INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Migration to Microservices: Effects on Energy Consumption, Performance and Maintainability

JOÃO PEDRO DA COSTA CAMPELO
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Migration to Microservices: Effects on Energy Consumption, Performance and Maintainability

João Pedro da Costa Campelo

Dissertation for a Master's Degree in Informatics Engineering, Specialisation in Software Engineering

Supervisor: Isabel Azevedo

Porto, September, 2025

Statement of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section "Ethical considerations" of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P. PORTO.

ISEP, Porto, September, 2025

Dedicatory

“To my family and friends for all the support over the years.”

by João Campelo.

Abstract

The migration to microservices has become a common practice in modern software development. This procedure involves dividing large monolithic applications into several small services that can be independently created, deployed, and scaled on their own. However, like any approach, it has its advantages, such as increasing performance, scalability, and robustness of the system; the impact of this transition on energy consumption has been the subject of discussion.

Microservices enable better resource management by allowing upscaling or downscaling of components, hence leading to underutilization of resources. The use of microservices can lead to energy savings by using fewer resources to achieve the same results. Furthermore, containerization technologies can improve resource utilisation by minimising virtualisation overheads.

However, the higher level of complexity of the microservices architectures may lead to increased energy consumption because such architectures often imply an increased level of network communication and data exchange between services.

Therefore, the impacts of the transition to microservices on energy use are multidimensional and can be best understood when considering the application size and complexity, the infrastructure, and the particular details of the microservices architecture. While there are potential energy savings, it is important to thoroughly weigh the trade-offs and potential impacts on energy consumption before making the decision to migrate to microservices.

The study shows that moving a monolithic application to microservices through pattern-based migration resulted in better system maintainability but led to increased energy usage and performance costs. The application of Green Software Patterns, such as containerisation, asynchronous communication and autoscaling, helped mitigate unnecessary waste and aligned resource usage with workload demand. The CQRS pattern achieved superior read-heavy operation performance yet distributed communication expenses resulted in decreased performance. The maintainability analysis showed that decoupling resulted in more than 90% reduction of propagation cost. The study demonstrates that migration and decomposition patterns increase the number of services which require additional energy, but sustainable design methods enable long-term advantages in scalability and adaptability and environmental sustainability.

Keywords: Energy Consumption, Microservices, Migration Patterns, Decomposition Patterns, Monolithic, Quality Attributes

Resumo

A migração para microserviços tornou-se uma prática comum no desenvolvimento de software moderno. Este procedimento baseia-se na divisão de grandes aplicações monolíticas em vários serviços pequenos que podem ser criados, implementados e escalados de forma independente. No entanto, como qualquer abordagem, tem as suas vantagens, como o aumento do desempenho, da escalabilidade e da robustez do sistema; o impacto desta transição no consumo de energia tem sido alvo de discussão.

Os microserviços permitem uma melhor gestão dos recursos, permitindo aumentar ou reduzir a escala dos componentes, o que leva à subutilização dos recursos. A utilização de microserviços pode levar à poupança de energia, uma vez que utiliza menos recursos para atingir os mesmos resultados. Ademais, as tecnologias de contentorização podem melhorar a utilização de recursos, minimizando as despesas gerais de virtualização.

No entanto, o maior nível de complexidade das arquiteturas de microserviços pode levar a um maior consumo de energia, uma vez que essas arquiteturas implicam frequentemente o aumento da comunicação e da troca de informação entre serviços.

Por conseguinte, os impactos da transição para microserviços na utilização de energia são multidimensionais e podem ser melhor compreendidos quando se considera a dimensão e a complexidade da aplicação, a infraestrutura e os detalhes específicos da arquitetura dos microserviços. Embora existam potenciais poupanças de energia, é importante avaliar cuidadosamente as soluções e os possíveis impactos no consumo de energia antes de tomar a decisão de migrar para os microserviços.

O estudo mostra que a migração de um monolítico para microserviços resultou numa melhor manutenção do sistema, mas conduziu a um aumento do consumo de energia e dos custos de desempenho. A aplicação de padrões de Green Software, como a contentorização, a comunicação assíncrona e o escalonamento automático, ajudou a mitigar o desperdício desnecessário e alinhou a utilização de recursos com a procura da carga de trabalho. O padrão CQRS alcançou um desempenho superior em operações com elevado consumo de leitura, mas os custos com comunicação distribuída resultaram num desempenho reduzido. A análise de manutenibilidade mostrou que o desacoplamento resultou numa redução de mais de 90% no custo de propagação. O estudo demonstra que os padrões de migração e decomposição aumentam o número de serviços que requerem energia adicional, mas os métodos de design sustentáveis permitem vantagens a longo prazo em termos de escalabilidade, adaptabilidade e sustentabilidade ambiental.

Palavras-chave: Consumo Energético, Microserviços, Padrões de Migração, Padrões de Decomposição, Monolítico, Atributos de Qualidade

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
AHP	Analytic Hierarchy Process
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command Line Interface
CPU	Central Processing Unit
CQRS	Command Query Responsibility Segregation
DDD	Domain-Driven Design
DRAM	Dynamic Random Access Memory
eBPF	extended Berkeley Packet Filter
GPU	Graphics Processing Unit
GQM	Goal Question Metric
KEPLER	Kubernetes-based Efficient Power Level Exporter
kWh	Kilowatt-hours
OS	Operating System
RAPL	Running Average Power Limit
REST	Representational State Transfer
SOA	Service-Oriented Architecture

1 Introduction

In this chapter, the dissertation to be presented is introduced as part of the master's degree in Software Engineering at the Engineering Institute of Porto. The first chapter of the study also contains the statement of the problem, together with the general objectives of the study. Furthermore, the chapter ends with a brief on the organisation of the document.

1.1 Context

Currently, the software development industry plays a decisive role in energy consumption. According to Erdenesanaa [1], Companies are continuously investing in the AI domain, creating more energy-consuming models and applications, and the energy consumption of software applications is constantly increasing due to the rise of artificial intelligence users.

However, with the advent of microservices, software developers can reduce the energy consumption of these applications by breaking down monolithic applications into smaller and independent microservices.

Nevertheless, there is a catch. While microservices can increase resource efficiency, they can also result in increased energy usage because of the additional network communication and data transfer required between services. Thus, it is important to consider the trade-offs and possible effects on energy usage before deciding to move to microservices.

Ultimately, the process of decreasing software application energy consumption stands as a vital method for companies to achieve lower carbon emissions and create a sustainable environment. The evaluation of software development environmental effects leads to climate change solutions that support high-performance application development.

1.2 Problem Statement

The Green Software Foundation is an organisation whose goal is to advocate for sustainable practices in software development. It seeks to increase understanding of the effects of software on energy use and greenhouse gas emissions and promotes the adoption of green software development guidelines. The foundation offers information and advice for developers and companies to create energy-optimal and environmentally responsible software. It also stresses the need to consider energy consumption and environmental impact at all stages of the software development life cycle, including design, architecture, deployment, and maintenance. The Green Software Foundation's goal is to help decrease the carbon footprint and establish a sustainable and eco-friendly software industry by encouraging green software practices [2].

While many works have compared monolithic and microservices solutions under different quality attributes, the effect on energy consumption is underexplored. Some specific patterns were examined by Khomh and Abtahizadeh [3], but none focused on migration or decomposition patterns [4]. Based on their experiments, they concluded that “migrating an application to a microservices architecture can improve the application's performance while significantly reducing its energy consumption” [3]. However, this work examines “the impact on energy consumption of six cloud patterns (i.e., Local Database Proxy, Local Sharding-Based Router, the Priority Message Queue, Competing Consumers, Gatekeeper, and Pipes and Filters patterns)” [3].

According to C. Joseph, energy efficiency and green techniques are greatly emphasised. There is a need to analyse the energy consumption patterns of different microservices and the nodes on which they run, and thus, allow researchers to develop approaches that can realise the orchestration and other energy-conscious configuration tasks [5].

The research of Berry et al. presents additional evidence through their investigation of monolithic versus microservice architecture performance, availability and energy usage. This research shows that microservices provide better scalability and resource efficiency during high load conditions, but their energy usage depends on scaling and configuration methods [6].

The selection of architectural style, together with the timing for implementation, requires further evaluation. Such decisions often receive insufficient evaluation regarding their advantages and disadvantages at the appropriate time.

Although the migration from monolithic architectures to microservices has been widely studied, its quantifiable impact on energy consumption, performance, and maintainability throughout a phased migration process remains largely unexplored.

1.3 Objectives and Research Process

The main objective is to quantitatively assess the effects of migration and decomposition patterns on energy consumption, performance and maintainability.

During the migration and decomposition, energy consumption, performance and maintainability for microservices-based architectures were measured.

By doing this, the work offers valuable insights for anyone considering a migration and decomposition of a monolithic into a microservice-based architecture and the relation between energy consumption, performance and maintainability.

Thus, the research questions of this work are the following:

- RQ1: How do the migration and decomposition patterns impact energy consumption?
- RQ2: How do migration and decomposition patterns affect performance?
- RQ3: How do migration and decomposition patterns affect maintainability?

To tackle practical issues in the real world, software engineering research can use a research model proposed by Oates [7]. This model helps analyse and evaluate research undertaken by others to assess whether studies carried out by others provide the needed evidence.

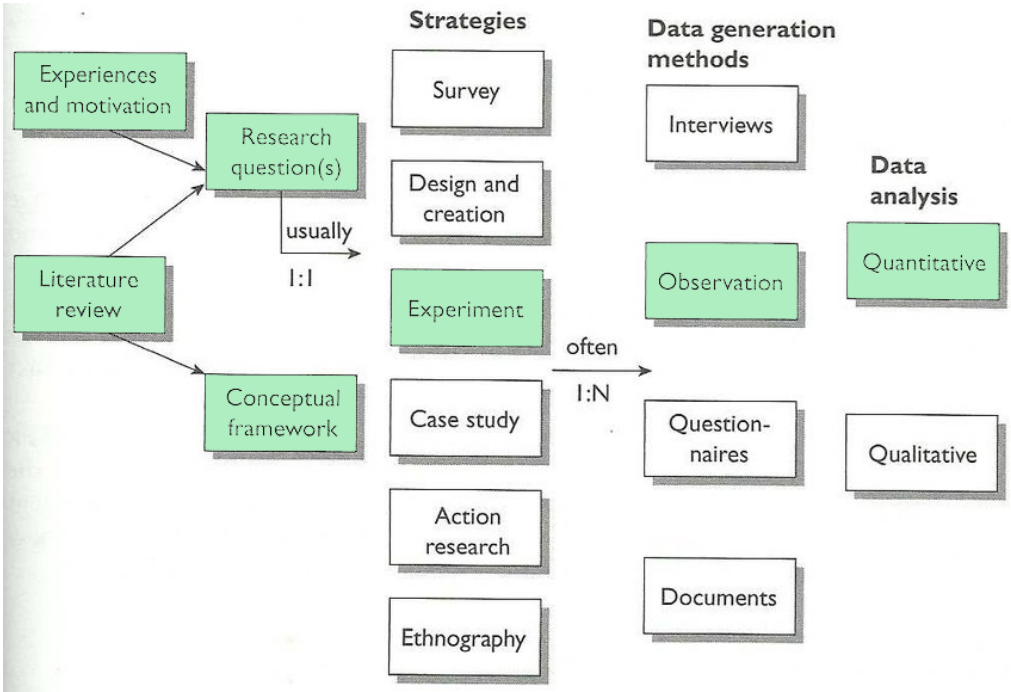

Figure 1 – Model of the research process [7]

In this work, it will be followed the green-coloured roadmap highlighted in Figure 1, which consists of 7 parts:

- **Experiences and motivations:** Brief explanation about the motivation, limitations, and strengths of doing this work.

- **Literature Review:** Involves critically evaluating previous work and identifying themes linking different authors' works. It aims to help determine what has been previously studied and which topics require further exploration.

- **Conceptual Framework:** Explicit structure about the way of thinking about a research topic and how the process is accomplished.

- **Research questions:** Typically defined as a clear, concise, and focused query that a researcher aims to answer through their research.

- **Strategy:** It is the overall approach to answering the research question. Experimentation will be the strategy for answering the research questions for this work.

- **Data generation methodology**: How to produce empirical (field) data or evidence. This work will use the observation method because it analyses collected data from the measurement tool of energy consumption, and which quality attributes that were previously selected.

- **Data analysis**: Based on the data collected during the experiment, quantitative analysis is needed using simple descriptive statistical techniques because all the collected data are numbers.

1.4 Ethical Considerations

The research followed the ethical standards established by Polytechnic University of Oporto (P.PORTO) as defined in the Order no. 11171/2020 – "Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto" [8].

The research work adhered to Article 2 – "Princípios Gerais"/General Principles, which required integrity, honesty, and scientific rigour in its execution. All research findings and conclusions in this work derive from independent investigation. The data remained free from falsification and omission, and manipulation. The research maintained intellectual honesty throughout both the writing and experimentation phases.

The research complied with Article 3 – "Deveres Gerais da Comunidade P.PORTO"/ General Obligations of the P.PORTO Community by upholding institutional and academic rules, particularly concerning intellectual property rights and authorship responsibilities and

responsible research conduct. All external sources, together with tools and references, received proper citation.

The project completion followed Article 6 – “Deveres Específicos dos Estudantes”/ Students' Specific Duties by avoiding any form of unauthorised collaboration. The software created for this research project was developed from start to finish exclusively for this work. The research used external tools only for academic support functions that stayed within acceptable limits. Artificial intelligence tools were used to assist with wording and grammar suggestions in limited sections, representing less than 5% of the document. The AI tools helped with output review and integration, but did not assist in generating original ideas or conducting analysis or producing results. AI tools were used solely for text drafting support and did not contribute to idea generation, analysis, or result production, but it was reviewed and integrated their output to maintain academic integrity.

The research activities maintained scientific rigour and transparency and ensured reproducibility through all research activities according to Article 10 – “Deveres Específicos das Pessoas que Desenvolvem Investigação”/ Specific Duties of Research Developers. The experimental procedures, including energy consumption measurements across microservice architectures, received proper documentation and methodological attention to ensure reliable findings.

1.5 Structure of Document

This section exposes the structure of the document and briefly describes what is discussed in each chapter by summarising its content. The present document is composed of 6 chapters and 7 appendices:

- Chapter 1 – The Introduction chapter gives an overview and a short explanation of the term being discussed. It aims to define the background, the issue to be investigated, and the goals to be achieved. This chapter ends with a section that explains the structure of the document.

- Chapter 2 – The Background chapter explains why this specific research topic is important and crucial to comprehending the main aspects of the study.

- Chapter 3 – The State of the Art chapter is divided into five sub-chapters and explores the current knowledge on the topic. The first sub-chapter presents a systematic mapping study aimed at understanding current research on the topic. The second subchapter concentrates on the interaction between energy consumption in software applications, while comparing energy and power consumption, the impact of software development and programming languages on power consumption and the various methodologies and tools for measuring energy consumption. The third sub-chapter discusses and explains the correlation between the Green Software patterns and the

Microservices patterns. The fourth chapter outlines the crucial tools and platforms required to oversee and fine-tune microservices architectures and containerization. The final sub-chapter highlights work related to the topic of this chapter.

- Chapter 4 – The Design chapter is divided into three sub-chapters. First of all, it presents the desktop description which was employed for this research. Then it outlines all the decisions made to perform the experiment, in other words, the tool selection for measuring energy consumption. Last, it provides a description of the project developed for use in the controlled experiments, including the business context, application architecture, implementation, definition of the migration roadmap and the key performance indicators.

- Chapter 5 – The Implementation chapter explains the modifications made to the original application and their progression towards the final solution. Additionally, this chapter describes the measurements of each metric that were performed and presents the results obtained and finishes with an overall conclusion summary.

- Chapter 6 – Finally, the Conclusions chapter provides a summary of the work done, discusses potential validity threats, outlines future work, and highlights contributions.

- Appendix A – The appendix details the project's expected phases and estimated timeframes.

- Appendix B – The appendix explains the AHP method which determined the tool for energy consumption measurement.

- Appendix C – The appendix explains the AHP method which determined the project to be used on this study.

- Appendix D – The appendix outlines the Grafana configuration to setup the Kepler dashboard and its functionality for retrieving energy consumption data.

- Appendix E – This appendix presents the results related to energy consumption obtained in the course of this study.

- Appendix F – This appendix presents the performance results obtained in this study.

- Appendix G – This appendix provides the maintainability results derived from the study.

2 Background

This chapter aims to offer a systematic examination of the monolith and microservice-based architectures, together with their benefits and drawbacks.

2.1.1 Monolithic Architectures

Monolithic architecture refers to a software design approach that entails building an entire application as a single, tightly coupled unit, with all components and modules of the application interconnected and interdependent, thus forming a single, monolithic codebase. In contrast to other architectural styles, such as Microservices or Service-Oriented Architecture (SOA), the application is not broken down into minor, independent services that communicate with each other [9].

When the application was still small, its monolithic architecture offered numerous benefits, such as simplicity in development since it was focused on building a single application. The code and database schema changes could be easily made and deployed without much hassle. The testing was also a breeze since end-to-end testing was executable by launching the application, invoking the REST API, and testing the User Interface with Selenium. Deploying an artifact on a server was also straightforward, as was scaling the application by running multiple instances behind a load balancer [10].

Regarding C. Richardson [10], if a monolithic application outgrows due to the increasing size of the development team and code base. As a result, the development becomes slow and painful, making agile development and deployment impossible:

- **Maintainability:** The application becomes too complex, making fixing bugs and implementing new features difficult and time-consuming. The overwhelming complexity leads to missed deadlines and a downward spiral. Migrating to a microservices architecture may be a better approach for a complex application.

- **Development:** The large size of the application causes overloads and slows down development. Building and starting the application takes a long time, significantly impacting productivity during the edit-build-run-test loop.

- **Continuous Integration/Continuous Delivery (CI/CD) complexity:** The application faces challenges with deploying changes into production, which becomes a long and painful process. The testing takes a long time due to the complexity of the code base, and some parts even require manual testing. It takes a couple of days to complete a testing cycle. The monolithic system may experience frequent production outages due to its lack of reliability and the difficulty of testing its large codebase. Bugs often make their way into production due to a lack of testability and fault isolation.

- **Scalability:** The monolithic will have problems with scaling, as it requires replicating the entire application, even if only specific components need additional features.

- **Flexibility:** The monolithic architecture makes adopting new frameworks and languages challenging, as the entire application typically relies on a single technology stack. Upgrading the application is expensive and risky, and the team wastes significant time dealing with incompatible frameworks.

2.1.2 Microservices Architectures

According to Newman [9], Microservices are services modelled around a specific business domain and can be released independently. They follow a service-oriented architecture and believe in establishing service boundaries. One of their key features is the ability to be deployed independently. Additionally, they are technology-agnostic, which offers several advantages.

These services operate independently, and each one runs on its own process. They are responsible for performing basic actions and are accessed only through an Application Programming Interface (API) to ensure impermeability.

Each service is designed to handle specific tasks that are often Domain-Driven Design (DDD) oriented [10]. The services are exposed via network endpoints, making their capabilities available to users.

Services are loosely coupled, maintainable, testable, and designed around business capabilities. Small teams own them, and they can be independently deployed. These are the defining characteristics of a service, although not everyone may agree on them.

As stated by Dragoni [11], microservices have the potential to improve various quality attributes. However, in order to achieve their full potential, specific guidelines must be followed. Each microservice should be small, independent, and have well-defined boundaries (Figure 2). They should be designed to isolate failures and embrace an automation culture.

