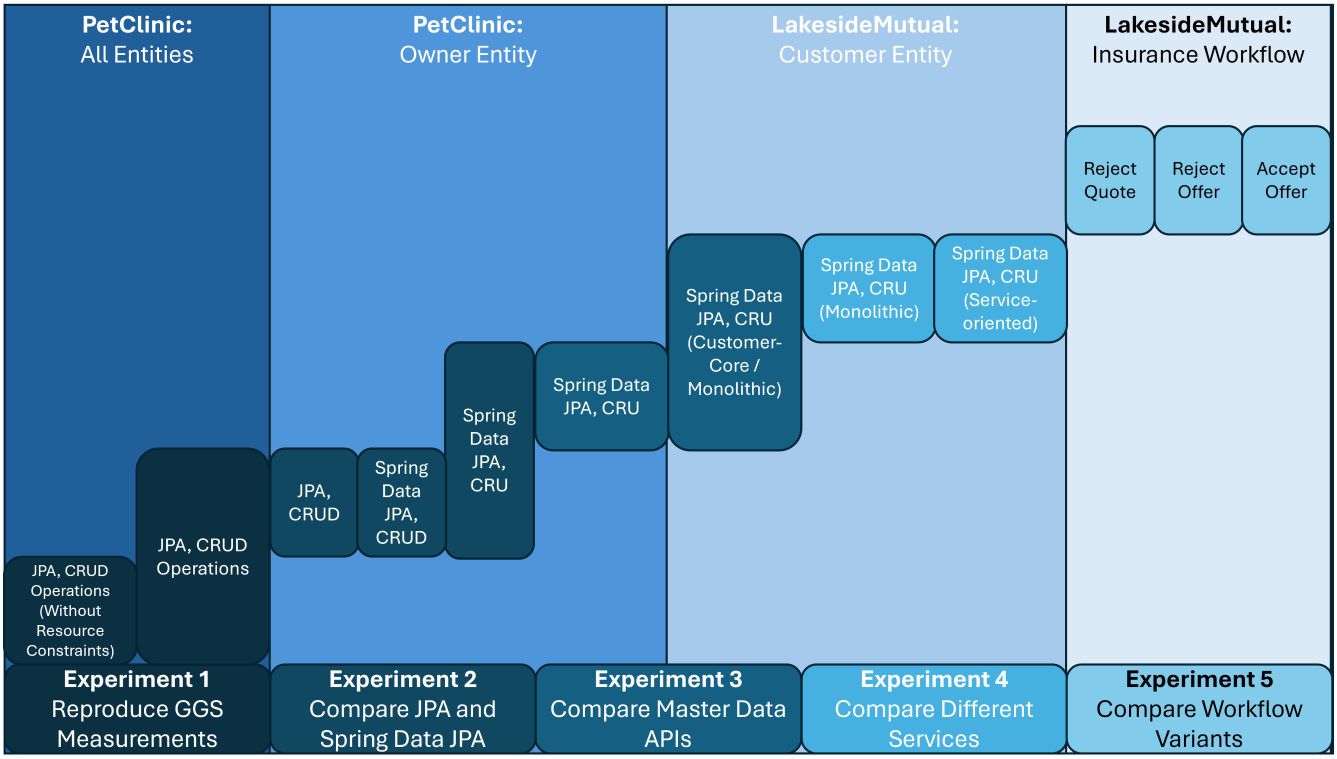

Figure 35. All experiments and their test scenarios illustrated as a staircase

The first experiment utilizes the PetClinic application and executes the existing test plan including all entities. It includes two scenarios, one without resource constraints and one with resource constraints. The second experiment reduces the number of entities to the owner entity. It compares JPA with Spring Data JPA in the first two scenarios, and then compares CRUD versus CRU in scenario two and three. The third experiment utilizes the same Spring Data JPA CRU setup from the second experiment and compares it to the LakesideMutual customer entity. The LakesideMutual operations are only conducted on the monolithic customer-core service to achieve a fair comparison. The fourth experiment compares the same monolithic customer-core results with the service-oriented results. The fifth experiment compares the three LakesideMutual insurance workflow variants with each other. This last experiment is detached from the other experiments and does not compare itself to the previous results.

4.1. PetClinic Experiment: Establish a Baseline

This first experiment is based on the GGS blog and aims to reproduce the measurements with the same tools and configurations. It intends to achieve similar results and confirm the local measurement setup to establish a baseline for this thesis. This experiment covers two scenarios. The first test scenario runs the application without any resource constraints. The second test scenario runs the application with the respective resource constraints parameters described by Listing 2 in Section 3.4. This section visualizes the experiment results and summarizes the findings of the first PetClinic experiment.

4.1.1. Experiment on Windows

The experiment collects data on the performance and the energy efficiency of the PetClinic application running on Windows. The results of interest are the impact of resource constraints and the correlation between performance and energy efficiency. Subsection 2.2.1 states that performance is characterized by latency and throughput. Table 11 summarizes the average latency and throughput reported by JMeter.

Table 11. The latency and throughput of the PetClinic application on Windows

Test Execution	Average Latency	Average Throughput
Without Resource Constraints	681ms	1.7 Requests per second
With Resource Constraints	652ms	6.3 Requests per second

The results indicate that the resource constraints have a positive impact on the performance, as the latency is reduced and the throughput is increased. This observation reflects in the overall processing time of the test execution extracted from the JMeter log file. Figure 36 visualizes the processing time for all requests sent to the PetClinic application.

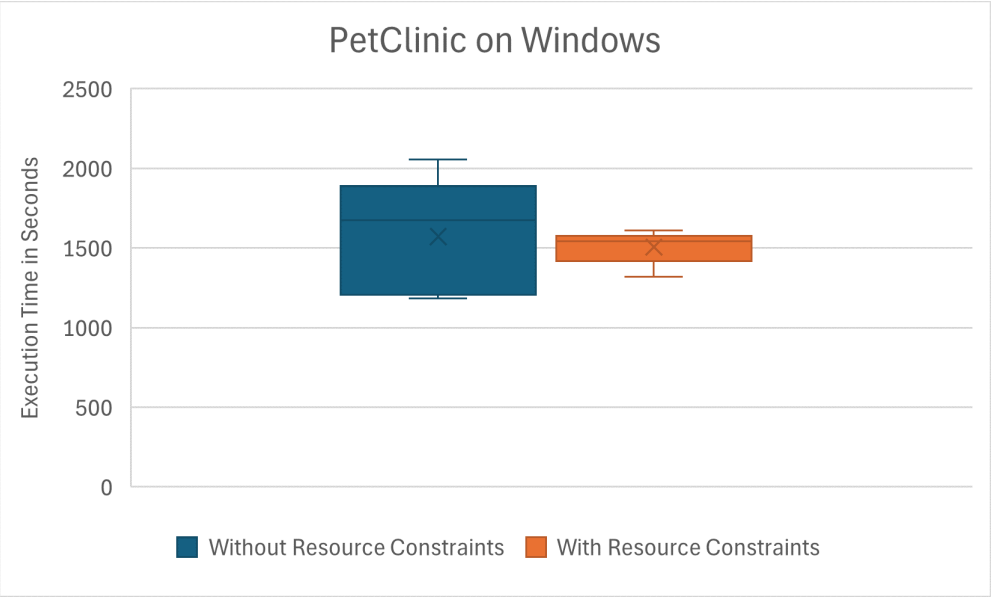

Figure 36. The processing time for all requests sent to the PetClinic application reported by JMeter on Windows

The box plot visualizes the distribution of processing times for all requests in a constrained and a non-constrained test execution. The two whiskers show the minimum and maximum processing times. The box shows the interquartile range between the first and third quartile. The line inside the box represents the median, while the *X* symbol represents the mean.

The vertical axis visualizes the processing time in seconds. The box plot reveals a significant distribution of processing times for the non-constrained test executions compared to the constrained test executions. The median and mean are slightly higher for the non-constrained test execution. This diagram supports the initial observation that the constrained test scenario achieves a better performance.

The performance results and processing times can be compared with the total energy consumption of the test executions. Figure 37 visualizes the energy consumption of the PetClinic application extracted from the JoularJX report files.

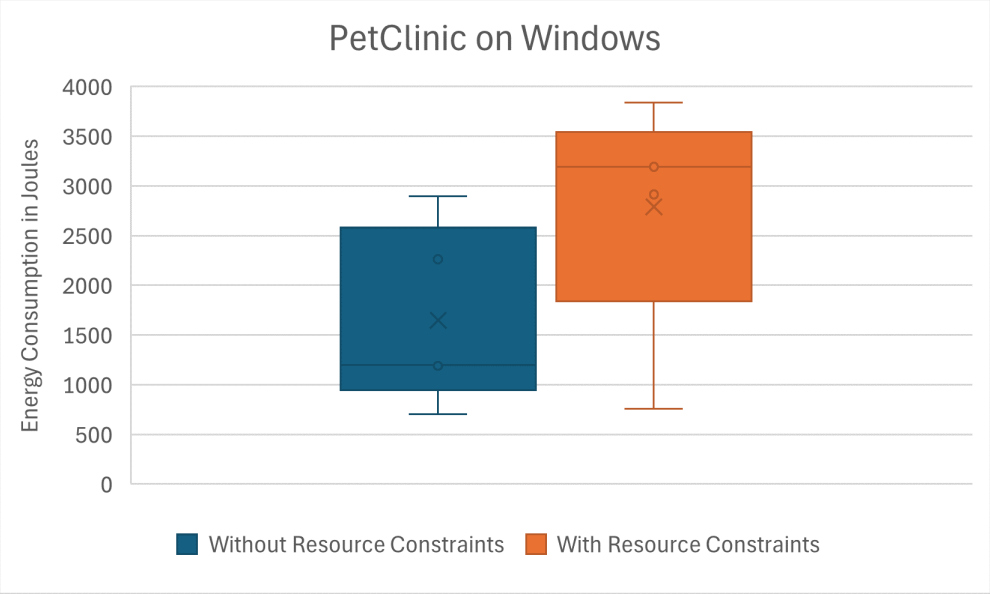

Figure 37. The energy consumption of the PetClinic application measured with JoularJX on Windows

The vertical axis visualizes the energy consumption in Joules. The box plot illustrates that the energy consumption is distributed similarly for non-constrained as well as constrained test executions. It appears that JoularJX measured significant outliers, especially for the constrained test executions. When comparing the median, the constrained test scenario consumes about 2'000 Joules more energy than the non-constrained scenario.

The total energy consumption is distributed among different Spring Boot controllers. These controllers receive and handle the requests sent by JMeter. Figure 38 visualizes the energy consumption of all Spring Boot controllers.

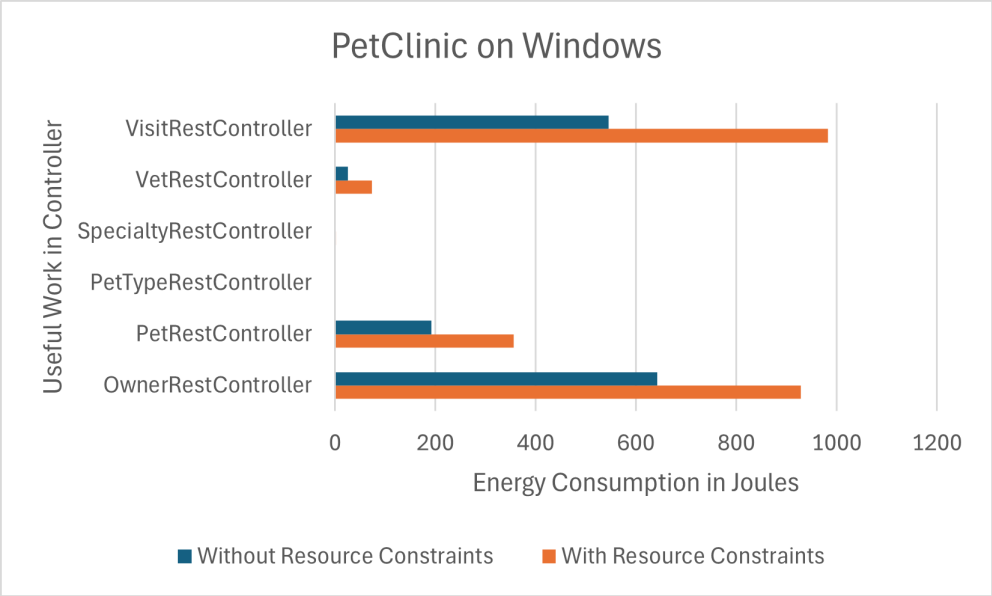

Figure 38. The energy consumption of all Spring Boot controllers on Windows

The horizontal axis refers to the energy consumption in Joules, while the vertical axis lists the controller classes. The horizontal axis reveals a difference in absolute energy consumption between the non-constrained and constrained test executions. The constrained scenario generally consumes significantly more energy over all controllers. In the specific case of the *OwnerRestController*, the constrained test scenario consume 250 to 300 Joules more for the same amount of work.

The controller classes aggregate the energy consumption of all operations within the respective controller class. Figure 39 visualizes a granular view of the energy consumption of all operations in the PetClinic application.

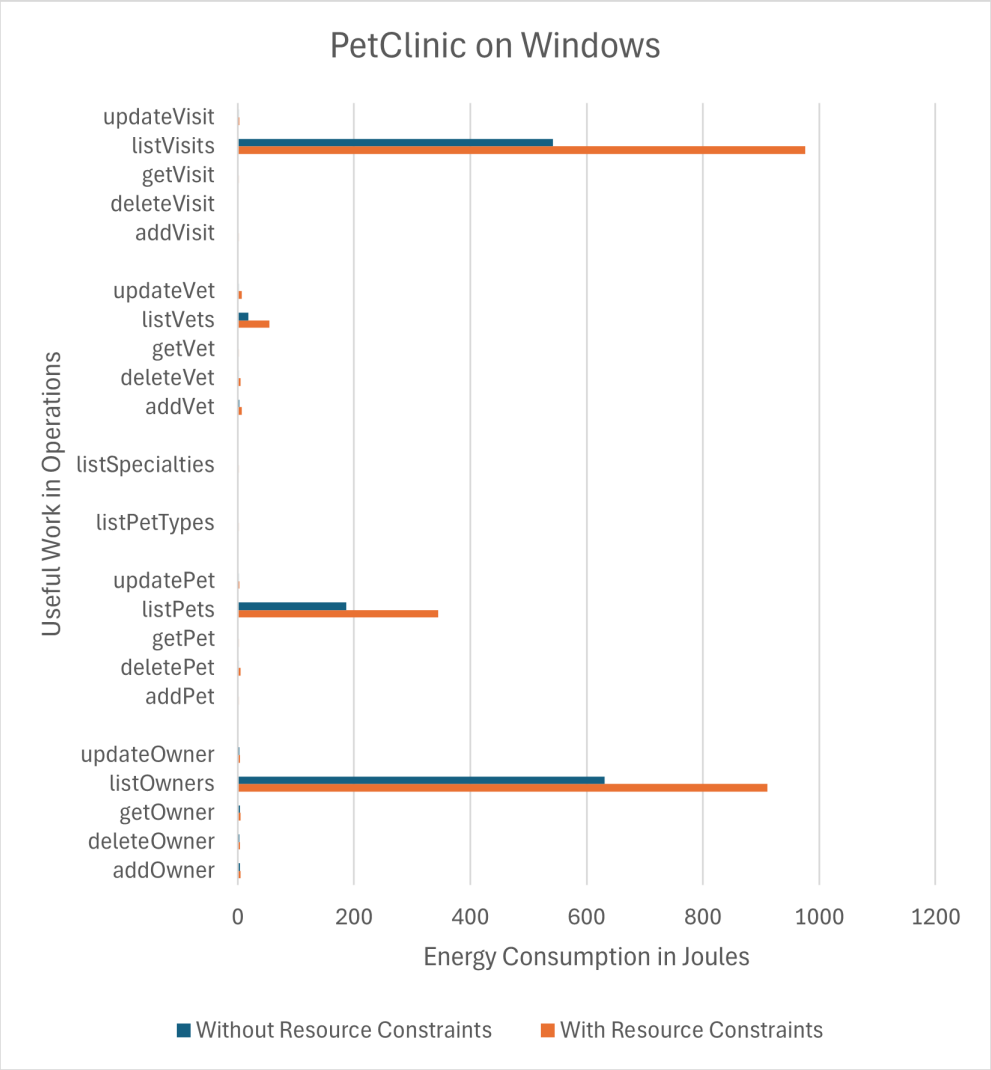

Figure 39. The energy consumption of all operations on Windows

The vertical axis lists all operations within controller classes. It is evident that the *listVisits*, *listVets*, *listPets*, and *listOwners* consume considerably more energy than the remaining operations. It appears that the amount of energy consumed by other operations is almost negligible.

The difference in absolute energy consumption between the two test executions is also visible. Again, the *listOwners* operation consumes 250 to 300 Joules more on the constrained test executions as opposed to the non-constrained ones. The discrepancy in absolute energy consumption between the two test scenarios reflects in their energy efficiency.

Subsection 2.2.2 states that resource and energy efficiency is characterized in useful work done per Joule. The energy efficiency of the system is calculated by dividing the useful work by the total energy consumed to perform said useful work. For this experiment, the entire test plan consisting of 11'500 requests is considered useful work. The total energy consumption refers to the average energy consumption of the entire PetClinic application across all test executions for each scenario. Table 12 summarizes the energy efficiency of the PetClinic application on Windows.

Table 12. The energy efficiency of the PetClinic application on Windows

Test Execution	Useful Work (Amount of Requests)	Total Energy Consumption	Energy Efficiency
Without Resource Constraints	11'500 Requests	1'406.88 Joules	8.17 Requests per Joule
With Resource Constraints	11'500 Requests	2'344.25 Joules	4.91 Requests per Joule

The calculations reveal that the non-constrained test executions achieve a better energy efficiency in terms of requests per Joule. The constrained test executions perform, on average, about three requests per Joule worse than the non-constrained ones. While the constraints improve the performance, they also increase the energy consumption. This sums up the experiment results on Windows, Subsection 4.1.2 reports on the experiment results on Linux.

4.1.2. Experiment on Linux

The experiment on Linux refers to the same requirements and presets as the experiment on Windows described in Subsection 4.1.1. The results of primary interest are the differences and similarities of the measurements on Windows and Linux.

Table 13 summarizes the average latency and throughput for the test executions reported in the JMeter result file.

Table 13. The performance analysis of the PetClinic application on Linux

Test Execution	Average Latency	Average Throughput
Without Resource Constraints	350ms	13.4 Requests per second
With Resource Constraints	351ms	13.4 Requests per second

The results indicate that the resource constraints have a negligible impact on the performance, as the latency and throughput are almost equal. This observation is visible in the overall processing time of the test executions extracted from the JMeter log file. Figure 40 visualizes the processing time for all requests sent to the PetClinic application.

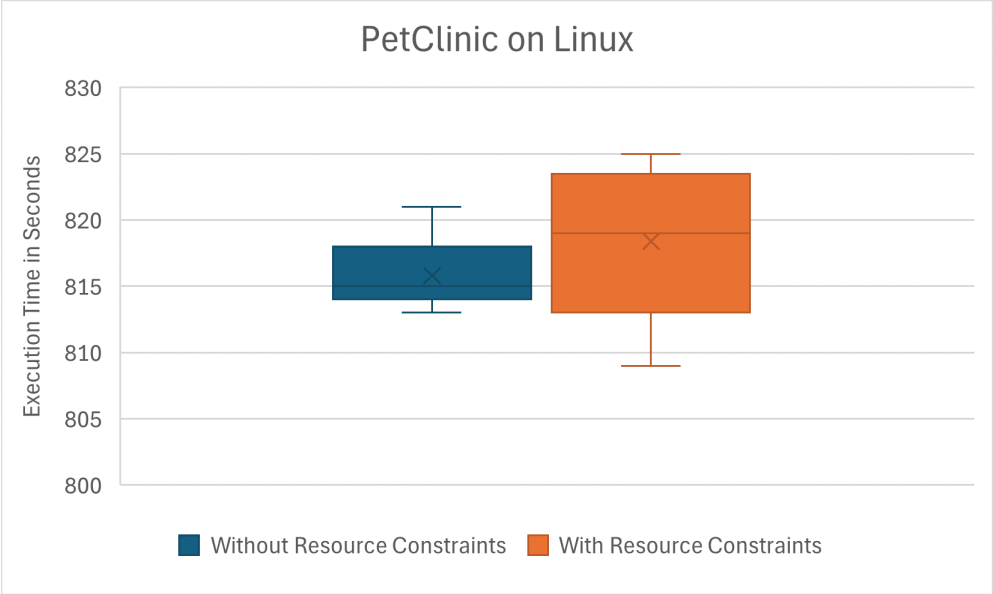

Figure 40. The processing time for all requests sent to the PetClinic application reported by JMeter on Linux

The vertical axis visualizes the processing time for all requests in seconds. The results reveal that the resource constraints have a slightly negative impact on the processing time. However, it is worth noting that the discrepancy between the two outliers is only about 15 seconds, which seems negligible in the context of the overall processing time.

The equal performance reflects in equal energy consumption for both test scenarios. Figure 41 visualizes the energy consumption of the PetClinic application extracted from the JoularJX report files.

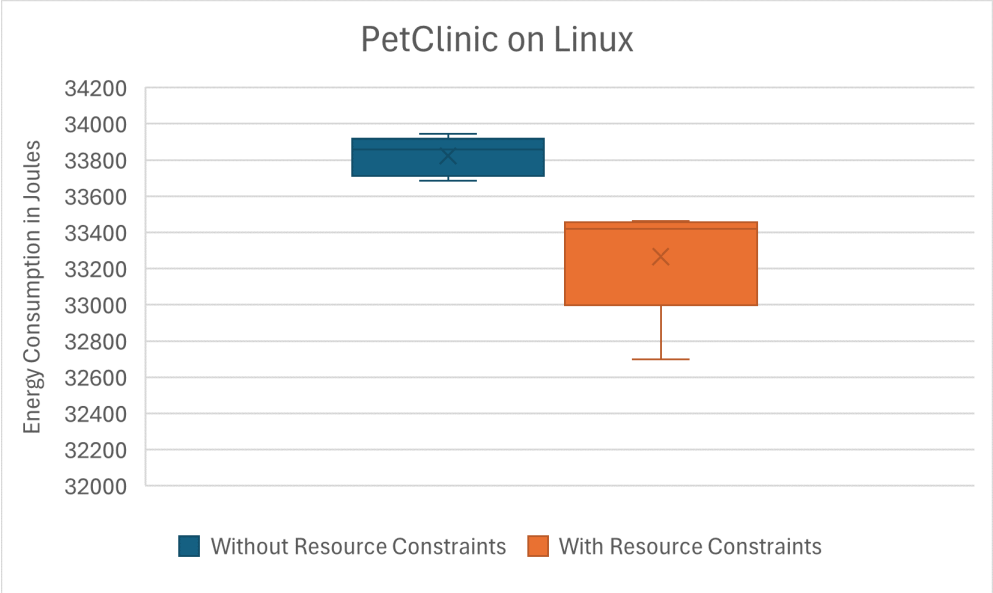

Figure 41. The energy consumption of the PetClinic application measured with JoularJX on Linux

The vertical axis visualizes the energy consumption in Joules. The test executions appear to have a large distribution of energy consumption, especially for the constrained test scenario. However, the relative difference of about 1'000 Joules between the constrained and non-constrained test executions is only about 3% of the total energy consumption. It is worth noting that the absolute energy consumption on Linux is up to ten times higher than on Windows.

The massive increase in energy consumption raises the question of how it is distributed across the system. Figure 42 visualizes the energy consumption distribution across the Spring Boot controllers.

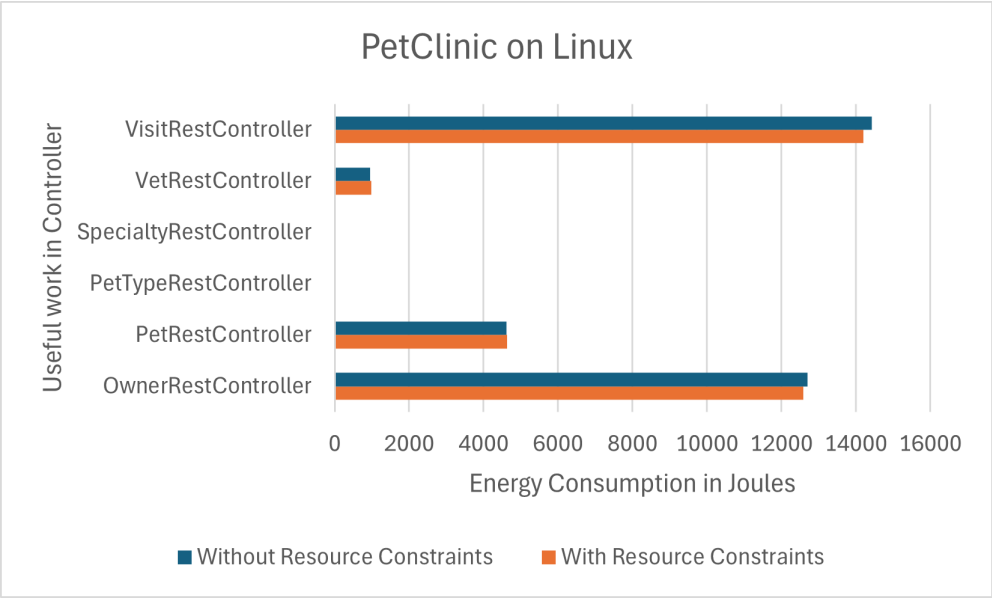

Figure 42. The energy consumption of all Spring Boot controllers on Linux

The horizontal axis visualizes the energy consumption in Joules. The vertical axis lists the controller classes. The diagram reveals that the absolute energy consumption is almost equal between both test scenarios. The relative distribution of energy consumption is similar to the one on Windows described in Figure 38.

Figure 43 visualizes a granular view of the energy consumption of all operations aggregated in the Spring Boot controllers.

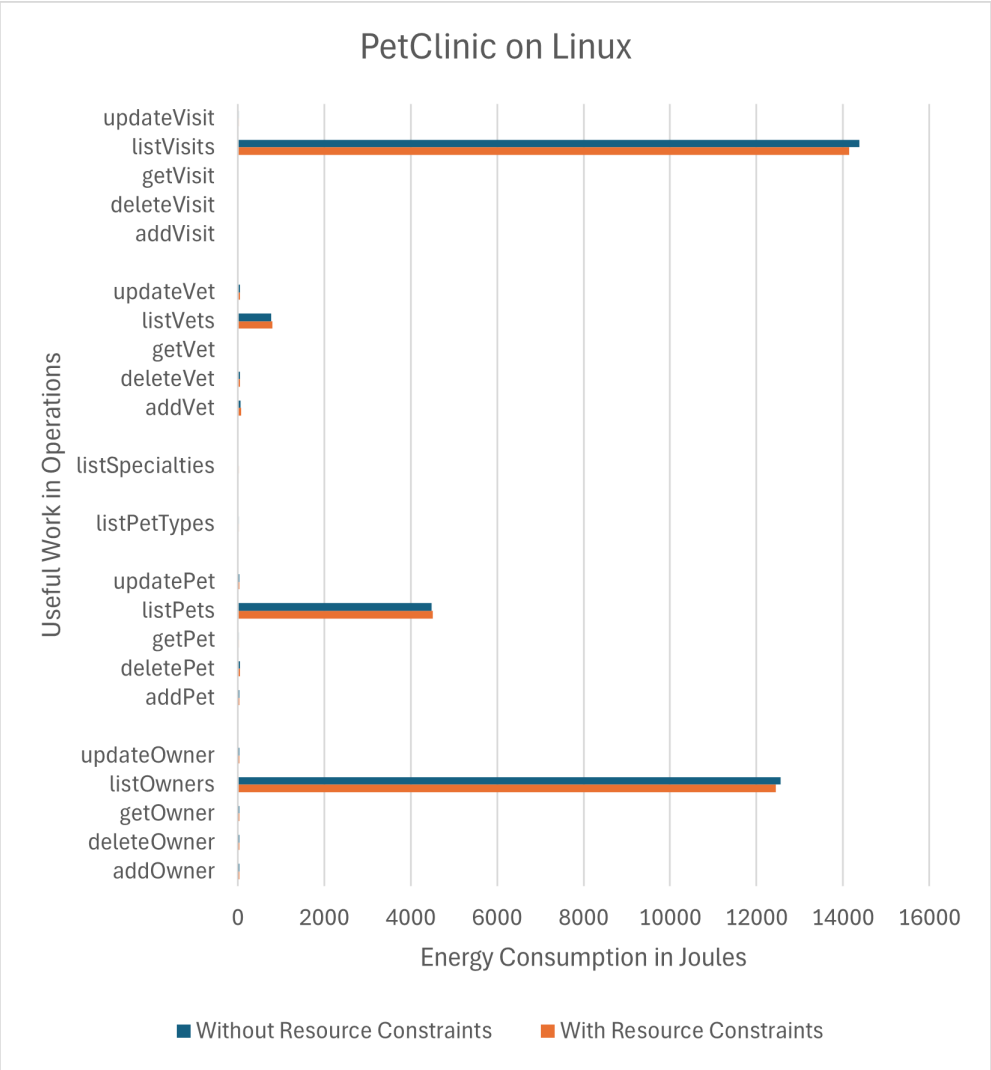

Figure 43. The energy consumption of all operations on Linux

The vertical axis lists the operations. The results on Linux align with the results on Windows described in Figure 39. The energy consumption is distributed similarly across all operations. The *listVisits*, *listVets*, *listPets*, and *listOwners* operations consume significantly more energy than the remaining operations.

The observed results suggest just minor deviations in energy efficiency for both test scenarios. Table 14 summarizes the energy efficiency of the PetClinic application on Linux.

Table 14. The energy efficiency of the PetClinic application on Linux

Test Execution	**Useful Work (Amount of Requests)**	**Total Energy Consumption**	**Energy Efficiency**
Without Resource Constraints	11'500 Requests	32'725.28 Joules	0.35 Requests per Joule
With Resource Constraints	11'500 Requests	32'433.56 Joules	0.36 Requests per Joule

The difference in energy efficiency between both test scenarios is marginal. These minor deviations confirm that the resource constraints have a negligible impact on the energy efficiency on Linux. However, the energy efficiency is considerably lower on Linux compared to Windows indicating that the Linux machine consumes more energy to process the same amount of requests. This sums up the experiment results on Linux, Subsection 4.1.3 summarizes the experiment results and discusses the findings.

4.1.3. Experiment Summary

The experiment results on Windows and Linux provide valuable insights into the performance and energy consumption of the PetClinic application. The test executions on the two test environments reveal differences in performance and resource and energy efficiency. Figure 44 aggregates Figure 36 from Subsection 4.1.1 and Figure 40 from Subsection 4.1.2. Figure 44 illustrates the significant difference in execution time for all requests between Windows and Linux.

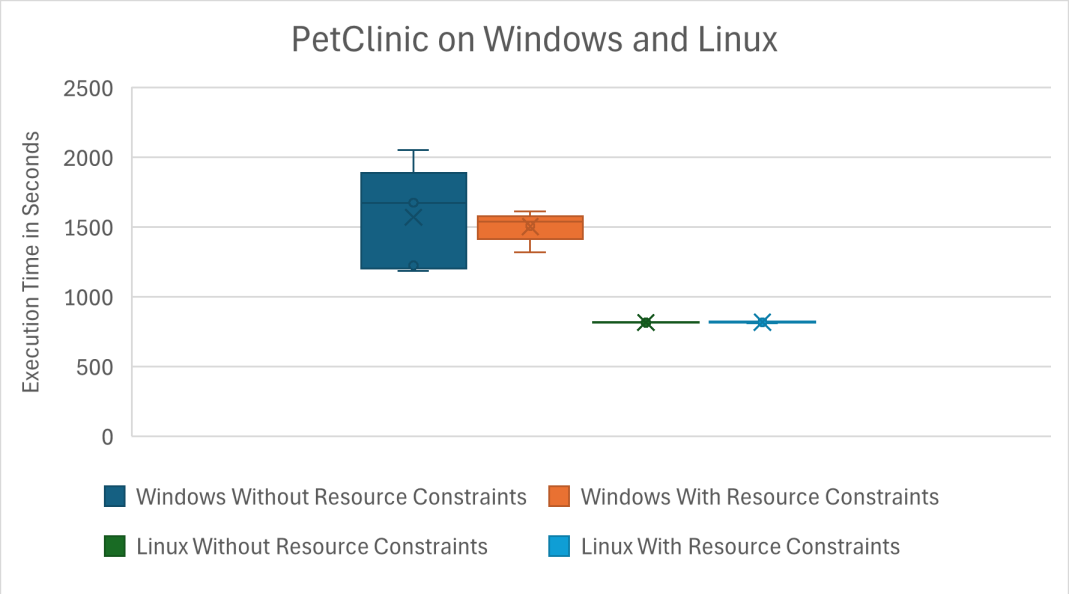

Figure 44. Comparison of execution times between Windows and Linux

The vertical axis visualizes the execution time in seconds. Test executions under Windows take up to twice as long as on Linux. Additionally, the test executions under Windows have a significantly broader distribution of execution times than the ones under Linux.

This observation is reflected in the energy consumption of the PetClinic application. Figure 45 aggregates Figure 37 and Figure 41 respectively. Figure 45 visualizes the massive energy consumption differences between Windows and Linux.

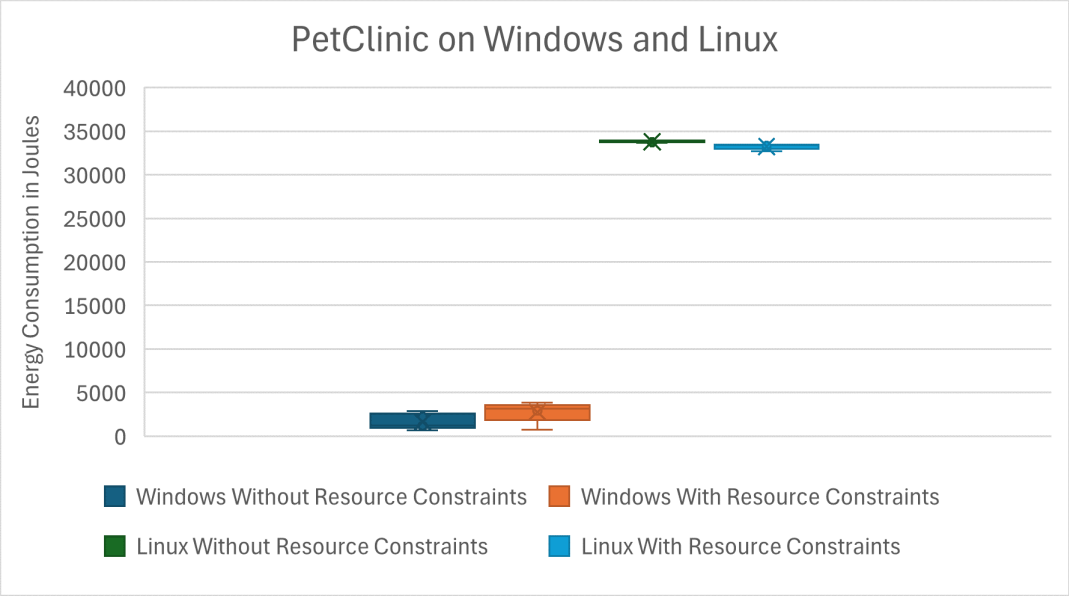

Figure 45. Comparison of energy consumptions between Windows and Linux

The absence of an automated test script on Windows likely explains the broad distribution and the high energy consumption outliers. A possible explanation is that the application is not immediately stopped after the load testing finishes, leading to a longer execution time and higher energy consumption. Eventually, an automated test execution is recommended to remove human interference and reduce potential variance in the measurements.

The results further indicate an inverse correlation between performance and energy efficiency. Worse performance on Windows is associated with lower energy consumption resulting in better energy efficiency. Better performance on Linux is associated with higher energy consumption resulting in worse energy efficiency. The resource constraints on Windows seem to reinforce this observation as the constrained test executions achieve a better performance but consume more energy. Resource constraints allow the JVM to allocate more resources, which in turn consume more energy. The consistent results on Linux are likely due to the powerful hardware. It appears that the Linux environment is able to allocate more resources to the JVM by default. In order to achieve consistent results within and across different systems, we decide to run all test executions with resource constraints.

The findings of these experiments are consistent with the observations in the GGS blog. The author of the GGS blog measured the energy consumption of the PetClinic application on a macOS device and reported similar results. Figure 46 illustrates results from the GGS blog that show the relative energy consumption of operations measured with JoularJX.

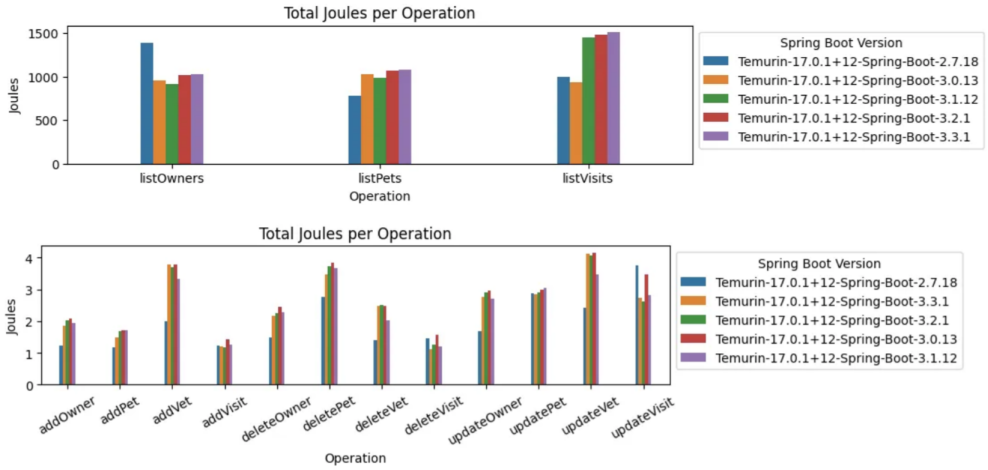

Figure 46. The energy consumption of all operations on macOS from the GGS blog [51]

The vertical axis reveals a massive difference in energy consumption of over 1'000 Joules between the *listOwners*, *listPets*, and *listVisits* operations at the top, and the remaining operations at the bottom. This behaviour appears to be consistent across all three test environments and leads to the conclusion that the energy consumption of operations mainly depends on the amount of data processed. The *list* operations process all entities in the database table in each request. In comparison, the *addOwner* operation only processes one entity per request.

Section 4.2 adapts the PetClinic test plan and the application configuration to solely measure the owner endpoints of the PetClinic again. Measuring the owner endpoint is a preparatory step for the measurements of LakesideMutual.

4.2. PetClinic Experiment: Compare JPA and Spring Data JPA

This section uses an adapted version of the original PetClinic test plan, evaluated in Section 4.1, to measure the owner HTTP endpoints only. The experiment covers three scenarios. The first scenario includes the entire set of create, read, update and delete (CRUD) operations on owners utilizing native JPA to access the database. The second scenario executes the same set of operations utilizing Spring Data JPA to access the database. The third scenario covers just the CRU operations, without the delete operation, utilizing Spring Data JPA to access the database. Omitting the delete operation and using Spring Data JPA is a preparatory step that is necessary to compare the results to the LakesideMutual application in Section 4.3. All test runs are conducted with resource constraints to reduce the complexity when comparing the results.

4.2.1. Experiment on Windows

The results of interest in this experiment are the impact of different database access technologies and the impact of omitting the delete operation. Table 15 lists the performance results for the three test scenarios.

Table 15. The latency and throughput of the PetClinic owner endpoint on Windows

Test Execution	Average Latency	Average Throughput
JPA With CRUD	526ms	6.1 Requests per second
Spring Data JPA With CRUD	474ms	4.0 Requests per second
Spring Data JPA With CRU	796ms	3.2 Requests per second

The first scenario with native JPA and CRUD establishes a baseline for this experiment as the remaining test configurations are equal to the experiment in Section 4.1. The second scenario with Spring Data JPA and CRUD appears to perform slightly better when it comes to latency but worse when it comes to throughput. The third scenario with Spring Data JPA and CRU reveals the worst performance of all scenarios. It roughly shows a 300ms increase in latency and a 0.8 requests per second decrease in throughput.

Figure 47 illustrates the total execution time for all requests sent to the PetClinic owner endpoint.

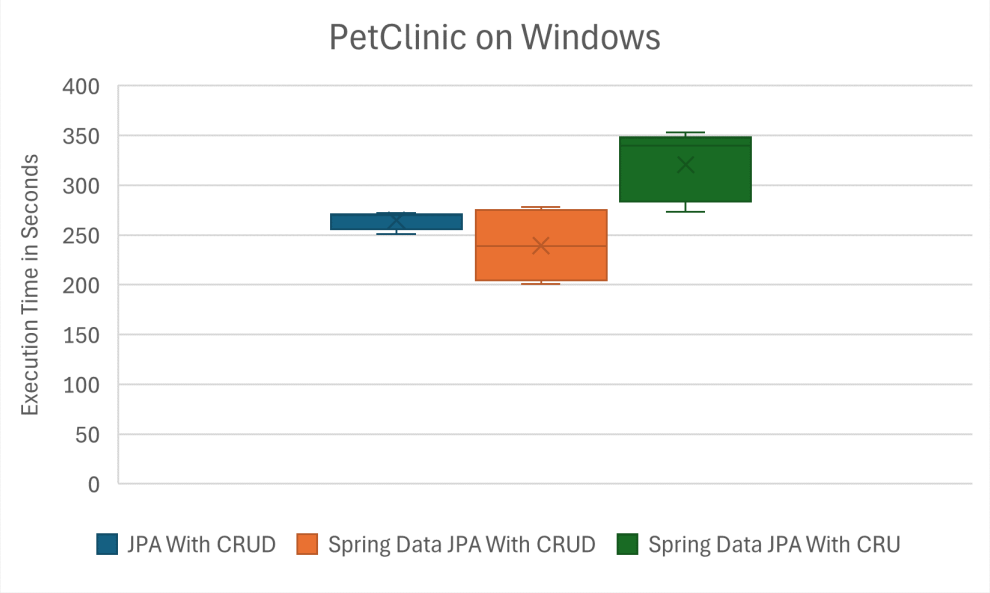

Figure 47. The processing time for all requests sent to the PetClinic owner endpoint reported by JMeter on Windows

The vertical axis illustrates the execution time for all requests in seconds. The JPA CRUD scenario reveals a narrow distribution of execution times. The Spring Data JPA scenarios have rather broad distributions of execution times. This could be a symptom of utilizing Spring Data JPA instead of native JPA. It is surprising that the Spring Data JPA CRU scenario, which executes fewer requests, requires more execution time.

The total energy consumption reflects the deviations in the performance results. Figure 48 illustrates the energy consumption details of the PetClinic application.

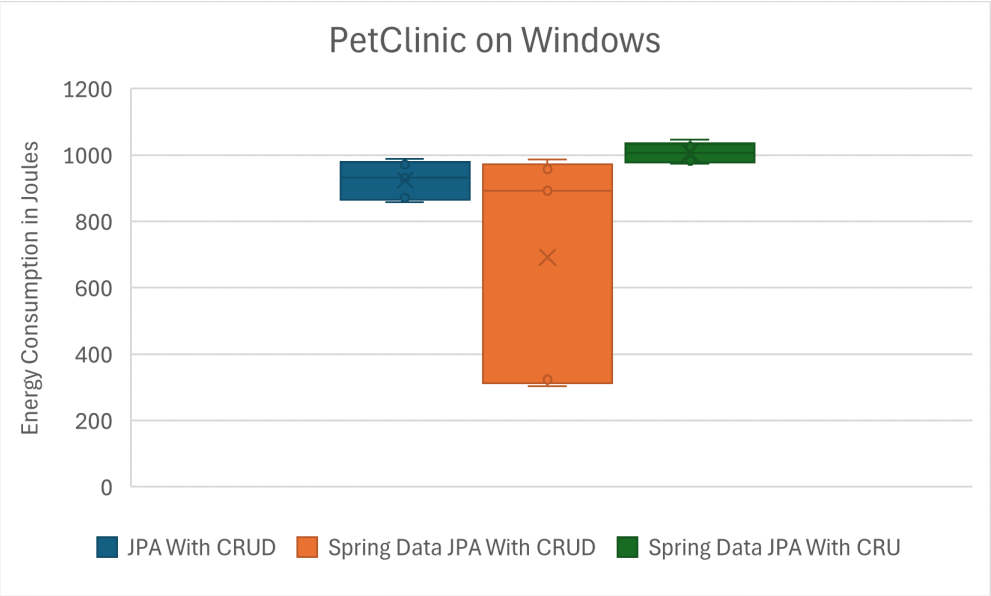

Figure 48. The energy consumption of the PetClinic application measured with JoularJX on Windows

The JPA CRUD and Spring Data JPA CRU scenarios show a consistent energy consumption of about 900 and 1000 Joules respectively. The Spring Data JPA CRUD scenario shows significant deviations when it comes to energy consumption. Its third quartile and the median value are at around 900 Joules. The diagram illustrates outliers at around 350 Joules, which is more than 50% less energy consumption compared to the median.

This raises the question if the outliers affect the distribution of the energy consumption. Figure 49 illustrates the distribution of energy consumption for all owner operations.

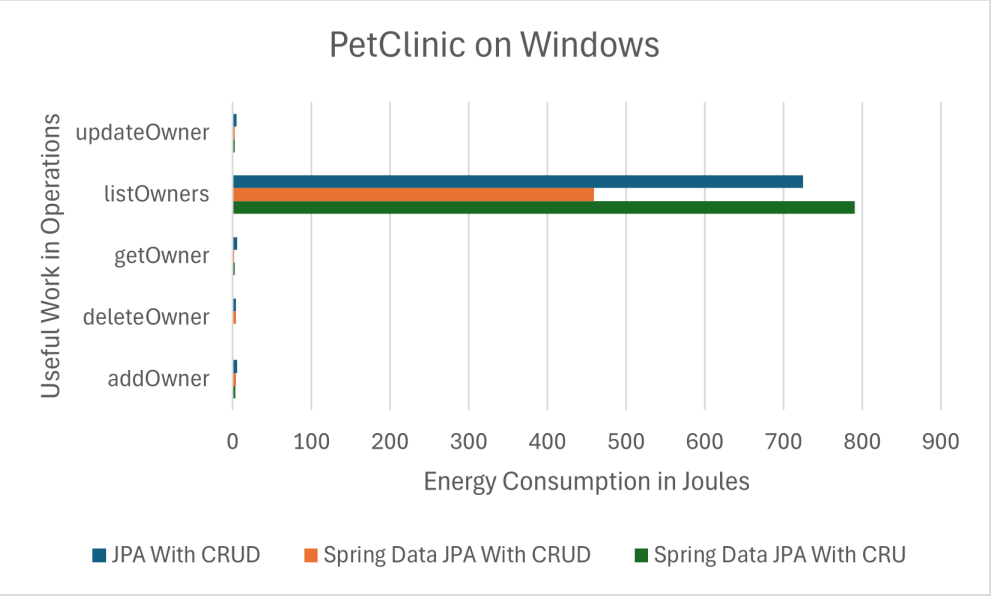

Figure 49. The energy consumption of all owner operations on Windows

It appears that the outliers in Figure 48 mainly affect the *listOwners* operation. The Spring Data JPA CRUD scenario reveals a deviation of about 250 to 300 Joules for the *listOwners* operation. The remaining operations are mainly unaffected, due to their overall low energy consumption.

This massive deviation reflects in the energy efficiency. Table 16 lists the energy efficiency for the three test scenarios.

Table 16. The energy efficiency of the PetClinic application on Windows

Test Execution	Useful Work (Amount of Requests)	Total Energy Consumption	Energy Efficiency
JPA With CRUD	2'500 Requests	924.18 Joules	2.71 Requests per Joule
Spring Data JPA With CRUD	2'500 Requests	692.49 Joules	3.61 Requests per Joule
Spring Data JPA With CRU	2'000 Requests	1006.97 Joules	1.99 Requests per Joule

The JPA CRUD scenario consumes about 230 Joules more than the second scenario. The Spring Data JPA CRUD scenario appears to be the most energy efficient with about 700 Joules and 3.61 requests per Joule. However, it is important to note that the outliers positively affect the average energy efficiency. The Spring Data JPA CRU scenario consumes just shy of 100 Joules more energy than the first scenario and therefore is the least energy efficient. This sums up the experiment results on Windows, Subsection 4.2.2 reports on the experiment results on Linux.

4.2.2. Experiment on Linux

The experiment on Windows revealed significant deviations in the energy consumption of the test scenario with Spring Data JPA and the delete operation. It is unclear whether these deviations are due to the database access technology or due to other factors such as the operating system. The experiment on Linux aims to clarify this question. The results of interest in this experiment are again the impact of a different database access technology and the impact of omitting the delete operation. Table 17 lists the performance results for the three test scenarios on Linux.

Table 17. The latency and throughput of the PetClinic owner endpoint on Windows

Test Execution	Average Latency	Average Throughput
JPA With CRUD	321ms	10.6 Requests per second
Spring Data JPA With CRUD	326ms	11.7 Requests per second
Spring Data JPA With CRU	470ms	8.4 Requests per second

The first two scenarios including the delete operation show a similar latency, but the second scenario shows a slightly higher throughput. The third scenario excluding the delete operation shows an increase in latency and a decrease in throughput. The results overall appear to be more consistent than the results on Windows.

Figure 50 illustrates the total execution time for all requests sent to the PetClinic owner endpoint.

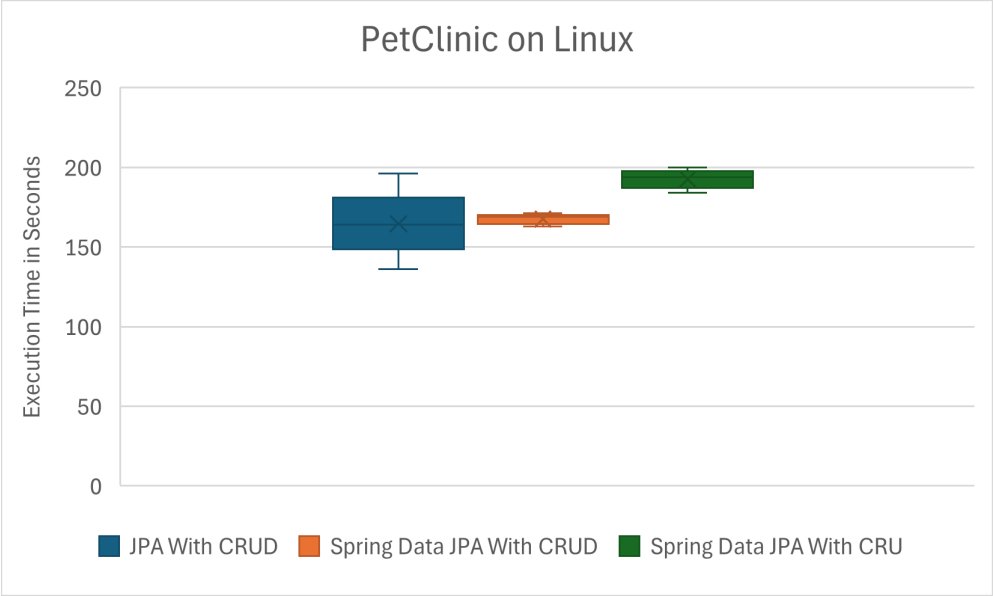

Figure 50. The processing time for all requests sent to the PetClinic owner endpoint reported by JMeter on Linux

The vertical axis illustrates the execution time for all requests in seconds. The results reveal a slightly different distribution of execution times compared to the results on Windows. The JPA CRUD scenario shows a broader distribution of execution times, while the Spring Data JPA scenarios show a more narrow distribution. The absolute execution times are much faster on Linux, similar to the first experiments described in Subsection 4.1.2. The median values reveal that the first two CRUD scenarios are almost equally performant, while the third one is the slowest of all scenarios.

Figure 51 illustrates the energy consumption of the PetClinic application.

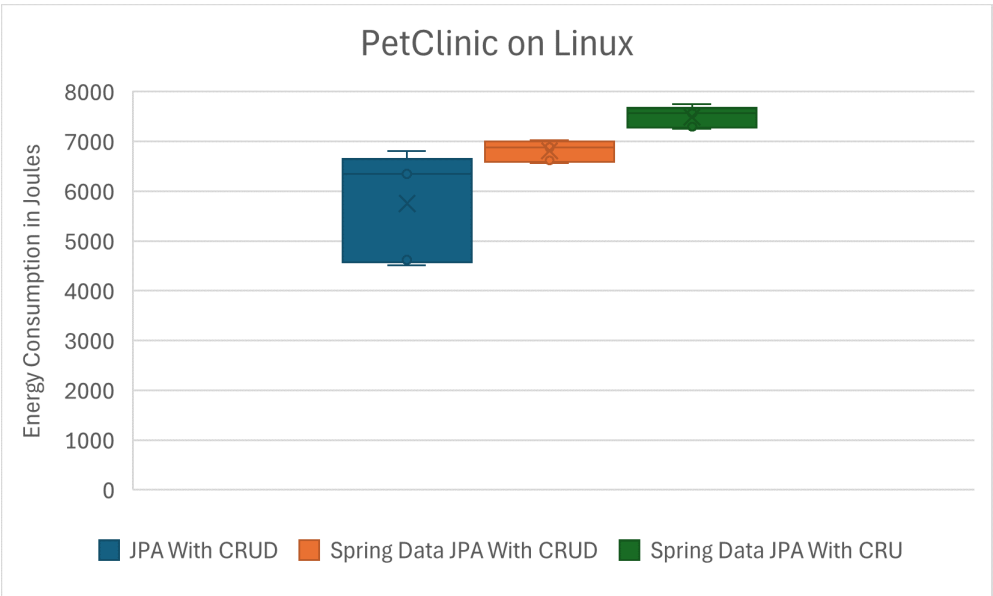

Figure 51. The energy consumption of the PetClinic application measured with JoularJX on Linux

The first test scenario with JPA and CRUD shows a broader distribution of energy consumption. The second scenario with Spring Data JPA and CRUD reveals a slightly higher but more narrow distribution of energy consumption. The third scenario with CRU has the highest energy consumption of all scenarios.

Figure 52 illustrates the distribution of energy consumption for all owner operations.

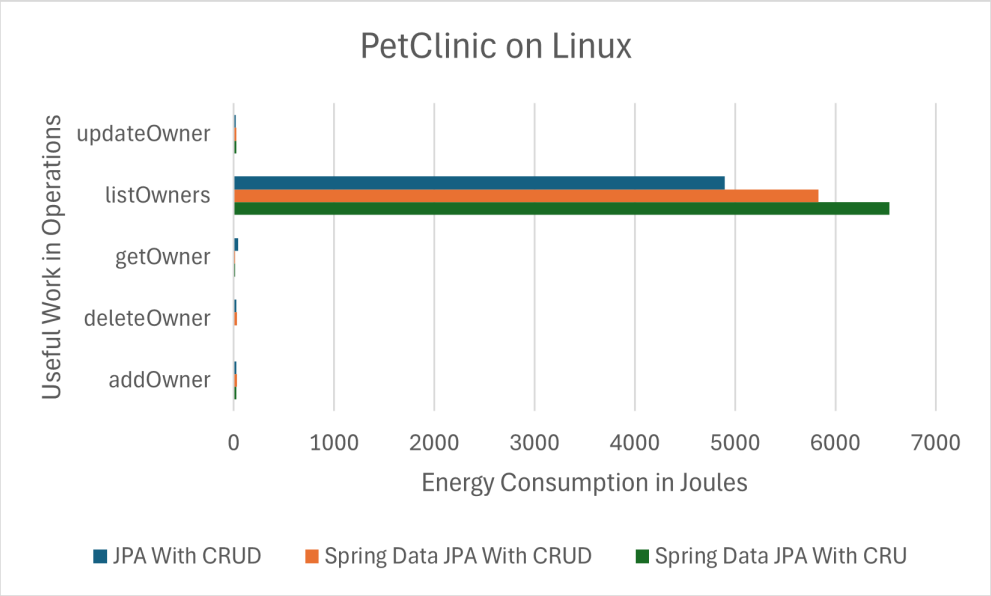

Figure 52. The energy consumption of all owner operations on Linux

The diagram reveals a similar energy distribution as in all previous experiments. Energy consumption deviations mainly affect the *listOwners* operation. It appears that the test scenario without the delete operations consumes the most energy. This results in a negative impact on energy efficiency.

Table 18 lists the energy efficiency for the three test scenarios.

Table 18. The energy efficiency of the PetClinic application on Linux

Test Execution	**Useful Work (Amount of Requests)**	**Total Energy Consumption**	**Energy Efficiency**
JPA With CRUD	2'500 Requests	5757.21 Joules	0.43 Requests per Joule
Spring Data JPA With CRUD	2'500 Requests	6813.93 Joules	0.37 Requests per Joule
Spring Data JPA With CRU	2'000 Requests	7492.56 Joules	0.27 Requests per Joule

The Spring Data JPA CRUD scenario consumes about 1000 Joules more than the JPA CRUD scenario. The Spring Data JPA CRU scenario in turn consumes about 700 Joules more than the Spring Data JPA CRUD scenario. This implies a negative impact on energy efficiency, especially for the third scenario, which executes 500 requests less than the other two scenarios. This sums up the experiment results on Linux, Subsection 4.2.3 summarizes the experiment results and discusses the findings.