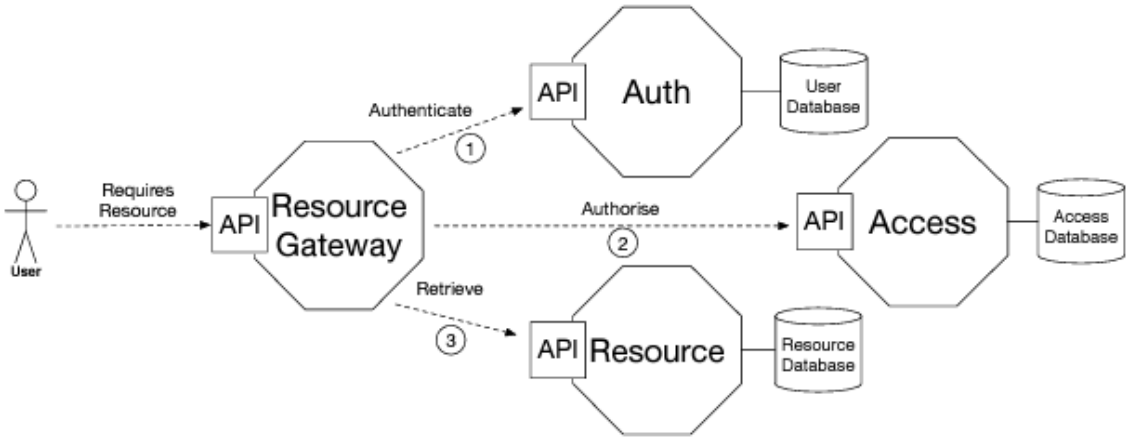

Figure 2 – Example of microservice architecture [11]

Two different architectural styles for developing software systems are SOA and Microservices.

The SOA uses heavyweight technologies like SOAP and WS for communication and relies on a shared database. It is commonly used to integrate large applications [12].

On the other hand, Microservices are considered "SOA done right" because they use lightweight communication technologies such as HTTP Representational State Transfer (REST), Remote Procedure Call (RPC), or message brokers [13]. This architecture follows a database-per-service approach, resulting in a greater degree of modularity. It is characterised by having several smaller services rather than a few large ones, making it suitable for complex systems that require flexibility and scalability.

2.1.2.1 Advantages of Microservices Architectures

The adoption of microservices architecture became popular because it provides organisations with agility and independence between teams, as well as infrastructure automation capabilities and evolutionary design opportunities. Traditional applications face negative effects on productivity due to their growing size and complexity, which leads to infrequent new version releases. Microservices architecture allows organisations to achieve rapid releases and system maintainability, and reuse capabilities, together with scalability features and agile development alongside DevOps principles [11] [14]. Organisations can achieve various benefits by implementing a microservices architecture system. The microservice architecture enables organisations to reduce time-to-market while gaining a competitive advantage.

The microservice architecture has the following benefits:

- **Team Workflow:** The microservices architecture enables several independent cross-functional teams to work independently on their assigned bounded contexts, which results in small teams per context (Figure 3). It consists of various deployable services, which enable the deployment pipeline to operate separately across each service [15].

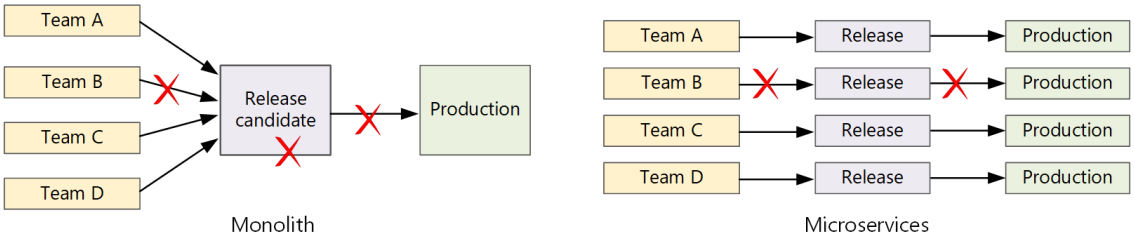

Figure 3 – Deployment of monolithic and microservices apps [15]

Multiple independent services enable short deployment cycles because they operate independently without requiring coordination. The accelerated deployment of new features lets organisations maintain market agility by delivering new features quickly.

- **Availability and Fault Tolerance:** Through deployment independence, microservices allow organisations to perform zero planned downtime or experience minimal service disruptions. The implementation of process automation through microservices provides teams with the ability to deliver new service versions in seconds. Dragoni [11] notes that distributed system availability becomes complicated to manage because monolithic applications need high availability mechanisms to stop failures from destroying the entire system. Overall, microservices offer many advantages over traditional monolithic applications, and teams can benefit from adopting a microservices architecture to build modern applications.

- **Scalability:** Microservices can be scaled individually for specific business capabilities [10]. At the same time, monoliths require the replication of the entire application or an increase in computing power. This flexibility makes microservices more agile and better suited for applications with high user demand and strict non-functional requirements. Cloud computing options also offer the ability to scale resources on demand and utilise auto-scale capabilities.

- **Maintainability:** The long-term maintainability and modifiability of applications based on microservices exceed that of traditional monolithic systems. The initial development phases might present increased complexity, yet every service maintains specific business capabilities which enable straightforward maintenance along with internal structure tracking. Modern applications benefit from effortless modifiability as their essential characteristic. Each service needs to operate independently from others to achieve this goal while minimising inter-service dependencies, which facilitates easy maintenance and functionality extension [10].

- **Deployability:** Modularity in software architecture supports the integration of a DevOps culture, including monitoring, testing, and deployment matters, that aligns well with the granular nature of microservices. Automation of processes is a key fundamental of the DevOps methodology that can significantly reduce time to market by making release cycles shorter [9].

- **Technology Heterogeneity:** According to Newman [9] microservices architectures enable technology heterogeneity. The system consists of numerous services that interact through defined APIs to function independently with their individual technology stacks. Each team can select the most appropriate tool or framework for their specific use cases because the service-only scope of the decision does not require technology commitment.

2.1.2.2 Challenges of Microservices Architectures

While adopting microservices architecture can offer several benefits, it comes with a few significant drawbacks that should be carefully evaluated before making the switch, according to Newman [9].

One of the most notable downsides is the added complexity that arises from managing multiple services. This complexity can lead to increased operation and maintenance costs [4]. Also, the implementation and deployment of microservices requires significant expertise and resources, which may not be readily available.

Another disadvantage is the rise in network latency when services need to communicate with each other, causing delays and decreased performance, which can negatively impact user experience and satisfaction.

Because microservices architecture relies on distributed systems, it may be more prone to security vulnerabilities and breaches [9]. Nevertheless, it can be especially concerning since each service presents a possible vulnerability that hackers could take advantage of.

Technology overload is one major challenge that comes with a microservices architecture. Each service may use different technologies and programming languages based on specific requirements. While having this flexibility is an advantage, it can lead to a varied technology stack, which demands proficiency in multiple tools and languages.

Monitoring performance, identifying issues, debugging, and maintaining the overall system health of a distributed system with many microservices can be difficult. This challenge becomes more complex when dealing with multiple independently deployable services.

Testing is also a challenge in a microservices environment. Traditional monolithic testing approaches may need to be revised to ensure the correct functioning of a distributed system. Challenges include managing dependencies, testing data consistency, and dealing with integration issues.

2.2 Monolithic Migration and Decomposition

The size, complexity and number of systems to be managed and maintained can become unmanageable when they are complex and diverse. One possible way to meet these challenges is to adopt a structure based on microservices. Migration from a monolithic application to microservices is a methodology of dividing the application into discrete, autonomous services. It has certain advantages, including the ones related to the flexibility of the system, its scalability, and durability. Organisations can benefit from enhanced flexibility, improved fault tolerance, and more efficient development processes by adopting a microservices-oriented architecture. Nevertheless, this migration process is a detailed task that demands thorough planning and implementation, as discussed in sections 1.1.1 and 1.1.2.

When migrating from monolith to microservices, it is recommended to proceed step by step and use the experience gained to adjust the process as needed [4]. The migration process can be broken down into two main parts: splitting the monolith and decomposing the database.

2.2.1 Migration patterns

Migration patterns are the ways and means of implementing the transition from one system or architecture to another. Most of the time, this is done without affecting the normal operation and the user's interaction with the application. Each pattern addresses the specific challenges associated with migrating software systems.

The Strangler Fig pattern is a commonly implemented approach during system rewrites that allows new and old systems to coexist [4]. This pattern facilitates incremental migration to a new system while affording the new system ample time to grow and potentially replace the old one. The main benefit of this approach is the possibility to resume or suspend the migration process and still be able to use the new system features that have been introduced up to that point. When applying this pattern to software, every step towards the new application architecture of the software must be such that it can be easily undone, thus reducing the risk of change. This way, every step is reversible and simple and gives more control over the migration process. In summary, the Strangler Fig pattern is an effective strategy for managing system rewrites, allowing for the gradual migration to a new system while minimising risk (Figure 4).

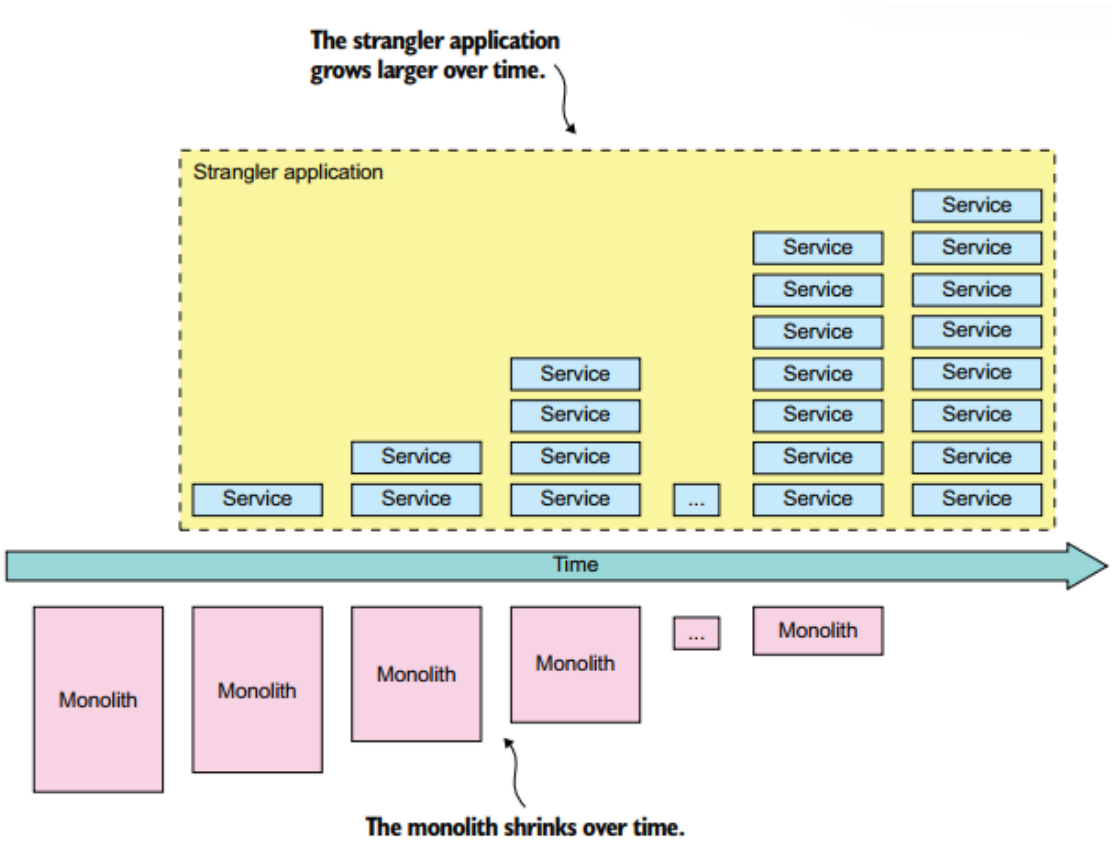

Figure 4 – The strangler fig application over time [10]

The process of migrating from a monolithic system to a series of microservices has often used the strangler fig pattern. The implementation of a strangler fig pattern is predicated on three critical steps, as depicted in Figure 5:

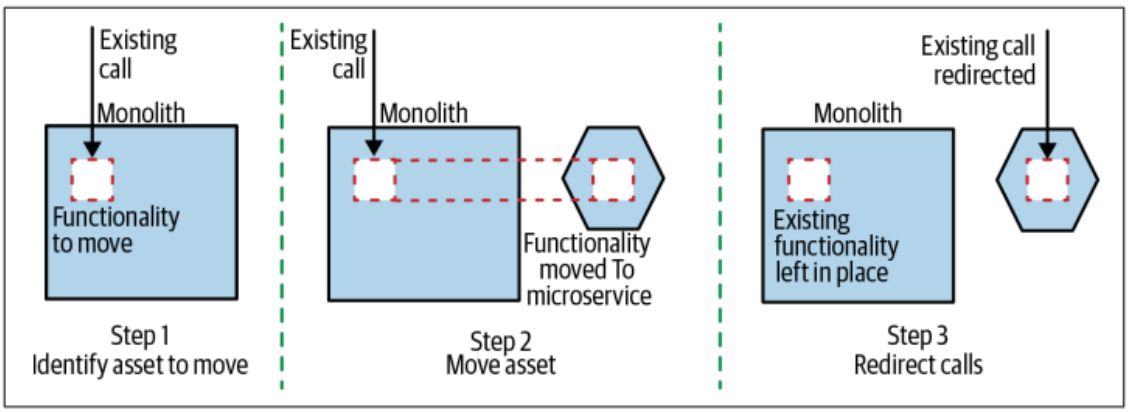

Figure 5 – An overview of the strangler pattern [4]

First, one must identify the functionalities of the current system that require migration and implement them in the new microservice. After the new implementation has been developed, rerouting calls from the monolith to the new microservice is necessary.

A crucial aspect of the strangler application approach is its ability to facilitate quick and easy rollback of changes in case of mistakes. Given that errors can occur, it is important to adopt strategies that allow us to make them as cheaply as possible (via small steps) and rectify them speedily. If other functionalities within the monolith also utilise the extracted functionality, modifying how those calls are made is also imperative.

2.2.2 Decomposition patterns

As previously discussed, the implementation of microservices comes with a variety of options to extract functionality. Data management is a crucial aspect that needs to be addressed. The optimal approach for microservices is to practice information hiding, which often involves completely encapsulating data storage and retrieval mechanisms within each microservice [4]. This necessitates the separation of the monolithic database to achieve the full benefits of transitioning to a microservices architecture.

Nevertheless, splitting a database is a complex and challenging undertaking, and it requires careful consideration of several factors, such as data synchronisation during the transition, the decomposition of logical and physical schema, transactional integrity, latency, and more.

According to Newman [4], the following are the migration patterns:

- Database-as-a-Service
- Synchronise Data in Application
- Split Table

2.2.2.1 Database-as-a-Service

Sometimes, it may be necessary to access a database to execute queries. This may be because of the need to fetch or query large amounts of data or because third-party toolchains already necessitate a SQL endpoint. In such scenarios, it may be logical to enable users to view data that the service manages in a database. However, it is crucial to isolate the database exposed to external consumers from the database we utilise within our service boundary [4].

A feasible approach is to create a dedicated database tailored to be presented as a read-only endpoint. This database should be populated when changes are made to the data in the underlying database. In essence, a service may expose a stream of events as one endpoint, a synchronous API as another, and a database to external consumers as a third endpoint [4].

2.2.2.2 Synchronise Data in Application

The pattern involves migrating part of an existing database or the entire one to a new one. The first step is to do a batch migration of data from the old system into the new one and run it while keeping the existing system running. A change data capture process was implemented to

capture changes since the import. Once the databases were in sync, a new application version was deployed to write data to both databases. The application was tested to ensure that data was correctly written to both sources and that the new database operated within acceptable tolerances. Once enough confidence had been built up in the new system, it switched to using the new database as the source of truth. Finally, the old schema was safely removed [4].

2.2.2.3 Split Table

The Split Table pattern suggests that it's necessary to split the tables apart in the existing schema before separating the schemas and notes that declaring a foreign-key relationship may not be useful if the tables are moved into separate databases. The text also addresses the issue of multiple pieces of code updating the same column. It emphasises the importance of keeping the state machines for domain entities inside a single service boundary. A big problem with splitting tables like this is that it loses the safety given by database transactions [4].

2.3 Quality Attributes in Microservices-based architecture

The architecture of a software application refers to its high-level structure, which comprises constituent parts and the dependencies between them [10].

Quality attributes are properties of a system that can be measured or tested to indicate how well it meets the needs of stakeholders.

Traditionally, architecture has been concerned with scalability, reliability, and security [10]. However, today, it is equally important that the architecture is capable of supporting the rapid and safe delivery of software. The microservices architecture was designed with the primary objective of improving the maintainability, testability, and deployability of an application. Its adoption can result in very important enhancements in the software development processes.

The project objective, mentioned in section 1.3, is to study how energy consumption relates to performance and maintainability throughout migration and decomposition operations. The solution evaluation section focuses on quality attributes because they represent the most critical factors that affect the system. In this section, therefore, the solution evaluation section focuses on quality attributes because they represent the most critical factors that affect the system.

The ISO/IEC 25000 series, known as System and Software Quality Requirements and Evaluation, provides a unified framework to evaluate software quality [16]. Among these series, ISO 25010 presents a model of characteristics and sub-characteristics for describing software products' quality and software quality in use [17]. This model can be used to measure and evaluate the quality of software products.

Microservices architecture involves decomposing a system into small, loosely coupled services that can be developed, deployed, and scaled independently. Applying ISO/IEC 25000 and 25010

can help ensure the quality of each microservice as well as the overall system. This way, it was considered interesting to analyse the quality attributes below to verify if the results allow any conclusion to be drawn.

2.3.1 Maintainability

This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment and in requirements [17, p. 2]. This characteristic is composed of the following sub-characteristics:

- **Modularity –** The extent to which a program system is made up of individual parts that have a low impact on other parts when one part is changed.
- **Reusability –** Degree to which an asset can be used in multiple systems or utilised to create other assets.
- **Analysability –** The measure of how easy and efficient it is to evaluate the effects of a proposed change on a system's components, diagnose product defects or failure causes, and identify parts that need modification.
- **Modifiability –** The capability of a system to be modified without compromising existing product quality or introducing defects.
- **Testability –** measures how well a system can be tested and whether the established criteria have been met. This involves determining the effectiveness and efficiency of the criteria and tests performed.

When supporting maintainability through clean coding practices, modular design, and automated testing, microservices become easy to modify, debug, and enhance.

2.3.2 Energy Efficiency

The increasing adoption of mobile devices, Internet of Things, and cloud services has made energy efficiency a critical concern. The energy cost associated with running and cooling large data centres has led to exploring alternative solutions, such as putting data centres in space or underwater [18]. Energy efficiency has become a quality attribute that architects must consider when designing a system. However, most architects and developers lack suitable design concepts and an understanding of energy efficiency requirements. In addition, energy efficiency must be balanced with performance and availability, requiring engineers to reason about trade-offs consciously [17].

2.3.3 Performance

This characteristic is defined as the performance compared to the resources employed in the study context [17]. This characteristic is composed of the following sub-characteristics:

- **Time behaviour** – The extent to which a system's response, processing times and throughput rates conform to the specified requirements is known as time behaviour.

- **Resource utilisation** – The system's adherence to requirements regarding the resources used and their types when performing functions.

- **Capacity** – Scope to which the maximum limits of a system parameter meet the requirements.

When monitoring tools measure performance, they can optimise the response time and throughput of each microservice, ensuring scalability and efficient resource usage.

3 State of the art

This chapter aims to review the literature, starting with the definition of the review, detailing the research methodology and the results obtained. The literature review is followed by current knowledge about monolithic migration and decomposition to microservices architectures, energy consumption in software development, and energy consumption measurement methodology and tools.

3.1 Research Methodology

In order to collect and understand the current knowledge of how migration and decomposition patterns affect energy consumption in migration to microservices, the first step is to establish an appropriate research methodology. In this case, the use of a Systematic Mapping Study was established. This method enables gathering and synthesising information about a specific field and presenting it in the form of maps, creating a clear structure and classification system. This study can be replicated since the search strategy is well-defined [19].

In order to define a research scope, it was necessary to first define the research questions in section 1.3. After that, the research questions need to be used as a basis for developing research strings, which are then utilised on the search engines of choice to gather all papers that correspond to them [19].

Then, the next step was to choose search engines to find useful literature:

- **ACM Library**: a digital library of publications in computer science and related fields.
- **arXiv.org:** an open-access repository of preprint articles in physics, mathematics, computer science, quantitative biology, quantitative finance, and statistics, where researchers often share their latest findings before formal peer review.