4.2.3. Experiment Summary

The experiments on Windows and Linux provide insights into the impact of using native JPA or Spring Data JPA. Additionally, they reveal how omitting the delete operation affects the performance and energy consumption. The results on both test environments show deviations and unexpected behaviour, making it difficult to draw conclusions.

On Windows, the Spring Data JPA CRUD scenario has a medium to high performance, because it has the lowest latency, a medium throughput, and the lowest median execution time. Additionally, it has the lowest energy consumption, which results in the highest energy efficiency. These results indicate that there is a strong correlation between performance and energy efficiency. However, the Spring Data JPA CRUD scenario is heavily affected by outliers as shown in Figure 48, which makes the results unreliable. It is unclear what causes the inconsistencies in the results.

Surprisingly, the Spring Data JPA CRU scenario on Windows consumes the most energy even though it executes 500 requests less than the other two scenarios. This behaviour would be explainable by the fact that Spring Data JPA performs worse than JPA, but in that case, the Spring Data JPA CRUD scenario should not be the most energy efficient. This is a clear indication that the results are inconsistent and that the outliers in the Spring Data JPA CRUD scenario heavily affect the average energy efficiency.

The test executions on Linux appear to be more consistent, but they also show unexpected results. The JPA CRUD scenario appears to correlate with the initial measurements in Subsection 4.1.2 with a slightly worse performance and energy efficiency. It is surprising that a subset of the original test plan achieves worse results in terms of performance and energy efficiency. The idle energy consumption may affect the overall energy consumption significantly in perspective to the small amount of requests performed.

On Linux, the Spring Data JPA CRUD scenario even reveals a slightly higher energy consumption than the JPA CRUD scenario. This indicates that Spring Data JPA consumes more energy than native JPA. Spring Data JPA builds on top of JPA and adds additional abstractions and complexity to facilitate database interactions for developers. It is likely that the additional complexity leads to an increase in energy consumption. JPA as an application programming interface relies on the Java Database Connectivity (JDBC) to interact with the database. Future work could investigate if using JDBC directly instead of JPA or Spring Data JPA would lead to an even better performance and energy efficiency.

The Spring Data JPA CRU scenario confirms the surprising result observed on Windows. It consumes the most energy and has the worst performance. It remains unclear why fewer operations consume more energy in the exact same test setup. If Spring Data JPA is the reason for this behaviour, then the Spring Data JPA CRUD scenario should have a worse performance and energy efficiency.

All results on Linux indicate a strong correlation between performance and energy efficiency. The JPA CRUD scenario reveals the best performance and the highest energy efficiency, as opposed to the Spring Data JPA CRU scenario, which has the worst performance and the lowest energy efficiency.

Section 4.3 considers the result of the Spring Data JPA CRU test scenario and compares it to the LakesideMutual customer endpoint. This allows to compare the results across two different enterprise applications.

4.3. PetClinic and LakesideMutual Experiments: Compare Master Data APIs

This section refers to the adapted test plan used in the previous experiment in Section 4.2. This experiment compares the create, read, and update operations of the PetClinic owner endpoint with the LakesideMutual customer endpoint. The first scenario refers to the PetClinic owner endpoint and the second scenario refers to the LakesideMutual customer endpoint.

Section 2.1 establishes that the PetClinic application follows a monolithic architecture, while LakesideMutual is built with a service-oriented architecture. The monolithic architecture inherently requires the startup of the entire application, while the service-oriented architecture would allow the startup of selected services. Starting a single service can potentially lead to lower energy consumption and thus to non-meaningful results. We decide to run all experiments starting LakesideMutual with all four backend services to ensure a fair comparison of the two applications. This experiment aims to compare two different enterprise applications with each other.

4.3.1. Experiment on Windows

The results of interest in this experiment are similarities and differences in performance and energy consumption across the two enterprise applications. Table 19 lists the performance results for both applications.

Table 19. The latency and throughput of the PetClinic and LakesideMutual applications on Windows

Test Execution	Average Latency	Average Throughput
PetClinic Owner	796ms	3.2 Requests per second
LakesideMutual Customer	43ms	5.8 Requests per second

The results reveal that the PetClinic application achieves an almost twenty times higher average latency than the LakesideMutual application. LakesideMutual is able to achieve 2.6 requests per second more throughput than the PetClinic application for the same operations. This leads to differences in the total execution times for both scenarios.

Figure 53 illustrates the total execution time for all requests sent to the owner and customer endpoint.

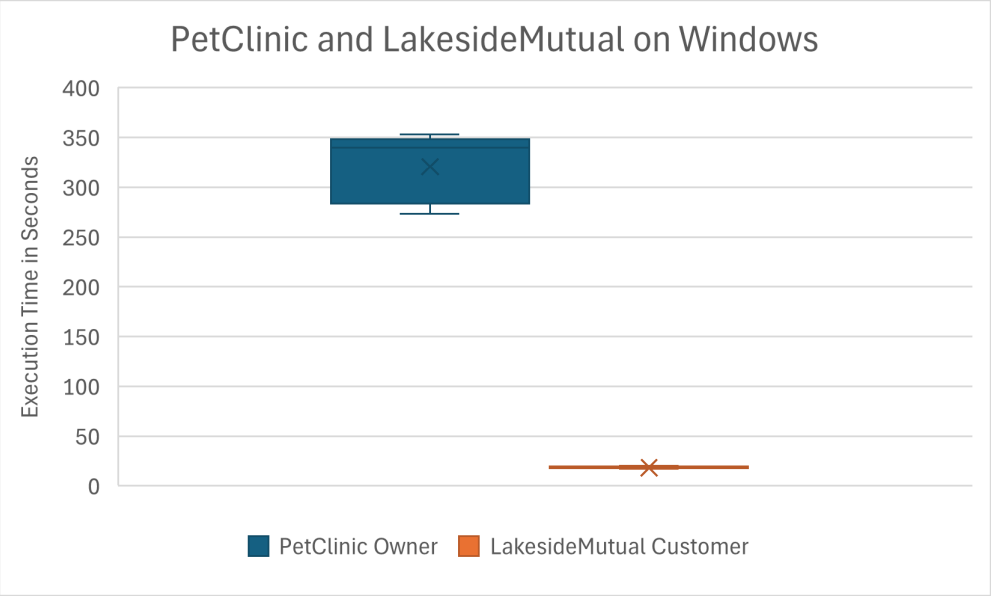

Figure 53. The processing time for all requests sent to the PetClinic owner and LakesideMutual customer endpoints reported by JMeter on Windows

The box plot confirms that the PetClinic application requires approximately ten times more time to process the same requests as the LakesideMutual application. The PetClinic scenario reveals a larger variance in the execution time, suggesting that LakesideMutual is more consistent than the PetClinic.

The total energy consumption reflects the massive difference. Figure 54 illustrates the energy consumption details of the PetClinic and LakesideMutual application.

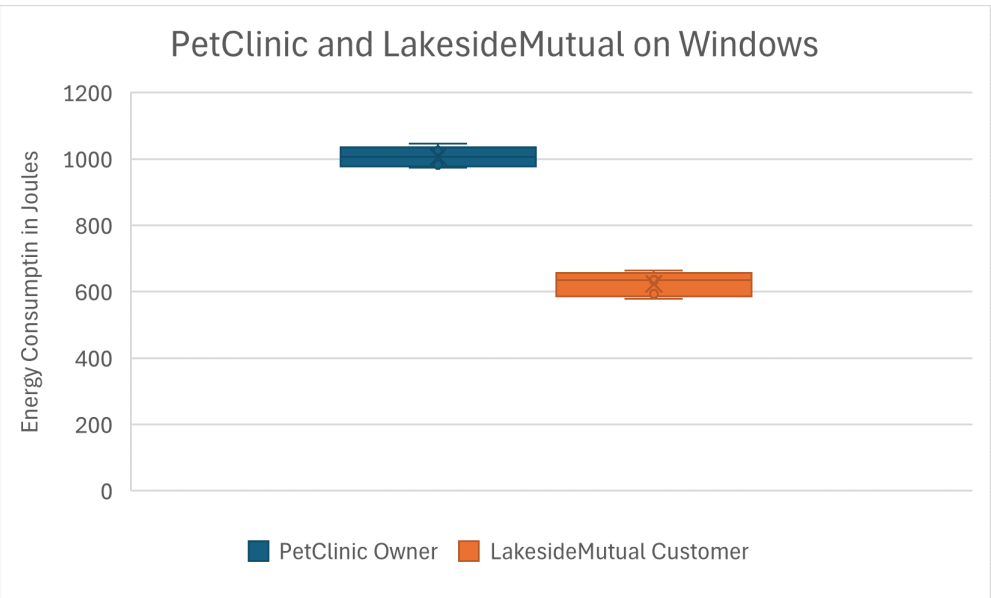

Figure 54. The energy consumption of the PetClinic and LakesideMutual applications measured with JoularJX on Windows

The diagram illustrates that the PetClinic application consumes about 1'000 Joules as opposed to the LakesideMutual application, which consumes about 600 Joules. This confirms that the PetClinic application not only requires more time but also consumes more energy than the LakesideMutual application to process the same requests.

As LakesideMutual consists of four backend services, the total energy consumption also includes the energy consumption of idle running services. Figure 55 visualizes the energy consumption of the Spring Boot controller of both applications.

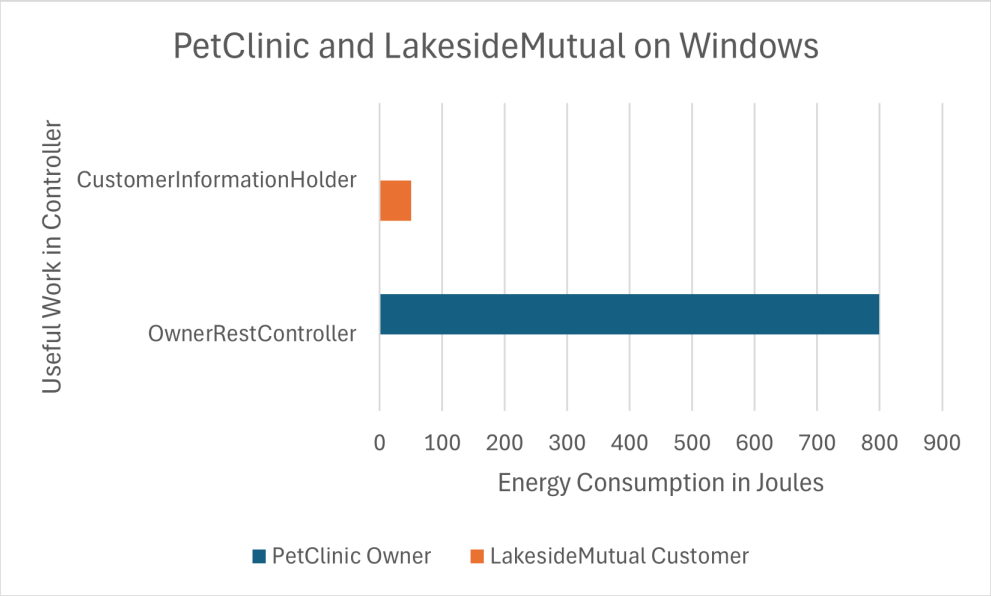

Figure 55. The energy consumption of the PetClinic and LakesideMutual controller measured with JoularJX on Windows

The PetClinic owner controller consumes about 800 Joules, 200 Joules less than the entire application consumes. This indicates that remaining parts of the application that are not under test consume 200 Joules for arbitrary reasons. The LakesideMutual customer controller on the other hand consumes just about 50 Joules to perform the same operations. The remaining 550 Joules are consumed by the remaining operations and services that are not under test. These results illustrate the significant difference in energy consumption between the two applications.

Figure 56 shows a granular view of the energy consumption across the involved operations.

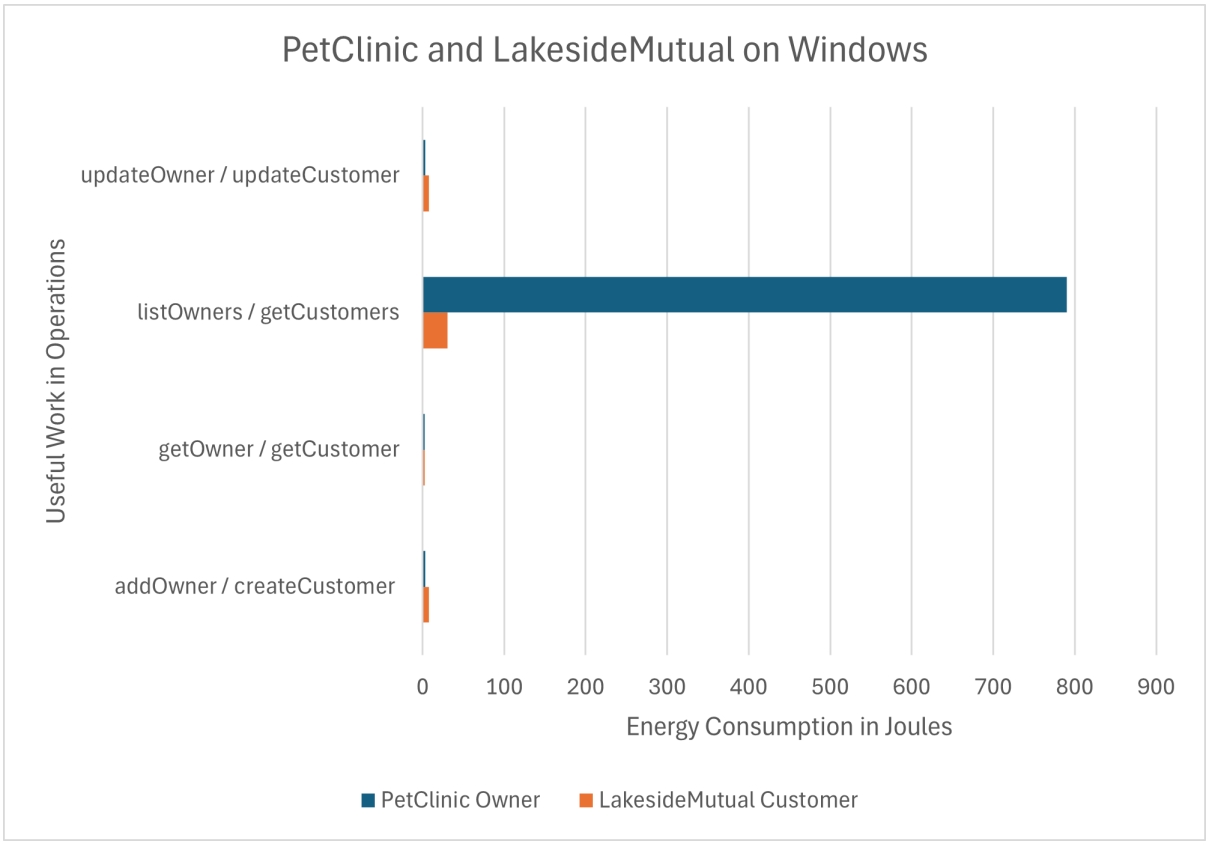

Figure 56. The energy consumption of the PetClinic owner and LakesideMutual customer operations measured with JoularJX on Windows

The diagram reveals that the *getCustomers* operation consumes about 15 to 20 times less energy than the *listOwners* operation in the PetClinic. Both applications have the same amount of test data stored in the database for customers and owners. It appears that the implementation of the PetClinic is less efficient than the LakesideMutual implementation. This in turn leads to a higher energy efficiency of the LakesideMutual application.

Table 20 lists the energy efficiency of the PetClinic and LakesideMutual applications.

Table 20. The energy efficiency of the LakesideMutual application on Windows

Test Execution	**Useful Work (Amount of Requests)**	**Total Energy Consumption**	**Energy Efficiency**
PetClinic Owner	2'000 Requests	1006.97 Joules	1.99 Requests per Joule
LakesideMutual Customer	2'000 Requests	623.85 Joules	3.21 Requests per Joule

The results confirm that the PetClinic application consumes about 400 Joules more energy for the same amount of work than the LakesideMutual application. Therefore, LakesideMutual achieves a higher energy efficiency than the PetClinic application. This sums up the experiment results on Windows, Subsection 4.3.2 reports on the experiment results on Linux.

4.3.2. Experiment on Linux

The experiment on Windows reveals a massive difference in energy consumption between the PetClinic and LakesideMutual. The test executions on Linux aim to confirm the observations on a different test environment. The results of interest are whether the reduced energy consumption for LakesideMutual can be confirmed.

Table 21 lists the performance results for both applications.

Table 21. The latency and throughput of the PetClinic and LakesideMutual applications on Linux

Test Execution	Average Latency	Average Throughput
PetClinic Owner	470ms	8.4 Requests per second
LakesideMutual Customer	43ms	16.2 Requests per second

The impression of PetClinic being slower than LakesideMutual is confirmed. The absolute numbers for the two test environments deviate, but the PetClinic is still significantly slower than LakesideMutual. Surprisingly, the latency of the LakesideMutual application is the same as on Windows. The LakesideMutual application seems to be even more performant on Linux than on Windows given the higher throughput.

Figure 57 illustrates the total execution time for all requests.

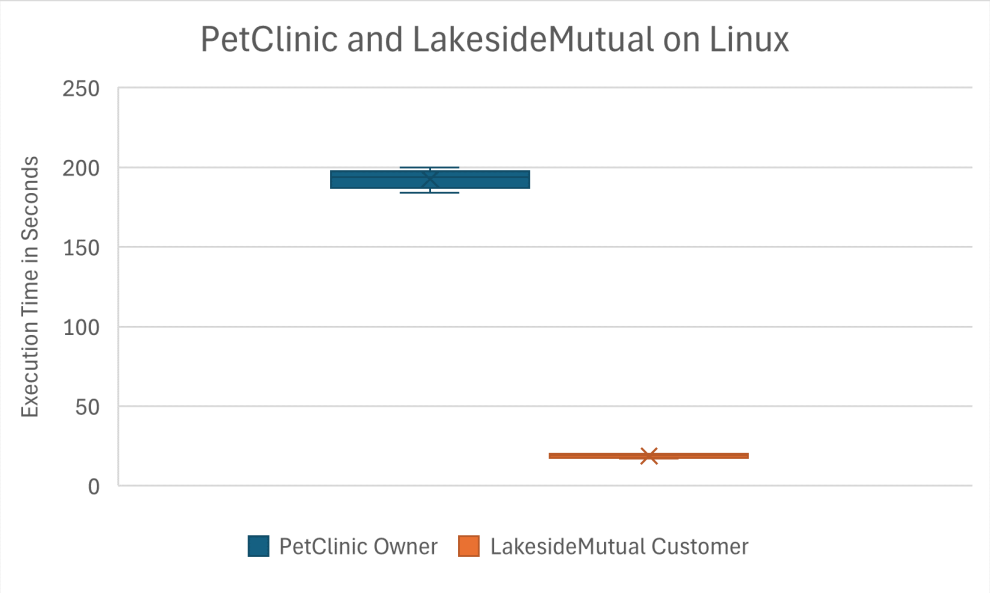

Figure 57. The processing time for all requests sent to the PetClinic owner and LakesideMutual customer endpoints reported by JMeter on Linux

The diagram reveals a similar picture as on Windows. The overall execution time is slightly faster on Linux compared to Windows. The PetClinic application requires about 200 seconds to process the same requests as the LakesideMutual application is able to process in about 25 seconds.

Figure 58 visualizes the energy consumption of the Spring Boot controllers.

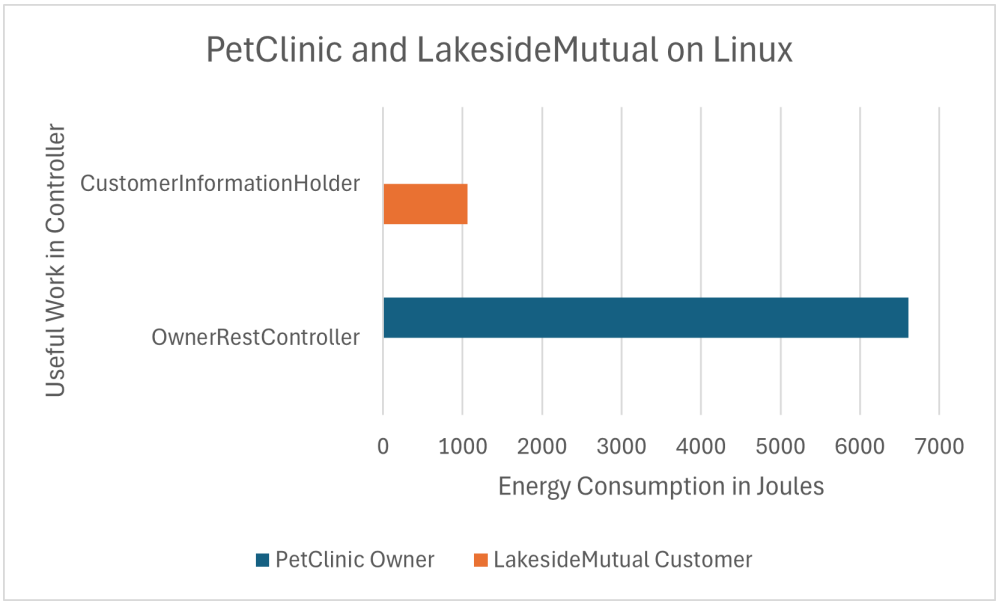

Figure 58. The energy consumption of the PetClinic and LakesideMutual controller measured with JoularJX on Linux

Similarly to the Windows experiment, the PetClinic controller consumes significantly more energy than the LakesideMutual controller. On Linux, the OwnerRestController consumes about 6.5 times more energy than the CustomerInformationHolder, as opposed to about 15 times more energy on Windows.

Figure 56 shows the energy consumption for each operation.

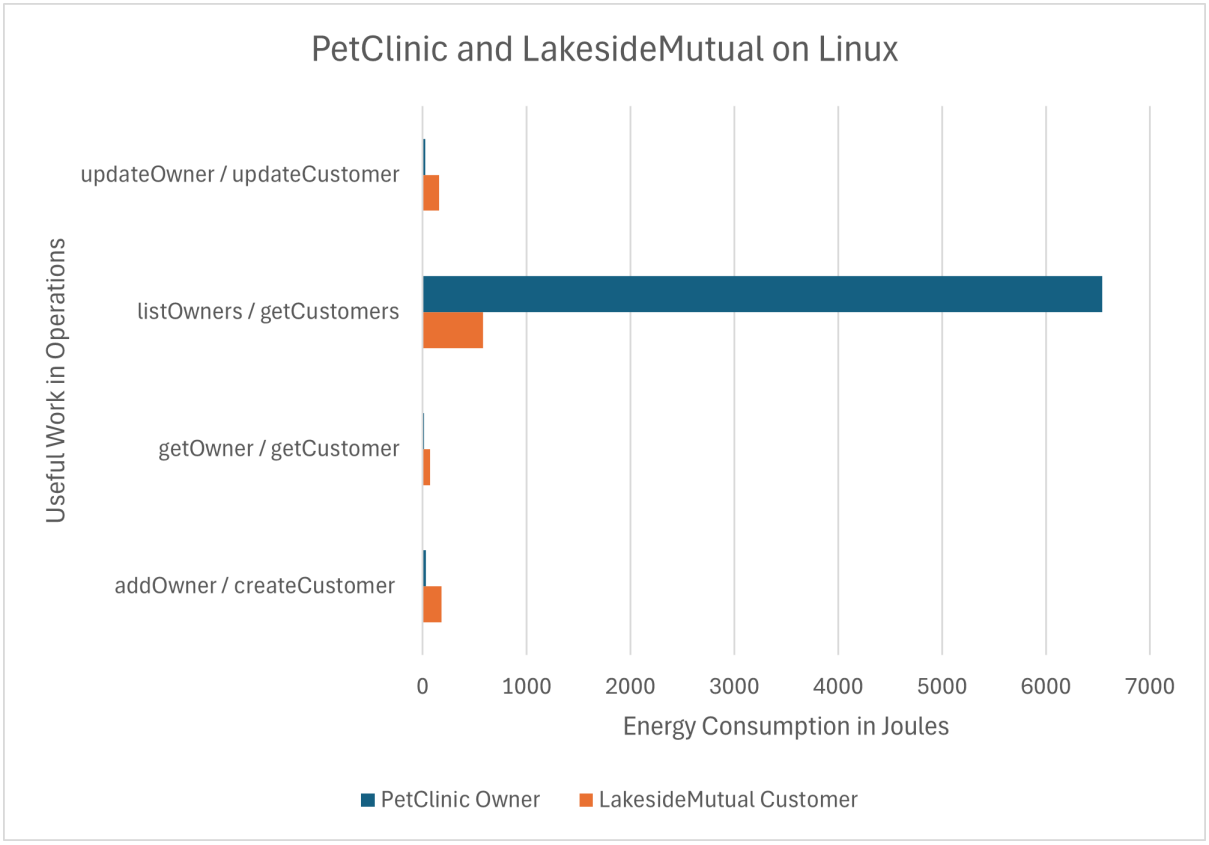

Figure 59. The energy consumption of the PetClinic owner and LakesideMutual customer operations measured with JoularJX on Linux

The relative distribution of energy consumption across the operations is equal to the one on Windows. The *listOwners* operation consumes the most energy, while all remaining operations in the PetClinic scenario consume an almost negligible amount of energy. The results for the LakesideMutual scenario show a more even distribution between the *getCustomers* operation and the remaining operations. The *getCustomers* operation still consumes the most energy with about 500 Joules.

Table 22 lists the energy efficiency of the PetClinic and LakesideMutual applications.

Table 22. The energy efficiency of the PetClinic and LakesideMutual application on Linux

Test Execution	Useful Work (Amount of Requests)	Total Energy Consumption	Energy Efficiency
PetClinic Owner	2'000 Requests	7492.56 Joules	0.27 Requests per Joule
LakesideMutual Customer	2'000 Requests	5494.01 Joules	0.36 Requests per Joule

Linux achieves a worse energy efficiency than Windows and the absolute difference in efficiency is just about 0.09 requests per Joule as opposed to 1.22 requests per Joule on Windows. However, the results confirm that the LakesideMutual application is significantly more performant and energy efficient than the PetClinic application. This sums up the experiment results on Linux, Subsection 4.3.3 summarizes the experiment results and discusses the findings.

4.3.3. Experiment Summary

The experiments aim to compare the performance and energy efficiency across two different enterprise applications. The results indicate massive performance and energy consumption differences between the two applications. The relative distribution of energy consumption appears to be comparable across the two applications and both test environments. The *getCustomers* and *listOwners* operations consume the most energy in all experiments.

The previous experiments in Section 4.2 revealed inconsistent results for test executions with and without the delete operation. The results indicated that Spring Data JPA consumes more energy to perform the same, or even less, work. This experiment relies on the same setup with Spring Data JPA and does perform similar create, read, and update operations. It appears that Spring Data JPA is not the only reason for the high energy consumption of the PetClinic application. The massive performance and energy consumption differences between the PetClinic and LakesideMutual suggest that the implementation of the PetClinic application is less efficient than the LakesideMutual implementation.

Furthermore, the results suggest an inverse and a strong correlation between performance and energy efficiency. The inverse correlation appears between the two test environments. The Linux environment is able to achieve a better performance at the cost of a worse energy efficiency, while the Windows environment achieves a better energy efficiency at the cost of a worse performance. A strong correlation occurs between the two applications. The implementation of LakesideMutual leads to a better performance and energy efficiency than the PetClinic application.

This section relied on the LakesideMutual customer core service to test the customer endpoint. Section 4.4 considers the same LakesideMutual customer test scenario and extends the setup to compare a single running service with multiple running services. This setup allows for a comparison of a monolithic architecture with a service-oriented architecture.

4.4. LakesideMutual Experiment: Compare Different Services

This section measures the customer endpoints of the LakesideMutual application in two variants and compares the results. This experiment consists of two scenarios and refers to the different services described in Subsection 2.1.3.

The first scenario covers the create, read, and update operations directly on the customer core service. Directly using the customer core service resembles the monolithic architecture of the PetClinic application and establishes a baseline for the second scenario. The first scenario is therefore referred to as the *monolithic scenario*. The second scenario covers the same set of operations using the customer management and the customer self-service services. Both services communicate with the customer core service. Using all three services aligns with the service-oriented architecture of the LakesideMutual application and mimics its real-world usage. The second scenario is therefore referred to as the *service-oriented scenario*. The experiments are solely conducted with resource constraints to reduce the complexity when comparing the results.

4.4.1. Experiment on Windows

The results of interest in this experiment are the differences between a single service call and multiple service calls. Table 23 lists the performance results for the two test scenarios.

Table 23. The latency and throughput of the LakesideMutual customer endpoint on Windows

Test Execution	Average Latency	Average Throughput
Monolithic	43ms	5.8 Requests per second
Service-oriented	68ms	7.6 Requests per second

The results reveal a similar latency for both test scenarios. The service-oriented scenario reports an increase of 25ms in latency and 1.8 requests per second in throughput. This increase is explainable by the fact that the customer management and customer self-service services perform additional network calls to the customer core service. Chances are that multiple services are able to handle more requests in parallel, which would indicate that the overall execution time is lower.