- **B-on**: a Portuguese online library that provides access to scientific journals, books, and databases.
- **IEEE Xplore**: a digital library of engineering, technology, and computer science publications.
- **ScienceDirect**: a platform that provides access to journals, books, and articles in science, technology, and medicine.
- **SpringerLink**: a platform that provides access to various disciplines' journals, books, and articles.

The search keywords in Table 1 and search sub-keywords in Table 2 were formulated to carry out the search based on the research questions.

Table 1 – List of keywords defined

Keyword 1	**Keyword 2**	**Keyword 3**
Software	Microservice	Energy Consumption

Table 2 – List of sub-keywords defined

Sub-keyword 1	**Sub-keyword 2**	**Sub-keyword 3**	**Sub-keyword 4**	**Sub-keyword 5**
Maintainability	Performance	Security	Flexibility	Scalability

The search keywords in Table 1 were created to conduct the research and combined in order to understand the current state of the studies.

The search sub-keywords in Table 2 were used to understand the most studied system quality attributes and the existing gaps.

As the keyword “Energy Consumption” is covered in different areas, all searches took into account the following criteria in Table 3.

Table 3 – Search common criteria

Criteria	**Data**
Area of Study	Computer Science
Date	Between 2013 and 2024
Type of Sources	Conference Materials; Journal Articles; Trade Publications;
Language	English

The first results were obtained after these criteria were applied to the research engines (Table 4).

Table 4 – Initial search results

Keyword combination	ACM Library	arXiv.org	B-on	IEEE Xplore	ScienceDirect	SpringerLink
Software & Microservice & Maintainability	1596	41	3	75	901	201
Software & Microservice & Performance	1968	130	34	979	1109	933
Software & Microservice & Security	1177	42	30	489	825	637
Software & Microservice & Flexibility	1043	33	13	181	793	449
Software & Microservice & Scalability	1790	77	19	475	1115	659
Software & Energy Consumption & Maintainability	15160	41	14	24	14889	252
Software & Energy Consumption & Performance	22590	266	0	5645	22404	8804
Software & Energy Consumption & Security	8834	37	15	1368	10411	4461
Software & Energy Consumption & Flexibility	9722	38	0	505	11798	2782
Software & Energy Consumption & Scalability	18708	33	5	424	18911	3242
Software & Microservice & Energy Consumption & Maintainability	297	0	4	2	315	21
Software & Microservice & Energy Consumption & Performance	354	2	6	36	391	164
Software & Microservice & Energy Consumption & Security	232	1	9	7	316	124
Software & Microservice & Energy Consumption & Flexibility	217	1	1	8	293	91
Software & Microservice & Energy Consumption & Scalability	332	0	2	13	387	114
Software & Microservice & Energy Consumption	359	2	8	74	402	172

Once all documents have been gathered, they must be reviewed to determine which ones meet the specific inclusion and exclusion criteria. After that, a classification scheme should be

developed to categorise the relevant documents. The extracted data can then be organised into frequency tables or bubble plots for better visual representation [19].

To refine the results, the following exclusion criteria were applied in Table 5:

Table 5 – Research exclusion criteria

	Criteria
Inclusion	The paper's abstract mentions all keywords and sub-keywords except the "Software" keyword.
Exclusion	The paper is not available in its full format. The paper only mentions microservices and/or energy consumption in its body of work.

The summary of relevant papers, using the criteria, can be found in Table 6.

Table 6 – Relevant paper's result

Keyword combination	ACM Library	arXiv.org	B-on	IEEE Xplore	ScienceDirect	SpringerLink
Software & Microservice & Maintainability	84	41	1	59	32	201
Software & Microservice & Performance	234	124	11	398	91	933
Software & Microservice & Security	72	37	0	160	43	637
Software & Microservice & Flexibility	72	31	0	102	43	449
Software & Microservice & Scalability	229	74	0	259	99	659
Software & Energy Consumption & Maintainability	247	41	1	15	274	252
Software & Energy Consumption & Performance	1607	236	241	2739	1901	8804
Software & Energy Consumption & Security	180	33	9	487	360	4461
Software & Energy Consumption & Flexibility	173	36	0	265	340	2782
Software & Energy Consumption & Scalability	631	32	3	212	762	3242
Software & Microservice & Energy Consumption & Maintainability	3	0	0	1	1	21
Software & Microservice & Energy Consumption & Performance	5	2	5	8	7	164
Software & Microservice & Energy Consumption & Security	2	1	0	2	0	124
Software & Microservice & Energy Consumption & Flexibility	1	1	2	4	5	91
Software & Microservice & Energy Consumption & Scalability	6	0	1	5	5	114
Software & Microservice & Energy Consumption	10	2	14	22	13	172

The findings highlight the significant trends, gaps and opportunities in the intersection of software, energy consumption and performance. The research shows a strong focus on optimising software performance to increase energy efficiency. The performance of microservices is often a topic of discussion, as a result, highlighting the industry's focus on

delivering the best results possible. At present, there is a lack of research on microservices and energy consumption in relation to maintainability. Future research should investigate how microservices migration impacts maintainability through energy consumption analysis. The discussion of performance for sustainable software practices requires the inclusion of energy consumption metrics. The long-term benefits of focusing on maintainability through energy consumption include reduced total energy costs and improved system reliability.

3.2 Energy Consumption in Software Applications

The growing need for sophisticated software applications results in higher power consumption, which produces additional greenhouse gas emissions. Efforts are required to produce sustainable software products by examining all steps of the software engineering process.

3.2.1 Sustainability in Software Engineering

Sustainable software engineering aims to create reliable, eco-friendly software that meets users' needs without compromising future generations [20]. Green and sustainable software engineering is the skill of creating software while continually assessing its positive and negative effects on sustainable development. The details are maintained as records and used for software product process optimisation.

3.2.1.1 Impact of Programming Languages on Power Consumption

Energy efficiency has become an important concern not only for hardware manufacturers but also for software developers. To this end, researchers have carried out investigations on the dynamics of energy efficiency of data structures, Android language, programming practices and applications for both desktop and mobile devices. They have also created ways of predicting the energy consumption in various software systems [21].

In this paper [21], the authors collected data on energy consumption, execution time, and memory requirements for each of the compilable and executable solutions. As the energy consumption was measured, they used Intel's Running Average Power Limit (RAPL) tool, which is accurate in measuring energy consumption at a granular level. The tool can be called from any C or Java program. Each benchmark solution was run and measured for energy consumption and execution time, and this was done ten times to get ten samples each of energy consumption and execution time. They followed the same approach when collecting results for memory usage in order to reduce the effects of cold starts and cache, analyse the measurements, and avoid outliers.

Before analysing energy consumption in software applications, it would be beneficial to examine the differences and similarities between power and energy. Power differs from energy because it represents the amount of energy transferred during a specific time period. The physical definition of power measures the amount of energy transferred per unit of time

through watts (*W*), while energy measures the total amount of energy consumed or provided during a specific period through joules (*J*). The formula $E = P\Delta t$ expresses energy as power multiplied by time, where P represents power in watts and Δt represents time in seconds [22]. The connection between power and time does not follow a simple pattern because shorter time periods lead to reduced energy usage. The power variable in the equation remains a variable that affects total energy consumption because it does not maintain a constant value. Hence, there are differing conclusions on whether energy and time are directly related or not. Some studies support a direct relationship between the two variables, while others show the opposite.

The study resulted in Figure 6, which describes a table with columns representing the average values for energy consumed (Joules), time of execution (milliseconds), the ratio between energy and time, and peak memory usage (Mb) for different programming languages. The first column categorises the languages as compiled (c), interpreted (i), or virtual-machine languages (v) and may include symbols indicating their position if ordered by execution time or peak memory usage.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 6 – Normalised global results for Energy, Time, and Memory [21]

Even if it is widely known that the top three programming languages, namely C, C++, and Rust, are heavily optimised and efficient in terms of execution performance, the results also support this claim. Based on this information, we hypothesised that these languages would also result in efficient energy consumption, as they possess a significant advantage in one of the variables that influence energy consumption, even if they consume more power on average.

This research may appear peculiar and impractical, but it could help understand the influence of programming languages on energy consumption and help develop more efficient programming languages.

This parameter is becoming increasingly significant and must be considered in the present and future, and this new parameter can be considered in the selection process for programming languages. Moreover, a faster language is not always the most energy-efficient.

3.2.2 Measurement methods

Over the last decade, several power monitoring tools have been developed, each with different approaches and levels of accuracy [23]. These tools can be categorised into two groups: hardware-based and software-based.

3.2.2.1 Hardware-based Approach

The energy consumption gets measured through power meters, which operate as hardware systems. Such devices function as digital measurement instruments which directly measure power or energy consumption from the power supply through Joules or Watts measurement.

The tools require a direct socket connection followed by a computer or server connection before reaching the meter. The measurement of computer or server power consumption needs to happen at the beginning and during software execution to achieve precise energy consumption values. The software's energy usage can be determined by measuring the computer's power consumption at the beginning and during the software run, then taking the difference between these two values.

The Monsoon serves as a precise power meter which operates frequently. Monsoon Solutions Inc. sells this power meter for $998 on its official website [24]. The device functions to measure energy usage from mobile devices, while researchers use it to analyse mobile application power consumption. Users can link this tool to computers to obtain measurement data and view charts [25].

The second is WattsUp Pro, which has been used in a few research studies, including the one by Acar et al. [26]. The device tracks power usage in real time, and it also measures total energy usage. The device shows the collected data on its display screen, and users can transfer data to spreadsheets by using a USB port. Logger Pro and LabQuest apps enable users to generate

graphs according to Hirst et al. [27]. The WattsUp Pro demonstrated high accuracy, but it is no longer in the market, and the only way to get this product now is through the used product markets.

Table 7 – Hardware-based tools

	Monsoon	**WattsUp Pro**
Measurement method	Entire Machine	Entire Machine
Precision	Precise	Precise
Cost	$998,0	Not available

It is important to note that low-cost power meters, readily available from regular online shops, are not designed for measuring software energy consumption. Their primary function focuses on determining the energy usage of household appliances. This can be a problem when trying to use them for software energy measurement, as their design may not be optimal for this use. They may not be very accurate in this use, and they usually have a screen only that does not have the capability of storing data. This can make the measurement process more complicated while also becoming less precise due to these limitations.

3.2.2.2 Software-based Approach

The approach involves using software tools to quantify system energy usage. These tools operate at no cost and function by making predictions instead of direct measurement of energy consumption. These software tools need other hardware tools to perform their calculations. The RAPL serves as the basis for some tools to generate their estimates. The RAPL registers on Central Processing Units (CPU) function to track power usage at millisecond intervals, which enables them to measure CPU package and Dynamic Random Access Memory (DRAM) energy usage. The research performed by Khan et al. [28] proved that RAPL tools deliver highly accurate results.

The software-based approach usually has no cost, and this section will mention some possible tools for use in this approach, divided by measurement method [2].

The Green Software Foundation's GitHub repository, “Awesome Green Software”, is a curated list of tools and resources that promote sustainable software practices. The repository categorises tools into segments, but this work will focus on three main segments: code-based, general-purpose, and operating system-based (OS-based) [29]. Each segment focuses on enhancing the environmental efficiency of software development and deployment.

The code-based tools help developers write more energy-efficient and sustainable code by providing real-time feedback and optimisation suggestions to minimise the software's carbon footprint.

Table 8 – Code-based energy measurement tools [29]

	JoularJX	**Tracarbon**
Description	Software power monitoring at the source code level in real-time.	Tracks your device's energy consumption and calculates your carbon emissions using your location.
Programming Language	Java	Python
Cost	Free	Free
Operating System	Windows Linux	Linux Mac
Energy Measurement	CPU	CPU Memory Graphics Processing Unit (GPU)

The general-purpose tools offer insights and solutions to reduce energy consumption and enhance efficiency in various technological and operational environments beyond software development.

Table 9 – General-purpose energy measurement tools [29]

	Kubernetes-based Efficient Power Level Exporter (Kepler)	**PowerJoular**	**scaphandre**
Description	It uses an extended Berkeley Packet Filter (eBPF) to probe energy-related system stats and exports as Prometheus metrics.	Monitor, in real time, the software's power consumption and hardware components.	Power measurement (bare metal hosts, Prometheus, within a Docker container, etc.)
Cost	Free	Free	Free
Operating System	Linux	Linux	Windows Linux
RAPL usage	Intel RAPL	Intel/AMD RAPL	Intel/AMD RAPL
Energy Measurement	CPU GPU	CPU	CPU

The OS-based tools focus on optimising the energy efficiency of entire OSs. These tools work at the kernel level or provide system-wide optimisations to enhance power management across all running applications and processes.

Table 10 – OS-based energy measurement tools [29]

	ipmitool	**PowerAPI**	**turbostress**
Description	Get the power consumption of a bare metal machine through the DCMI (IPMI extension).	Software-defined power meter to estimate real-time process-scale power consumption.	Generates load and outputs computer power metrics for this load.
Cost	Free	Free	Free
Operating System	Linux	Linux	Linux
Energy Measurement	CPU	CPU	CPU

3.2.2.3 Summary

The hardware is very precise because it is connected directly to the socket, but the RAPL method was shown to be accurate.

The main limitation of the hardware approach is the high installation cost and the limited scalability, as it requires an additional hardware device [23] [30]. However, software-based tools are free and are available for download online, and if the software in question is running on a cloud-based platform, where physical server access is not possible, this approach cannot be employed.

3.3 Green Software Patterns

The growing call for environmentally sustainable practices has reached the field of software engineering, with a focus on minimising the energy consumption of software applications. Green Software Patterns [2] provide strategies for reducing the environmental impact of software systems. The implementation of these patterns works well in monolithic and microservices architectures because each architecture presents different optimisation possibilities and implementation challenges.

One efficient method involves compressing stored data. By decreasing the size of data stored in databases or file systems, less storage space is needed, which in turn reduces energy usage associated with storage devices [31]. Similarly, compressing transmitted data can significantly reduce the energy required for data transmission over networks. This not only saves bandwidth but also eases the load on network infrastructure, contributing to overall energy savings [32].

The approach of containerising workloads serves both monolithic and microservices architectures by providing essential benefits. Containers permit a lightweight and efficient way to run applications, allowing for better resource utilisation and reducing the overhead compared to traditional virtual machines [33]. By optimising average and peak CPU utilisation,

software can run more efficiently. This involves adjusting applications to use the CPU resources more effectively during both regular and peak loads, ensuring that energy is not wasted during idle times or excessive spikes in demand [34] [35].Optimising storage utilisation involves managing storage resources more efficiently, ensuring that unnecessary data is removed and data retrieval processes are streamlined. This reduces the energy storage devices consume and improves overall system performance [36]. Additionally, reducing transmitted data decreases the energy used for data transmission and improves application performance and user experience by minimising latency [37].Optimising storage utilisation involves managing storage resources more efficiently, ensuring that unnecessary data is removed and data retrieval processes are streamlined. This reduces the energy storage devices consume and improves overall system performance [36]. Additionally, reducing transmitted data decreases the energy used for data transmission and improves application performance and user experience by minimising latency [37].

When not in use, scaling down applications is a powerful pattern for saving energy. This involves dynamically adjusting the resources allocated to applications based on their current usage [38].

For Kubernetes-managed environments, scaling down applications and workloads based on relevant demand metrics can result in significant energy savings. This includes scaling down Kubernetes applications when not in use and scaling Kubernetes workloads according to demand, ensuring efficient resource usage only when necessary [39] [40].

Time-shifting Kubernetes cron jobs is another valuable pattern. By scheduling resource-intensive tasks during off-peak hours, energy consumption can be spread more evenly, reducing the load on the infrastructure during peak times and optimising overall energy usage [41].

Incorporating asynchronous network calls instead of synchronous ones can contribute to energy efficiency. Asynchronous calls allow applications to perform other tasks while waiting for network responses, leading to better CPU resource utilisation and reducing system idle time [42].

Optimising the impact on customer devices and equipment involves designing software to minimise the energy consumption of client devices. This can be achieved through efficient coding practices, reducing unnecessary computations, and ensuring the software is not overly demanding on client hardware [43].

Regular vulnerability scanning stands as a fundamental requirement. Software security and the absence of vulnerabilities prevent security breaches, which otherwise could result in resource consumption and energy waste [44].

These patterns enhance both sustainability goals and application effectiveness and performance optimisation. The industry needs to adopt these practices to establish a sustainable technological framework that is environmentally conscious during its ongoing development.

3.3.1 Microservices Patterns

The Green Software Patterns can be effectively integrated with various Microservices Patterns to enhance energy efficiency and optimise resource utilisation.

“Compress Transmitted Data” is a pattern that helps to save energy in data transmission over networks. Especially if we consider the “API Gateway” pattern, which can benefit from data compression before transmitting it. Moreover, the “Event Sourcing” pattern can also be associated with this green pattern as it sends incremental changes instead of whole data sets, which reduces the energy consumption due to the reduced bandwidth requirement.

“Optimising Average CPU Utilisation” is the process of controlling CPU resources during normal conditions to conserve energy. This pattern applies to several microservices patterns, such as the “API Gateway”, which manages CPU resources during normal operations, and “Database-per-Service”, where proper database operations lead to the reduction of CPU load. “Health Check” Patterns ensure that resources are used efficiently during normal operations, while “Saga Patterns” manage distributed transactions to minimise CPU usage. “Service Discovery” Patterns reduce CPU load during normal operations, and deploying services as containers provides lightweight and efficient resource usage. The “Command Query Responsibility Segregation” (CQRS) pattern [45] separates read and write operations to reduce CPU load, and the “Strangler Fig” Pattern improves CPU usage by gradually replacing legacy systems with more efficient components.

“Optimising Peak CPU utilisation” is crucial for preventing unnecessary energy use during periods of high demand. This pattern is relevant to the “API Gateway”, which handles peak loads efficiently. The “Database-per-Service” pattern benefits from efficient handling of peak database demands. “Health Check” Patterns prevent resource overuse during peak times, and “Saga” Patterns handle transaction peaks efficiently. “Service Discovery” Patterns manage peak discovery requests efficiently, and deploying services as containers allows for better handling of peak loads than traditional virtual machines. The “CQRS” pattern efficiently manages peak query and command loads, and the “Strangler Fig” Pattern ensures that peak load energy use is managed well during system transitions.

Another very important green pattern is “Reducing Transmitted Data”, which helps to save bandwidth and energy. The “Event Sourcing” pattern matches this pattern well, as it sends out smaller, incremental changes instead of the entire dataset.

“Using Asynchronous Network Calls Instead of Synchronous” can greatly improve CPU resource utilisation by enabling applications to do other work while waiting for network responses. This green pattern is relevant to “Event Sourcing”, which improves the use of resources by using asynchronous calls. “Saga Patterns” are also enhanced by asynchronous transactions, and “Service Discovery Patterns” use asynchronous discovery calls to improve resource utilisation.

The green pattern of “Containerising Workloads” serves two purposes by reducing overhead costs and maximising resource utilisation. The deployment of services through containers

requires this approach because containers maintain lightweight and efficient resource usage. The “Strangler Fig” Pattern requires containerisation of new components to enhance resource management.

“Scaling Down Applications When Not in Use” involves dynamically adjusting resources to usage, so as to avoid power leakage. This pattern is relevant to “Health Check Patterns”, which guarantee that resources are used optimally as per demand. The deployment of services as containers enables energy conservation through dynamic scaling functionality.

The “Scaling Down Kubernetes Applications When Not in Use” utilises Kubernetes workload management features to adjust to demand levels for energy conservation purposes. The pattern connects to “Health Check” Patterns and service container deployment, and “CQRS” workload management, because Kubernetes provides dynamic scaling capabilities.

By integrating these Green Software Patterns with corresponding Microservices Patterns, developers are enabled to build environmentally friendly software systems that reduce environmental impact and optimise performance.

3.4 Technologies

The chapter provides essential information about the required tools and platforms that enable software architecture management through microservices and containerization. The chapter presents information about three essential categories of tools, which include orchestration tools and message brokers, and performance testing tools. Each section explores the core technologies, their functionalities, and their comparative advantages to comprehensively understand their roles in contemporary software development and deployment.

3.4.1 Orchestration

Complex applications that use microservices alongside containerised deployments across multiple hosts gain benefits from orchestration tools [46]. These tools enable developers to integrate distributed containers and control their operation during the development and deployment phases. The two most popular orchestration tools are Docker Swarm and Kubernetes.

Docker Swarm functions as a cluster management solution that operates with Docker Engine to distribute container management across multiple machines. The tool operates within the Docker ecosystem because it shares the same Docker Command Line Interface (CLI), which simplifies transitions from Docker CLI commands to Docker Swarm. Docker Swarm does not provide built-in monitoring and tracing functions, so users need to implement third-party solutions. The services operating within Docker Swarm clusters can experience effortless scaling through the automatic addition of new instances.

Kubernetes emerged from Google in 2014 as a strong orchestration solution which enables users to schedule containers while performing auto-scaling and self-healing, and providing native monitoring and health checks. The unique CLI of this system provides advanced features that make its learning process more complex. Docker Swarm and Kubernetes both enable developers to specify application operation definitions and component interaction rules.

The comparison between Docker Swarm and Kubernetes orchestration tools [47] appears in Table 11.

Table 11 – Comparison between Docker Swarm and Kubernetes

	Docker Swarm	**Kubernetes**
GUI	Third-party dashboards are necessary	The dashboard is integrated
Scalability	The system scales better than Kubernetes, with a maximum increase of 5 times.	The system contains 5000 nodes and 150,000 pods within its clusters. Auto-scaling is more mature.
Logging and Monitoring	Third-party tools are necessary	Inbuilt tools available
Node Support	Over 2000 nodes	Support for up to 5000 nodes
Availability	When an existing host fails, containers automatically restart on a new host.	Pods undergo health checks directly.

3.4.2 Message Brokers

In a microservices context, asynchronous communication using a choreography style is considered best practice. This approach needs a message broker to facilitate the communication between services. Asynchronous communication can be described in different ways: notifications, request/async response, or publish/subscribe [48]. Queuing is commonly used for one-to-one communication, while the publish/subscribe approach is employed for one-to-many communication.

The evaluation of a message broker requires assessment of scalability features, together with message persistence capabilities and supported messaging approaches and delivery guarantees and integrations and latency performance. The three main asynchronous messaging technologies used today are Apache Kafka, together with RabbitMQ and ActiveMQ.

The distributed streaming platform Apache Kafka operates as a high-performing, scalable system rather than a standard message broker, while remaining difficult to understand. The official documentation states that this system delivers messages with latency rates below 2ms when processing numerous messages. Topics in Kafka clusters function as message channels which support multiple partitions that enable parallel message consumption through load balancing. Kafka's distributed commit log maintains message persistence while preserving the

order of all sent events. Kafka supports only the publish/subscribe messaging model, which functions as a one-to-many communication pattern [49].

RabbitMQ emerged in 2007 as a general-purpose message broker which operates using queues. The system supports operation through both queuing and publish/subscribe styles. RabbitMQ supports multiple queue protocols, including STOMP, AMQP, HTTP and others, whereas Kafka accepts only primitives and binary messages. This technology provides support for both persistent and temporary message persistence. RabbitMQ includes message prioritisation functionality, which Kafka lacks [50].

Research findings demonstrated that Kafka and RabbitMQ show equivalent performance in pub/sub operations while producing similar results regarding latency and scalability, and throughput capabilities [51].

3.5 Related Works

A recent systematic review of energy consumption in microservices architectures [52] strongly emphasises performance metrics. The focus is also driven by the significance that performance assumes in the effectiveness of the microservices. They are essential metrics that must be met to guarantee the effectiveness of the product and to fulfil the needs of users. The second most important metric is elasticity, which is used to manage resources and their dynamics. It reduces power consumption and improves the overall performance by the ability of the system to adjust its resources to the demand of the workload.

Even if the review demonstrates that performance and elasticity have received extensive research, but it reveals a deficiency in sustainability metrics studies. Although the current research on sustainable practices remains limited because there exists a substantial opportunity for additional investigation. Future research should concentrate on developing methods to maximise the renewable energy utilisation and enhance data centres' energy efficiency and reduce microservices' operational environmental impact. The long-term sustainability and ethical operation of cloud-based services depend on resolving these sustainability issues. This research demonstrates why researchers should extend their analysis to include sustainability metrics together with performance and elasticity metrics to create environmentally friendly microservices architectures.

In addition to the discussion on microservices performance and sustainability, Faustino et al. [53] offer an in-depth study on migrating from a monolithic to a microservice architecture through a gradual migration strategy. Their research focuses on the common challenges of scalability and agility faced by large software development projects when shifting from monolithic systems to microservices. The research stands out for utilising a modular monolith as an intermediate step, which enables a controlled and organised migration process.

The authors describe a detailed case study involving migrating a large monolithic system to a microservice architecture. The process starts with the refinement of the monolith to its modular

form, i.e. modular monolith. This intermediate step aims at breaking down inter-module relationships, domain entities split, and module interfaces designed to control the system's complexity before migration. In the next stage, these modules are refactored to become microservices. The study also emphasises the need to redesign module interfaces to enable distributed communication technologies like REST for synchronous communication and message brokers for asynchronous communication. Performance optimisations like cache implementation and remote invocation count reduction are also crucial in this stage to address the performance deterioration typical of microservice architectures.

Faustino et al. [53] assess the migration effort and performance effect through a series of tests on the migrated system. They conclude that the modular monolith stage is a substantial refactoring effort, especially on dependency management and performance tuning. However, this study provides a more favourable environment for subsequent facet-to-facet refactoring towards the microservice architecture. Faustino et al. [53] propose a stepwise approach for migrating systems from a modular monolith to microservices, which provides insights and practical guidelines for software architects planning similar migrations.

Overall, this research contributes to the body of knowledge by demonstrating a structured and systematic method for migrating monolithic systems to microservices, addressing the technical and organisational challenges involved. Combined with the insights from the systematic review on energy consumption, these studies provide a comprehensive view of current challenges and strategies in microservices architecture, highlighting the need for performance optimisation, sustainability, and careful migration planning.

4 Design

The chapter describes the decisions taken to conduct the controlled experiment, which includes choosing the energy consumption measuring tool and creating the project to be used in the experiments.

4.1 Selection of Tools

Although the objectives are to study the impact of migration and decomposition patterns on energy consumption, performance and maintainability, one question remains: how do you measure these quality attributes?

The Sonargraph and Apache JMeter were chosen for this study due to their deep understanding of these tools. Sonargraph is a commercial static analysis tool that focuses on architecture and design quality by detecting code dependencies, structural erosion, cyclic dependencies, and other architectural violations. It provides essential maintainability indicators such as structural debt index, modularity metrics, and architecture conformance checks to support the long-term sustainability of software systems [54]. Apache JMeter functions as an open-source performance testing tool which tests application scalability and responsiveness through load simulation to measure response times and throughput, and error rates. The research used these tools to perform a complete evaluation of maintainability and performance within its scope. [55]

For the energy consumption, this study was conducted on a desktop with the following specifications: Linux Debian 12.5.0 OS, kernel version 6.1.0-37, with 32GB of RAM, and an Intel Core I7-14700F processor.

Based on the tools presented in sections 3.2.3 and 3.2.4, the options include power meters and software-based approaches and their combinations and one software-based approach with power meters. The available alternatives are:

- **Alternative 1:** Use a power meter for energy consumption measurement.
- **Alternative 2:** Use a Code-based Software tool to measure energy consumption.
- **Alternative 3:** Use an OS-based Software tool to measure energy consumption.
- **Alternative 4:** Use a Software General proposes tool to measure energy consumption.
- **Alternative 5:** Use a combination of multiple Software-based tools.
- **Alternative 6:** Use a combination of a power meter with a Software-based tool.

The selection of alternatives will use the Analytic Hierarchy Process (AHP) method developed by Thomas L. Saaty [56]. The following step requires the establishment of decision criteria. The defined decision criteria consisted of:

- **Simplicity:** How easy is it to perform energy estimation
- **Reliability:** The accuracy and precision of energy estimation methods
- **Functionalities:** The number of tool functionalities can be interesting to the investigation
- **Cost:** Financial cost associated with measuring the energy consumption

The initial stage of the AHP approach requires developing a decision hierarchy tree, which appears in Figure 7:

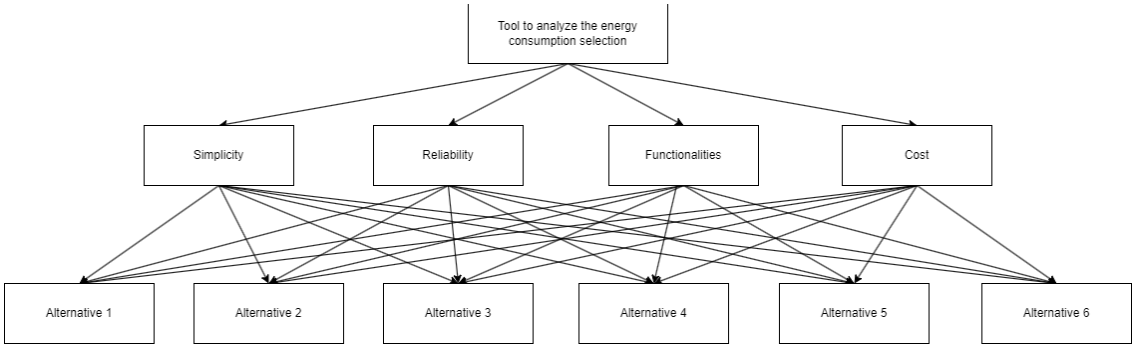

Figure 7 – Decision Hierarchical Tree – Tool

After analysing the values presented in the Appendix B, it can be concluded that Alternative 4 has the highest relative priority in the criteria of Simplicity. In terms of reliability, Alternative 6 has the highest score. Alternative 5 has the greatest score in terms of functionality. As for cost, Alternatives 2, 3, and 4 share the highest value. The final calculation involves multiplying the relative vectors' matrix for each alternative shown in the tables above against the previous priority vector from in Table 22 to generate the global weights for each alternative (Table 12). Each alternative received an overall weight ranking to determine the best solution among them:

Table 12 – Global weight of tool alternatives

	Global Weight	Rank
Alternative 1	0,0968	5
Alternative 2	0,2102	2
Alternative 3	0,2102	2
Alternative 4	0,2110	1
Alternative 5	0,1474	3
Alternative 6	0,1244	4

Based on the AHP analysis, the general-purpose software-based tool (Alternative 4) achieved the highest global weight and is thus the optimal choice. Alternatives 2 and 3 are tied as the second-best options. Kepler stands out as an excellent choice due to its integration with Kubernetes, providing fine-grained, per-process and per-container energy metrics. By combining real-time hardware metrics with machine learning models, Kepler achieves very high accuracy (mean squared error as low as 0.010). Additionally, it automatically calibrates power models and follows GHG Protocol principles to fairly attribute energy use, making it ideal for sustainability-focused initiatives. This combination of accuracy, extensibility, and native support for containerised workloads makes Kepler a powerful framework for energy-aware cloud operations [57].

4.2 Project

4.2.1 Selection of Project

When developing a software system, one of the first strategic decisions requires selecting between adapting an existing project or building a customised solution from scratch. In this case, two approaches were evaluated to determine which offers a more suitable basis for a robust and scalable system, while minimising implementation time and maintenance effort.

As outlined in section 4.2, the decision will be guided using the AHP method. The assessment will be organised based on standards including Complexity and Scalability, Modularity and Architecture, Implementation and Documentation. This method enables us to compare the projects objectively and identify which one best meets the project's objectives and requirements. This way, the following alternatives are available through this approach:

- **Alternative 1:** Use an existing project and adapt it.
- **Alternative 2:** Creating a customised solution from scratch.

The decision criteria defined were:

- **Complexity and Scalability:** The structure of each project is designed to scale in large environments.
- **Modularity and Architecture:** Evaluates the modularity and architecture of each system (microservice vs. monolithic).
- **Implementation:** Level of difficulty in understanding and implementing the code.
- **Documentation:** Quantity and quality of documentation and resources to help with implementation.

The first step of the AHP method is creating a decision hierarchy tree, presented in Figure 8:

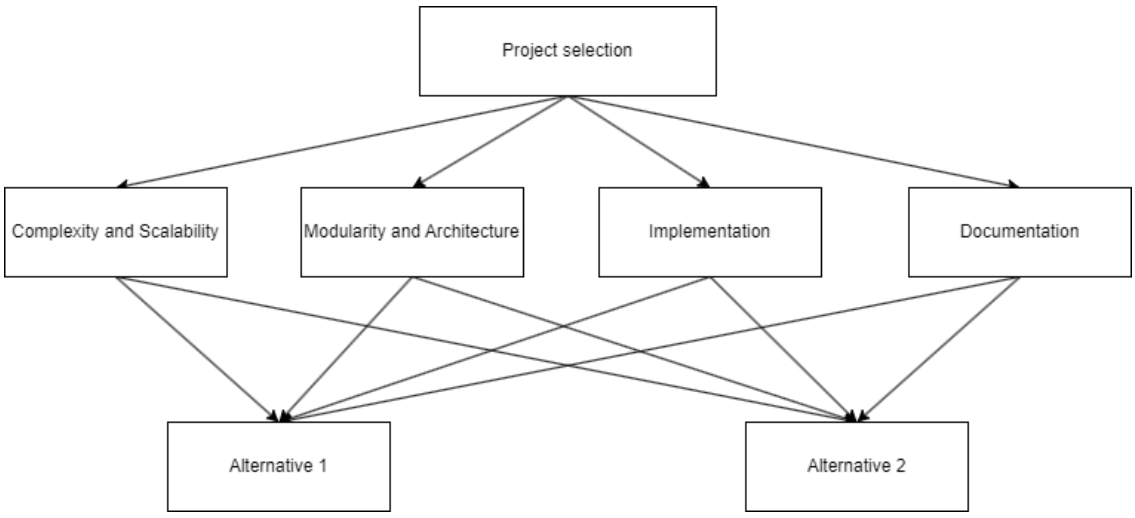

Figure 8 – Decision Hierarchy Tree – Project

After analysing the values presented in the Appendix C, it is clear that Alternative 1 stands as the top choice for the Implementation and Documentation criteria. Alternative 2 has the greatest score in terms of Complexity and Scalability, and Modularity and Architecture.

At last, the global weight for each alternative (Table 13) is the result of multiplying the priority vector, in Table 29, by the matrix of the relative vectors for each alternative represented in the tables above. In addition, the overall weight of each alternative was ranked to define the best alternative:

Table 13 – Global weight of project alternatives

	Global Weight	**Rank**
Alternative 1	0,4841	2
Alternative 2	0,5159	1

This analysis indicates that Alternative 2 is the best option. In other words, creating a customised solution from scratch is the chosen approach.

4.2.2 Project description

A reference monolithic application was designed and developed from scratch for this research. It allows for full control and flexibility, which would be unlikely with a proprietary system, a situation often encountered in experiments. The test application used for conducting the case studies is a retail platform. As mentioned in section 2.2, the complexity level of this application was kept to a minimum to enable the utilisation of the techniques.

The reason for selecting this context is also related to numerous suggestions for defining the domain boundaries of a domain for a retail platform. Although many publications are dedicated to breaking a monolithic system, this dissertation's focus is not specifically on that topic. However, the domain was separated into independent services discussed in section 5.1.

The project is a monolithic Java application that uses the Spring Boot framework for its development, Spring Security for user authentication and authorisation, and it connects to the MySQL database via Spring Data JPA.

The developed project was named “retailproject” and can be accessed on GitHub through the repository at: https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/tree/DIMEI-September-2025. [58]

The Domain Model diagram in Figure 9 displays all the basic entities and their connections, displaying all the basic entities and their correlations.

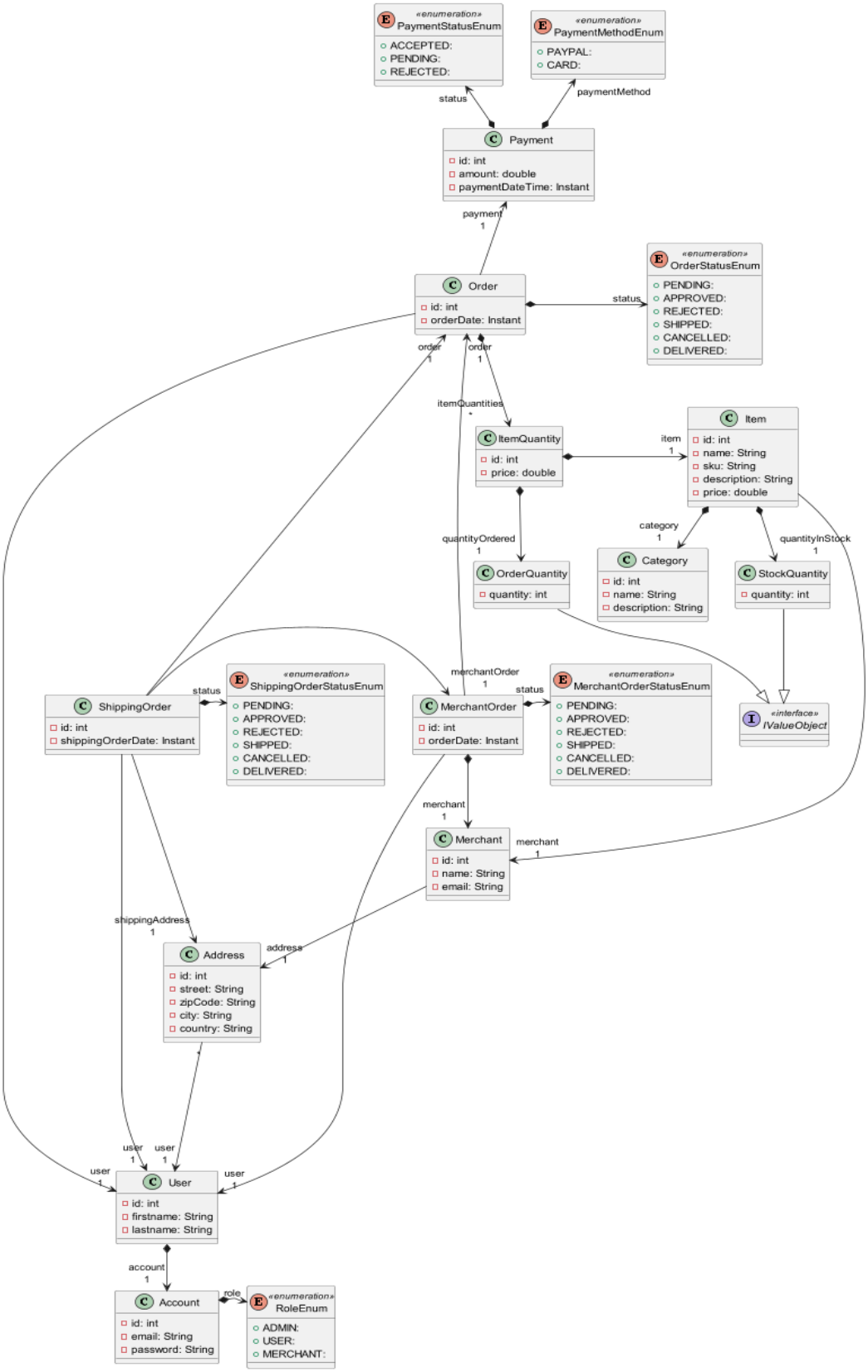

Figure 9 – Domain Model diagram

4.2.3 Business Context

This section presents the application's use cases, which help clarify the system's business context.

They are identified and described in Table 14. The objective is to convert the existing monolith application into microservices while keeping the functionalities intact and ensuring that tests are developed to ensure that the use cases remain unaffected during the migration process.