Figure 60 illustrates the total execution time for all requests sent to the customer endpoint.

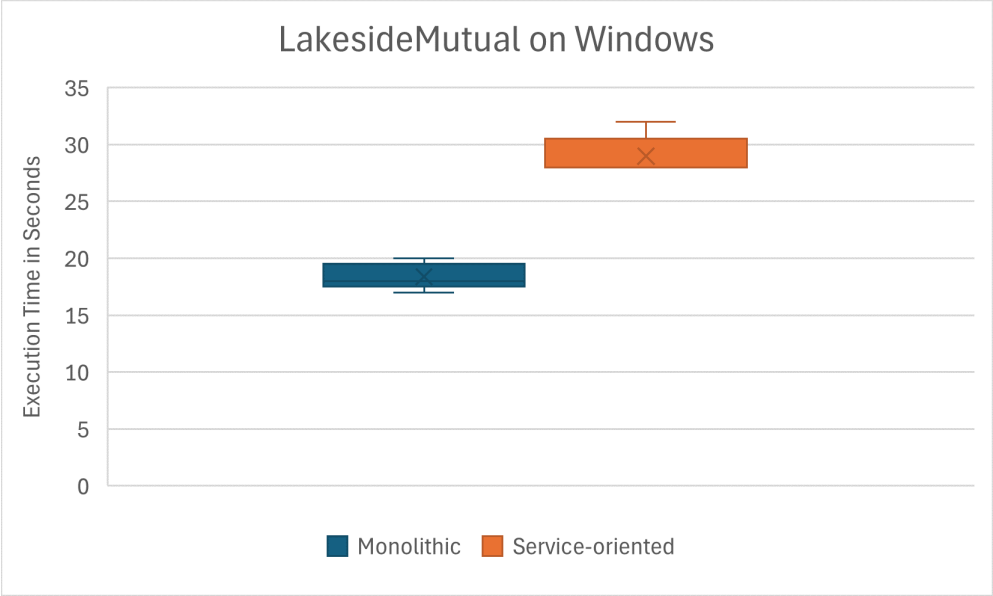

Figure 60. The processing time for all requests sent to the LakesideMutual customer endpoint reported by JMeter on Windows

The vertical axis visualizes the execution time for all requests in seconds. The previous assumption that multiple services are able to handle more requests in parallel is not confirmed by the results. They require about ten seconds more to complete all requests, which is about one third of the total execution time. This should also reflect in the energy consumption of the application.

Figure 61 illustrates the total energy consumption of the LakesideMutual application.

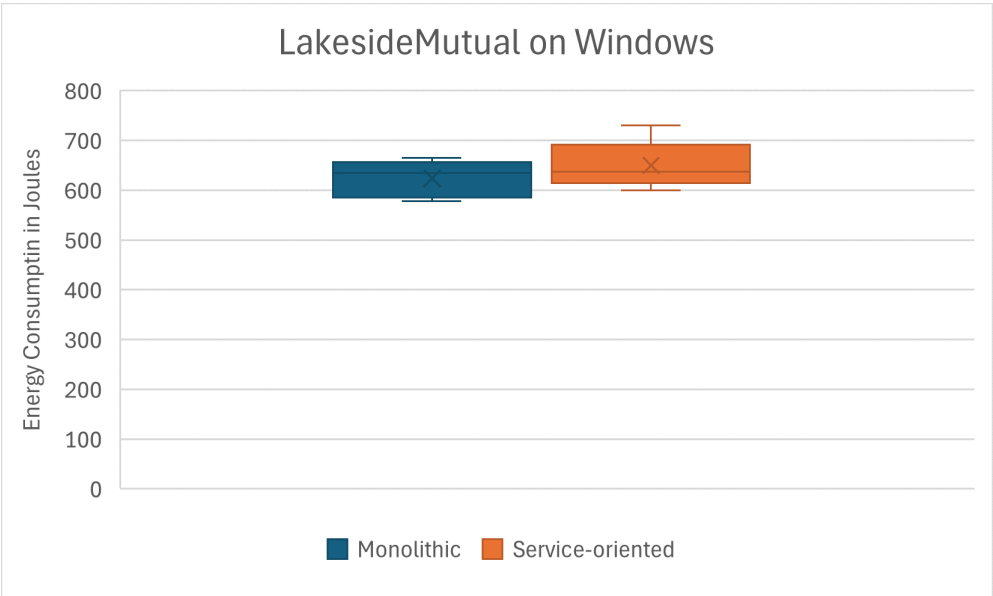

Figure 61. The energy consumption of the LakesideMutual application measured with JoularJX on Windows

The diagram reveals an almost equal median for both scenarios with a slightly higher energy consumption for the service-oriented scenario. This is explainable by the additional network calls between the services. The total energy consumption of the LakesideMutual application aggregates all four running services. Figure 62 splits the total energy consumption into the three main services involved in the customer test.

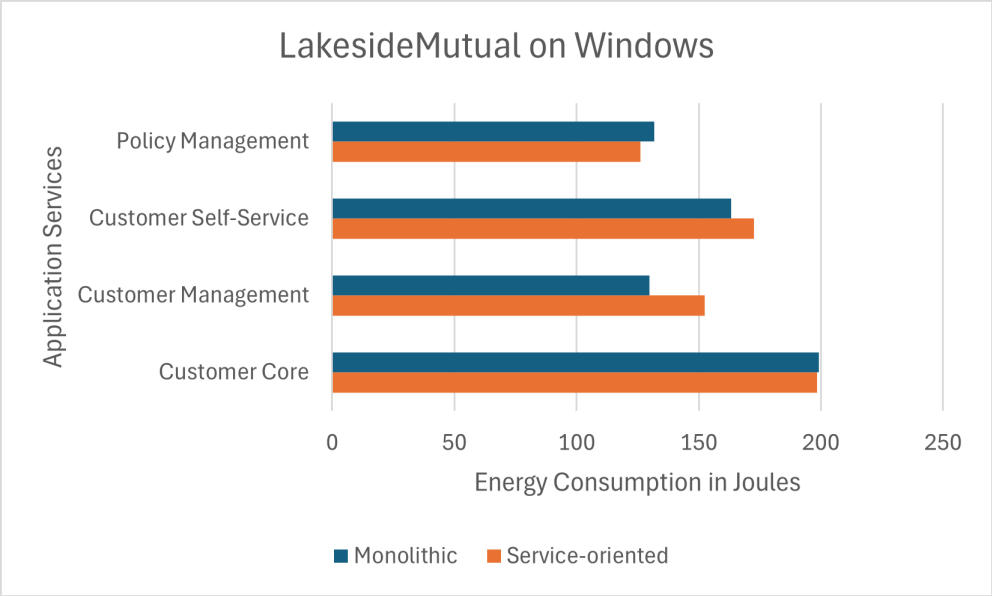

Figure 62. The energy consumption of the LakesideMutual services measured with JoularJX on Windows

This diagram shows the idle energy consumption for the policy management service. The customer core service consumes about the same amount of energy in both scenarios. The service-oriented scenario consumes slightly more energy in the customer self-service and customer management services. This is expected due to the additional network calls between the services.

The customer core, customer management, and customer self-service services have a specific *CustomerInformationHandler* class each. They are responsible to handle the respective customer requests in each service. Each CustomerInformationHolder class has methods to handle customer information and methods to handle the common logic such as creating a response object. Figure 63 visualizes the distribution of energy consumption across all customer operations.

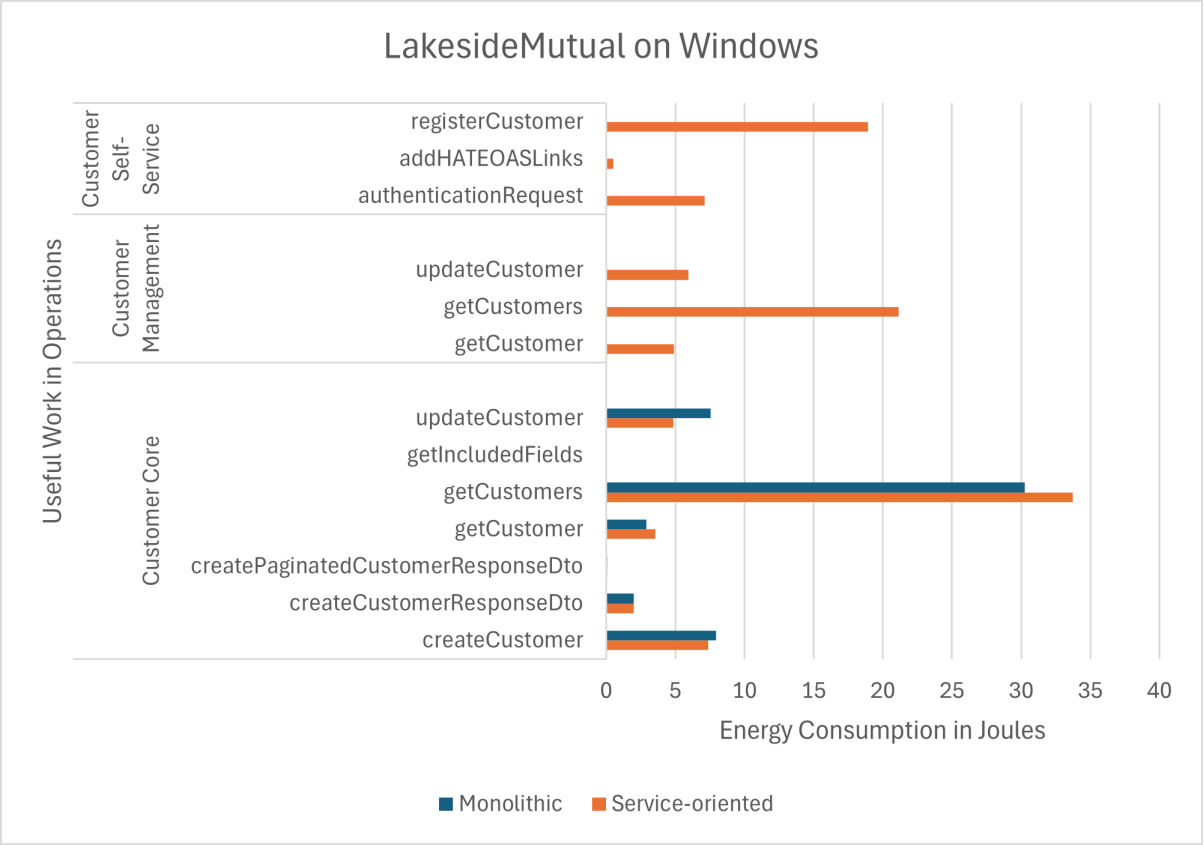

Figure 63. The energy consumption of the LakesideMutual customer operations measured with JoularJX on Windows

The results for the monolithic test scenario confirm that only methods in the customer core service are affected by the test plan. The *getCustomers* operation of the customer core service consumes about 30 Joules, significantly more than the remaining operations. The next most energy-consuming operations are the *updateCustomer* and *createCustomer* operations with about 7 Joules. The helper methods *getIncludedFields*, *createPaginated_CustomerResponse_Dto*, and *createCustomerResponseDto* consume almost negligible amounts of energy.

The service-oriented scenario including the customer management and customer self-service services shows a similar distribution of energy consumption. The *getCustomers* operation consumes the most energy in both the customer core and customer management services. The *updateCustomer* operations and the *createCustomer* or *registerCustomer* operations consume the second most energy. The helper methods consume a negligible amount of energy.

Table 24 lists the energy efficiency of the LakesideMutual application.

Table 24. The energy efficiency of the LakesideMutual application on Windows

Test Execution	**Useful Work (Amount of Requests)**	**Total Energy Consumption**	**Energy Efficiency**
Monolithic	2'000 Requests	623.85 Joules	3.21 Requests per Joule
Service-oriented	2'000 Requests	649.74 Joules	3.08 Requests per Joule

The monolithic scenario consumes slightly less energy than the service-oriented scenario. Therefore, it has a slightly higher energy efficiency of 3.21 requests per Joule. This sums up the experiment results on Windows, Subsection 4.4.2 reports on the experiment results on Linux.

4.4.2. Experiment on Linux

The experiment on Linux has the same requirements and presets as the experiment on Windows in Subsection 4.4.1. It further investigates the effects of a single service call versus multiple service calls. The primary results of interest in this experiment are the differences between a monolithic and a service-oriented architecture.

Table 25 lists the performance results for the customer requests on Linux.

Table 25. The latency and throughput of the LakesideMutual customer endpoint on Linux

Test Execution	Average Latency	Average Throughput
Monolithic	43ms	16.2 Requests per second
Service-oriented	63ms	15.7 Requests per second

The performance results reveal that the Linux environment is able to achieve the same latency as the Windows environment but roughly double the throughput. The throughput of the monolithic scenario is slightly higher on Linux as opposed to Windows.

Figure 64 illustrates the total execution time for all requests.

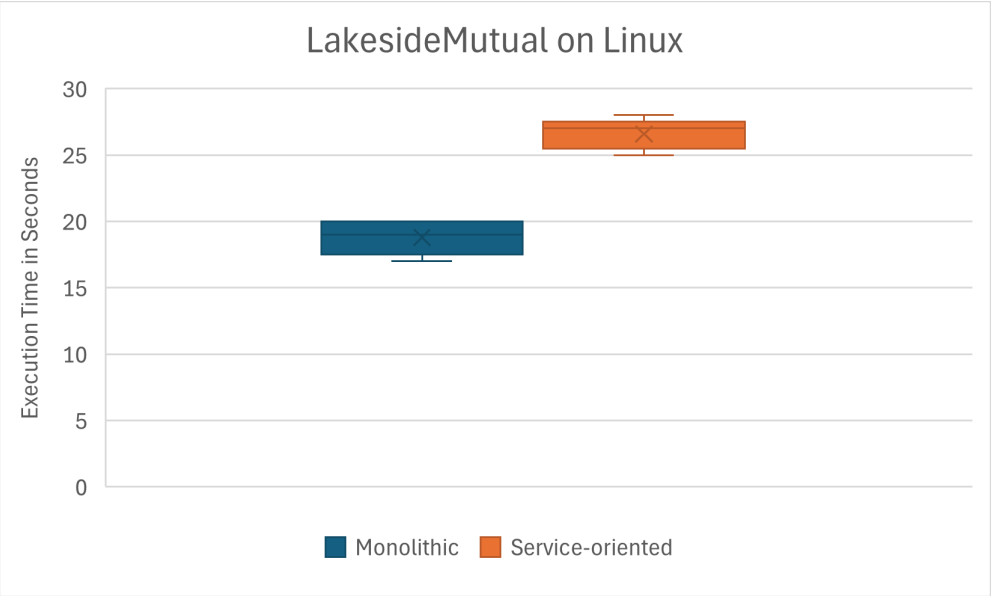

Figure 64. The processing time for all requests sent to the LakesideMutual customer endpoint reported by JMeter on Linux

The vertical axis refers to the execution time in seconds. The diagram is almost identical to the results on Windows, visualized in Figure 60. The service-oriented scenario including the customer management and self-service services requires about five seconds more to complete all requests than the monolithic one.

Figure 65 visualizes the total energy consumption of the LakesideMutual application.

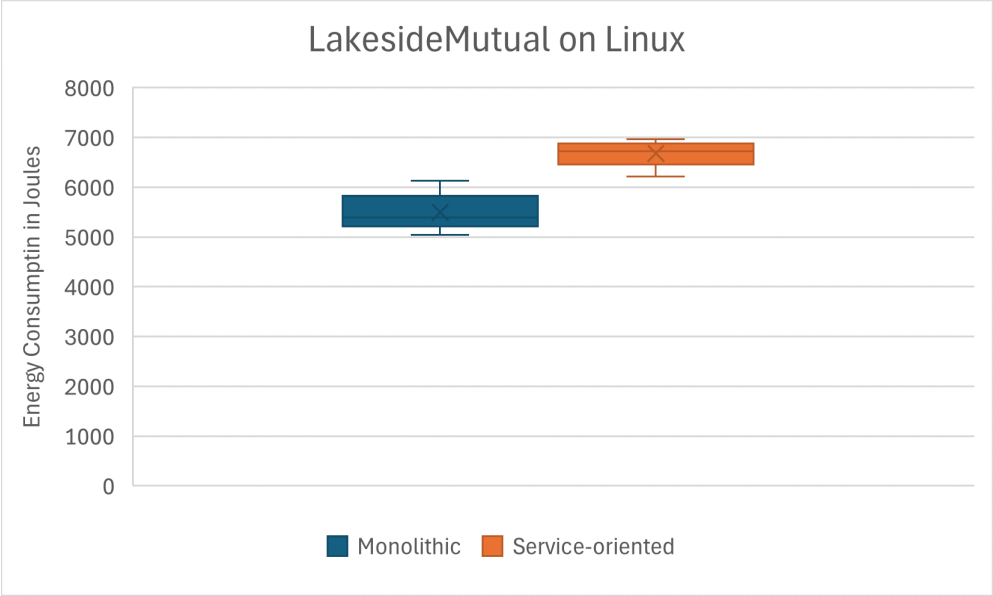

Figure 65. The energy consumption of the LakesideMutual application measured with JoularJX on Linux

The relative distribution of energy consumption for both scenarios is similar to the results on Windows. When it comes to the absolute energy consumption, the experiment on Linux consumes ten times more energy than the experiment on Windows. The results confirm that multiple service calls consume slightly more energy than a single service call.

Figure 66 illustrates the energy consumption of LakesideMutual per service.

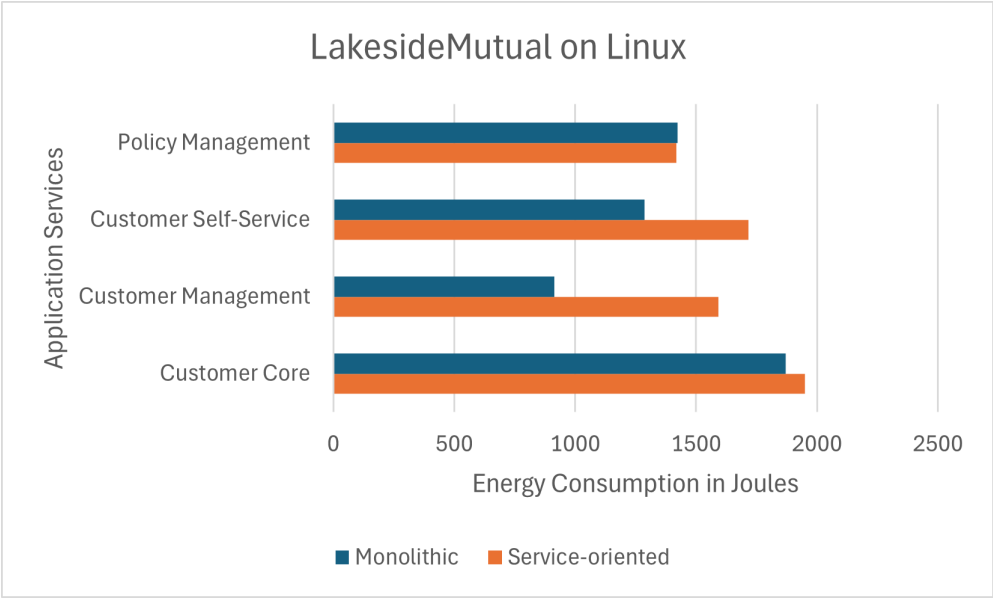

Figure 66. The energy consumption of the LakesideMutual services measured with JoularJX on Linux

The customer core service consumes the most energy in both scenarios. This is expected as the customer core service is responsible for the customer data and handles all requests in the background. The results confirm the observations on Windows. It is expected that the idle policy management service consumes an almost equal amount of energy in both scenarios.

Figure 67 displays a granular distribution of energy consumption across all involved operations.

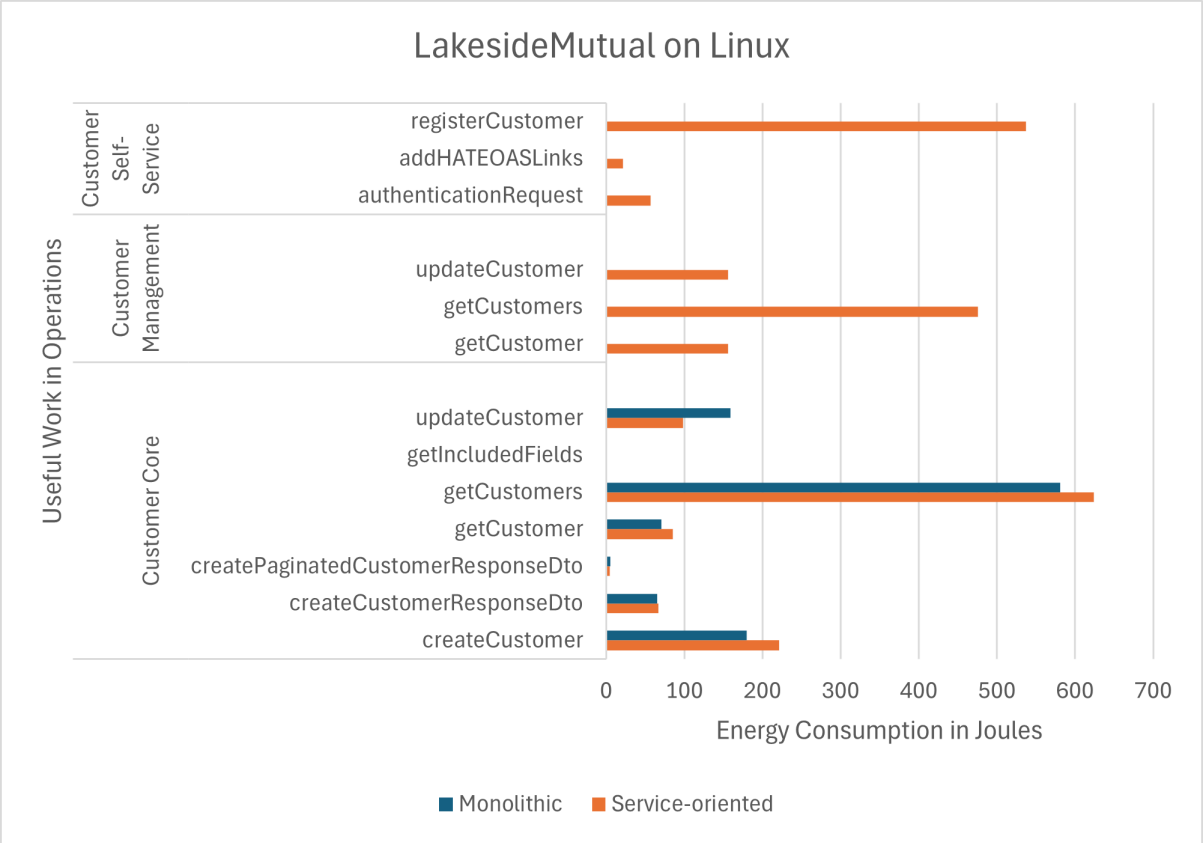

Figure 67. The energy consumption of the LakesideMutual customer operations measured with JoularJX on Linux

The results confirm that in both scenarios the *getCustomers* operation in the customer core and customer management service consumes the most energy. It appears that the *registerCustomer* operation is the key factor for the increased energy consumption in the customer self-service service. It is likely that multiple database queries affect the energy consumption.

The increase in energy consumption for multiple service calls reflects in a decreased energy efficiency. Table 26 lists the total energy consumption and energy efficiency of the LakesideMutual application.

Table 26. The energy efficiency of the LakesideMutual application on Linux

Test Execution	**Useful Work (Amount of Requests)**	**Total Energy Consumption**	**Energy Efficiency**
Monolithic	2'000 Requests	5494.01 Joules	0.36 Requests per Joule
Service-oriented	2'000 Requests	6678.17 Joules	0.30 Requests per Joule

The results confirm that multiple service calls slightly reduce the energy efficiency by about 0.06 requests per Joule. This sums up the experiment results on Linux, Subsection 4.3.3 summarizes and discusses the experiment findings.

4.4.3. Experiment Summary

The experiments using LakesideMutual aim to reproduce the master data measurements on the PetClinic application and compare a monolithic and a service-oriented architecture. The results reveal great similarities between the two test scenarios and between the two test environments.

Section 4.1 and Section 4.2 indicated that absolute numbers can not be compared within the same application across system boundaries. When it comes to latency and total execution time, this experiment indicates that absolute numbers can be compared across different test environments for the same application. The latency between the two test scenarios is almost equal in both test environments. Using multiple services consistently results in a higher latency. This is expected due to the additional network calls between the services. In case the services are deployed on different servers, potentially farther apart, the latency is expected to increase significantly. The overall execution time is slightly higher on Windows than on Linux, but the distribution is almost equal. Using multiple services is slightly slower than using a single service on both test environments.

However, absolute numbers for throughput and energy consumption still reveal significant differences across system boundaries. Using multiple services is slightly faster in terms of throughput than using a single service on Windows. On Linux, the opposite is true. The results do not allow to draw a conclusion whether the throughput is better for a single service or multiple services. The total energy consumption is significantly higher on Linux than on Windows.

The relative distribution of energy consumption is similar for both test scenarios across both test environments. This suggests that the application itself is comparable across system boundaries. Unsurprisingly, the customer core service consumes the most energy in both test scenarios. This is expected as the other two services depend on the customer core to handle the customer data. It is interesting that the power consumption of the customer self-service heavily depends on the *registerCustomer* operation. Chances are that not only the amount of entities handled by the service, but also the amount of database queries has a massive impact on the energy consumption.

The margins in this experiment are too small to draw a conclusion about the correlation between performance and energy efficiency. One could argue that this resembles the hypothesis that there is no correlation between performance and energy efficiency. We specify that this experiment does not provide sufficient evidence to support this claim. Especially with a service-oriented architecture, the performance is highly dependent on the network and would require further investigation.

All experiments so far solely focused on simple CRU(D) operations. The last experiment utilizes the LakesideMutual application to measure a more complex, business relevant workflow. Section 4.5 reports on the results and compares the findings.

4.5. LakesideMutual Experiment: Compare Workflow Variants

This experiment aims to measure the performance and energy efficiency of the LakesideMutual application with a focus on complex business workflows instead of CRUD operations. The test plan is designed to perform a workflow simulating interactions between customers and policy managers. The workflow can end in three different outcomes, the experiment executes a separate test scenario for each outcome and compares the results. The workflow ends either when a policy manager rejects the insurance quote

request, when the customer rejects the insurance offer made by the policy manager, or when the customer accepts the insurance offer. Accepting the offer results in a successful insurance policy creation and leads to further processing steps. The experiment is solely conducted with resource constraints to reduce the complexity when comparing the results.

4.5.1. Experiment on Windows

The primary results of interest in this experiment are the comparability of performance and relative energy distribution in complex business workflows. The different test plans do not allow for a direct comparison with other experiments, but the results may indicate similarities between them.

Table 27 lists the performance results of the LakesideMutual workflows on Windows.

Table 27. The latency and throughput of the LakesideMutual Workflows on Windows

Test Execution	Average Latency	Average Throughput
Reject Insurance Quote	211ms	5.0 Requests per second
Reject Insurance Offer	229ms	7.1 Requests per second
Accept Insurance Offer	305ms	8.2 Requests per second

The results show a slight increase in latency and throughput for each subsequent workflow. It appears that an increase of requests leads to a higher latency while the system is able to process more requests overall. This is explainable by the fact that many small requests can be processed faster than a few large requests.

The results indicate that the quote rejection scenario should be the first one to finish. Figure 68 illustrates the total execution time for all requests sent to the LakesideMutual application.

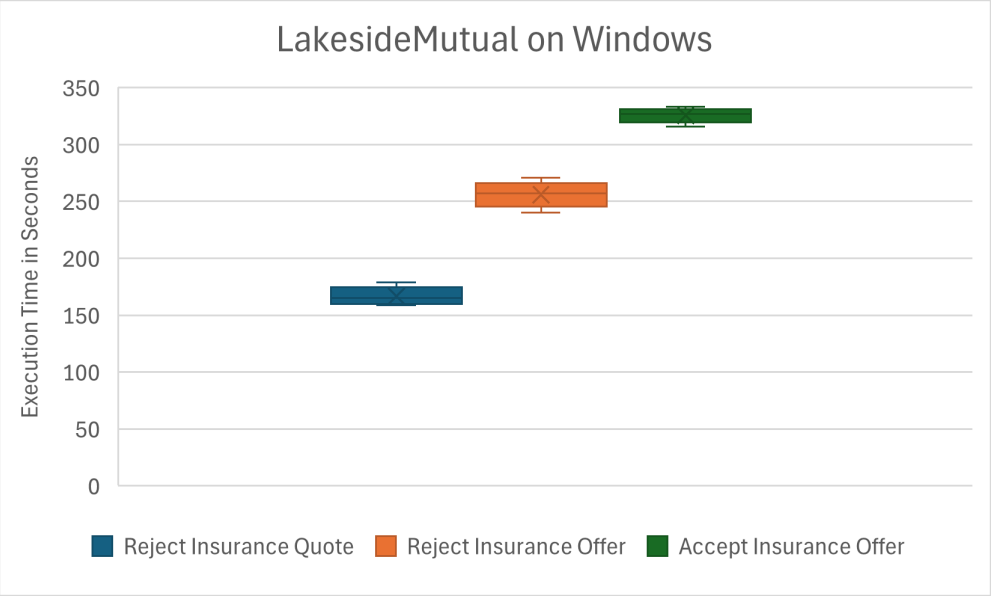

Figure 68. The processing time for all requests sent to the LakesideMutual application reported by JMeter on Windows

The diagram illustrates that the quote rejection scenario is the first one to finish processing the requests. The offer rejection scenario, which executes more requests and has a higher latency, is the second one to finish the requests. The offer acceptance scenario, which creates policies and has the highest latency, is the last one to finish processing the requests.