Table 14 – Functional requirements

Functional Requirement	Description
UC-001	The user must be able to register itself.
UC-002	The admin user must be able to register an admin user.
UC-003	The admin user must be able to register as a merchant user.
UC-004	The admin user must be able to get a list of categories.
UC-005	The admin user must be able to create a category.
UC-006	The admin user must be able to update a category.
UC-007	The admin user must be able to delete a category.
UC-008	The admin user must be able to get a list of all items.
UC-009	The merchant user must be able to get a list of his items.
UC-010	The admin user must be able to get an item using its identifier.
UC-011	The merchant user must be able to get his item using its identifier.
UC-012	The merchant user must be able to create an item.
UC-013	The admin user must be able to update an item.
UC-014	The admin user must be able to delete an item.
UC-015	The admin user must be able to get a list of merchants.
UC-016	The admin user must be able to get a merchant using its identifier.
UC-017	The admin user must be able to create a merchant.
UC-018	The admin user must be able to update a merchant.
UC-019	The admin user must be able to delete a merchant.
UC-020	The admin user must be able to get a list of all merchant orders.
UC-021	The merchant user must be able to get a list of his merchant orders.
UC-022	The admin user must be able to get a merchant order using its identifier.
UC-023	The merchant user must be able to get his merchant order using its identifier.
UC-024	The merchant user must be able to update his merchant order.
UC-025	The admin user must be able to get a list of orders.
UC-026	The user must be able to get a list of his orders.
UC-027	The admin user must be able to get an order using its identifier.
UC-028	The user must be able to get his order using its identifier.
UC-029	The user must be able to create his order.
UC-030	The admin user must be able to delete an order.
UC-031	The admin user must be able to cancel an order using its identifier.
UC-032	The user must be able to cancel his order using its identifier.
UC-033	The admin user must be able to update an order.
UC-034	The admin user must be able to get a list of shipping orders.
UC-035	The user must be able to get a list of his shipping orders.
UC-036	The admin user must be able to get a shipping order using its identifier.

UC-037	The user must be able to get his shipping order using its identifier.
UC-038	The admin user must be able to update a shipping order.

4.2.4 Architecture

The base application was structured in a modular way using a DDD approach. This approach involves modelling software using

the same terminology as the real-world entities of the domain, which can be highly beneficial when creating business software that resolves real-world problems [59].

The application in question is structured in an n-tier architecture, which means that the components of the application are divided into logically separated layers. Each layer is responsible for specific types of entities, and interactions between components are unidirectional. This architectural style provides structure and organisation to the software, with logical boundaries and some modularity. The goal is to prevent software that lacks structure and organisation by defining clear responsibilities.

The n-tier architecture used for developing the monolithic solution consists of controller components, service components, and repositories, as presented in Figure 10. The controller components receive incoming requests, which are processed by the service components, which may depend on each other. The service components use repositories to interact with the database and modify data.

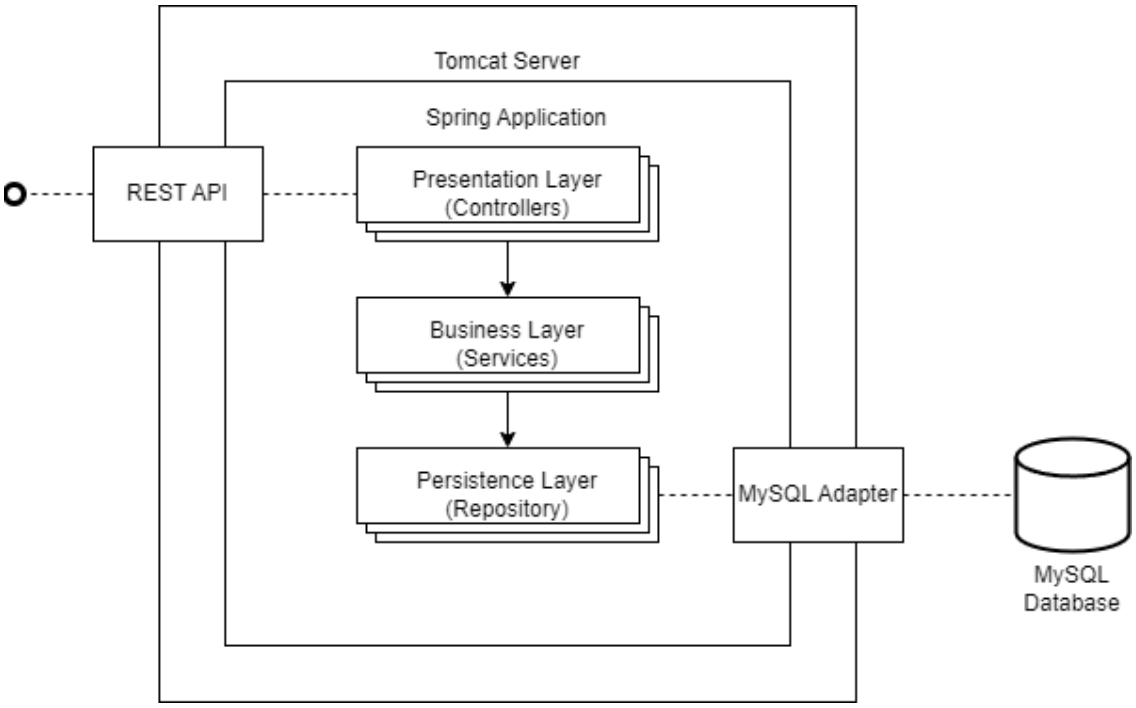

Figure 10 – Base monolithic solution architecture

To support the functionalities in this system, a collaboration of components in different layers is required. Figure 11 represents the implementation view of the selected project.

4.2.5 Implementation

The monolithic application operates as a single process that uses one technology stack and connects to one database. The entire application exists as a Java-based system that uses Spring framework version 3.2.2. JPA functioned as the Object Relational Mapping system to perform database operations, while MySQL operated as the production database. The implementation required uniform technology stacks throughout because using different stacks would increase validity risks. The selection of this specific technology stack happened because of existing familiarity with it.

The functionalities described in section 4.3.1 exist now as a single deployable unit, which is prepared for production deployment. The application operates as a stateless system to enhance scalability and availability, and performance. Multiple instances can operate at once because server-side sessions have been replaced with JSON Web Tokens, which authenticate users through token decoding. The system uses a JSON-based RESTful API as its backend because HTML page server-side rendering was outside the project's scope for optimisation purposes.

Moreover, the implementation adhered to SOLID design principles [60] because experts state that the monoliths become difficult to expand and maintain over time. The developers created separate components with unique functionalities to stop dependency conflicts that would reduce application extensibility.

The traditional application implementation includes a single component and a database that contains all data. The business functionalities exist only at the method-invocation level for local processing. Atomicity, Consistency, Isolation and Durability (ACID) transactions enable developers to easily insert operations that provide complete data integrity mechanisms. A single component contains all codebases, which presents both advantages and disadvantages for developers.

Although distributed implementations present challenges, they naturally lead to enhanced modularity.

A Java project containing dozens of classes emerges from monolithic application development, although its organised structure with minimal dependencies makes it easy to navigate. The development of a sustainable monolithic application demands absolute modularity through independent module creation.

With that in mind, the application structure developed for the final version appears in a class diagram that displays the principal participating classes (Figure 11) and shows the different layers as packages.

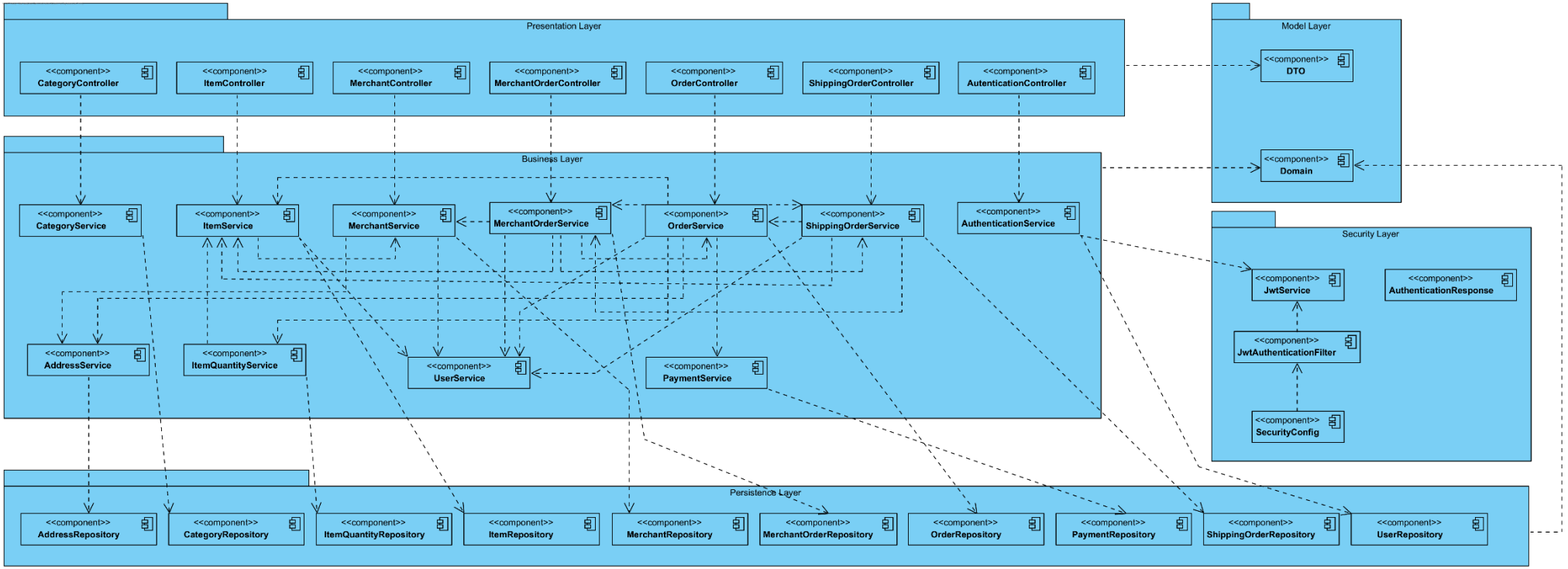

Figure 11 – Monolithic Layer Diagram

- **Presentation Layer:** The only function of this layer is to provide the interface, which includes defining the endpoints and forwarding the requests to the underlying business layer. The business layer is the layer that is immediately below this one.

- **Business Layer:** The core logic of the system is embedded in this layer. This layer gets data-driven processes from the persistence layer that is below it.

- **Persistence Layer:** This layer is in charge of data retrieval, storage and modification from the database.

- **Model Layer:** This layer holds the data, thereby preventing it from being exposed to clients of the presentation layer through DTOs. The Domain is used between the presentation layer and business layer, and also between the business layer and the persistence layer.

- **Security Layer:** This layer, together with the Business Layer, restricts the access of data-driven processes from the persistence layer.

4.2.6 Migration Roadmap

The migration process starts by conducting a thorough evaluation of the monolithic application's present state. The assessment includes evaluating core components and dependencies to identify existing pain points. The current architecture needs a thorough understanding because it serves as the foundation for creating a smooth transition to microservices-based systems.

The Strangler Fig pattern should be used throughout the entire migration and decomposition process. The system transition from monolithic to microservices will occur through a step-by-step process that combines new service development with traffic routing from the original system to the modern system. Initially, parts of the monolith are rewritten as microservices, while the rest of the application remains intact, ensuring minimal disruption.

A significant change occurs during the Strangler Fig implementation: transitioning from a single, monolithic database to a database-per-service. The Split Table pattern is utilised to logically distribute data across different services, allowing each service to manage its own subset of the data. As shown in Figure 12, microservices interact with a dedicated database to maintain their specific data domains.

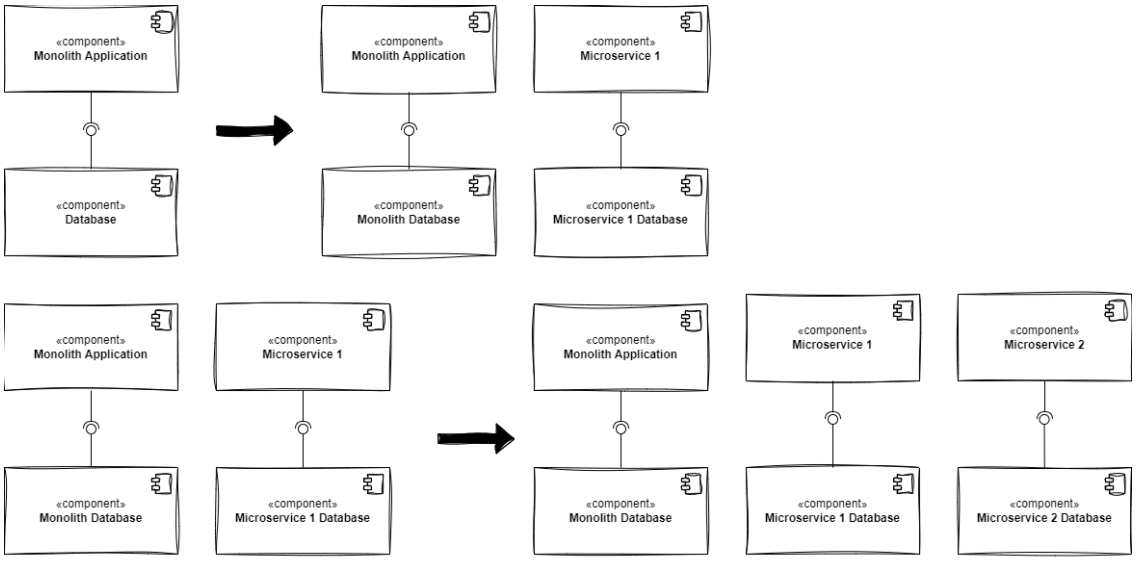

Figure 12 – Strangler Fig and Database per Service pattern

In order to effectively manage this distributed system, service discovery, specifically server-side service discovery, is used. Microservices register themselves with a service registry, enabling them to find the locations of other microservices dynamically. This is complemented by an API Gateway that acts as a single entry point for client requests, handling routing, authentication, and rate limiting tasks.

The application uses various patterns to ensure that data consistency is maintained across these distributed databases. The Synchronise Data in Application pattern ensures that data remains consistent between the old monolithic system and the new microservices during the transition. The Event Sourcing pattern is used to capture every change to the data as a sequence of events, providing a reliable audit trail and enabling the reconstruction of the state by replaying these events [61]. Additionally, Saga patterns, specifically choreography-based Sagas, are implemented to manage complex transactions that span multiple microservices, ensuring eventual consistency without relying on distributed transactions [62].

In the last phase, the system is further optimised with the application of the CQRS pattern as presented in Figure 13. CQRS divides the read and write operations into distinct models, improving data management efficiency and scalability [45]. This is achieved by incorporating the Split Table and Database-per-Service patterns, which guarantee that each microservice manages its data autonomously and effectively.

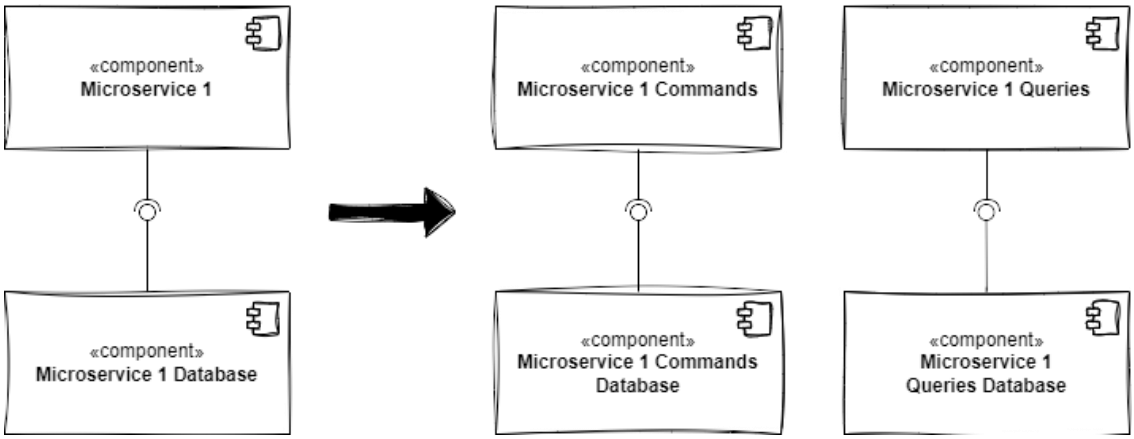

Figure 13 – CQRS pattern

This migration roadmap will result in five migration phases as described in section 5.1.1.

4.2.7 Key Performance Indicators

This project aims to evaluate the energy consumption, performance, and maintainability associated with each migration and decomposition pattern described in section 4.3.4. Hence, all the key performance indicators are linked to these three characteristics.

The Goal Question Metric (GQM) approach provides a measurement model that can be used to determine the metrics in accordance with the goals. This model has three levels: Conceptual, Operational and Quantitative levels [63].

The GQM measurement model is structured in three levels [64]:

- Conceptual Level (Goal): A goal is defined for an object (such as a product, process, or resource) for a specific purpose, based on a particular quality focus, from a chosen point of view, and within a defined environment.

- Operational Level (Question): A set of questions is developed to determine how the goal can be achieved, using a characterisation model. These questions help analyse the object of measurement with respect to a specific quality concern and assess it from the selected viewpoint.

- Quantitative Level (Metric): Several metrics are identified and linked to each question to provide quantitative answers. These metrics may be objective, relying solely on the object being measured, or subjective, influenced by the stakeholder's perspective and context.

The GQM model starts with the formulation of a goal, which is further decomposed into a number of questions to capture the major aspects of the goal. The same metric serves multiple

purposes when applied to various questions under the same goal. After establishing the model, researchers need to identify data collection strategies for measuring the defined metrics [64].

At the conceptual level, the project goal is to explore the effects of migration and decomposition patterns on energy consumption, performance and maintainability, as defined in section 1.3.

Based on this goal, the operational level specifies a set of questions mapping to concrete components of the system being evaluated. The questions are as follows:

- **Q1**: How do migration and decomposition patterns affect energy consumption?
- **Q2**: How do migration and decomposition patterns affect performance?
- **Q3**: How do migration and decomposition patterns affect maintainability?

4.2.7.1 Energy Consumption

In Table 15 presents the metric used to evaluate the energy consumption.

Table 15 – GQM Energy Consumption metrics

Question	**Metric**
How do migration and decomposition patterns affect energy consumption?	**M1** – Total Power Consumption by Container – Kepler

The Kepler tool can aggregate package/socket energy consumption of CPU, DRAM, GPUs, and other host components for a given container and aggregate the energy consumption of all containers running on the node and OS.

4.2.7.2 Performance

In Table 16, it is possible to verify the metric that will be used to evaluate the performance.

Table 16 – GQM Performance metrics

Question	**Metric**
How do migration and decomposition patterns affect performance?	**M2** – Throughput – JMeter **M3** – Average response time (ms) – JMeter **M4** – Error rate – JMeter

When looking at performance, throughput is the system capacity under heavy usage becomes evident through the number of requests an application handles each second. Average response time tells us how long, on average, it takes for the server to reply to a request in mili-seconds, with quicker times meaning a smoother experience for users. The error rate indicates the number of requests that fail to connect which shows application stability through lower values. [55]

4.2.7.3 Maintainability

In Table 17, it is possible to verify the metric that will be used to evaluate the maintainability.

Table 17 – GQM Maintainability metrics

Question	**Metric**
How do migration and decomposition patterns affect maintainability?	**M5** – Maintainability Level – Sonargraph **M6** – Propagation Cost – Sonargraph **M7** – Average Complexity – Sonargraph

In Sonargraph, the Maintainability Level shows how healthy and sustainable a system's structure is over the long term. It takes into account things like cyclic dependencies, duplicated code, and architectural issues, with higher scores meaning the system is better organised and easier to adapt in the future. Propagation Cost measures the distance that changes in one section of code will affect other parts of the system with lower values indicating more modular systems that experience fewer secondary effects. The Average Complexity metric shows the overall complexity of the codebase through its numerical value where lower numbers indicate straightforward logic that is simple to understand and test and maintain but higher numbers point to complex sections that require more management effort. [54].

The value is expressed on a continuous scale from 0 to 100, where higher values reflect a well-structured and maintainable system, and lower values indicate severe architectural problems and erosion. A score near 100 represents an architecturally sound system with low structural debt, whereas values approaching 0 highlight systems that are costly to maintain and difficult to evolve. By offering a clear, numerical representation, the Maintainability Level enables teams to monitor trends over time, assess the impact of refactoring, and compare the sustainability of different projects consistently.

With the goal, questions, and metrics defined, Figure 14 illustrates the resulting GQM map.

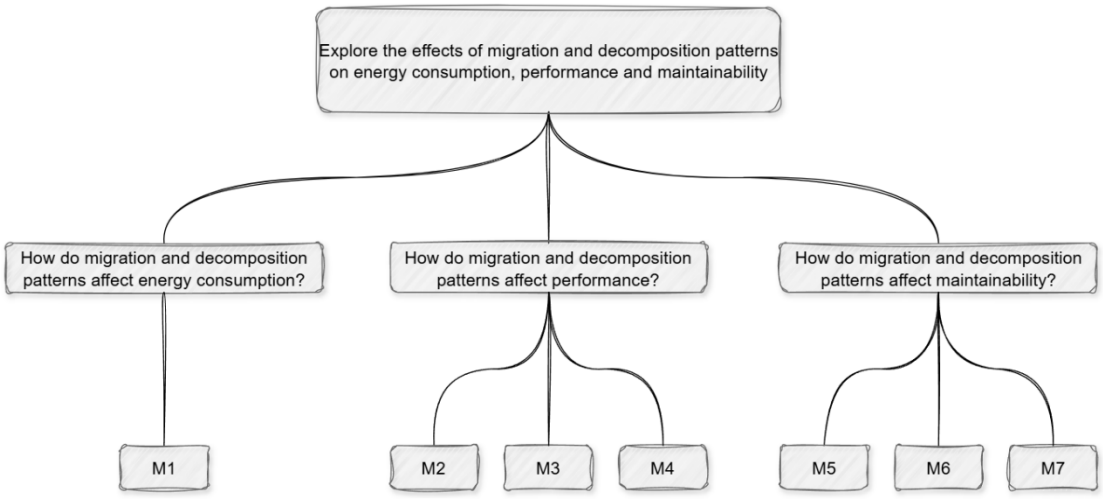

Figure 14 – GQM map

5 Implementation

This chapter describes the migration and decomposition to microservices, which includes the migration process and the monolith segregation, following up on what was mentioned in the chapter 4. In addition, a description of the tool configuration that measures the energy consumption, performance and maintainability and presents the results obtained for each metric.

5.1 Migration and Decomposition to microservices

The shift from monolithic architecture to microservices can be done in steps, and in the initial stages, the first step is to determine the different functional domains in the monolithic system. For each domain, a corresponding microservice is defined that encapsulates the particular business logic.

The migration from monolithic architecture to microservices architecture will be planned through structured planning based on DDD practices, which Eric Evans proposed. DDD presents specific practices and principles which help this transition by defining Bounded Contexts.

The software development method known as DDD focuses on deep domain understanding to direct system design. It consists of two main concepts: the domain, which defines the software's target knowledge area and domain modelling, which produces models that capture domain concepts and entities. All team members utilise the ubiquitous language to explain domain concepts, and entities represent domain concepts through their distinct identity and value objects define domain concepts through their attributes. A repository serves as an interface to manipulate and access aggregates, while aggregates define a collection of entities and value objects that function as a single unit. Aggregates have a root entity known as the root, which serves as the entry point for any operations within the aggregate. This root ensures the consistency of the aggregate’s state. Essential domain operations are handled by services that operate independently of entities and value objects [59].

The DDD concept of Bounded Contexts represents a vital element for performing microservices migration. A Bounded Context defines an explicit boundary where a particular model receives its definition and application [59].

The components of a monolithic architecture maintain strict coupling between each other. Bounded Context identification enables domain partitioning into smaller sections, which allows independent microservice migration of each section. Every Bounded Context should become an independent microservice, which operates autonomously, thus minimising system coupling and enabling independent evolution and scalability of each microservice [59].

Each microservice maintains a clear purpose through Bounded Contexts, which also ensures the system has a uniform domain model. The system becomes simpler to comprehend and sustain maintenance operations. The interaction between microservices occurs through predefined APIs or events. Bounded Contexts establish clear interfaces that enable microservices to function independently without developing excessive dependencies [59].

As shown in Figure 15, this project's domain model was divided into 5 bounded contexts:

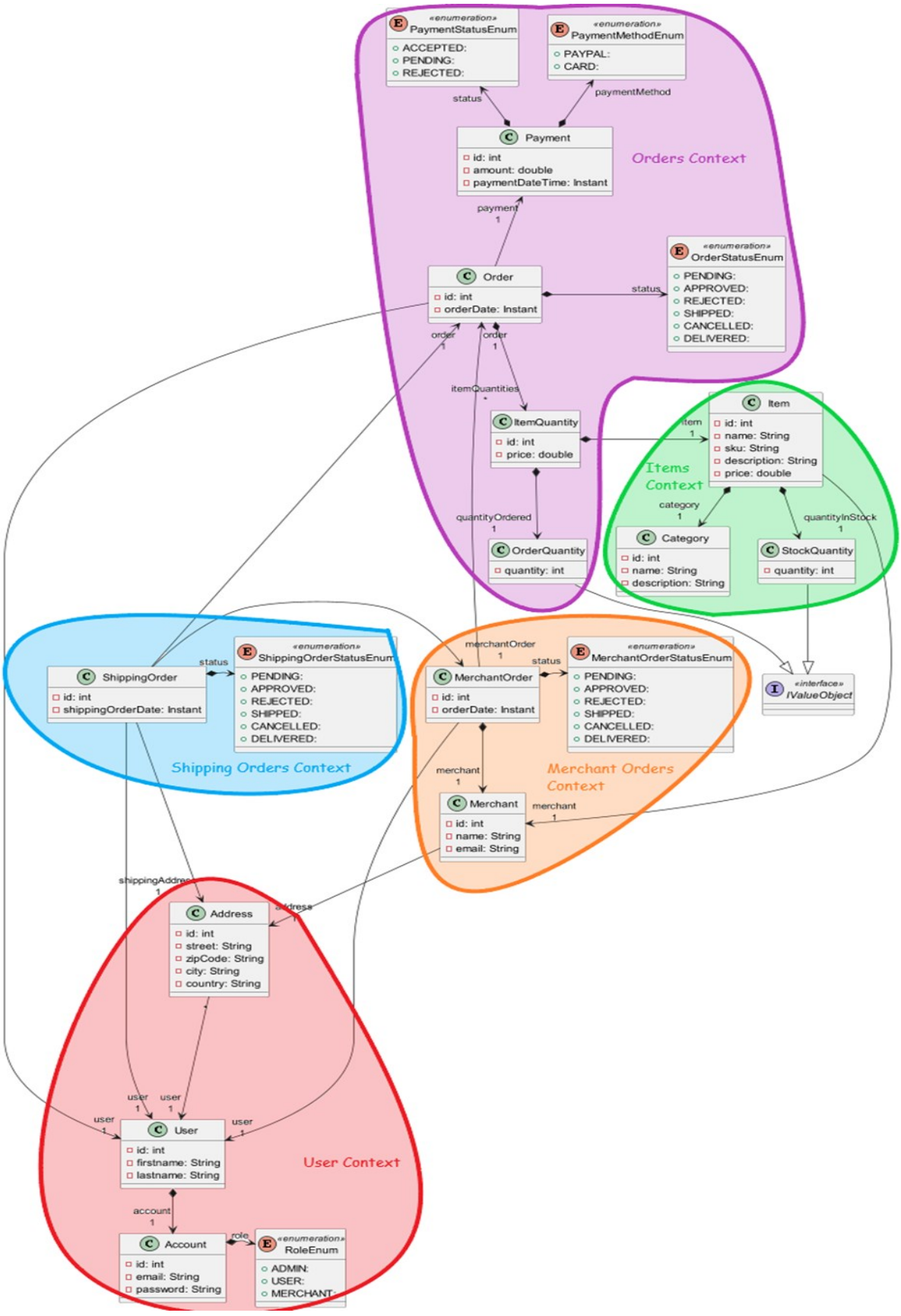

Figure 15 – Project bounded contexts

After the definition of the bounded contexts, it starts to analyse the bounded contexts' dependencies on each other as represented on Table 18.

Table 18 – Bounded Context dependencies

Bounded Context	**Root (Aggregates)**	**Dependencies**	
		In	**Out**
Item Context	Item	2	1
Merchant Order Context	Merchant Order	2	2
Order Context	Order	2	2
Shipping Order Context	Shipping Order	3	0
User Context	User	0	3

To determine the bounded context's migration based on the incoming ("In") and outgoing dependencies ("Out"), it's important to begin with the contexts with fewer "In" and more "Out" dependencies. This grants the contexts more independence to be migrated first, facilitating the migration between them.

The User Context required immediate migration because this context holds authentication logic that other contexts need, and it's independent.

The Items Context follows next because it only depends on the Merchant Order Context and is required for the Order Context.

The Order Context needs migration as the next step because it depends on both the User Context and the Item Context, which have already been migrated and are required by two other contexts. The last two services to migrate are the Merchant Order Context and Shipping Order Context.

Therefore, the migration order is: User Context, Item Context, Order Context, and, finally, Merchant Order Context and Shipping Order Context.

Based on the migration order, this migration was divided into 6 distinct phases.

- Phase 0 – Monolith: At the beginning, the application functions as one monolithic unit where every feature exists within a single project structure. At this stage, the system has high internal cohesion but suffers from limited scalability and difficulties with maintenance and continuous deployment. Green Software Patterns introduced at this stage include containerizing workloads and optimizing both average and peak CPU utilization to prepare the system for sustainable scaling.

- Phase 1 – User Module Migration: The monolith-breaking process starts by separating the authentication module into a User Microservice alongside an API Gateway, which functions as a service entry point and a Load Balancer for request distribution. The system now includes a Discovery Service, which handles service registration to support

dynamic service discovery capabilities. The User microservices will maintain direct point-to-point communication with the other services. Each microservice at this stage chooses to use its own independent database, which leads to total service isolation and enables autonomous scaling. The pattern provides better performance along with service autonomy, yet generates data consistency issues. During this phase, workloads are containerized, applications and Kubernetes deployments are scaled down when not in use, and CPU utilization continues to be optimized to reduce unnecessary resource consumption.

- Phase 2 – Item Module Migration: The Item Microservice stands as the main development during this phase. The system implements Saga Transactions together with Messaging and Event-Driven Architecture patterns to achieve asynchronous consistency between the Item Microservice and the remaining monolithic system. Through event-based communication between services, the system becomes more scalable while decreasing inter-service dependencies. Green Software Patterns applied here include replacing synchronous calls with asynchronous network calls, reducing transmitted data between services, and applying compression to transmitted data, lowering both bandwidth usage and energy consumption.

- Phase 3 – Order Module Migration: The Order Microservice introduction causes business logic to spread between multiple domains, thus demanding distributed transaction handling and eventual consistency, thus reinforcing messaging and events. Containerized workloads remain in use, while optimization of both average and peak CPU utilization continues, ensuring sustainable resource consumption as the system grows.

- Phase 4 – Merchant Order and Shipping Order Modules Migration: The final stage of monolith decomposition involves extracting the Merchant Order and Shipping Order microservices, enhancing domain-driven separation and system decoupling. Green Software Patterns at this stage include continued containerization of workloads, compression and reduction of transmitted data, scaling Kubernetes workloads based on relevant demand metrics, and ensuring applications scale down automatically when idle.

- Phase 5 – Adoption of CQRS: The CQRS pattern serves as the final implementation, which separates the read model from the write model for optimised performance and scalability, especially when dealing with high read traffic. The pattern improves both business logic structure and readability by enabling customised read and write models according to specific needs. The implementation occurs through staged development, which preserves system functionality during new capability deployment. The architecture becomes more robust, adaptable and ready for continuous growth through the implementation of modern practices and proven patterns. At this stage, all Green Software Patterns are fully embedded: compressing transmitted data, asynchronous communication, reducing transmitted data, containerized workloads,

CPU utilization optimization, scaling down idle applications, and demand-driven scaling of Kubernetes workloads.

Based on what has been described, the architectural transformation includes multiple distinct components which support modularisation, scalability and asynchronous communication across services.

The following table presents the essential components which form the new microservices architecture:

Table 19 – Microservices Components

Components Description	**Services Components**
The Load Balancer is a Spring Boot application to distribute network traffic dynamically across resources.	Load Balancer
The API Gateway is a service registry that uses Eureka Server to allow the services to connect with each other.	API Gateway
The Message Broker is a Spring Boot application that uses RabbitMQ to facilitate asynchronous communication between services by sending and receiving messages through queues.	Message Broker
The Read application is a Spring Boot application that provides an HTTP API to retrieve business context data.	Item Read App Merchant Read App Merchant Order Read App Order Read App Shipping Order Read App
The Write application is a Spring Boot application that provides an HTTP API for creating or updating business context data.	Item Write App Merchant Write App Merchant Order Write App Order Write App Shipping Order Write App
The Read database is a MySQL database that provides shipping orders' data to the Read application.	Item Read Database Merchant Read Database Merchant Order Read Database Order Read Database Shipping Order Read Database
The Write database is a MySQL database that provides shipping orders' data to the Write application.	Item Write Database Merchant Write Database Merchant Order Write Database Order Write Database Shipping Order Write Database
The Authentication application is a Spring Boot application that provides an HTTP API for retrieving, creating or updating business context data.	Authentication App
The Authentication database is a MySQL database that provides authentication data to the Authentication application.	Authentication Database

With that in mind, the final microservices architecture developed is described in a components diagram that displays all the microservices and their supporting infrastructure as detailed (Figure 16).

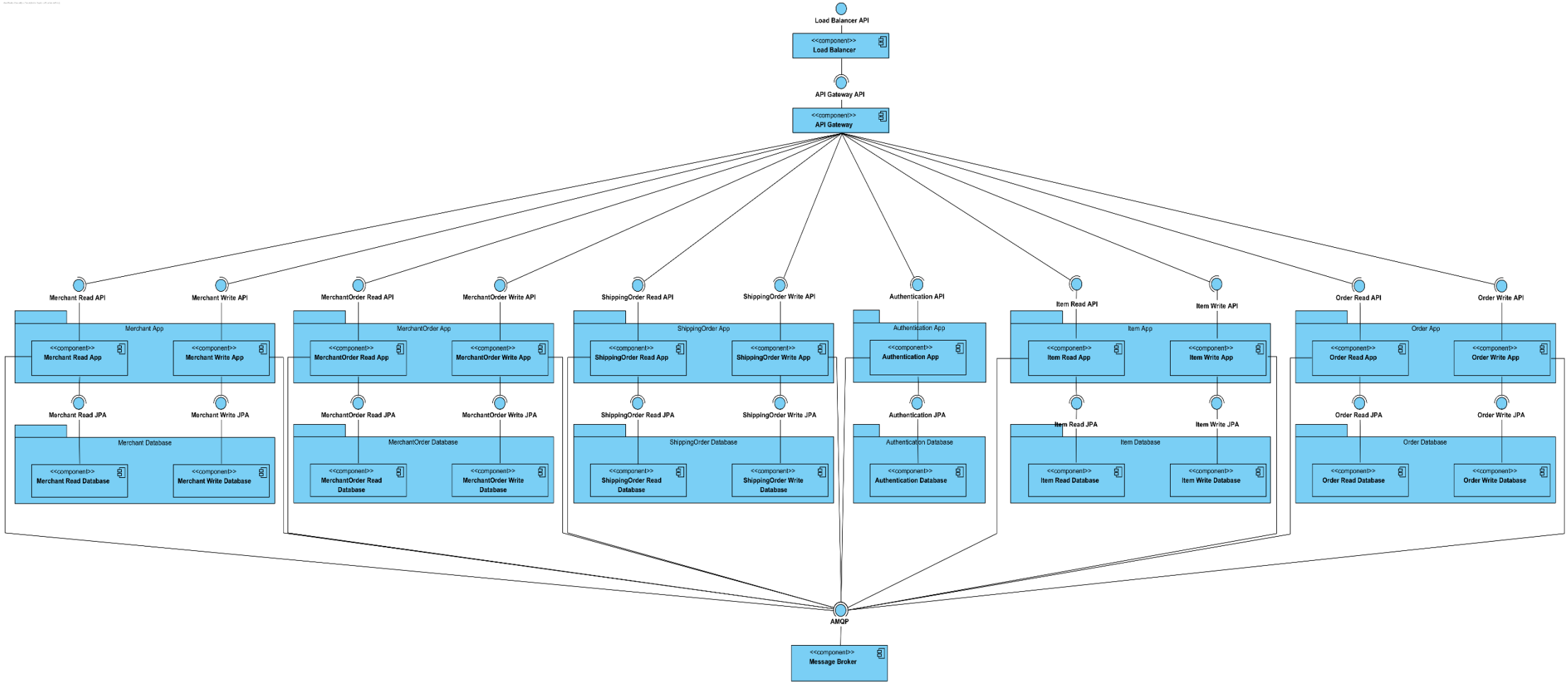

Figure 16 – Microservices Architecture

For each phase, a Jenkins file is provided in the Phase Jenkins subfolder, and the Kepler setup is already included within it, as presented in the Table 20. [58]

Table 20 – Jenkins pipeline configuration

Phase	Jenkins folder
0	/Monolith/jenkins
1	/PhaseOne/jenkins
2	/PhaseTwo/jenkins
3	/PhaseThree/jenkins
4	/PhaseFour/jenkins
5	/PhaseFive/jenkins

5.2 Energy Consumption Measurement

Local cluster deployment of Kepler using Kind (Kubernetes in Docker) requires multiple precise technical procedures to guarantee proper system deployment and operation. The following procedure describes the process in detail.

The first necessary step requires initialising a local kind cluster. This cluster setup will configure the local k8s cluster, as well, the Prometheus and Grafana instances. After the cluster configuration, it needs to deploy the Kepler instance on the k8s cluster. [65]

The Kepler dashboard setup is detailed in the Appendix D.

The results obtained for each phase can be seen in the Project Repository - Kepler Results [66] and each phase has the three performance tests (Thread Group, BZM and JP@GC).

The energy consumption results are presented in the Figure 17, based on the Table 35 in the Appendix E.

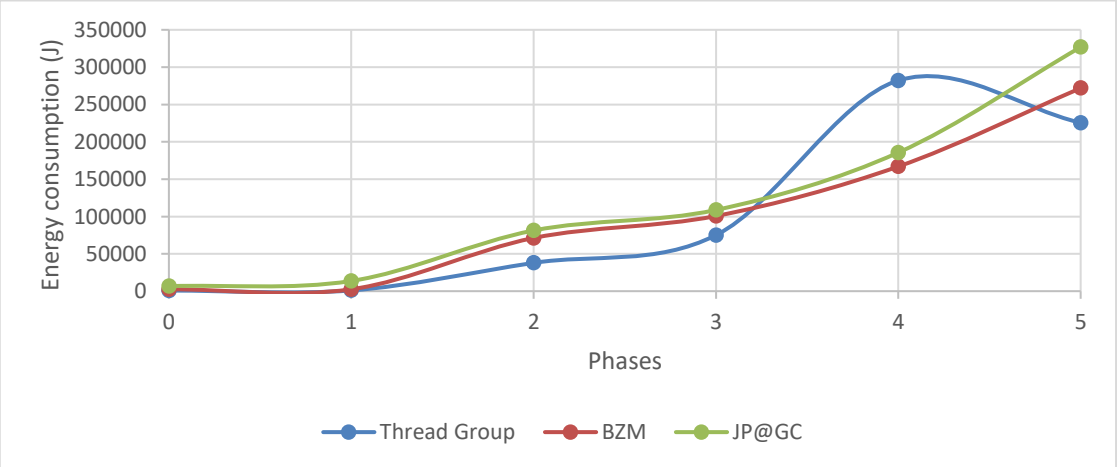

Figure 17 – Energy Consumption Results (Joules)

At the start of Phase 0, when the system was still monolithic but already using green practices such as containerisation and scaling down idle workloads, energy use was very low — about 919 J for the Thread Group, 2,576 J for BZM, and 6,744 J for JP@GC.