This raises the question of how the energy consumption is affected by the different workflows. Figure 69 visualizes the energy consumption of each scenario.

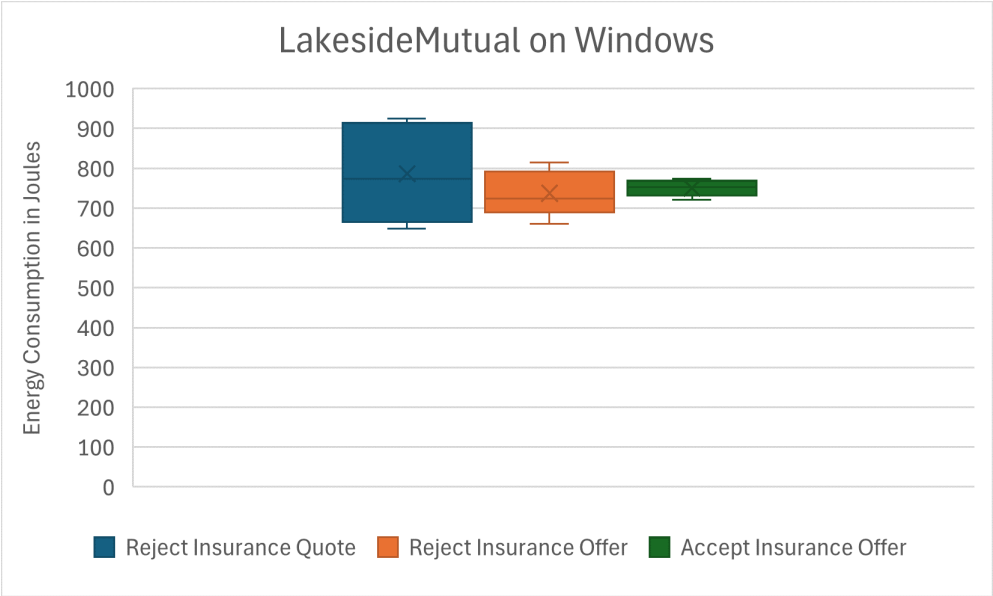

Figure 69. The energy consumption of the LakesideMutual application measured with JoularJX on Windows

The box plot reveals a surprising result. The quote rejection scenario, which executes the least amount of requests, has the highest median value and largest third quartile range. This suggests that this scenario consumes more energy than the other two scenarios.

It is possible to evaluate which service consumes the most energy, especially in the quote rejection scenario. Figure 70 illustrates the energy consumption for each LakesideMutual service across all three scenarios.

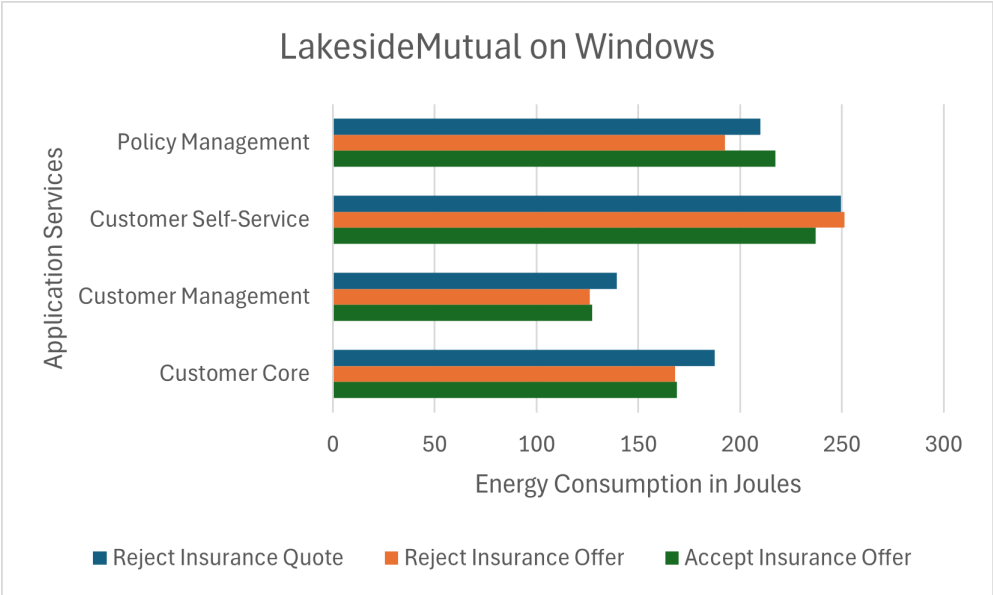

Figure 70. The energy consumption of the LakesideMutual services measured with JoularJX on Windows

It appears that the quote rejection scenario consumes more energy in the customer core and customer management services than the other two scenarios. The results show that the relative distribution of energy consumption between the services is similar across all three scenarios. A more granular analysis of the energy distribution is required to determine the cause of the high energy consumption in the quote

rejection scenario. Figure 71 provides a detailed analysis for the distribution of energy across all operations involved.

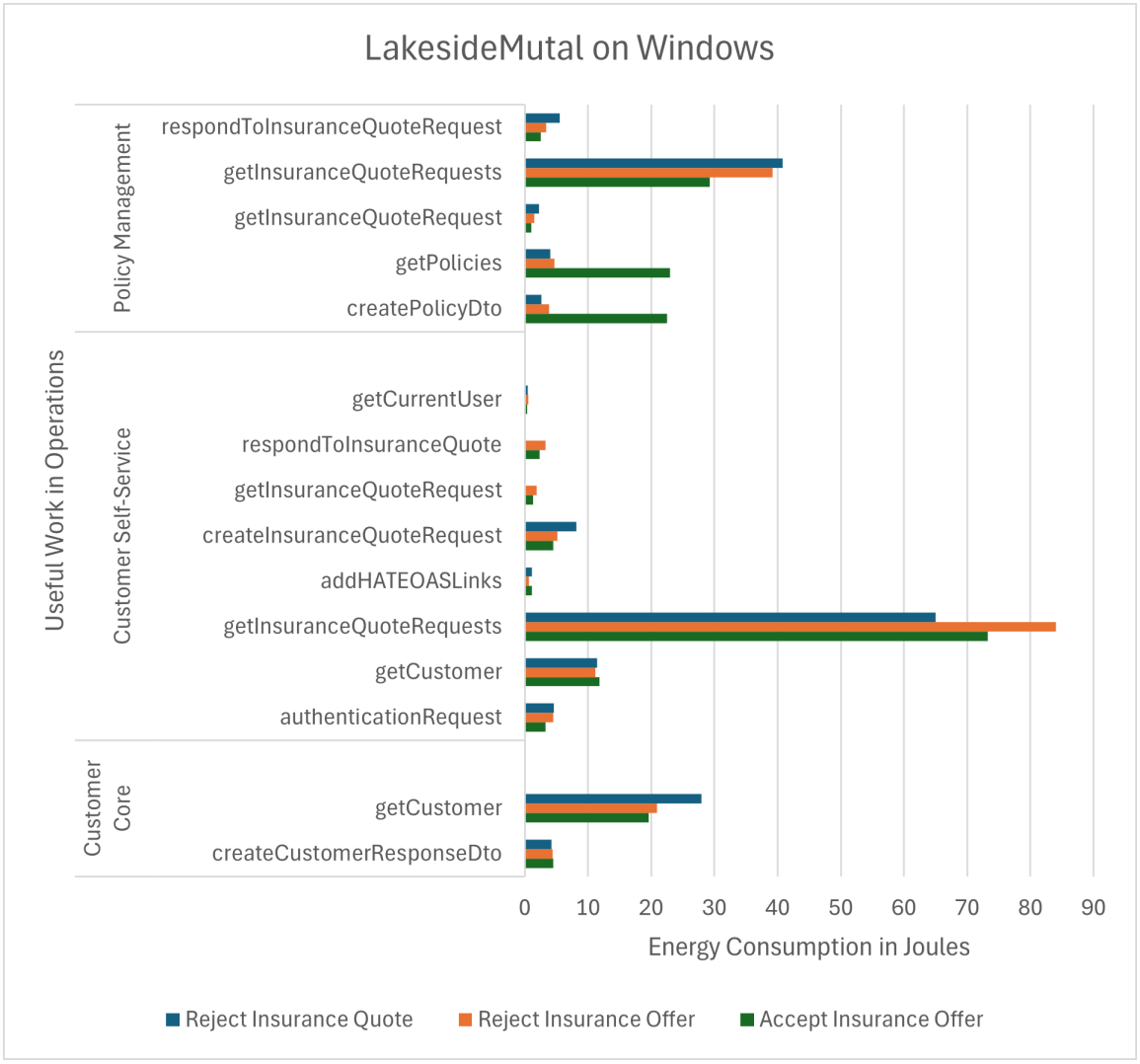

Figure 71. The energy consumption of the LakesideMutual operations measured with JoularJX on Windows

The results suggest that the quote rejection test scenario consumes significantly more energy for the *getCustomer* operation. The test executions do not provide a clear explanation for this behaviour. The results further indicate that the offer acceptance scenario consumes a lot more energy for the *createPolicyDto* and *getPolicies* operation.

The *getInsuranceQuoteRequests* operations consume the largest amount of energy in all three scenarios. This behaviour is consistent with the previous experiments as multiple entities are handled in a single request.

The difference in energy consumption can be misleading as the quote rejection scenario executes fewer requests due to the fact that the workflow finishes earlier. It is important to compare the energy consumption of the workflows with the amount of requests processed. Table 28 lists the amount of requests processed, the total energy consumed, and the energy efficiency for each test scenario.

Table 28. The energy efficiency of the LakesideMutual application on Windows

Test Execution	Useful Work (Amount of Requests)	Total Energy Consumption	Energy Efficiency
Reject Insurance Quote	2'696 Requests	786.41 Joules	3.43 Requests per Joule
Reject Insurance Offer	4411 Requests	737.63 Joules	5.98 Requests per Joule
Accept Insurance Offer	4411 Requests	750.71 Joules	5.88 Requests per Joule

The results confirm that the quote rejection test scenario consumes the most energy and is the least energy efficient. It remains unclear why this scenario consumes more energy for fewer requests. Interestingly, the offer rejection scenario is the most energy efficient. This is explainable by the additional processing steps that are required in the offer acceptance scenario to create a policy after accepting the insurance offer. This sums up the experiment results on Windows, Subsection 4.5.2 reports on the experiment results on Linux.

4.5.2. Experiment on Linux

The experiment on Windows raises one big question: Why does the quote rejection scenario with the fewest requests consume the most energy? The experiments on Linux aim to reproduce the results and to find an explanation for the unexpected behaviour. The results of primary interest is the relative distribution of energy consumption across test scenarios, services, and operations.

Table 29 lists the performance results of the LakesideMutual workflows on Linux.

Table 29. The latency and throughput of the LakesideMutual Workflows on Linux

Test Execution	Average Latency	Average Throughput
Reject Insurance Quote	144ms	11.7 Requests per second
Reject Insurance Offer	148ms	15.5 Requests per second
Accept Insurance Offer	184ms	13.9 Requests per second

The results indicate similar behaviour for LakesideMutual as the experiment on Windows. The latency and throughput increases slightly for each subsequent scenario. However, on Linux, the offer rejection scenario has the highest throughput, as opposed to the offer acceptance scenario on Windows.

Figure 72 illustrates the total execution time for all requests.

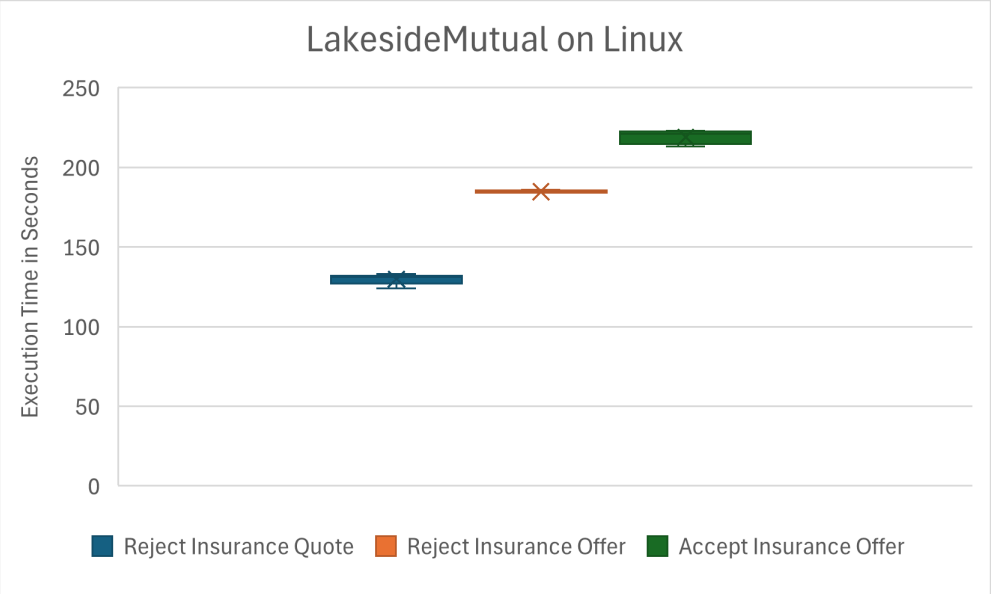

Figure 72. The processing time for all requests sent to the LakesideMutual application reported by JMeter on Linux

The results reveal an equivalent behaviour as the experiment on Windows. The box plots have a slightly more narrow distribution than the results on Windows.

Figure 73 illustrates the total energy consumption for each test scenario.

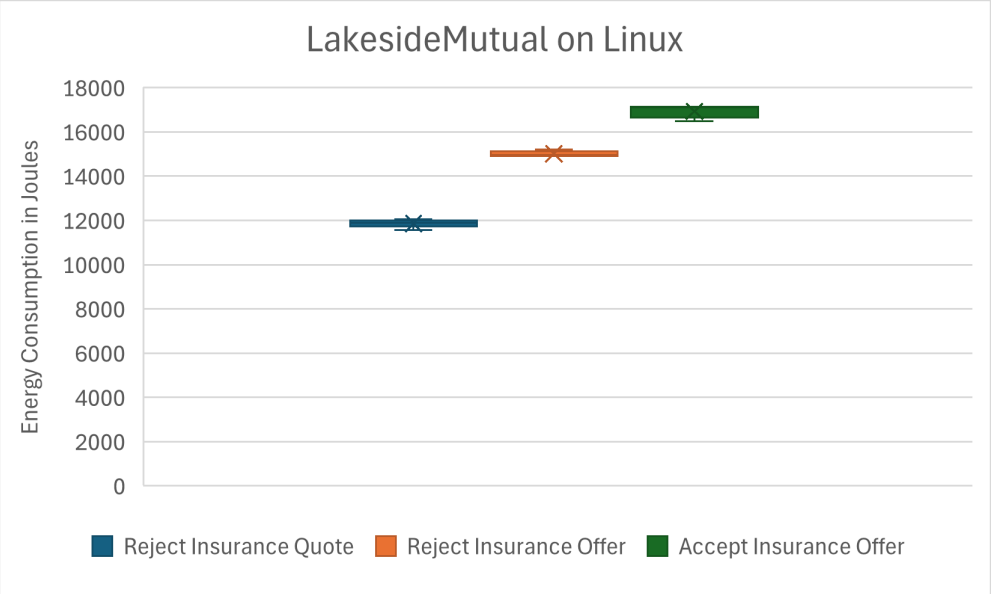

Figure 73. The energy consumption of the LakesideMutual application measured with JoularJX on Linux

The box plot reveals similarities to the distribution of execution times shown in Figure 72. The result on Linux appear to be more intuitive as the ones on Windows visualized in Figure 69. The quote rejection scenario with the least amount of requests consumes the least amount of energy. The offer rejection scenario has a medium energy consumption. The offer acceptance scenario, which subsequently creates policies, consumes the most energy.

This raises the question if the relative distribution of energy consumption is significantly different between the Linux and Windows environments. Figure 74 shows the energy consumption of each LakesideMutual service across all three scenarios.

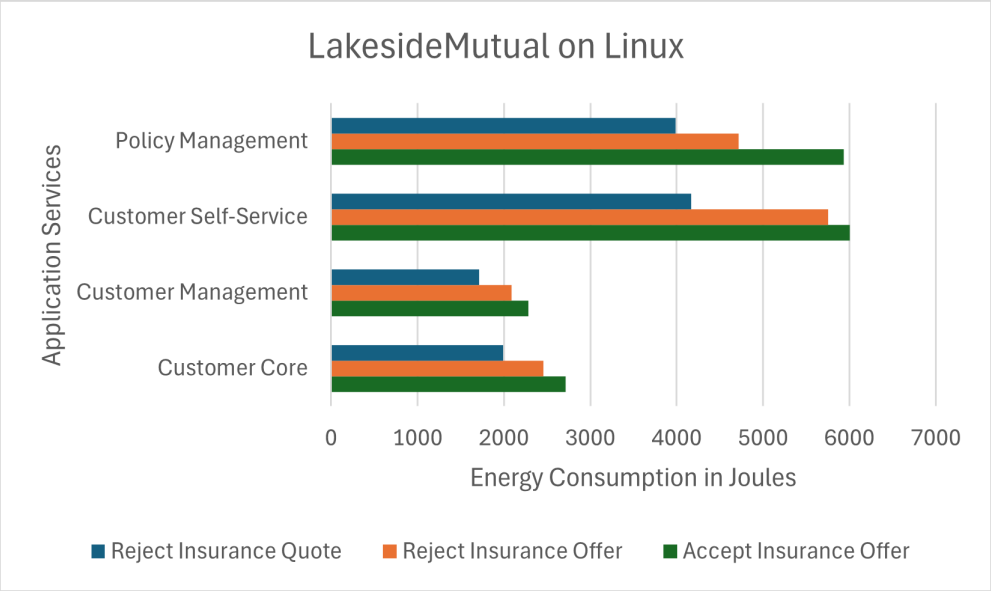

Figure 74. The energy consumption of the LakesideMutual services measured with JoularJX on Linux

A comparison between this figure and Figure 70 does not reveal fundamental differences. The relative distribution between the services is similar across all three scenarios. The diagram visualizes that the energy distribution is similar even within each service. The quote rejection scenario consumes the least amount of energy across all services while the offer acceptance scenario consumes the most energy.

Figure 75 provides a detailed analysis for the distribution of energy across all operations involved.

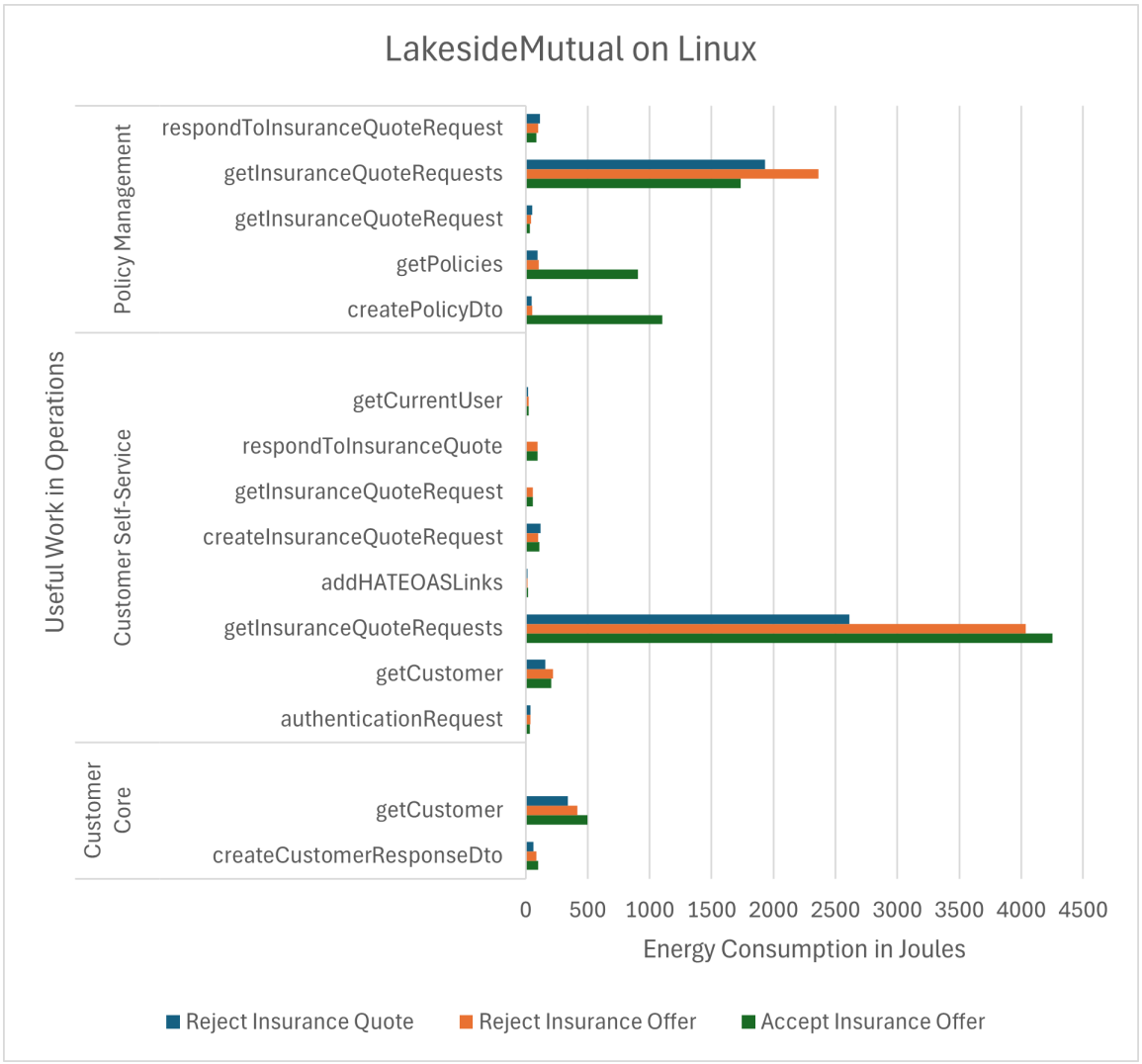

Figure 75. The energy consumption of the LakesideMutual operations measured with JoularJX on Linux

The results show a similar distribution of energy consumption as the experiment on Windows. It appears that the main difference between Linux and Windows is the energy consumption for *getCustomer* in the customer core service. On Windows, the quote rejection scenario consumes significantly more energy during the *getCustomer* operation as shown in Figure 71. The reason for this behaviour is unclear and would require further investigation.

Table 30 lists the energy efficiency of the LakesideMutual application on Linux.

Table 30. The energy efficiency of the LakesideMutual application on Linux

Test Execution	Useful Work (Amount of Requests)	Total Energy Consumption	Energy Efficiency
Reject Insurance Quote	2'696 Requests	11869.54 Joules	0.23 Requests per Joule
Reject Insurance Offer	4411 Requests	15014.87 Joules	0.29 Requests per Joule
Accept Insurance Offer	4411 Requests	16932.13 Joules	0.26 Requests per Joule

The results are surprisingly similar to the experiment on Windows. The quote rejection scenario has the worst energy efficiency even though it consumes the least amount of energy. The offer rejection scenario is more energy efficient than the offer acceptance scenario, however, the margins between the three scenarios are small. This sums up the experiment results on Linux, Subsection 4.5.3 summarizes and discusses the findings of the LakesideMutual workflow experiments.

4.5.3. Experiment Summary

The experiments described in this section can not be compared directly with the previous experiments. The previous experiments focused on CRU(D) operations while this experiment focuses on complex business workflows. The previous experiments already indicated that absolute numbers are not comparable across different systems and applications. It appears that even the same application on the same system can produce deviating results depending on the test scenario.

Figure 76 aggregates and compares the total energy consumption for the previous experiments.

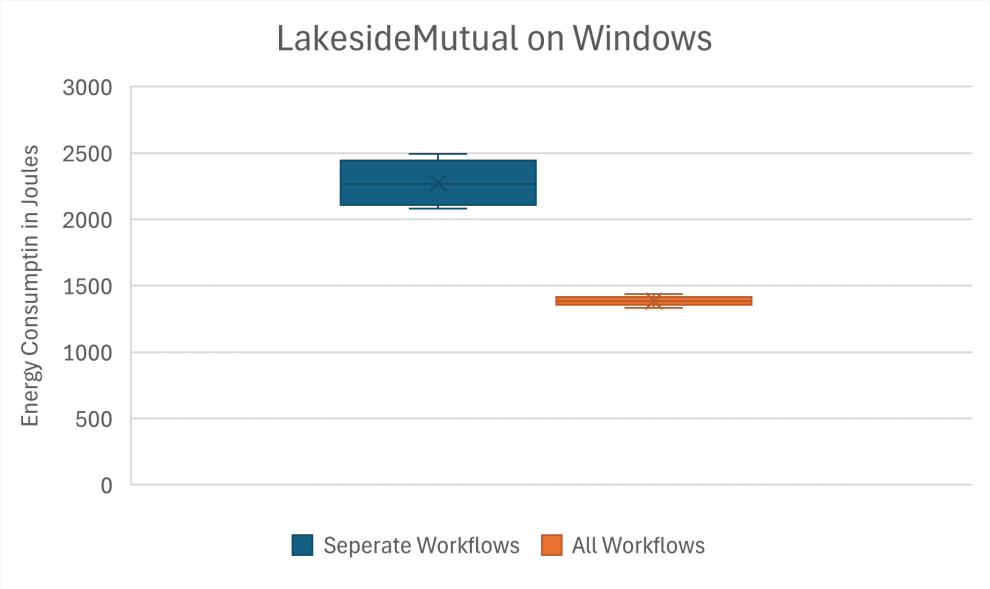

Figure 76. A comparison of aggregated energy consumption of the LakesideMutual application on Windows

The first box plot shows the aggregated total energy consumption for all three previously established test

executions. The second box plot shows the total energy consumption for the same amount of requests performed in one test execution. Surprisingly, even though the same amount of requests is processed, the energy consumption is significantly different. When all requests are executed in one test execution without starting and stopping the application, the energy consumption is significantly lower. Figure 77 visualizes the same results for the Linux environment.

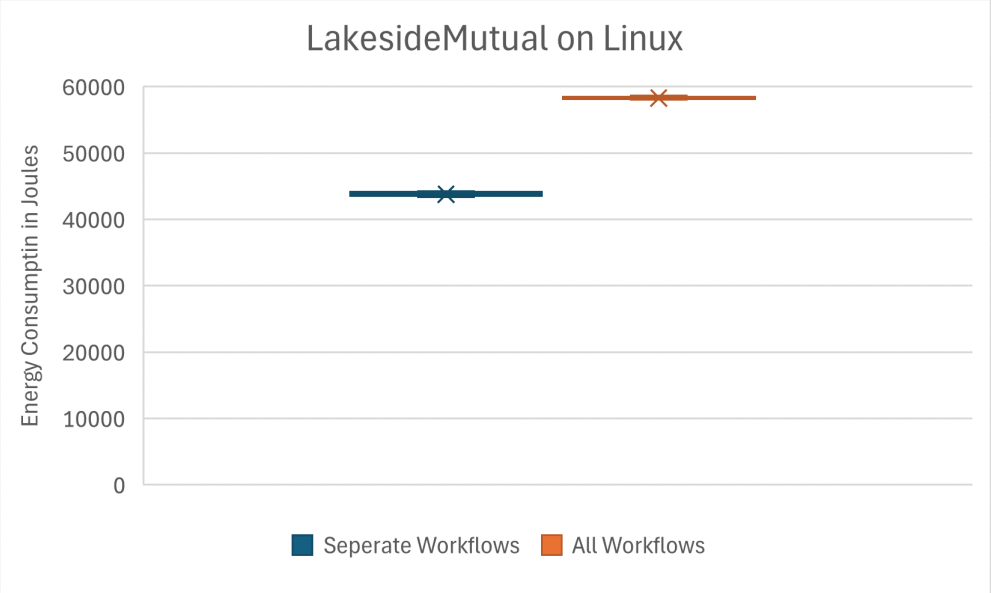

Figure 77. A comparison of aggregated energy consumption of the LakesideMutual application on Linux

The diagram reveals that the observed behaviour on Windows is the polar opposite on Linux. The second box plot illustrates that the energy consumption is significantly higher when all requests are executed in one test execution. The results indicate that neither different applications, nor different systems, nor different test plans with equal requests can be compared directly.