In Phase 1, after migrating the user microservices with patterns like Load Balancer and Service Registry, consumption hardly moved: the Thread Group rose by just 9%, while BZM fell slightly and JP@GC increased by 102%, reflecting service initialisation overhead.

The real shift came in Phase 2, when item microservices were migrated with Saga, Domain Events, Messaging, and Database-per-Service. Energy leapt — Thread Group jumped by more than 3,700%, BZM by over 2,700%, and JP@GC by almost 500% compared with Phase 1.

As more services were added in Phase 3 (orders), the figures roughly doubled again, and in Phase 4 (merchant and shipping orders together), Thread Group peaked at an increase of around 650% compared with Phase 3.

Finally, in Phase 5, the introduction of CQRS shifted the profile: Thread Group actually dropped by about 20%, while BZM grew by 63% and JP@GC by 75%, showing the heavier memory and database load of read-only models. Thanks to green software patterns such as data compression, asynchronous calls, and Kubernetes autoscaling, these increases reflected genuine workload demand rather than wasted energy.

5.3 Performance Measurement

The performance evaluation covered all migration stages according to section 4.3.7.2. The evaluation uses Apache JMeter to measure throughput as its main assessment metric.

Apache JMeter presents different thread group configurations to model user load scenarios along with traffic pattern simulation capabilities. The following sections examine the Standard JMeter Thread Group, BZM Concurrency Thread Group, and JPGC Ultimate Thread Group, detailing their configurations and emphasising the necessity of resetting the Kepler-kind cluster before each execution to ensure consistent and reliable test conditions.

The research employs multiple JMeter tests to achieve diverse system performance assessments. The Standard Thread Group defines fundamental system behaviour when under constant load, while the BZM Concurrency Thread Group tests system stability under continuous concurrent operations, and the JPGC Ultimate Thread Group models various traffic patterns to evaluate system performance under stress and spike, and endurance conditions.

5.3.1 Standard JMeter Thread Group

The default JMeter Thread Group provides basic load testing capabilities. The number of threads in this setup stands at fifty to enable concurrent user requests. Each user in the test

performs the test once because the loop count is one and the ramp-up period is zero, which starts all users simultaneously, as shown on Figure 18.

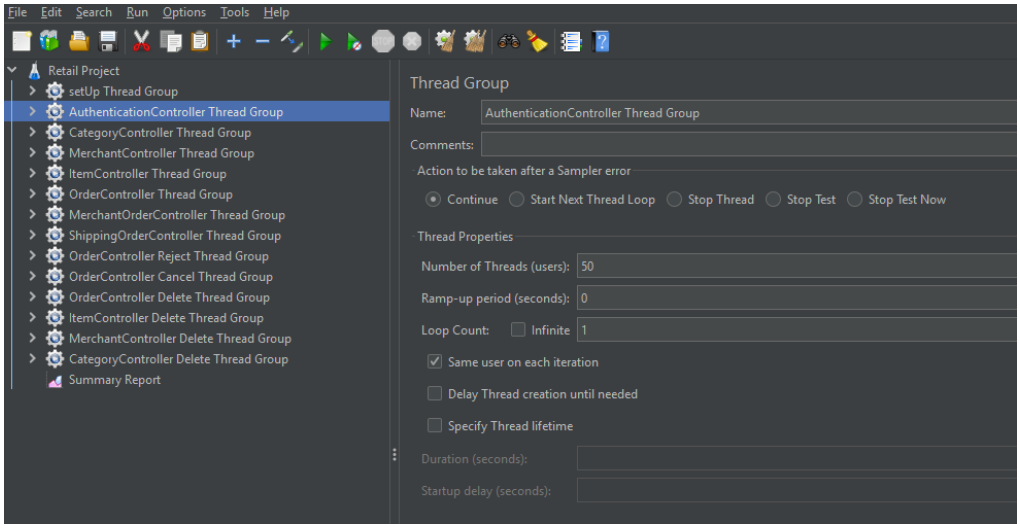

Figure 18 – Standard JMeter Thread Group configuration

5.3.2 BZM (BlazeMeter) Concurrency Thread Group

The BZM Concurrency Thread Group enhances JMeter's capabilities by providing finer control over thread behaviour, ensuring a gradual and structured increase in concurrent users. This test setup is present in Figure 19, the target concurrency is defined as ten, with a ramp-up time of five seconds. The ramp-up step count is left empty, meaning a linear increase in user load. The system holds the target concurrency rate for thirty seconds before termination.

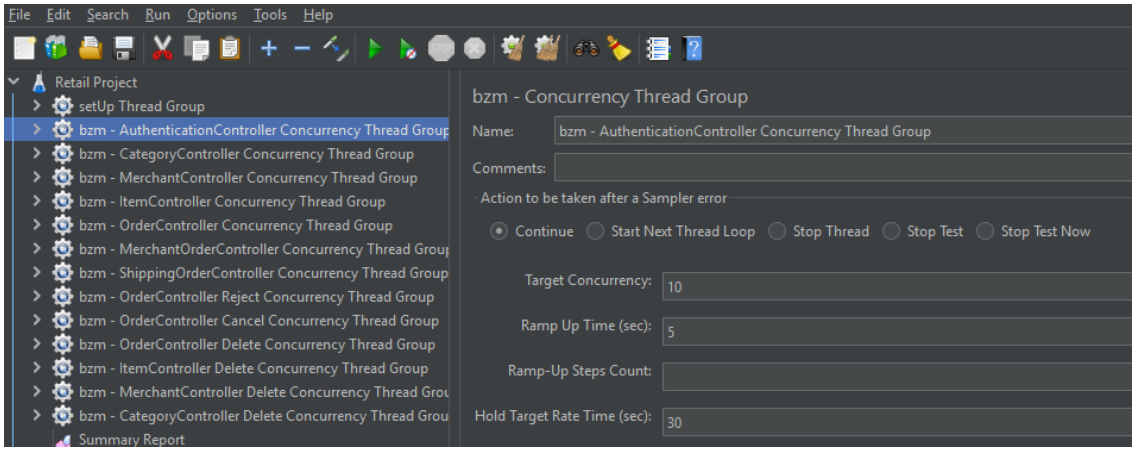

Figure 19 – BZM Concurrency Thread Group configuration

5.3.3 JPGC Ultimate Thread Group

The JPGC Ultimate Thread Group offers high flexibility in defining user load profiles with custom ramp-up and ramp-down sequences. In this configuration, represented in Figure 20, the test begins with ten threads and no initial delay. The startup time is set to ten seconds, ensuring a gradual increase in concurrent users. The system sustains the simulated user load for twenty seconds before initiating a controlled shutdown over ten seconds.

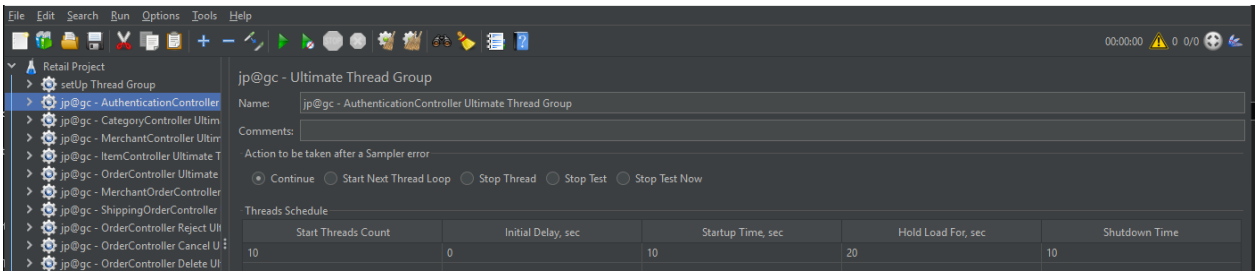

Figure 20 – JPGC Ultimate Thread Group configuration

5.3.4 Performance results

The results obtained for each phase can be seen in the [Project Repository - JMeter Results]. [67]

As outlined in Table 36, Table 37and Table 38 of Appendix F, the final results are represented in Figure 21, Figure 22 and Figure 23.

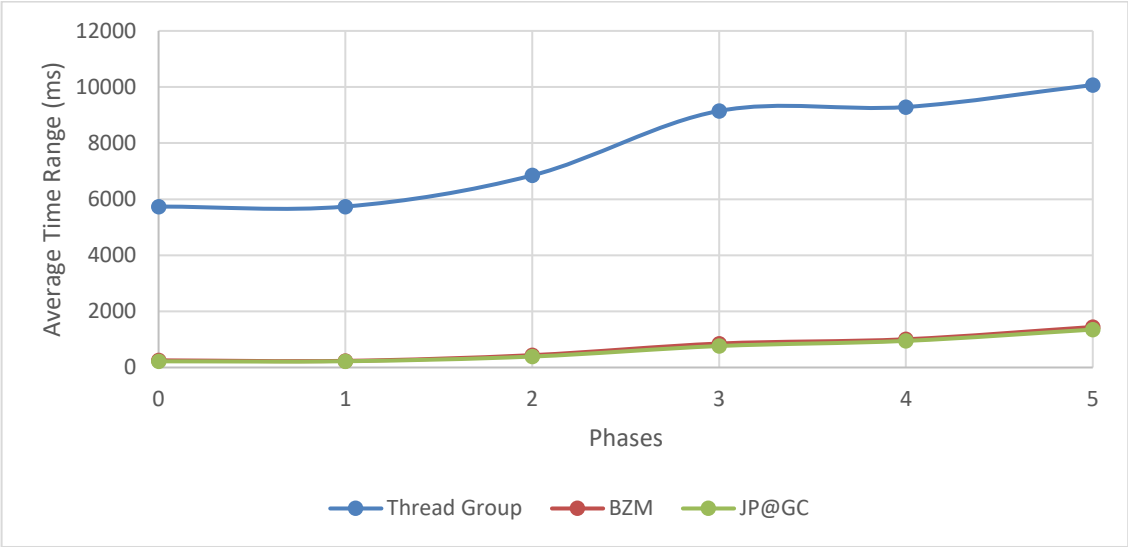

Figure 21 – Performance Average response time Results

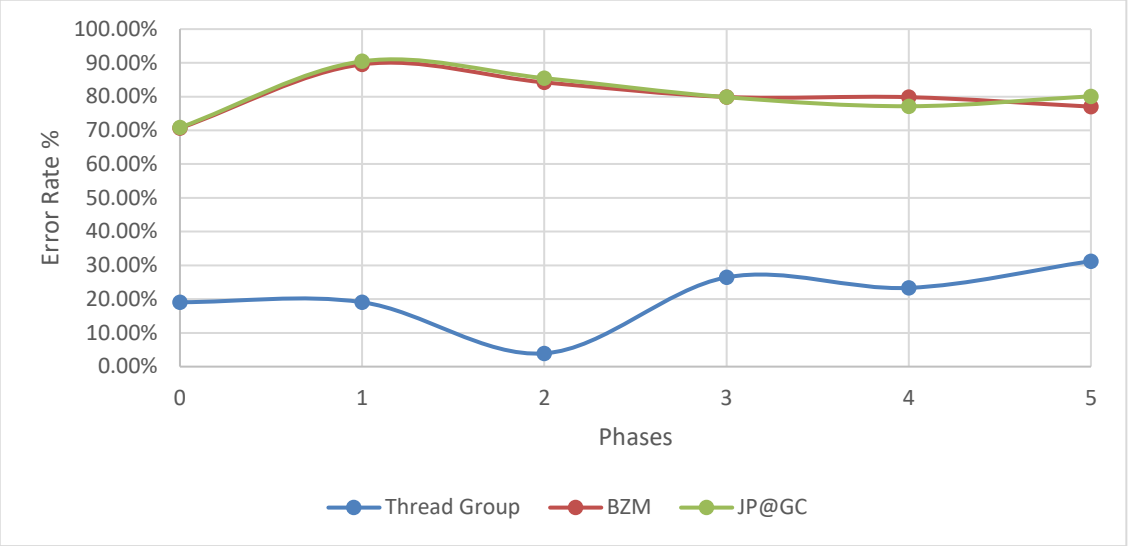

Figure 22 – Performance Error rate Results

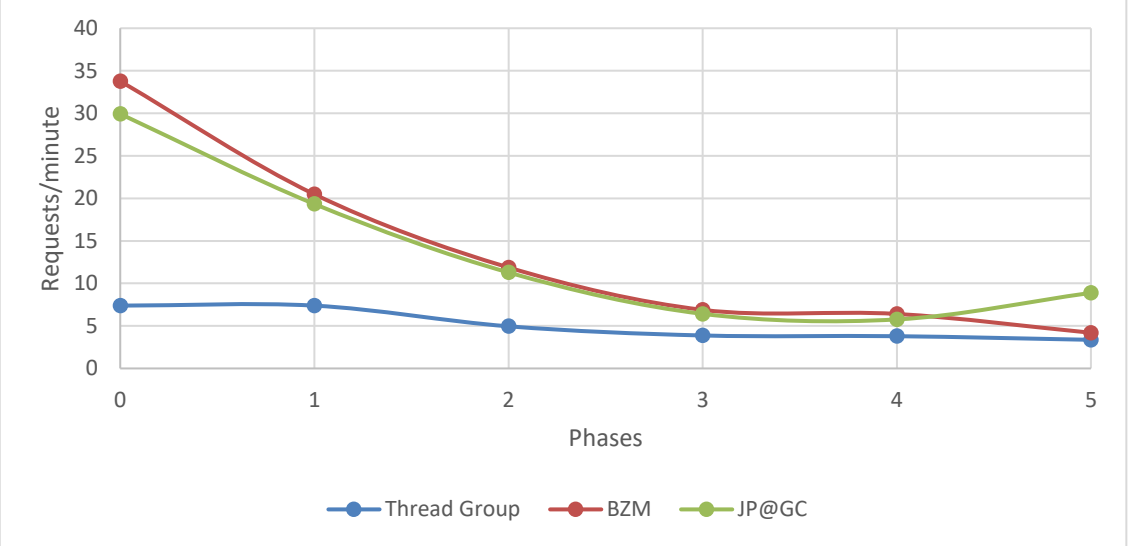

Figure 23 – Performance Throughput Results

The Thread Group maintained a constant request rate of 7.38 requests per second during Phases 0 and 1 yet BZM and JPGC experienced performance drops of 39.4% and 35.4% respectively. The Thread Group maintained an average response time of 5.7 seconds, but plugin scenarios experienced longer response times because API composition created additional fan-out. Error percentages were still relatively low, at around 0.2% for Thread Group and 0.7% for BZM and JPGC. The combination of load balancing with API composition patterns and compression as a green pattern results in read-heavy tests which produce lower throughput but maintain stable response times in the main flow.

The items microservice migration during Phase 2 resulted in a significant decline of throughput performance, where Thread Group reached 4.95 requests per second (-32.9%) and BZM

throughput dropped to -64.8% and JPGC throughput reached -62.3%. Thread Group response times reached an average of 6.8 seconds while showing increased performance instability. Error rates improved slightly for Thread Group, down to 0.04%, but plugin error rates remained high at close to 0.9%. The use of sagas and domain events, and messaging systems led to lower throughput and slower response times compared to the monolithic system in every workload tested. The delays introduced by distributed architectures, the eventual-consistency behaviour of operations, and the added cost of data synchronization were reflected in the results. The system performance showed some improvement because of asynchronous calls and reduced data transfer, but CPU overhead from compression and saga orchestration operations stayed as the main performance bottleneck.

The throughput experienced a decline in Phase 3 after order migration because Thread Group processed 3.87 requests per second, which represented a -47.5% decrease, while BZM reached -79.6% and JPGC reached -78.6%. Average response times continued to increase as Thread Group achieved a new high of 9.1 seconds. Error rates ticked up to around 0.26% for Thread Group, with plugins still close to 0.9%. The system performance declined because the system processed multiple sagas and heavy broker operations, which resulted in extended response times.

The production rate reached a stable point at minimal levels during Phase 4. Thread Group was 3.79 requests per second (-48.7%), BZM -81.0%, JPGC -80.8%. The average response time rose slightly because Thread Group took 9.3 seconds to complete its operations. Error rates stayed at 0.23% for Thread Group. The system reached its performance limit at this point because broker and database operations became the main bottlenecks, which made new microservices only slightly more inefficient. The efficiency of autoscaling and application scaling down during idle periods proved to be more effective than their contribution to throughput recovery.

The CQRS system with read and write microservices became operational during Phase 5. The throughput measurements showed different results during this phase because Thread Group reached 3.37 requests per second, while BZM throughput reached -87.6% and JPGC throughput improved from -80.8% in Phase 4 to -70.3%. Average response times increased again for Thread Group to just over 10 seconds. Error rates for Thread Group rose slightly to 0.31%. The system demonstrates CQRS functionality through these modifications because read-intensive operations gained speed from basic queries yet write-intensive operations became slower because of increased write operations. The green patterns decreased data transmission and compression rates, which led to better read performance but the projection workloads against writes restricted write throughput.

5.4 Maintainability Measurement

The document section 4.3.7.3 describes measurement methods for Maintainability, but the Sonargraph Maintainability level requires local setup to read the maintainability during each migration phase.

For each migration phase, a new Sonargraph system is created using the default quality model to establish a clean analysis baseline. Each service is then organised into its own Java Module, ensuring clear separation and visibility of dependencies. Finally, the maintainability level metric is retrieved to measure the system's overall health for the current phase.

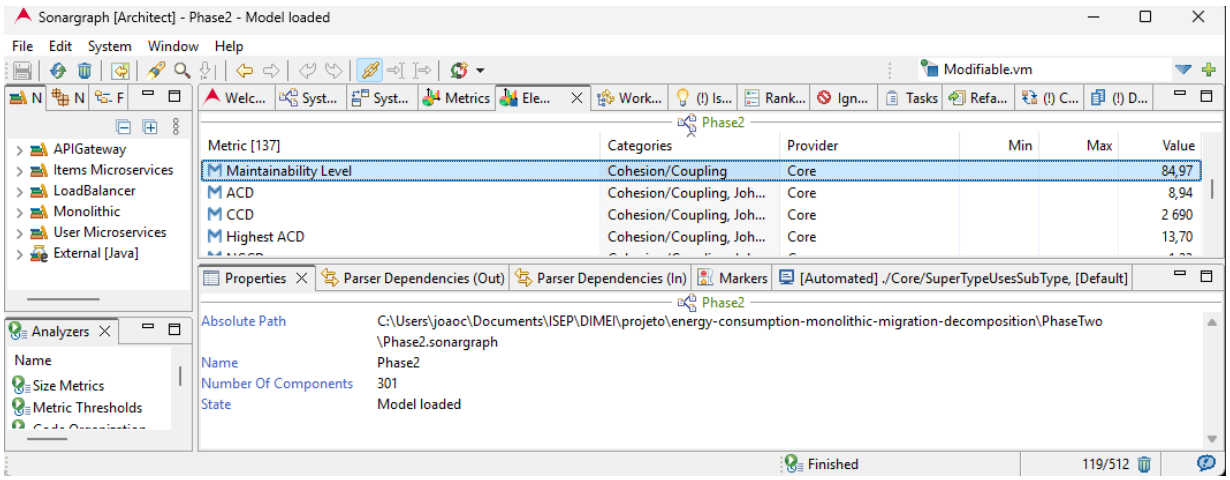

Figure 24 – Sonargraph Measurements setup

After setting up the Sonargraph, the results obtained for each phase can be seen in the [Project Repository - Sonargraph Results]. [68]

The final results are illustrated in Figure 25, Figure 26 and Figure 27, , which are based on the data provided in Table 39 of Appendix G.