On the other hand, the results in this experiment confirm that the relative distribution of energy consumption can be compared not only across different systems, but also across different applications and even across different types of useful work. The results suggest that complicated business processes have similar energy distribution patterns as CRUD operations. It appears that the main factors for energy consumption are the amount of entities processed per request and the amount of database accesses required. Chances are that implementation details, such as database sequence allocation size, the database access technology, or algorithms and data structures have a significant impact on the energy consumption.

This summarizes the LakesideMutual experiment and its results. It is also the last experiment in this thesis. Chapter 5 considers the findings of all experiments, compares them with each other, and discusses the results in detail.

5. Discussion

This chapter analyses, interprets, and compares the results of the experiments conducted in this thesis. It then generalizes the findings for other Spring Boot applications, other web-based applications, and software-intensive systems in general. It provides a list of related work and reflects on the research objectives and achievements, on the challenges faced during the experiments, and the reviewed literature. Eventually, it provides an outlook on possible next steps in the fields of software measurement and energy-efficient software.

5.1. Analysis and Interpretation of Measurement Results

This section analyzes the experimental results presented in Chapter 4 and discusses their implications. The discussion alternates between the findings of this thesis and the evidence from additional literature to support the findings. Key takeaways are highlighted in the text and summarized at the end of each subsection.

5.1.1. Performance

All experiments report deviating performance measurements across different enterprise applications and test environments. To examine on one specific example, the results report varying performance measurements for the PetClinic application on Windows and Linux.

Table 31. The latency and throughput deviations across experiments on Windows and Linux

Test Plan	Latency	Throughput
PetClinic All Entities CRUD on Windows	652ms	6.3 Requests per second
PetClinic All Entities CRUD on Linux	351ms	13.4 Requests per second
PetClinic Owner CRUD on Windows	526ms	6.1 Requests per second
PetClinic Owner CRUD on Linux	321ms	10.6 Requests per second
PetClinic Owner CRU + Spring Data JPA on Windows	796ms	3.2 Requests per second
PetClinic Owner CRU + Spring Data JPA on Linux	470ms	8.4 Requests per second

The Linux environment achieves a better performance in terms of latency and throughput than Windows. The performance measurements exhibit similar evolution patterns across the test plans. The first two rows executing the entire test plan achieve a higher latency and throughput on both environments compared to row three and four executing the owner test plan. When comparing row three and four to row five and six, the latency increases on both test environments, while the throughput decreases. The absolute numbers alone do not provide sufficient evidence on why performance evolves the way it does.

The author of the Growing Green Software blog states that a test scenario should not be compared across system boundaries. He experienced similar deviations, as observed in this thesis, with other systems and suggested to compare results solely within the same system. His statement refers to absolute numbers in general, but especially to performance comparisons.

Absolute numbers can not be compared across different test environments.

Azimi et al. investigate an enhanced operating system support for multicore processors [106]. They conclude that the hardware resources, in their case the processor, and the operating system have a significant impact on the performance and energy efficiency of applications. Becker and Chakraborty investigate an Intel processor its microarchitecture and its impact on performance measurements [107]. Their experiments confirm that measurement setups can significantly impact performance measurements depending on the selected analysis method.

The literature confirms that even minor changes in the processor architecture and operating system can affect the performance and energy efficiency of applications. This thesis relies on two entirely different hardware resources and operating systems to evaluate whether the results can be compared even across system boundaries. It appears that the performance results are not comparable due to the variations; both, the GGS blog author and the literature, support this finding.

Different hardware resources and operating systems can significantly affect the performance of an application.

Another experimental finding refers to the impact of Spring Data JPA on performance compared to native JPA. The PetClinic measurements reveal lower performance results when using Spring Data JPA instead of JPA on the same test environment. Spring Data JPA builds on top of JPA and adds complexity and overhead to the database access mechanism to facilitate the database access for software developers.

Bonteanu and Tudose measure and compare JPA, Hibernate, and Spring Data JPA against well-known databases, such as MySQL, Oracle, SQLServer, and PostgreSQL [108]. They conclude that Hibernate and JPA achieve similar performance results and confirm that Spring Data JPA comes with additional overhead. Colley, Stanier, and Asaduzzaman investigate on the impact of object-relational mapping (ORM) frameworks on performance [109]. They conclude that ORM frameworks negatively affect performance in terms of query execution duration. They identify problematic areas and provide potential mitigations, but many mitigations include configuring the ORM, adapting the query, or tuning the database.

The selected ORM framework can significantly affect the performance of an application.

The high loads during the experiments generated excessive heat on the Windows device. Operating systems apply thermal throttling to prevent overheating by reducing the processor clock speed. Rao and Vrudhula elaborate on optimal processor throttling under thermal conditions [110].

The impact of thermal throttling on processors [110]

> The throttling mechanism allows the processor to gracefully handle workloads with a mix of high and low power tasks by running low power tasks at full speeds, and the more intense ones at lower speeds. The ideal DTM strategy maintains the chip temperature at or below the specified maximum with minimum performance loss due to throttling.
>
> — Rao and Vrudhula

Thermal throttling mechanisms can have an impact on performance measurements and should be considered when executing extensive performance tests.

Subsection 2.2.1 refers to academic literature and grey literature to establish latency, throughput, and the average execution time as a common denominator for performance measurements. The selected approach does not provide sufficient evidence to compare performance results. It appears necessary to conduct multiple performance measurements from different perspectives to gain a holistic view of the performance of an application. Different configurations and metrics provide additional clues to compare, interpret, and understand performance measurements.

Figure 78 lists important parameters for performance testing scenarios in distributed software applications, proposed by Denaro, Polini, and Emmerich [6].

Workload	Number of clients Client request frequency Client request arrival rate Duration of the test
Physical resources	Number and speed of CPU(s) Speed of disks Network bandwidth
Middleware configuration	Thread pool size Database connection pool size Application component cache size JVM heap size Message queue buffer size Message queue persistence
Application specific	Interactions with the middleware - use of transaction management - use of the security service - component replication - component migration Interactions among components - remote method calls - asynchronous message deliveries Interactions with persistent data - database accesses

Figure 78. Parameters for performance testing in a distributed software application [6]

They state that performance testing can yield different results depending on the usage of services, middleware, and deployment environments. The authors suggest that a performance test should consider application-specific use cases, such that the most critically interactions are covered. When applying performance measurements in practice, it is important to define which aspects of an enterprise application shall be evaluated. Furthermore, Denaro et al. emphasize on early and continuous performance testing to avoid performance problems in later stages of the software development process [6]. They conclude that empirical testing outperforms performance estimation models.

Multiple metrics and parameters should be considered to evaluate the performance of an application. Performance measurements should be conducted early and continuously throughout the software development process.

Key takeaways about performance:

- Absolute performance numbers are not comparable across different test environments.
- Performance measurements can be affected by the hardware resources and operating systems.
- Performance measurements can be affected by the selected ORM framework.
- Performance measurements can be affected by thermal throttling mechanisms.
- Latency, throughput, and average execution time are not sufficient to gain a holistic view of the performance of an application.

- Additional metrics and parameters should be considered to evaluate the performance of an application.
- Performance measurements should be conducted early and continuously throughout the software development process.

5.1.2. Resource and Energy Efficiency

The previous Performance Subsection 5.1.1 suggests that absolute performance numbers are not comparable across different test environments. The experimental results reveal that numerical values for energy consumption are not comparable either.

Capra et al. investigate the energy consumption of management information systems (MIS) in the context of ERPs, CRMs and DBMS [111]. The authors state that different management information systems significantly deviate in their energy consumption [57]. The experiments were conducted on a Windows and a Linux test environment, the authors report remarkable differences between the two operating systems. The authors conclude that the infrastructure layer, such as operating systems or Java Virtual Machines, affects the energy efficiency of applications.

Anagnostopoulou, Dimitrov, and Doshi confirm that the power management of operating systems can affect the energy savings for enterprise servers [112]. The article refers to a Linux test setup and does not provide further information on other operating systems. Chances are that different operating systems use different power management mechanisms, affecting the energy efficiency respectively.

Absolute energy consumption numbers are not comparable across different test environments.

The experiments reveal similar relative distributions of energy consumption across different operating systems, different enterprise applications, and even across different sets of operations (useful work). A comparison between the initial PetClinic experiment and the GGS blog results reveals the same relative distribution of energy consumption. The CRU operations on the PetClinic and LakesideMutual confirm that *listOwner* and *listCustomer* consume the most energy. Even the LakesideMutual insurance policy workflow allows to observe similar results.

The relative distribution of energy consumption remains consistent and can be compared across different test environments, enterprise applications, and sets of operations (useful work).

The PetClinic and LakesideMutual measurements reveal an increase in energy consumption for operations that read all records from a database table. The *listOwner*, *listPet*, *listVet*, and *litVisit* operations of the PetClinic application are examples for this behaviour. The *getCustomers*, *getInsuranceQuoteRequests*, and *getPolicies* operations of LakesideMutual consume significantly more energy than the remaining operations. The results suggest that the amount of energy consumption correlates with the amount of records fetched from the database.

Bonteanu and Tudose investigate the execution time for create, read, update, and delete operations on well-known databases [108]. They conclude that each database has its own performance characteristics and that operations can achieve different results on different databases. Figure 79 visualizes that different databases can be optimized for specific operations and may not perform well for other operations.

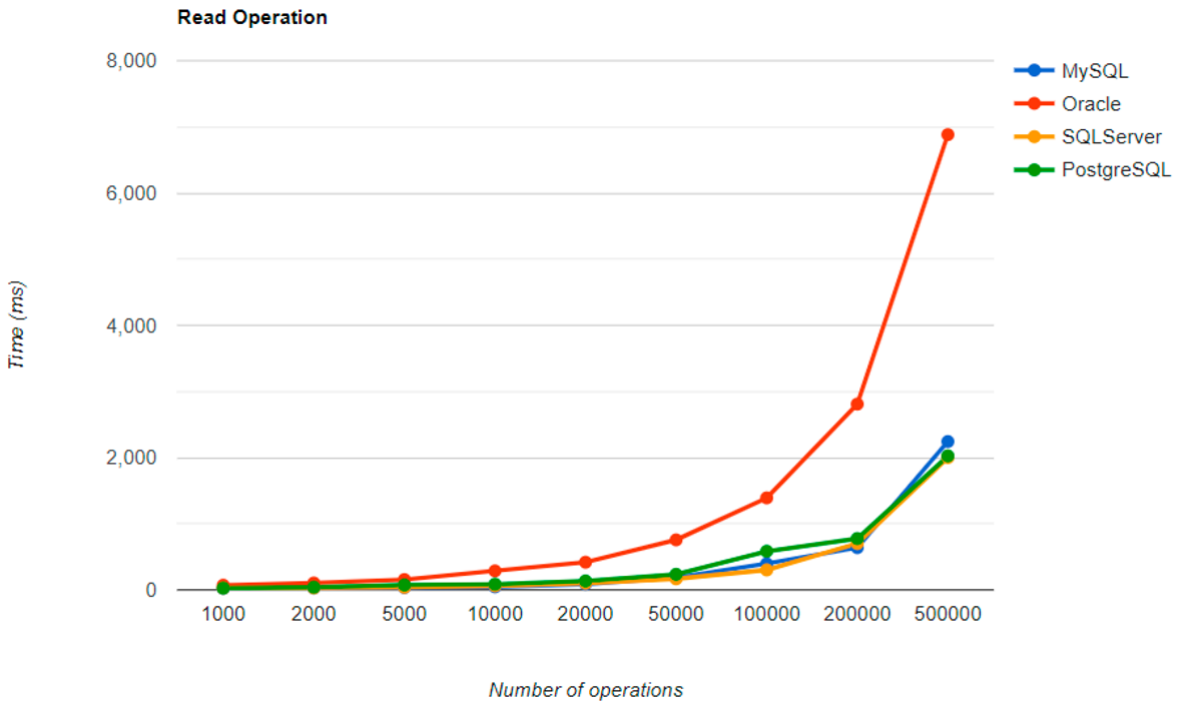

Figure 79. The execution time for the read operation on multiple databases using Spring Data JPA [108]

The graph shows an exponential increase in execution time for the read operations on all databases. The execution time for the MySQL database is around 2'000 milliseconds for 50'000 entries, while the Oracle database takes three times as long.

According to Bonteanu and Tudose, the read operation is the most efficient operation on a database. Their results suggest that the create and update operations are the worst performer on a MySQL database depending on whether JPA, Hibernate, or Spring Data JPA is used. In contrast, the results of this thesis suggest that fetching all records from a database table is the worst performing operation in an enterprise application. It is important to note that their results are based on performing all operations for 50'000 entries, but it remains unclear how they perform the read operations. Chances are that they perform one read operation at the time to fetch one record by its primary key. In our experiments, one operation fetches all records from the database, as opposed to just one record.

The amount of records fetched from the database affects the execution time and therefore the energy consumption of operations.

Furthermore, the LakesideMutual measurements reveal that the *registerCustomer* operation consumes a substantial amount of energy. The implementation performs multiple database calls to store the customer and fetch the logged-in user. Listing 12 shows the implementation storing a new customer in the database, then fetching an already logged-in customer via email, and eventually storing the logged-in user with an updated customer id.

Listing 12. The implementation of the registerCustomer operation in the LakesideMutual application

```
public ResponseEntity<CustomerDto> registerCustomer(...) {
    ...
    CustomerDto customer = customerCoreRemoteProxy.createCustomer(dto);
    UserLoginEntity loggedInUser = userLoginRepository.findByEmail(loggedInUserEmail);
    loggedInUser.setCustomerId(new CustomerId(customer.getCustomerId()));
    userLoginRepository.save(loggedInUser);
    ...
}
```

The results suggest that the energy consumption of an arbitrary operation in the context of enterprise applications heavily depends on database-related aspects; including but not limited to the amount of requests sent to the database, the amount of records fetched from the database, the allocation size of the database sequence, the amount of records stored when performing bulk operations, and the database access technology.

The amount of database requests and other database-related aspects can significantly affect the energy consumption of an operation.

Subsection 2.2.2 refers to academic and grey literature, which lacks a common definition for resource and energy efficiency in the context of software engineering. The literature mentions the term *useful work* to calculate the *energy efficiency factor* for an enterprise application. We propose to utilize the INVEST mnemonic as a structured approach to define the term useful work in the context of this thesis. It allows to scope the measurement by defining relevant use cases for the test scenarios and identifying sets of operations that represent useful work. This approach was successfully applied during this thesis but could be further elaborated in future work.

The current state of the art in software engineering does not seem to provide guidelines on how to write energy-efficient software in general. Each application is unique and requires an analysis and a tailored approach to improve its energy efficiency. Future work could investigate on how to continuously measure and improve the energy efficiency of an application.

The INVEST mnemonic can be used to define useful work in the context of software engineering. Useful work helps to scope the energy efficiency measurements.

Key takeaways about resource and energy efficiency:

- Absolute energy consumption numbers are not comparable across different test environments.
- The relative distribution of energy consumption remains consistent and can be compared across different test environments, enterprise applications, and sets of operations (useful work).
- The amount of records fetched from the database affects the energy consumption of an operation.
- The amount of database requests and other database-related aspects can significantly affect the energy consumption of an operation.
- The INVEST mnemonic can be used to define useful work in the context of software engineering.
- Useful work helps to scope the energy efficiency measurements.

5.1.3. Correlation Between Performance and Energy Efficiency

Subsection 2.2.3 elaborates three hypotheses on the correlation between performance and energy efficiency. This thesis proposes that there is either no correlation, an inverse correlation, or a strong correlation.

The inverse correlation hypothesis states that additional hardware resources may increase performance but decrease energy efficiency. The findings suggest that different hardware setups lead to an inverse correlation. The more powerful the hardware in terms of CPU clock speeds or memory read/write speeds, the better the performance, but the more energy is consumed. Vice versa, the less powerful the hardware, the worse the performance, but the less energy is consumed. Table 32 shows an example for inverse correlation between performance and energy efficiency on Windows and Linux.

Table 32. An example for inverse correlation between performance and energy efficiency on Windows and Linux

Test Execution	Average Latency	Average Throughput	Energy Efficiency
PetClinic Owner on Windows	796ms	3.2 Requests per second	1.99 Requests per Joule
PetClinic Owner on Linux	470ms	8.4 Requests per second	0.27 Requests per Joule

The PetClinic results on Windows run on a less powerful hardware setup and therefore produce a higher latency and process less than half the throughput compared to Linux. But the test executions on Windows achieve a much better energy efficiency than on Linux.

Different hardware setups can lead to an inverse correlation between performance and energy efficiency.

The strong correlation hypothesis suggests that varying implementation details can affect the performance and energy consumption of an application positively or negatively. The experiment with the PetClinic application on Linux described in Subsection 4.2.2 reveals that the same test plan executed with JPA achieves a better performance and energy efficiency than with Spring Data JPA. Additionally, the experiments in Section 4.3 confirm that the implementation details of an enterprise application can increase the performance and energy efficiency likewise. The LakesideMutual application achieves better performance and energy efficiency results than the PetClinic application across both test environments. Table 33 shows an example for strong correlation between performance and energy efficiency on Windows.

Table 33. An example for strong correlation between performance and energy efficiency on Windows

Test Execution	Average Latency	Average Throughput	Energy Efficiency
PetClinic Owner	796ms	3.2 Requests per second	1.99 Requests per Joule
LakesideMutual Customer	43ms	5.8 Requests per second	3.21 Requests per Joule

The numbers reveal that LakesideMutual processes the same amount of useful work much faster with a higher energy efficiency.

Varying implementation details can lead to a strong correlation between performance and energy efficiency.

The findings of this thesis are indicative and not conclusive. The experiments do not provide clear information on what causes the correlations between performance and energy efficiency. Further research could investigate on this topic and provide more insights and explanations.

5.2. Generalization of Measurement Results

This section considers the findings of all experiments described in Chapter 4 and their implications on other software systems. The main findings include that absolute numbers are not comparable across system boundaries, while relative distributions of energy consumption remain consistent. This section generalizes the findings to other Spring Boot applications, web-based applications, and software-intensive systems in general.

5.2.1. Other Spring Boot Applications

From personal experience, we specify that many Spring Boot applications follow a layered or N-tier architecture and run a client-server architecture. The PetClinic and LakesideMutual act as servers and provide HTTP endpoints to clients. Depending on the context and requirements, applications may additionally implement other architectural styles such as a service-oriented architecture, a serverless architecture, or an asynchronous architecture.

Subsection 5.1.1 analyzes the performance deviations of the two Spring Boot applications across Windows and Linux. The findings of this thesis indicate that even minor changes in the configurations such as swapping JPA for Spring Data JPA can affect the performance and overall energy consumption of an application [108]. We expect similar Spring Boot applications, with different configurations and running on a different hardware setup, to achieve diverging performance results.

Subsection 5.1.2 analyzes the resource and energy efficiency deviations of the two Spring Boot applications across Windows and Linux. The results suggest that the relative distribution of energy consumption is comparable across different Spring Boot applications. We expect similar Spring Boot applications to consume varying amounts of energy, while revealing similar relative distributions of energy consumption.

When measuring different architectural styles, such as serverless or asynchronous architectures, the measurement methods and tools may need to be adapted. As a serverless application may not run continuously, additional metrics such as "cold start time" are necessary to gain a holistic view of the performance. When measuring the energy consumption of an asynchronous application, it may be necessary to consider the energy consumption of the message broker or the event bus.

5.2.2. Other Web-Based Applications

The following list provides examples of other web-based applications selected based on their popularity and anecdotal evidence. They include but are not limited to enterprise applications, browser games, social media, and online learning platforms.

- E-commerce applications such as Amazon.
- Issue and project tracking software such as Jira.
- IT service management platforms such as ServiceNow.
- Browser games such as 2048.
- Social media platforms such as Facebook.
- Online learning platforms such as Udemy.

Web-based enterprise applications typically perform a similar set of operations to retrieve, update, and delete data from a database. The PetClinic provides endpoints to perform CRUD operations on the database. LakesideMutual provides endpoints to manage customer data, insurance policies, and customer inquiries in rather complex business workflows across multiple distributed services. Amazon allows customers to search for products (read data), add products to a shopping cart (create data), remove products (delete data), and place an order (update data). Jira and ServiceNow allow users to create, read, update, and delete issues, tickets, or requests.

Different enterprise applications may provide different business functionalities and processes that differ in their complexity and longevity. We specify that their business processes boil down to the same database-related aspects and operations. Enterprise applications eventually perform CRUD operations because databases inherently work this way. Exactly these CRUD operations enable a comparison of the relative distribution of energy consumption across different enterprise applications, according to Subsection 5.1.2. The database itself is an important factor that affects the performance and energy consumption of an arbitrary enterprise application [108].

Other web-based applications like browser games require a real-time, low-latency connection to the server. The user interacts with the game in real-time, and the server needs to control the game state, such as actions, positions, collisions, and scores. Database queries are probably too slow for real-time applications, so they may use in-memory data structures or caching mechanisms. Future work could investigate if the operations performed on such in-memory data structures or caches reveal a similar relative distribution of energy consumption as the CRUD operations performed on a database.

Social media platforms like Facebook probably implement complex algorithms to recommend content to users. Chances are that these algorithms constantly run in the background to analyze user behavior and interactions. Measurements need to be adapted such that they cover the impact of these algorithms on the overall performance and energy consumption of the application. It would be particularly interesting to optimize these algorithms as they are used in a large scale.

Online learning platforms like Udemy probably rely on content delivery networks to deliver video content to users, or they may use document databases to store course content. The main load of users streaming videos is probably outsourced to the content delivery network, while the application itself is responsible for managing user accounts, course enrollments, and progress tracking. The application still performs CRUD operations on a database to manage user accounts and course content. These operations could reveal a similar relative distribution of energy consumption as the operations performed in the PetClinic and LakesideMutual applications.

5.2.3. Other Software-Intensive Systems

Other software-intensive systems may diverge from web-based applications completely, such as desktop and mobile applications, games, embedded systems or controllers. Examples for other software-intensive systems may include but are not limited to:

- Desktop applications such as Microsoft Word.
- Mobile applications such as the Revolut banking app, the WhatsApp messenger, or the Garmin fitness app.
- Games including desktop, mobile, and console games such as Minecraft.
- Embedded systems such as automated thermostats.
- Controller systems such as a mechanical actuator.
- Smart Factories (Industry 4.0) such as automated production lines.

We selected these examples again because of their popularity and based on anecdotal evidence. Desktop and mobile applications are widely used in everyday life, while games are a popular form of entertainment. Embedded systems and controllers are used in various industries, such as automotive, manufacturing, and home automation. Last but not least, smart factories are a hot topic in manufacturing and are often referred to as Industry 4.0.

Microsoft Word can be used offline, without client-server communication, and does not rely on a database to store data. Desktop applications can store data on the local file system or in a cloud storage service. Load testing may require different methods and tools, such as simulating keyboard and mouse inputs, or measuring the time it takes to open and save files. Energy consumption measurements may focus on the application's resource usage, such as CPU and memory, rather than network requests or database operations.

Games, as mentioned before, require a real-time, low-latency connection to the server. While latency is a very common and important quality aspect of games, a generic load and performance test may not be applicable. Benchmarks may focus on the time it takes to load a level, the frame rate, or the time it takes to respond to user inputs. CRUD operations on in-memory data structures or caches may reveal a similar relative distribution of energy consumption as the CRUD operations performed on a database. The scope of useful work needs to be defined according to the game mechanics and the expected user interactions.

Embedded systems are small parts of a larger system and are designed for a specific functionality [113]. They receive or collect data from the environment and perform a specific task. Control systems are a subset of embedded systems and can act in open or closed loop systems. "A control system is a set of mechanical or electronic devices that regulates other devices or systems by way of control loops" [114]. Figure 80 shows an example of a closed loop control system.

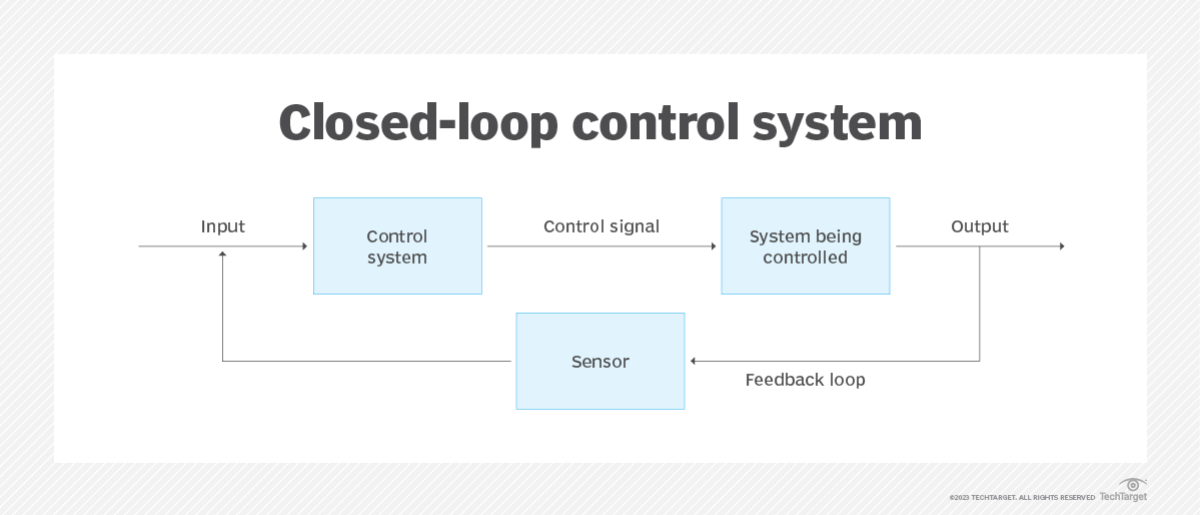

Figure 80. An example for a closed loop control system [114]

Such systems operate in a different context, with different requirements and constraints, but they still receive data from connected devices and send data to connected systems. A controller in a fridge may monitor the temperature with a sensor and turn on the compressor when the temperature exceeds a certain threshold.

Smart factories are a combination of industrial automation and the Internet of Things (IoT) [115]. Embedded systems and other software platforms are connected with each other and exchange messages via message brokers. They may use different means of communication such as Bluetooth, Wi-Fi, or Zigbee. Energy consumption measurements may be adapted to measure the amount of data per message in bytes and the amount of messages exchanged. Performance tests may generate messages with different sizes and measure the time it takes to send and receive these messages.

Each type of software system has its own context, requirements, and constraints. Each system may implement different architectural styles and patterns, which require different measurement methods and tools. Some may require huge data sets for load testing, others may not run continuously or do not even serve HTTP endpoints. Measuring performance requires multiple different metrics and parameters to gain a holistic view of the system under test. The selected approach of defining useful work to measure energy efficiency proofs to be applicable but needs to be adapted to the specific context of the software system. INVEST helps to define useful work but may not be suitable for different types of software systems. Future work could further elaborate on useful work in combination with INVEST and how to use it in actual software projects.

5.3. Related Work

This section references related work in the field of software measurements related to performance and resource and energy efficiency. Some of the important resources covered in this thesis are journal articles, the Green Software Foundation (GSF), and the Green Software (GGS) blog.

Denaro et al. report on performance characteristics and why performance is an important requirement for software projects [6]. They suggest an approach for performance testing for distributed enterprise applications beginning in early stages of the development process. Capra et al. suggest a measure for energy efficiency and provide a methodology to measure the energy efficiency of software applications

[57]. They conclude that different software designs have significant impact on energy efficiency. Guldner et al. elaborate the *Green Software Measurement Model (GSMM)*, which combines measurement models, setups, and methods from multiple research groups [12]. This approach appears to cover a wide range of software measurements and is a good starting point for future research.

The GSF is a non-profit organization that aims to promote the development of sustainable software [55]. It is a good source of information on the topic of green software and energy efficiency in general. We expect them to gain more influence in the future and to become a leading organization when it comes to standards, tooling, and best practices.

The GGS blog reports on practice-oriented methods to performance and energy consumption measurements [11]. It is an inspiration for the experiments in this thesis and an interesting source of information. The blog provides beginner-friendly articles for people who would like to get into the field of software measurements and benchmarking. Practitioners can find an easy introduction to the topic with practical examples.

There are many more articles online we have not discussed in this thesis, just two of them are mentioned here. An article on energy based performance tuning for large scale high performance systems from 2012 by Laros et al. reports on "energy savings opportunities of up to 39% with little to no impact on run-time performance" [116]. A later 2016 article by Jin et al. investigates improvements in energy efficiency when using parallel programming and power-saving features [117]. The topics seem interesting and relevant for improving energy efficiency, especially in the context of cloud computing. Similar methods may be applied to improve the energy efficiency of enterprise applications.

The list is not exhaustive and only provides a few examples of the many articles available on the topics of software measurements, performance, and resource and energy efficiency. The research field of green software appears to be growing, further research is needed to improve the energy efficiency of software systems and eventually the ecological footprint. Section 5.4 reflects on the achievements and challenges of this thesis.

5.4. Retrospective

This section reflects on the research objectives, the challenges faced, and the added value for the target audience. This thesis is successfully completed based on the defined objectives but leaves room for further improvements and future work.

5.4.1. Research Objectives

This thesis aims to investigate whether, how and why the state of the art in the field of performance and efficiency measurements reported in the academic literature differs from practice. Section 1.1 separates the objective into five sub-objectives and formulates them as research questions.

1. How are the two types of software quality attributes performance and resource and energy efficiency defined a.) in the scientific literature and in official standards (ISO/IEC/IEEE) and b.) in the gray literature (e.g., Q42, Growing Green Software blog)?

The quality attribute of performance is widely accepted and defined in standards [37], academic literature [6], and grey literature [41] [42] [43]. This thesis relies on latency and throughput as measurable aspects of performance and combines them with the average execution time for the test plan. The selected approach turns out to be suitable for measuring the performance of an application from a client perspective. The approach accompanies the energy consumption measurements and puts them into perspective, but it is not able to provide sufficient insights into why the performance deviations occur. Additional metrics are required to gain a holistic view of the performance of an application.

The quality attribute of resource and energy efficiency lacks standardization and varies between different sources [12] [56] [57]. It appears that the term *useful work* is a common denominator in the definitions of resource and energy efficiency [12] [60], which finds its application in practice [58]. This thesis proposes to leverage the INVEST mnemonic [13] to define the scope of useful work. The selected approach seems suitable for the context of this thesis, and for a use in practice.

2. How do performance tests and energy efficiency/resource consumption measurements have to be set up so that their results are accurate, meaningful (with respect to the definitions from question 1) and reproducible (e.g., with respect to the FAIR criteria)?

The literature and practitioners suggest to set up a controlled test environment in order to achieve meaningful results [12] [46]. In case it is not feasible to use an isolated testing lab or mirror a production environment, resource constraints can guarantee or restrict resources for test executions [102]. The impact of resource constraints depends on the environment they are used in. The findings in this thesis suggest that they have a positive impact on systems with limited resources, and a negligible or even negative impact on systems with sufficient resources. Resource constraints definitely have their place when it comes to production deployments in practice, but their usage depends on specific requirements and environmental factors.

One part of a controlled test environment is an automated test execution script including a setup phase, multiple test steps, and a cleanup phase. This thesis relies on an automated test execution script on Linux. The tests on Windows are not fully automated and require manual intervention to conduct the experiments. Automated test executions should always be preferred over manual interactions to save time, ensure reproducibility, and reduce the probability of human error.

When it comes to achieving accurate and meaningful results, the practitioners suggest to use multiple iterations, calculate averages, and report outliers [46] [81]. Diagrams illustrate the results with outliers and deviations to provide a more realistic view of the test results [12]. Relying on multiple iterations, averages, visualizations, and outliers proves to be suitable when documenting measurement results in practice.

The industry standard of load testing with third party tools such as JMeter, Gatling, or Locust, is a simple and effective way to measure the performance of software systems. The configuration and usage of JMeter is rather straightforward with a low learning curve, which makes it a suitable tool for this thesis and practitioners in the industry. JMeter allows to store test plans in files, which can be easily shared, reused, and adapted. Additionally, all results are stored in structured files, which can be analyzed and visualized with JMeter. This makes it a good choice for documenting test plans and the corresponding results.

JoularJX as a measurement tool provides valuable insights into the energy consumption of Java applications, which helps to optimize the energy efficiency of software systems. JoularJX stores the

measurement results in .csv files, which can be evaluated with third party tools like Excel. It is a reliable, easy-to-use tool, and a versatile option for multiple operating systems. The selected test environment setup and measurement tools are suitable to gain accurate, meaningful, and reproducible results.

3. Is it possible to reproduce the measurements of the Spring Boot PetClinic sample that are reported in the Growing Green Software blog? Do the interpretations of the data given in the blog posts require clarification and discussion? How could the reported test and measurements be improved (taking the answers to questions 1 and 2 into account)?

The GGS blog measurements of the PetClinic application are successfully reproduced in this thesis and help to rule out potential misconfigurations and invalid results in the later experiments. The blog post interpretations of the data are clear and provide valuable insights into the energy consumption of the PetClinic across different Spring Boot versions. The findings of this thesis reveal significant performance differences between Windows and Linux compared to the GGS blog. The deviations occur due to different hardware resources and operating systems. However, the relative distribution of energy consumption for all involved operations is similar across the different operating systems.

The GGS blog introduces the term *useful work*, but does not further elaborate on it and uses kilowatt-hours and Joules instead. It does suggest to combine useful work with user stories or use cases. This thesis proposes to leverage the INVEST mnemonic to define the scope of useful work and to use it in practice. Future work could further elaborate on useful work in combination with INVEST to scope the test plan according to user stories or use cases. A real enterprise application could be measured and analyzed such that the selected user stories can be improved and optimized on a technical level.

4. When measuring selected use cases of the sample application LakesideMutual in the same way as the Spring Boot PetClinic sample, how do the two result sets compare? How can the differences be explained? Does the monolith version of LakesideMutual show a different behavior than the microservices version?

The initial test plan solely considers CRUD operations with JPA, while the adapted test plan considers CRU operations and workflows with Spring Data JPA. The process of adapting the test plan is covered in a total of five experiments. The test concept includes comparing CRUD and CRU operations, comparing JPA and Spring Data JPA, and comparing the monolith and microservices versions of LakesideMutual. The results confirm varying performance results between different operating systems and even between different implementation details, such as JPA or Spring Data JPA. The relative distribution of energy consumption remains consistent across the two operating systems, across database access technologies, across different enterprise applications, and even across different sets of operations.

Additionally, the correlation between performance and energy efficiency is analyzed. The findings suggest an inverse correlation between performance and energy efficiency when using different hardware resources. The Linux test environment runs more powerful hardware resources, achieves a higher performance, but consumes more energy than the Windows test environment. The results also suggest a strong correlation when comparing applications with varying implementation details. The PetClinic achieves better performance and energy efficiency with JPA than with Spring Data JPA. Eventually LakesideMutual achieves significantly better performance and energy efficiency than the PetClinic application for the same set of operations.

5. How can the results from questions 1 to 4 be generalized so that they can serve as guidelines and examples for future tests and measurements of a.) other Spring Boot applications b.) other Web-based applications c.) any distributed, software-intensive system?

The results of the PetClinic and LakesideMutual applications allow for a generalization of the findings for other Spring Boot applications, web-based applications, and other enterprise applications. Simple CRUD operations on databases are a common denominator for enterprise applications and can be used to generalize the findings. The results solely provide indications for other types of applications and act as a good starting point for further research.

All research questions are successfully answered, and the research objectives are met.

5.4.2. Challenges

We faced several challenges and learned valuable lessons throughout this thesis. The first challenge was to set up a controlled test environment and to configure the measurement tools correctly. A misconfiguration of JoularJX led to incorrect measurement results, which made it impossible to reproduce the results of the GGS blog post. Reproducing existing measurements before conducting our own measurements proved to be a valuable step in the process. It prevented us from misconfigurations and invalid results in the later experiments.

I learned to pay close attention to configuration details, test the setup thoroughly, and to document the test environment setup properly.

The second challenge was to automate the test executions on Windows. JoularJX works reliably when executed in foreground mode. We faced challenges on Windows when running the application in background mode. We were not able to terminate the process gracefully enough, resulting in lost energy consumption logs. Therefore, we had to run the tests in foreground mode and manually terminate the process after the test execution. This appears to be a limitation of JoularJX in combination with Windows, as the tool works as expected on Linux. We do not exclude a potential misconfiguration or misuse of the tool from our side. We encourage to further investigate this behaviour on Windows and potentially improve the tool.

I learned that it is important to test the measurement tools on all target operating systems and to ensure that they work as expected.

The third challenge was to achieve meaningful results across different test environments and different sets of operations. The two test environments differ in hardware resources and the applications under test differ in their functionality. We created a test concept that resembles a staircase approach, where each experiment and each scenario builds upon the previous one. Each test scenario specifically changes one aspect in the test setup to achieve comparable results. It allowed us to draw conclusions even across system boundaries and different enterprise applications.

I learned that it is important to have a clear test concept and to achieve meaningful results.

5.4.3. Added Value for the Target Audience

This thesis primarily targets academic personnel and practitioners in the field of software engineering and architecture. Professors, students, and researchers may benefit from the detailed research, the experimental setup, and the measurement results as a baseline for their own research. Software engineers and architects can apply the established measurement techniques and metrics to gain insights in their projects and to optimize their software systems. Additionally, product owners and project managers may refer to the findings and implications to understand the trade-offs between software quality, customer satisfactions, costs, and sustainability.

5.5. Outlook

This thesis focused on measurements of enterprise applications, specifically two Spring Boot applications written in Java. The Spring Boot framework comes with a lot of features and complex configurations, which have a potential impact on measurement results. Further research could address several aspects in connection with Spring Boot.

1. Different versions of Java, Spring Boot, and even database versions could be investigated.
2. The impact of database interactions can be isolated by comparing different database vendors, database connection technologies, database sequence allocation sizes, and database transaction properties (ACID).
3. Code changes related to Spring Boot, such as using @Autowired versus manual dependency injection, may reveal interesting results.

When it comes to other types of applications, such as embedded and control systems, or desktop and mobile applications, the findings of this thesis may not directly apply. An interesting next step could involve extending the measurements to other types of applications. This includes further methods and tools for measuring the performance of an application, as well as further elaborating useful work when it comes to measuring energy efficiency.

Cloud deployments are becoming increasingly popular, and the energy consumption of cloud-based applications is a growing concern. Sustainability and energy efficiency are becoming more important in the software engineering community. A Master's thesis could extend the work of this thesis by investigating the energy consumption of a real enterprise application running in a cloud environment. It appears that there is a disconnect between conceptual user stories and actual software implementation with respect to energy efficiency. The findings of this thesis can be used to identify the useful work of an application, measure its energy consumption, and evaluate its energy efficiency. The goal is to identify wasteful operations that degrade the energy efficiency, and to provide recommendations for optimizing the energy consumption of cloud-based applications.

The following activities should be achieved with the help of research, experiments, and if applicable, programming activities:

1. Identify how energy consumption measurements have to be setup so that their results are accurate, meaningful (with respect to the findings of the second project thesis), and reproducible.
2. Reproduce the existing measurements of the second project thesis on a bare-metal Linux server by applying the setup identified in the first step.

3. Deploy and measure the same application on a cloud server and compare the results to the ones obtained in the second step.
4. Analyze wasteful operations in the enterprise application and provide optimization recommendations with respect to useful work and user stories.
5. Generalize the findings from step one to four and establish guidelines for energy-efficient software engineering in cloud environments.

The scope of such a thesis could also be adapted to cloud-native application development. It is possible to investigate the energy consumption of cloud-native application traits by lifting and shifting an existing application to the cloud. This could include the deployment of an application on IaaS, and the subsequent refactorings to utilize cloud-native traits such as PaaS, serverless computing, and other cloud offerings.

Critical success factors for such a thesis may include:

- Relying on a real enterprise application, including complex domain models and long-running business processes, to establish a solid foundation for the research.
- Comparing on-premise and cloud deployments to identify differences in energy consumption.
- Achieving meaningful results by ensuring equal technology stacks, configurations, and application versions.

Such a thesis would provide valuable insights into the energy consumption of cloud-based applications and help to establish guidelines for energy-efficient software engineering. We aim to promote sustainable software development and reduce the environmental impact of growing cloud infrastructure.

6. Conclusion

This thesis set out to investigate whether and how the state of the art in the field of performance and efficiency measurements reported in the academic literature differs from practice. Performance measurements are well established in the academic literature. The selected approach of measuring performance with latency and throughput is not sufficient to gain a holistic view of the performance of an application. Additional metrics from different perspectives need to be evaluated to gain a comprehensive understanding of the performance of an application. In practice, measurements need to be conducted continuously to recognize performance issues early on in the software development process.

With respect to resource and energy efficiency measurements, the software engineering literature lacks a common definition of measurable aspects. The literature mentions the term *useful work* but does not provide a sufficient definition. This thesis successfully leverages the INVEST mnemonic to clarify the term *useful work* for energy efficiency measurements of enterprise applications. Future research could further elaborate the concept of useful work and how it can be applied to other types of applications.

The experiments are conducted with JMeter as a load testing tool and JoularJX as an energy consumption measurement tool. Best practices and methods mentioned in the literature are applied to the experiments, such as using automated test scripts with setup and cleanup phases, running multiple test executions, calculating average results, and using visualizations to interpret the results. Equal tools, versions, and configurations across all systems and applications are essential to achieve meaningful results.

The measurements confirm that the performance and energy consumption of applications are significantly influenced by external factors such as hardware, operating systems, and implementation details. The results suggest that the absolute performance of an application can not be compared across system boundaries and enterprise applications. They reveal that the relative distribution of energy consumption is comparable across different test environments, enterprise applications, and even sets of operations.

The experiments suggest that the performance correlates with the energy efficiency. The results indicate that different hardware resources lead to an inverse correlation between performance and energy efficiency. Systems with more powerful hardware resources perform better, but consume more energy, and vice versa. Furthermore, the results indicate that implementation and configuration details lead to a strong correlation between performance and energy efficiency. A test plan using JPA performs better with higher energy efficiency than the same test plan using Spring Data JPA. The LakesideMutual application differs in its implementation from the PetClinic application and achieves better performance and energy efficiency.

The achieved results and insights can be used by professors and students for their own research. Software engineers and architects can refer to the testing methods, tools, and configurations used in the experiments to measure their own applications. Product owners and project managers can refer to the result analysis and interpretation to take informed decisions about the implications of performance and energy efficiency.

Energy efficient software engineering is becoming increasingly important with respect to sustainability. We aim to continue this work in a follow-up Master's thesis and investigate the energy efficiency of cloud-native applications and cloud infrastructure. Such work contributes to a more sustainable software development process and eventually reduces the ecological impact of software systems.

7. Appendices

Appendix A: Glossary

arc42

arc42 is an open-source tool for architecture communication and documentation.

arc42 Quality Model

arc42 Quality Model (Q42) is an approach to analyse and increase product and system quality.

Application Programming Interface

An application programming interface (API) provides a set of functions to other software components or systems to interact with the software providing the API.

Class Diagram

The class diagram is a Unified Modeling Language (UML) diagram that describes the structure of software systems. It shows classes, attributes, operations and relationships between classes.

Cloud Computing

Cloud computing refers to multiple cloud services communicating with each other via internet protocols.

Cloud-native

Cloud-native is a term used to describe applications that are designed to run in cloud computing environments. This involves building, deploying and managing software applications.

Control System

A control system is an embedded system that interacts with the environment to control mechanical or electrical actuators.

Docker

Docker is a tool to deploy and run applications in virtual containers. Containerization allows developers to ship applications including all dependencies across different environments.

Docker Compose

Docker Compose is a tool that builds upon Docker and allows developers to define and run multi-container applications.

Embedded System

An Embedded system is part of a larger system and fulfills a specific task.

Energy Efficiency Factor

The energy efficiency factor describes the ratio of useful work done by a system to the energy consumed by the system.

Enterprise Application

An enterprise application is a software system that differs from other software systems in terms of complexity, longevity, and amount of users. Enterprise applications provide long-running business processes to multiple hundreds or thousands of users.

Entity-Relationship Diagram

The entity-relationship diagram (ERD) is a UML diagram that describes a data model and how entities relate to each other.

FAIR

FAIR stands for Findable, Accessible, Interoperable, and Reusable. The FAIR principles are a set of guidelines, which should be followed when publishing research data and all processing steps involved.

Growing Green Software

Growing Green Software (GGS) is a blog that reports on performance and efficiency measurements in the context of Java-based Spring Boot applications.

INVEST

INVEST is a mnemonic that stands for Independent, Negotiable, Valuable, Estimable, Small, and Testable. This set of criteria is used to assess the quality of user stories in agile software development and improve the size of work packages.

Java

Java is an open-source, cross-platform, object-oriented programming language that runs in the Java Virtual Machine (JVM).

Java Database Connectivity

The Java Database Connectivity (JDBC) is an API that describes how clients such as Java applications can access databases.

Java Persistence API

The Java Persistence API (JPA) is an interface that builds upon JDBC and eases the interaction with databases by introducing object-relational mapping.

Java Virtual Machine

The Java Virtual Machine (JVM) is a runtime environment that runs on top of an operating system. This allows Java to run on any platform that supports a JVM.

JMeter

Apache JMeter is an open-source load and performance testing tool. JMeter interacts with applications via APIs and simulates users interacting with the application.

JoularJX

JoularJX is an open-source tool that allows users to measure specific Java applications down to the method level.

LakesideMutual

LakesideMutual is an open-source project developed by the Eastern Switzerland University of Applied Sciences, which represents a fictitious insurance company called *Lakeside Mutual*. This software project resembles a distributed enterprise application and is used in this thesis to evaluate the established software quality attributes.

Latency

Latency describes the time it takes to send a request, process the request, and retrieve the response.

Master Data

In API design, master data describes all data that is long-living, rarely updated, and frequently referenced.

Measurement

A measurement is the definition and execution of a procedure to get a single data point. In the context of this thesis, a measurement measures a quality attribute of an architecture. Multiple measurement executions, multiple data points, can be aggregated in a metric.

Metric

A metric contains multiple measurement definitions and their respective data points. Metrics represent the deviation between multiple measurements over time.

Monolithic Architecture

A monolithic architecture provides the entire application functionality in a single deployment unit.

MySQL Database

MySQL is an open-source relational database management system (RDBMS) provided by Oracle. MySQL is a widely used database system and is used in this thesis to persist data.

Operational Data

In API design, operational data describes all data that is short-living, frequently updated, and rarely referenced.

Performance

Performance is an important non-functional requirement or software quality attribute for enterprise applications. Performance describes multiple characteristics of a system such as throughput and latency.

PetClinic

The PetClinic is an open-source project developed by the Spring community to demonstrate the Spring framework. The PetClinic resembles a monolithic enterprise application and is used in this thesis to reproduce the GGS blog measurements.

Running Average Power Limit

The Running Average Power Limit (RAPL) is a feature of Intel processors that allows users to measure the power consumption statistics of the CPU.

Resource and Energy Efficiency

Resource and energy efficiency is a software quality attribute that describes how well a system uses its resources and energy.

Service-Oriented Architecture

A service-oriented architecture (SOA) provides the entire application functionality in multiple deployment units. Each unit is called a service and usually interacts with other services via APIs.

Software Quality Attribute

A software quality attribute is a non-functional requirement that describes a certain quality aspect of a software system. An example of a software quality attribute is performance, which describes how well a system performs under certain conditions.

Spring Framework

The Spring framework is an open-source Java framework that enables developers to build production-ready enterprise applications. Spring is an alternative to other Java frameworks such as Jakarta EE.

Spring Boot Framework

Spring Boot is an open-source Java framework to develop applications based on Java and Spring. Spring Boot allows developers to use convention over configuration.

Spring Data JPA

Spring Data JPA builds upon JPA and introduces abstraction layers to ease the interaction with the database from a developer its perspective.

Test Plan

A test plan describes the entire test scenario including the test steps, the test data, the expected results, and the actual results.

Throughput

Throughput describes the amount of requests a system can process in a given time interval.

Useful Work

Useful work is a term introduced by academic literature to describe work that is done by a system. The definition of useful work depends on the domain context and requires further specification.

Appendix B: Bibliography

[1] GitHub Copilot · Your AI pair programmer. (2025). GitHub. Retrieved February 19, 2025, from https://github.com/features/copilot

[2] Scribbr. (2023, March 13). Wir korrigieren Dokumente für Schule, Studium und Job. Retrieved February 19, 2025, from https://www.scribbr.ch/

[3] DeepL. (n.d.). DeepL Translate: The world's most accurate translator. Retrieved February 19, 2025, from https://www.deepl.com/en/translator

[4] Elicit: the AI Research Assistant. (n.d.). Elicit. Retrieved February 19, 2025, from https://elicit.com/

[5] OpenAI. (n.d.). ChatGPT. ChatGPT. Retrieved June 18, 2025, from https://chatgpt.com/

[6] Denaro, G., Polini, A., & Emmerich, W. (2004). Early performance testing of distributed software applications. Workshop on Software and Performance (WOSP 2004), 94–103. https://doi.org/10.1145/974044.974059

[7] Brunnert, A. (2024). Green Software Metrics. ICPE '24 Companion, 287–288. https://doi.org/10.1145/3629527.3652883

[8] Kern, E., Dick, M., Naumann, S., Guldner, A., & Johann, T. (2013). Green Software and Green Software Engineering – Definitions, Measurements, and Quality Aspects. ICT4S 2013: Proceedings of the First International Conference on Information and Communication Technologies for Sustainability, ETH Zurich. https://doi.org/10.3929/ethz-a-007337628

[9] Ruch, J. (n.d.). Measuring Software Architecture Quality: Elaborating Metrics to Compare Enterprise Application Architectures. Not published.

[10] Ruch, J. (n.d.). EVA Advanced Software Architecture - Metrics for Observability. Not published.

[11] Stocker, M. (2024, December 23). Growing Green Software – medium. Medium. Retrieved February 19, 2025, from https://medium.com/growing-green-software

[12] Guldner, A., Bender, R., Calero, C., Fernando, G. S., Funke, M., Gröger, J., Hilty, L. M., Hörnschemeyer, J., Hoffmann, G., Junger, D., Kennes, T., Kreten, S., Lago, P., Mai, F., Malavolta, I., Murach, J., Obergöker, K., Schmidt, B., Tarara, A., . . . Naumann, S. (2024). Development and evaluation of a reference measurement model for assessing the resource and energy efficiency of software products and components—Green Software Measurement Model (GSMM). Future Generation Computer Systems, 155, 402–418. https://doi.org/10.1016/j.future.2024.01.033

[13] Wake, B. (2003, August 17). INVEST in good stories, and SMART tasks. XP123. Retrieved February 28, 2025, from https://xp123.com/invest-in-good-stories-and-smart-tasks/

[14] Fowler, M. (2002, November). Patterns of enterprise Application Architecture. O'Reilly Online Learning. Retrieved April 1, 2025, from https://learning.oreilly.com/library/view/patterns-of-enterprise/0321127420/

[15] Zimmermann, O. & Eastern Switzerland University of Applied Sciences. (2024, September 17). Application Architecture: Introduction & Architecturally Significant Requirements [Slide show]. Lecture, Switzerland.

[16] Hartwich, C. (2004). Chapter 3: The Distributed Architecture of MultiTiered Enterprise Applications. In Refubium FU-Berlin. Freie Universität Berlin. Retrieved April 1, 2025, from https://refubium.fu-berlin.de/bitstream/handle/fub188/4741/03_chapter3.pdf?sequence=4&isAllowed=y

[17] Evans, E. (2003, August). Domain-Driven Design: tackling complexity in the heart of software. O'Reilly Online Learning. Retrieved April 1, 2025, from https://learning.oreilly.com/library/view/domain-driven-design-tackling/0321125215/

[18] Hohpe, G., & Woolf, B. (2003, October). Enterprise integration patterns: Designing, building, and deploying messaging solutions. O'Reilly Online Learning. Retrieved April 1, 2025, from https://learning.oreilly.com/library/view/enterprise-integration-patterns/0321200683/

[19] Internal Revenue Service. (n.d.). Foreign Account Tax Compliance Act (FATCA) | Internal Revenue Service. Retrieved June 6, 2025, from https://www.irs.gov/businesses/corporations/foreign-account-tax-compliance-act-fatca

[20] Sarbanes-Oxley Compliance Professionals Association. (n.d.). Sarbanes-Oxley Act | Sarbanes-Oxley Compliance Professionals Association (SOXCPA). Retrieved June 6, 2025, from https://www.sarbanes-oxley-act.com/

[21] The Spring PetClinic community. (n.d.). Retrieved March 7, 2025, from https://spring-petclinic.github.io/

[22] Spring by VMware Tanzu. (n.d.). Spring Framework. Spring Framework. Retrieved April 30, 2025, from https://spring.io/projects/spring-framework

[23] Spring by VMware Tanzu. (n.d.-a). Spring boot. Spring Boot. Retrieved April 30, 2025, from https://spring.io/projects/spring-boot

[24] IBM. (2025, April 16). Java Spring Boot. What is Java Spring Boot? Retrieved April 30, 2025, from https://www.ibm.com/think/topics/java-spring-boot

[25] GitHub - spring-petclinic/spring-petclinic-rest: REST version of the Spring Petclinic sample application. (n.d.). GitHub. Retrieved March 7, 2025, from https://github.com/spring-petclinic/spring-petclinic-rest

[26] GitHub - spring-petclinic/spring-petclinic-angular: Angular 16 version of the Spring Petclinic sample application (frontend). (n.d.). GitHub. Retrieved March 15, 2025, from https://github.com/spring-petclinic/spring-petclinic-angular

[27] Harel, D., & Kupferman, O. (2002). On object systems and behavioral inheritance. IEEE Transactions on Software Engineering, 28(9), 889–903. https://doi.org/10.1109/tse.2002.1033228

[28] Ramachandrappa, N. C. (2024). SOLID design principles in software engineering. International Journal of Computer Trends and Technology, 72(9), 18–23. https://doi.org/10.14445/22312803/ijctt-v72i9p104

[29] Liskov, B. (1987). Data Abstraction and Hierarchy. Addison-Wesley. Retrieved June 6, 2025, from https://www.cs.tufts.edu/~nr/cs257/archive/barbara-liskov/data-abstraction-and-hierarchy.pdf

[30] Zimmermann, O., Stocker, M., Lubke, D., Zdun, U., & Pautasso, C. (2022, November). Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. O'Reilly Online Learning. Retrieved March 27, 2025, from https://learning.oreilly.com/library/view/patterns-for-api/9780137670093/

[31] Microservice-API-Patterns. (n.d.). GitHub - Microservice-API-Patterns/LakesideMutual: Example Application for Microservice API Patterns (MAP) and other patterns (DDD, PoEAA, EIP). GitHub. Retrieved April 22, 2025, from https://github.com/Microservice-API-Patterns/LakesideMutual

[32] Zimmermann, O., Stocker, M., Lubke, D., Zdun, U., & Pautasso, C. (n.d.). Patterns for API design. Microservice API Patterns (MAP). Retrieved June 6, 2025, from https://microservice-api-patterns.org/

[33] Wikipedia contributors. (2025, February 4). SMART criteria. Wikipedia. Retrieved March 1, 2025, from https://en.wikipedia.org/wiki/SMART_criteria

[34] Zimmermann, O., & Stocker, M. (2024). Design Practice Reference: Activities and Templates to Craft Quality Software in Style. In design-practice-repository. Leanpub. Retrieved March 1, 2025, from https://leanpub.com/dpr

[35] Software Engineering Institute. (2018). The SEI Quality Attribute Workshop. In Software Engineering Institute. Retrieved March 5, 2025, from https://insights.sei.cmu.edu/documents/2542/2018_010_001_513488.pdf

[36] Bass, L., Clements, P., & Kazman, R. (2021, August). Software Architecture in Practice, 4th Edition. O'Reilly Online Learning. Retrieved March 5, 2025, from https://learning.oreilly.com/library/view/software-architecture-in/9780136885979/

[37] ISO & IEC. (2023). ISO. Online Browsing Platform (OBP). Retrieved February 19, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en

[38] ISO, IEC, & IEEE. (2016). ISO. Online Browsing Platform (OBP). Retrieved March 1, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24748:-4:ed-1:v1:en

[39] IEEE Standard for Application and Management of the Systems Engineering Process. (2005). IEEE. https://doi.org/10.1109/ieeestd.2005.96469

[40] Starke, G., & Hruschka, P. (n.d.-b). Home. Arc42 Quality Model. Retrieved February 19, 2025, from https://quality.arc42.org/

[41] Starke, G., & Hruschka, P. (2000, September 2). Response time for image rendering. arc42 Quality Model. Retrieved February 19, 2025, from https://quality.arc42.org/requirements/response-time-for-image-rendering

[42] Starke, G., & Hruschka, P. (2022b, December 28). Latency. arc42 Quality Model. Retrieved March 1, 2025, from https://quality.arc42.org/qualities/latency

[43] Starke, G., & Hruschka, P. (2022c, December 28). Throughput. arc42 Quality Model. Retrieved March 1, 2025, from https://quality.arc42.org/qualities/throughput

[44] Burke, J. (2025, February 11). What is throughput? Search Networking. Retrieved March 1, 2025, from https://www.techtarget.com/searchnetworking/definition/throughput

[45] Starke, G., & Hruschka, P. (2022c, December 28). Scalability. arc42 Quality Model. Retrieved March 1, 2025, from https://quality.arc42.org/qualities/scalability

[46] Devoxx, & Baumgartner, P. (2024, October 10). Lean Spring Boot Applications for the Cloud by Patrick Baumgartner [Video]. YouTube. Retrieved March 2, 2025, from https://www.youtube.com/watch?v=s982aX2HSfk

[47] Patbaumgartner. (n.d.). talk-lean-spring-boot-applications-for-the-cloud/lean-spring-boot-applications-for-the-cloud.pdf at main · patbaumgartner/talk-lean-spring-boot-applications-for-the-cloud. GitHub. Retrieved March 2, 2025, from https://github.com/patbaumgartner/talk-lean-spring-boot-applications-for-the-cloud/blob/main/lean-spring-boot-applications-for-the-cloud.pdf

[48] Team, B. A., & Team, A. (2023, November 20). How to run web app testing. Blog About Software Development, Testing, and AI | Abstracta. Retrieved March 2, 2025, from https://abstracta.us/blog/performance-testing/how-to-do-performance-testing-for-web-application/

[49] Team, A. (2024, April 24). Performance Testing VS Load Testing | Abstracta. Blog About Software Development, Testing, and AI | Abstracta. Retrieved March 2, 2025, from https://abstracta.us/blog/performance-testing/performance-testing-vs-load-testing/

[50] Stocker, M. (2024a, November 16). Measuring Java Energy consumption - Growing Green Software - medium. Medium. https://medium.com/growing-green-software/measuring-java-energy-consumption-987654efdabb

[51] Stocker, M. (2024a, November 13). Evolution of energy usage in Spring Boot - Growing green Software - medium. Medium. https://medium.com/growing-green-software/evolution-of-energy-usage-in-spring-boot-69c7c372dba3

[52] Capra, E., Francalanci, C., & Slaughter, S. A. (2012). Measuring application software energy efficiency. IT Professional, 14(2), 54–61. https://doi.org/10.1109/mitp.2012.39

[53] ISO & IEC. (2016b). ISO. Online Browsing Platform (OBP). Retrieved February 19, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec:30134:-2:ed-1:v1:en

[54] ISO & IEC. (2016). ISO. Online Browsing Platform (OBP). Retrieved February 19, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec:30134:-1:ed-1:v1:en

[55] Green Software Foundation. (n.d.). GSF. Retrieved February 19, 2025, from https://greensoftware.foundation/

[56] Green-Software-Foundation. (n.d.). sci/SPEC.md at main · Green-Software-Foundation/sci. GitHub. Retrieved February 19, 2025, from https://github.com/Green-Software-Foundation/sci/blob/main/SPEC.md

[57] Capra, E., Francalanci, C., & Slaughter, S. A. (2011). Is software "green"? Application development environments and energy efficiency in open source applications. Information and Software Technology, 54(1), 60–71. https://doi.org/10.1016/j.infsof.2011.07.005

[58] Stocker, M. (2024a, November 23). Software Efficiency and Energy Consumption - Growing Green Software - medium. Medium. https://medium.com/growing-green-software/software-efficiency-and-energy-consumption-916b390593ec

[59] Cambridge University Press & Assessment. (2025). efficiency. In Cambridge Dictionary. Retrieved February 19, 2025, from https://dictionary.cambridge.org/dictionary/english/efficiency

[60] english/glossary.md · main · Green-Software-Engineering / Green Software Measurement Model · GitLab. (2023, September 8). GitLab. Retrieved February 20, 2025, from https://gitlab.rlp.net/green-software-engineering/gsmm/-/blob/main/english/glossary.md

[62] Mancebo, J., Calero, C., Garcia, F., Moraga, M. A., & De Guzman, I. G. (2021). FEETINGS: Framework for energy efficiency testing to improve environmental goal of the software. Sustainable Computing Informatics and Systems, 30, 100558. https://doi.org/10.1016/j.suscom.2021.100558

[63] Johann, T., Dick, M., Naumann, S., & Kern, E. (2012). How to measure energy-efficiency of software: Metrics and measurement results. GREENS 2012, Zurich, Switzerland, 51–54. https://doi.org/10.1109/greens.2012.6224256

[64] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH. https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/

[65] Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT Model: A reference model for green and sustainable software and its engineering. Sustainable Computing Informatics and Systems, 1(4), 294–304. https://doi.org/10.1016/j.suscom.2011.06.004

[66] Wikipedia contributors. (2024, December 30). Resource efficiency. Wikipedia. Retrieved February 19, 2025, from https://en.wikipedia.org/wiki/Resource_efficiency

[67] Starke, G., & Hruschka, P. (2022, December 28). Energy efficiency. arc42 Quality Model. Retrieved February 19, 2025, from https://quality.arc42.org/qualities/energy-efficiency

[68] Merriam-Webster. (2025). efficient. In Merriam-Webster Dictionary. Retrieved February 19, 2025, from https://www.merriam-webster.com/dictionary/efficient

[69] Starke, G., & Hruschka, P. (2023, July 23). Reduce energy consumption with every new version. arc42 Quality Model. Retrieved February 19, 2025, from https://quality.arc42.org/requirements/reduce-energy-consumption-with-new-version

[70] Starke, G., & Hruschka, P. (2023a, July 4). Save at least 20% of carbon emissions with every new

version. arc42 Quality Model. Retrieved February 19, 2025, from https://quality.arc42.org/requirements/carbon-efficiency-save

[71] Laaber, C., & Leitner, P. (2018). An evaluation of open-source software microbenchmark suites for continuous performance assessment. MSR '18: Proceedings of the 15th International Conference on Mining Software Repositories, 119–130. https://doi.org/10.1145/3196398.3196407

[72] Laaber, C., Würsten, S., Gall, H. C., & Leitner, P. (2020). Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), 989–1001. https://doi.org/10.1145/3368089.3409683

[73] Sealuzh. (n.d.). GitHub - sealuzh/jmh. GitHub. Retrieved February 24, 2025, from https://github.com/sealuzh/jmh

[74] Costa, D., Bezemer, C., Leitner, P., & Andrzejak, A. (2019). What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. IEEE Transactions on Software Engineering, 47(7), 1452–1467. https://doi.org/10.1109/tse.2019.2925345

[75] Lenka, R. K., Dey, M. R., Bhanse, P., & Barik, R. K. (2018). Performance and Load Testing: Tools and Challenges. International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering - (ICRIEECE), 2257–2261. https://doi.org/10.1109/icrieece44171.2018.9009338

[76] Yenugula, M., Kodam, R., & He, D. (2019). Performance and load testing: Tools and challenges. International Journal of Engineering in Computer Science, 1(1), 57–62. https://doi.org/10.33545/26633582.2019.v1.i1a.102

[77] Huerta-Guevara, O., Ayala-Rivera, V., Murphy, L., & Portillo-Dominguez, A. O. (2019). DYNAMOJM: A JMeter tool for performance testing using dynamic workload adaptation. In Lecture notes in computer science (pp. 234–241). https://doi.org/10.1007/978-3-030-31280-0_14

[78] Load testing designed for DevOps and CI/CD | Gatling. (n.d.). Gatling. Retrieved March 17, 2025, from https://gatling.io/

[79] Locust.io. (n.d.). Locust. Retrieved March 17, 2025, from https://locust.io/

[80] Jay, M., Ostapenco, V., Lefevre, L., Trystram, D., Orgerie, A., & Fichel, B. (2023). An experimental comparison of software-based power meters: focus on CPU and GPU. CCGrid 2023 - 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, 1–13. https://doi.org/10.1109/ccgrid57682.2023.00020

[81] Castor, F. (2024, July 16). Estimating the Energy Footprint of Software Systems: a Primer. arXiv.org. Retrieved March 17, 2025, from https://arxiv.org/abs/2407.11611v2

[82] De Souza, K. (n.d.). PowerKap - A tool for Improving Energy Transparency for Software Developers on GNU/Linux (x86) platforms. In Imperial. Imperial College London. Retrieved March 17, 2025, from https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1617-ug-projects/Krish-De-Souza---PowerKap%3B-A-tool-for-Improving-Energy-Transparency-for-Software-Developers-on-GNU.Linux-(x86)-platforms.pdf

[83] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., & Ou, Z. (2018). RAPL in action. ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 3(2), 1–26. https://doi.org/10.1145/3177754

[84] Intel. (n.d.). Running Average Power Limit Energy Reporting. Retrieved February 25, 2025, from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

[85] Noureddine, A. (2021, October 27). Measure the energy consumption of Java applications with

JoularJX. Noureddine. Retrieved February 25, 2025, from https://www.noureddine.org/articles/measure-the-energy-consumption-of-java-applications-with-joularjx

[86] Operating Systems and Middleware Group. (n.d.). GitHub - osmhpi/pinpoint: Perf-Inspired Energy Profiling Tool. GitHub. Retrieved June 6, 2025, from https://github.com/osmhpi/pinpoint

[87] Hoffmann, D., & Tarara, A. (n.d.). GitHub - green-kernel/powerletrics: Powermetrics for Linux. GitHub. Retrieved March 17, 2025, from https://github.com/green-kernel/powerletrics

[88] GitHub - powerapi-ng/smartwatts-formula: SmartWatts is a formula for a self-adaptive software-defined power meter based on the PowerAPI framework. (n.d.). GitHub. Retrieved March 17, 2025, from https://github.com/powerapi-ng/smartwatts-formula

[89] Chamberlin, S. (2020, September 14). Measuring your application power and carbon Impact (Part 1) - Sustainable software. Sustainable Software. Retrieved March 17, 2025, from https://devblogs.microsoft.com/sustainable-software/measuring-your-application-power-and-carbon-impact-part-1/?WT.mc_id=green-8660-cxa

[90] Noureddine, A. (n.d.-b). GitHub - joular/MacPowerMonitor: Power Monitor for macOS is a command line tool that read CPU's power consumption through Powermetrics tool. GitHub. Retrieved March 17, 2025, from https://github.com/joular/MacPowerMonitor

[91] Noureddine, A. (n.d.-a). GitHub - joular/joularjx: JoularJX is a Java-based agent for software power monitoring at the source code level. GitHub. Retrieved March 10, 2025, from https://github.com/joular/joularjx?tab=readme-ov-file

[92] Noureddine, A. (n.d.). Overview - JoularJX Documentation. GitHub. Retrieved February 25, 2025, from https://joular.github.io/joularjx/overview.html

[93] Raffin, G., & Trystram, D. (2024, January 29). Dissecting the software-based measurement of CPU energy consumption: a comparative analysis. arXiv.org. Retrieved March 16, 2025, from https://arxiv.org/abs/2401.15985

[94] Hubblo. (n.d.-b). GitHub - hubblo-org/windows-rapl-driver: Windows driver to get RAPL metrics from a bare metal machine. GitHub. Retrieved March 16, 2025, from https://github.com/hubblo-org/windows-rapl-driver?tab=readme-ov-file

[95] Noureddine, A. (n.d.-b). How JoularJX works - JoularJX Documentation. GitHub. Retrieved March 16, 2025, from https://joular.github.io/joularjx/ref/how_it_works.html

[96] Shiv, K., Iyer, R., Newburn, C., Dahlstedt, J., Lagergren, M., & Lindholm, O. (2003). Impact of JIT/JVM optimizations on JAVA application performance. Proceedings of the Seventh Workshop on Interaction Between Compilers and Computer Architectures (INTERACT'03, 5–13. https://doi.org/10.1109/intera.2003.1192351

[97] Horký, V., Libič, P., Steinhauser, A., & Tůma, P. (2015). DOs and DON'Ts of Conducting Performance Measurements in Java. ICPE'15, 337–340. https://doi.org/10.1145/2668930.2688820

[98] Energy Star. (2022). ENERGY STAR Program Requirements for Computers – Final Draft Test Method (Rev. May-2022). In Energy Star. Retrieved February 20, 2025, from https://www.energystar.gov/sites/default/files/asset/document/ENERGY%20STAR%20Draft%20Test%20Method%20for%20Computers.pdf

[99] Energy Star. (2022b). ENERGY STAR Program Requirements for Computers – Final Draft Test Method (Rev. July-2022). In Energy Star. Retrieved February 20, 2025, from https://www.energystar.gov/sites/default/files/asset/document/ENERGY%20STAR%20Computers%20Version%208.0%20Final%20Specification%20Rev.%20July%202022.pdf

[100] Stocker, M. (n.d.-b). GitHub - misto/spring-petclinic-energy-benchmarking. GitHub. Retrieved

March 10, 2025, from https://github.com/misto/spring-petclinic-energy-benchmarking
[101] Brunnert, A., & Krcmar, H. (2015). Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. Journal of Systems and Software, 123, 239–262. https://doi.org/10.1016/j.jss.2015.08.030
[102] The java Command. (2024, October 17). Docs Oracle. Retrieved March 3, 2025, from https://docs.oracle.com/en/java/javase/23/docs/specs/man/java.html
[103] Newland, C. (n.d.). VM Options Explorer - OpenJDK11 HotSpot. Chris Newland 2018-2024. Retrieved March 2, 2025, from https://chriswhocodes.com/
[104] Cloud Native BuildPacks. (n.d.). Cloud Native Buildpacks. Retrieved March 2, 2025, from https://buildpacks.io/
[105] Ruch. (n.d.). GitHub - j-ruch/LakesideMutual-energy-benchmarking. GitHub. Retrieved April 22, 2025, from https://github.com/j-ruch/LakesideMutual-energy-benchmarking
[106] Azimi, R., Tam, D. K., Soares, L., & Stumm, M. (2009). Enhancing operating system support for multicore processors by using hardware performance monitoring. ACM SIGOPS Operating Systems Review, 43(2), 56–65. https://doi.org/10.1145/1531793.1531803
[107] Becker, M., & Chakraborty, S. (2018). Measuring software performance on Linux. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1811.01412
[108] Bonteanu, A. M., & Tudose, C. (2024). Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA. Applied Sciences, 14(7), 2743. https://doi.org/10.3390/app14072743
[109] Colley, D., Stanier, C., & Asaduzzaman, M. (2018). The Impact of Object-Relational Mapping Frameworks on Relational Query Performance. 2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE), 47–52. https://doi.org/10.1109/iccecome.2018.8659222
[110] Rao, R., & Vrudhula, S. (2007). Performance optimal processor throttling under thermal constraints. CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 257–266. https://doi.org/10.1145/1289881.1289925
[111] Capra, E., Formenti, G., Francalanci, C., & Gallazzi, S. (2010). The impact of MIS software on IT energy consumption. AIS Electronic Library (AISeL). Retrieved April 5, 2025, from https://aisel.aisnet.org/ecis2010/95
[112] Anagnostopoulou, V., Dimitrov, M., & Doshi, K. A. (2012). SLA-guided energy savings for enterprise servers. IEEE International Symposium on Performance Analysis of Systems and Software, 120–121. https://doi.org/10.1109/ispass.2012.6189216
[113] Gillis, A. S., & Lutkevich, B. (2024, August 13). What is an embedded system? Search IoT. Retrieved April 30, 2025, from https://www.techtarget.com/iotagenda/definition/embedded-system
[114] Kirvan, P. (2023, March 31). control system. WhatIs. Retrieved April 30, 2025, from https://www.techtarget.com/whatis/definition/control-system
[115] Hanna, K. T., & Daniel, D. (2024, July 18). What is a smart factory? Search ERP. Retrieved April 30, 2025, from https://www.techtarget.com/searcherp/definition/smart-factory
[116] Laros, J. H., Pedretti, K. T., Kelly, S. M., Shu, W., & Vaughan, C. T. (2012). Energy based performance tuning for large scale high performance computing systems. HPC '12: Proceedings of the 2012 Symposium on High Performance Computing, 6. https://doi.org/10.5555/2338816.2338822
[117] Jin, C., De Supinski, B. R., Abramson, D., Poxon, H., DeRose, L., Dinh, M. N., Endrei, M., & Jessup, E. R. (2016). A survey on software methods to improve the energy efficiency of parallel computing. The

International Journal of High Performance Computing Applications, 31(6), 517–549. https://doi.org/10.1177/1094342016665471
[118] Gunnarsson, K., & Herber, O. (2020). The Most Popular Programming Languages of GitHub's Trending Repositories. DIVA. Retrieved April 30, 2025, from https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Akth%3Adiva-280113
[119] Jansen, P. (2022, June 3). TIOBE Index - TIOBE. TIOBE. Retrieved April 30, 2025, from https://www.tiobe.com/tiobe-index/
[120] Fowler, M. (n.d.). CQRS. martinfowler.com. Retrieved April 30, 2025, from https://martinfowler.com/bliki/CQRS.html
[121] Fowler, M. (n.d.-b). Event sourcing. martinfowler.com. Retrieved April 30, 2025, from https://martinfowler.com/eaaDev/EventSourcing.html
[122] Nanayakkara, C. (2023, July 9). Microservices Patterns: Event sourcing | Cloud Native Daily. Medium. https://medium.com/cloud-native-daily/microservices-patterns-event-sourcing-7c6e765681c1
[123] ISO & IEC. (2016c). ISO. Online Browsing Platform (OBP). Retrieved February 24, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec:25023:ed-1:v1:en
[124] ISO & IEC. (1999). ISO. Online Browsing Platform (OBP). Retrieved February 24, 2025, from https://www.iso.org/obp/ui/#iso:std:iso-iec:14756:ed-1:v1:en
[125] ISO & IEC. (2020). ISO. Online Browsing Platform (OBP). Retrieved February 24, 2025, from http://iso.org/obp/ui/#iso:std:iso-iec:21836:ed-1:v1:en
[126] "Resource constraints." (2024, September 3). Docker Documentation. Retrieved March 3, 2025, from https://docs.docker.com/engine/containers/resource_constraints/
[127] "Compose Deploy specification." (2025, January 29). Docker Documentation. Retrieved March 3, 2025, from https://docs.docker.com/reference/compose-file/deploy/#resources
[128] IBM Spectrum Symphony 7.3.1. (n.d.). IBM. Retrieved March 3, 2025, from https://www.ibm.com/docs/en/spectrum-symphony/7.3.1?topic=limits-control-groups-cgroups-limiting-resource-usage-linux
[129] Running PowerJoular in WSL · Issue #68 · joular/powerjoular. (n.d.). GitHub. Retrieved March 6, 2025, from https://github.com/joular/powerjoular/issues/68
[130] powercap or intel-rapl not supported · Issue #10160 · microsoft/WSL. (n.d.). GitHub. Retrieved March 6, 2025, from https://github.com/microsoft/WSL/issues/10160
[131] Ruch, J., & Noureddine, A. (n.d.). Measurement Deviation on Windows 11 · Issue #83 · joular/joularjx. GitHub. Retrieved March 16, 2025, from https://github.com/joular/joularjx/issues/83
[132] Bednarczuk, P., & Borsuk, A. (2022). EFFICIENTLY PROCESSING DATA IN TABLE WITH BILLIONS OF RECORDS. Informatyka Automatyka Pomiary W Gospodarce I Ochronie Środowiska, 12(4), 17–20. https://doi.org/10.35784/iapgos.3058
[132] Lago, P., Meyer, N., Morisio, M., Muller, H. A., & Scanniello, G. (2013). 2nd International workshop on green and sustainable software (GREENS 2013). 2013 35th International Conference on Software Engineering (ICSE), 1523–1524. https://doi.org/10.1109/icse.2013.6606768
[133] Starke, G., & Hruschka, P. (n.d.). arc42 Template Overview. Arc42. Retrieved February 19, 2025, from https://arc42.org/
[134] Hasselbring, W., Carr, L., Hettrick, S., Packer, H., & Tiropanis, T. (2020). From FAIR research data toward FAIR and open research software. It - Information Technology, 62(1), 39–47. https://doi.org/10.1515/itit-2019-0040

Appendix C: List of Figures

Figure 1. A mind map illustrating the main characteristics of enterprise applications
Figure 2. The UML class diagram of the PetClinic application
Figure 3. The entity-relationship diagram (UML class diagram) of the PetClinic application
Figure 4. The UML sequence diagram of the PetClinic application for retrieving all owners
Figure 5. The UML deployment diagram of the PetClinic application
Figure 6. A mind map illustrating the main characteristics of the PetClinic application
Figure 7. The UML class diagram of the customer core and customer management services
Figure 8. The entity-relationship diagram (UML class diagram) of the customer core and customer management services
Figure 9. The UML class diagram of the customer self-service service
Figure 10. The entity-relationship diagram (UML class diagram) of the customer self-service service
Figure 11. The UML class diagram of the policy management service
Figure 12. The entity-relationship diagram (UML class diagram) of the policy management service
Figure 13. The UML sequence diagram of the LakesideMutual application for retrieving all customers
Figure 14. The UML deployment diagram of the service-oriented LakesideMutual application
Figure 15. A mind map illustrating the performance software quality attribute
Figure 16. The definition of specific energy [57]
Figure 17. Relevant metrics for energy efficiency [12]
Figure 18. A mathematical definition of useful work [63]
Figure 19. A mind map illustrating the energy and resource efficiency software quality attribute
Figure 20. A mind map illustrating the correlation between performance and resource and energy efficiency
Figure 21. Example of a JMeter test execution [9]
Figure 22. Overview of the test environment and its components
Figure 23. The computer system architecture for Windows and Linux
Figure 24. The interaction between the sensors, the Scaphandre driver, and the MSRs [94]
Figure 25. The architecture of JoularJX [95]
Figure 26. The application monitoring cycles by JoularJX [95]
Figure 27. The statistical analysis of methods by JoularJX [95]
Figure 28. Summary of the system stack and the interaction of its components
Figure 29. A mind map illustrating the test environment, the system stack, and the interaction of the components
Figure 30. Test setup and cleanup [12]
Figure 31. The PetClinic test plan with global variables in JMeter
Figure 32. The PetClinic test plan with a thread group for the owner endpoint in JMeter
Figure 33. The PetClinic test plan with request configurations to fetch all owners in JMeter
Figure 34. LakesideMutual test plan with thread groups for workflows and customers
Figure 35. All experiments and their test scenarios illustrated as a staircase
Figure 36. The processing time for all requests sent to the PetClinic application reported by JMeter on Windows
Figure 37. The energy consumption of the PetClinic application measured with JoularJX on Windows
Figure 38. The energy consumption of all Spring Boot controllers on Windows
Figure 39. The energy consumption of all operations on Windows

Figure 40. The processing time for all requests sent to the PetClinic application reported by JMeter on Linux
Figure 41. The energy consumption of the PetClinic application measured with JoularJX on Linux
Figure 42. The energy consumption of all Spring Boot controllers on Linux
Figure 43. The energy consumption of all operations on Linux
Figure 44. Comparison of execution times between Windows and Linux
Figure 45. Comparison of energy consumptions between Windows and Linux
Figure 46. The energy consumption of all operations on macOS from the GGS blog [51]
Figure 47. The processing time for all requests sent to the PetClinic owner endpoint reported by JMeter on Windows
Figure 48. The energy consumption of the PetClinic application measured with JoularJX on Windows
Figure 49. The energy consumption of all owner operations on Windows
Figure 50. The processing time for all requests sent to the PetClinic owner endpoint reported by JMeter on Linux
Figure 51. The energy consumption of the PetClinic application measured with JoularJX on Linux
Figure 52. The energy consumption of all owner operations on Linux
Figure 53. The processing time for all requests sent to the PetClinic owner and LakesideMutual customer endpoints reported by JMeter on Windows
Figure 54. The energy consumption of the PetClinic and LakesideMutual applications measured with JoularJX on Windows
Figure 55. The energy consumption of the PetClinic and LakesideMutual controller measured with JoularJX on Windows
Figure 56. The energy consumption of the PetClinic owner and LakesideMutual customer operations measured with JoularJX on Windows
Figure 57. The processing time for all requests sent to the PetClinic owner and LakesideMutual customer endpoints reported by JMeter on Linux
Figure 58. The energy consumption of the PetClinic and LakesideMutual controller measured with JoularJX on Linux
Figure 59. The energy consumption of the PetClinic owner and LakesideMutual customer operations measured with JoularJX on Linux
Figure 60. The processing time for all requests sent to the LakesideMutual customer endpoint reported by JMeter on Windows
Figure 61. The energy consumption of the LakesideMutual application measured with JoularJX on Windows
Figure 62. The energy consumption of the LakesideMutual services measured with JoularJX on Windows
Figure 63. The energy consumption of the LakesideMutual customer operations measured with JoularJX on Windows
Figure 64. The processing time for all requests sent to the LakesideMutual customer endpoint reported by JMeter on Linux
Figure 65. The energy consumption of the LakesideMutual application measured with JoularJX on Linux
Figure 66. The energy consumption of the LakesideMutual services measured with JoularJX on Linux
Figure 67. The energy consumption of the LakesideMutual customer operations measured with JoularJX on Linux
Figure 68. The processing time for all requests sent to the LakesideMutual application reported by JMeter on Windows
Figure 69. The energy consumption of the LakesideMutual application measured with JoularJX on Windows
Figure 70. The energy consumption of the LakesideMutual services measured with JoularJX on Windows

Figure 71. The energy consumption of the LakesideMutual operations measured with JoularJX on Windows
Figure 72. The processing time for all requests sent to the LakesideMutual application reported by JMeter on Linux
Figure 73. The energy consumption of the LakesideMutual application measured with JoularJX on Linux
Figure 74. The energy consumption of the LakesideMutual services measured with JoularJX on Linux
Figure 75. The energy consumption of the LakesideMutual operations measured with JoularJX on Linux
Figure 76. A comparison of aggregated energy consumption of the LakesideMutual application on Windows
Figure 77. A comparison of aggregated energy consumption of the LakesideMutual application on Linux
Figure 78. Parameters for performance testing in a distributed software application [6]
Figure 79. The execution time for the read operation on multiple databases using Spring Data JPA [108]
Figure 80. An example for a closed loop control system [114]

Appendix D: List of Tables

Table 1. A comparison of the PetClinic and LakesideMutual enterprise applications in terms of their characteristics
Table 2. Measurable performance quality attributes and metrics (own presentment)
Table 3. An example of refined user stories according to INVEST
Table 4. A definition of useful work according to the INVEST acronym
Table 5. Specific definitions for efficiency terms (own presentment)
Table 6. Hypotheses for the correlation between performance and resource and energy efficiency
Table 7. An overview of methods and tools to measure performance and energy consumption
Table 8. Techniques to establish a controlled test environment and their applicability in the project thesis
Table 9. System specification of the Microsoft Surface Pro 9 and the Linux remote server
Table 10. Tools and versions used for the experiments
Table 11. The latency and throughput of the PetClinic application on Windows
Table 12. The energy efficiency of the PetClinic application on Windows
Table 13. The performance analysis of the PetClinic application on Linux
Table 14. The energy efficiency of the PetClinic application on Linux
Table 15. The latency and throughput of the PetClinic owner endpoint on Windows
Table 16. The energy efficiency of the PetClinic application on Windows
Table 17. The latency and throughput of the PetClinic owner endpoint on Windows
Table 18. The energy efficiency of the PetClinic application on Linux
Table 19. The latency and throughput of the PetClinic and LakesideMutual applications on Windows
Table 20. The energy efficiency of the LakesideMutual application on Windows
Table 21. The latency and throughput of the PetClinic and LakesideMutual applications on Linux
Table 22. The energy efficiency of the PetClinic and LakesideMutual application on Linux
Table 23. The latency and throughput of the LakesideMutual customer endpoint on Windows
Table 24. The energy efficiency of the LakesideMutual application on Windows
Table 25. The latency and throughput of the LakesideMutual customer endpoint on Linux
Table 26. The energy efficiency of the LakesideMutual application on Linux
Table 27. The latency and throughput of the LakesideMutual Workflows on Windows
Table 28. The energy efficiency of the LakesideMutual application on Windows
Table 29. The latency and throughput of the LakesideMutual Workflows on Linux
Table 30. The energy efficiency of the LakesideMutual application on Linux
Table 31. The latency and throughput deviations across experiments on Windows and Linux
Table 32. An example for inverse correlation between performance and energy efficiency on Windows and Linux
Table 33. An example for strong correlation between performance and energy efficiency on Windows

Appendix E: List of Listings

Listing 1. Example of a JMH test execution [50]
Listing 2. Example command to specify resource constraints on the JVM
Listing 3. Example command to run JMeter in non-GUI mode for load testing
Listing 4. Example command to run JoularJX as a Java agent attached to the JVM
Listing 5. JoularJX config.properties file for the PetClinic application
Listing 6. JoularJX config.properties file for the LakesideMutual application
Listing 7. Docker run command to start the MySQL database
Listing 8. Docker command to initialize the MySQL database with test data
Listing 9. Docker Compose command to start the MySQL databases
Listing 10. The command to start the PetClinic application with the correct system properties
Listing 11. The command to start the LakesideMutual customer core service with the correct system properties
Listing 12. The implementation of the registerCustomer operation in the LakesideMutual application