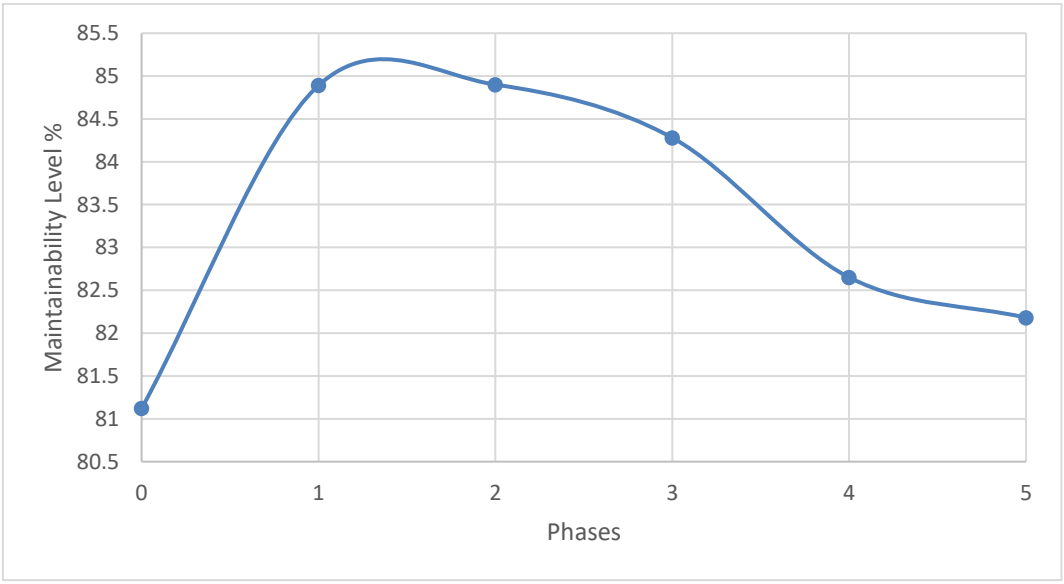

Figure 25 – Maintainability Level Results

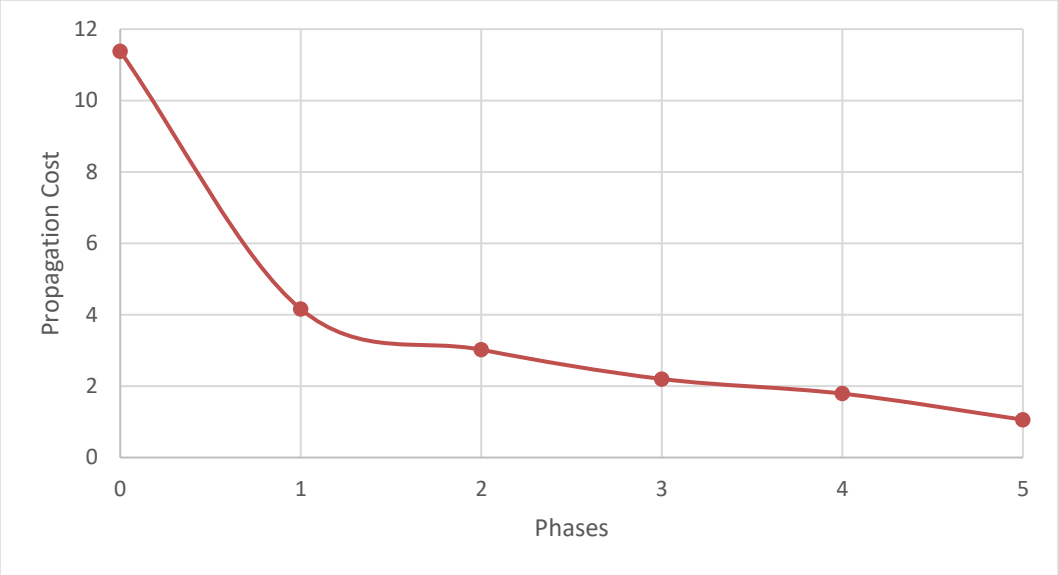

Figure 26 – Propagation Cost Results

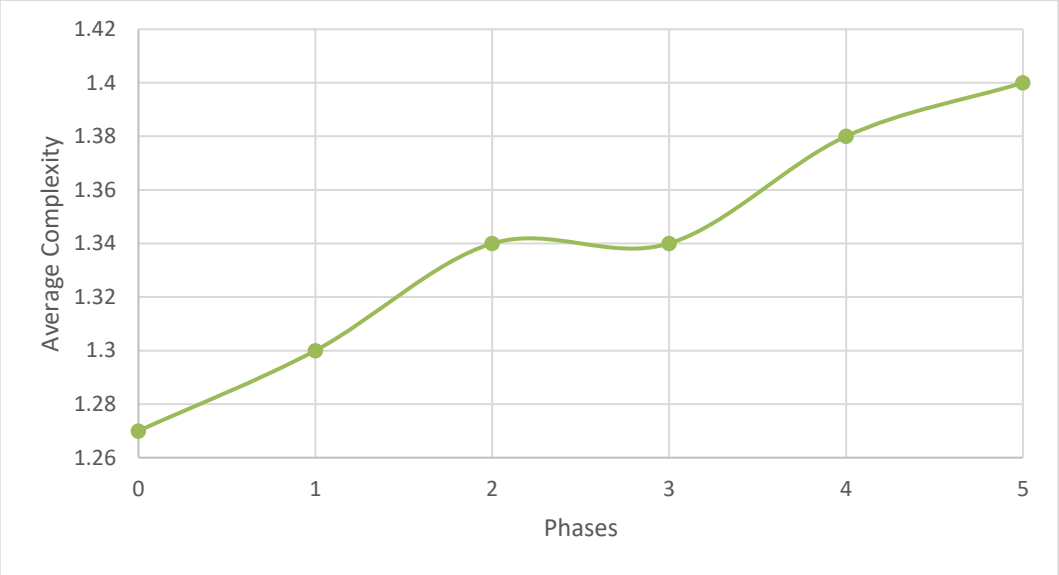

Figure 27 – Average Complexity Results

At Phase 0, the system was still a monolith. The system achieved a maintainability score of 81.1 while propagation cost reached 11.38 and average complexity measured 1.27. In short, it was tightly coupled but relatively simple.

In Phase 1, the user service was split out and supported by a load balancer, API gateway, service registry, self-registration, and its own database. The changes delivered two main advantages because maintainability increased to 84.9% while propagation cost decreased to 4.16, which represented a 63% improvement. The system complexity rose to 1.30 because of the additional infrastructure requirements.

The items service migration occurred during Phase 2 while the team implemented database-per-service patterns and sagas and domain events, and messaging patterns. These pushed propagation cost down further to 3.02 (-27% from the previous phase), but complexity also rose to 1.34. The maintainability score stayed at 84.9, which showed that the benefits of decoupling were balanced by the extra complexity that came from coordination logic.

In Phase 3, the orders service was added with the same approach. Propagation cost dropped again to 2.20 (-27%), but maintainability slipped slightly to 84.3. The system maintained stability after introducing the new service because the average complexity level remained at 1.34.

The merchant orders and shipping orders services became available during Phase 4. Propagation cost improved to 1.79 (-19%), but maintainability fell to 82.6 (-2%). The average complexity level reached 1.38, which showed that the system required more complex orchestration and coordination.

The CQRS implementation during Phase 5 required developers to establish separate read-only services that maintained their own database systems. This delivered the lowest propagation cost of the entire migration at 1.06 (-41% from Phase 4 and a 91% reduction compared with the monolith). However, average complexity rose again to 1.40, and maintainability slipped a little further to 82.2. The CQRS implementation resulted in excellent decoupling, but it added extra system elements, which made the system more operationally complex.

5.5 Summary

The evaluation of migration focused on three essential aspects, which included energy consumption, performance, and maintainability.

The results from Table 35 show that system complexity creates a direct relationship with resource usage. The monolithic baseline (Phase 0) proved to be the most energy-efficient, but energy consumption increased dramatically when services became distributed and the system adopted sagas and messaging, and database-per-service patterns (Phases 2–4). The company verified that increased usage came from actual workload needs instead of waste through the deployment of green practices, which included data compression and asynchronous communication, and Kubernetes autoscaling. The last stage (Phase 5) with CQRS integration resulted in two outcomes because it made read operations faster, but write operations needed more resources.

The performance analysis shows the various trade-offs that emerge when using a distributed system architecture, in Table 36, Table 37 and Table 38. The system performance deteriorated with each additional service because service coordination expenses and message processing time resulted in longer response times and more errors. The migration of the Order and Shipping contexts proved particularly demanding in this respect. The CQRS implementation during Phase 5 brought better performance for read-intensive operations, yet it created more work for write operations, which demonstrated both advantages and disadvantages of this method. The research indicates that organizations must monitor performance while making ongoing system improvements to achieve successful microservices operation at scale.

The results for maintainability (Table 39) showed positive results in all cases. The propagation cost decreased significantly by more than 90% when compared to the monolithic baseline, which proved that modular design with bounded contexts delivers substantial advantages. The maintainability score reached its highest point at 84.9% during the initial development phase, but it declined to 82.2% in subsequent stages although the benefits of service decoupling and improved independence compensated for these minor reductions. The research shows that microservices support ongoing system upkeep yet they introduce new operational difficulties.

The migration process proved that moving to microservices became possible through a systematic approach which followed Domain-Driven Design principles and sustainable software development practices. Although performance challenges remain due to the distributed nature of the architecture, the gains in modularity, maintainability, and scalability provide a solid foundation for future development. The research indicates that microservices become more resilient when paired with green software patterns, which also support the development of sustainable and adaptable software systems.

6 Conclusion and Future Work

This chapter details all of the findings made during this dissertation. Firstly, it details what has been achieved. Next, it describes the found limitations and the threats to validity. Finally, the chapter delineates the possible future.

6.1 Achievements

The study provides multiple essential contributions to monolithic application migration into microservices through performance, energy consumption and system maintainability. A brand-new retail platform received complete control over migration analysis through its development from scratch.

The Domain-Driven Design-based structured migration plan enabled the transition from a monolithic architecture to a microservices architecture. The implementation of Saga and Event-Driven Messaging and Database-per-Service and Command Query Responsibility Segregation (CQRS) patterns achieved functional preservation while delivering improved system modularity and scalability. The architecture received sustainability integration through the implementation of green software practices, which included containerization and asynchronous communication and workload autoscaling, and data compression.

The experimental assessment showed that the system exhibited multiple performance trade-offs. The monolithic system maintained its position as the most energy-efficient configuration according to Kepler measurements yet the CQRS phase demonstrated potential energy savings during read-heavy operations. Apache JMeter performance tests demonstrated the expected communication delays between distributed systems yet showed improved read operations under CQRS. The Sonargraph maintainability analysis demonstrated substantial architectural coupling reduction because the system experienced more than 90% lower propagation cost than the original system.

The study achieved three primary results through its work:

I. A functional approach for transforming monolithic systems into microservices
II. The implementation of sustainable design principles during system decomposition
III. The study provides quantitative data about how energy efficiency relates to performance and maintainability during system transformation. The study provides essential knowledge for both software developers and researchers who want to create sustainable software systems with better maintainability.

6.2 Threats to Validity

This work is based on the measurement of the CPU energy consumption when running an application. Despite being very careful, some aspects can be considered threats to the validity.

As explained in section 4.1, the study was conducted on a desktop with the following specifications: Linux Debian 12.5.0 OS, kernel version 6.1.0-37, with 32GB of RAM, and an Intel Core I7-14700F processor. This specification can interfere with energy consumption, leading to different results with different hardware setups.

About selecting a tool to measure energy consumption, Kepler, it is important to note that a new release became available during the study. As with any recent release, there may be some unexpected bugs or instabilities that could affect its performance. Despite this, the most recent version of the software was used in the study because it enabled evaluation that reflected the most up-to-date features and capabilities. This choice prioritized relevance and alignment with ongoing development; however, potential limitations due to unresolved issues in the new release should be acknowledged when interpreting the results.

The CPU used in this study could only measure energy consumption at the package and core levels. Measurements for DRAM and uncore components were not available. As a result, the reported energy data does not capture the full system consumption. This limitation should be considered when analysing the results.

In addition, when measurements were being taken, care was taken to ensure that the computer was always in the same condition.

Finally, the results presented in this study are dependent on the application used. It is not possible to guarantee that the results will be the same when using another application.

6.3 Future Work

The study has taken initial steps to investigate how migration and decomposition patterns affect system energy consumption, performance, and maintainability; some opportunities for future exploration remain.

Future research should investigate heterogeneous technology stacks that combine polyglot microservices that use different programming languages and frameworks and databases to discover new efficiency versus maintainability trade-offs. The evaluation of hybrid systems that combine modular monolithic with microservices architecture needs to be done to achieve simplicity while maintaining long-term operational sustainability.

The deployment environment serves as the main element for this evaluation. The study of multi-cloud platforms that integrate AWS and Azure, and GCP, and various orchestration approaches, will demonstrate how each provider handles system performance and energy consumption. The evaluation of communication mechanisms through synchronous and asynchronous methods, and API technologies (e.g., REST, gRPC, GraphQL) and middleware tools (e.g., RabbitMQ, Apache Kafka, Apache Camel) would provide complete details about distributed system performance and energy consumption.

The evaluation process for quality attributes requires additional attributes that go beyond the current selection to achieve more comprehensive results. The evaluation of monolithic-to-microservices migration requires assessment of scalability, flexibility, and security metrics.

In conclusion, this study adds to sustainable software design knowledge by demonstrating how architectural changes impact energy consumption alongside performance and maintainability. The study reveals multiple new directions for comprehensive investigation and enhancement, along with practical implementation in various real-world situations.

References

[1] D. Erdenesanaa, “A.I. Could Soon Need as Much Electricity as an Entire Country,” *The New York Times,* 2023.

[2] Green Software Foundation, “Manifesto,” [Online]. Available: https://greensoftware.foundation/manifesto. [Accessed 2023].

[3] F. Khomh and S. A. Abtahizadeh, “Understanding the impact of cloud patterns on performance and energy consumption.,” *Journal of Systems and Software,* vol. 141, pp. 151-170, 07 2018.

[4] S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, 1st ed., Beijing: O'Reilly Media, Inc., 2019.

[5] C. T. Joseph and K. Chandrasekaran, “Straddling the crevasse: A review of microservice software architecture foundations and recent advancements,” *Software: Practice and Experience,* vol. 49, no. 10, pp. 1448-1484, October 2019.

[6] V. Berry, A. Castelltort, B. Lange, J. Teriihoania, C. Tibermacine and C. Trubiani, “Is it Worth Migrating a Monolith to Microservices? An Experience Report on Performance, Availability and Energy Usage,” in *2024 IEEE International Conference on Web Services (ICWS)*, Shenzhen, China, 2024.

[7] B. J. Oates, Researching Information Systems and Computing, London: SAGE Publications, 2006.

[8] “Despacho n.º 11171/2020 | DR,” Diário da República, 12 November 2020. [Online]. Available: https://diariodarepublica.pt/dr/detalhe/despacho/11171-2020-148327696. [Accessed 1 May 2025].

[9] S. Newman, Building Microservices: Designing Fine-Grained Systems, Beijing: O'Reilly Media, Inc., 2021.

[10] C. Richardson, Microservices Patterns: With examples in Java, Shelter Island, New York: Manning Publications, 2019.

[11] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*, Cham, Springer International Publishing, 2017, pp. 195-216.

[12] N. M. Josuttis, SOA in Practice: The Art of Distributed System Design, 1st ed., Beijing: O'Reilly Media, Inc., 2008.

[13] M. Richards, Microservices vs. Service-Oriented Architecture, Sebastopol, Calif.: O'Reilly Media, Inc., 2015.

[14] S. Baškarada, V. Nguyen and A. Koronios, “Architecting Microservices: Practical Opportunities and Challenges,” *Journal of Computer Information Systems,* vol. 60, no. 5, pp. 428-436, 2020.

[15] M. Ekuan, “CI/CD for microservices - Azure Architecture Center,” [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd. [Accessed 10 December 2023].

[16] ISO/IEC, “ISO 25000 STANDARDS,” 2023. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards.

[17] ISO/IEC, “ISO 25010,” 2023. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[18] L. Bass, P. Clements and R. Kazman, Software Architecture In Practice, Fourth ed., Boston: Addison-Wesley, 2021.

[19] K. Petersen, R. Feldt, S. Mujtaba and M. Mattsson, “Systematic Mapping Studies in Software Engineering,” Swindon, GBR, 2008.

[20] M. Muthu, K. Banuroopa and S. Arunadevi, “Green and Sustainability in Software Development Lifecycle Process,” in *Sustainability Assessment at the 21st century*, IntechOpen, 2020.

[21] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva, “Energy efficiency across programming languages: how do energy, time, and memory relate?,” in *SPLASH '17: Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Vancouver, 2017.

[22] VEDANTU, “Energy Consumption Formula - Meaning, Calculation and FAQs,” [Online]. Available: https://www.vedantu.com/formula/energy-consumption-formula. [Accessed 28 June 2024].

[23] A. Noureddine, “PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools,” in *2022 18th International Conference on Intelligent Environments (IE)*, Biarritz, 2022.

[24] Monsoon Solutions Inc., “HIGH VOLTAGE POWER MONITOR,” [Online]. Available: https://www.msoon.com/high-voltage-power-monitor. [Accessed 05 April 2025].

[25] T. A. Ghaleb, “Software Energy Measurement at Different Levels of Granularity,” in *2019 International Conference on Computer and Information Sciences (ICCIS)*, Sakaka, 2019.

[26] H. Acar, G. I. Alptekin, J.-P. Gelas and P. Ghodous, “The Impact of Source Code in Software on Power Consumption,” *International Journal of Electronic Business Management,* vol. 14, pp. 42-52, 2016.

[27] J. M. Hirst, J. R. Miller, B. A. Kaplan and D. D. Reed, “Watts Up? Pro AC Power Meter for Automated Energy Recording: A Product Review,” *Behavior Analysis in Practice,* vol. 6, no. 1, pp. 82-95, 2013.

[28] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen and Z. Ou, “RAPL in Action: Experiences in Using RAPL for Power Measurements,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems,* vol. 3, no. 2, pp. 1-26, 30 June 2018.

[29] Green Software Foundation, “Green-Software-Foundation/awesome-green-software,” 2021. [Online]. Available: https://github.com/Green-Software-Foundation/awesome-green-software. [Accessed 13 12 2023].

[30] A. Noureddine, R. Rouvoy and L. Seinturier, “A review of energy measurement approaches,” *ACM SIGOPS Operating Systems Review,* vol. 47, no. 3, pp. 42-49, 26 November 2013.

[31] Green Software Practitioner, “Compress stored data,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/compress-stored-data/. [Accessed 2024].

[32] Green Software Practitioner, “Compress transmitted data,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/compress-transmitted-data. [Accessed 2024].

[33] Green Software Practitioner, “Containerize your workloads,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/containerize-your-workload-where-applicable. [Accessed 2024].

[34] Green Software Practitioner, “Optimize average CPU utilization,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/optimize-avg-cpu-utilization. [Accessed 2024].

[35] Green Software Practitioner, “Optimize peak CPU utilization,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/optimize-peak-cpu-utilization/. [Accessed 2024].

[36] Green Software Practitioner, “Optimise storage utilization,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/optimise-storage-resource-utilisation/. [Accessed 2024].

[37] Green Software Practitioner, “Reduce transmitted data,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/reduce-transmitted-data/. [Accessed 2024].

[38] Green Software Practitioner, “Scale down applications when not in use,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/scale-down-unused-applications/. [Accessed 2024].

[39] Green Software Practitioner, “Scale down Kubernetes applications when not in use,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/scale-down-kubernetes-workloads/. [Accessed 2024].

[40] Green Software Practitioner, “Scale Kubernetes workloads based on relevant demand metrics,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/scale-kubernetes-workloads-based-on-events/. [Accessed 2024].

[41] Green Software Practitioner, “Time-shift Kubernetes cron jobs,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/time-shift-kubernetes-cron-jobs. [Accessed 2024].

[42] Green Software Practitioner, “Use Asynchronous network calls instead of synchronous,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/use-async-instead-of-sync/. [Accessed 2024].

[43] Green Software Practitioner, “Optimize impact on customer devices and equipment,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/optimize-impact-on-customer-equipment. [Accessed 2024].

[44] Green Software Practitioner, “Scan for vulnerabilities,” [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/scan-for-vulnerabilities. [Accessed 2024].

[45] Microservices.io, “Microservices Pattern: Pattern: Command Query Responsibility Segregation (CQRS),” [Online]. Available: http://microservices.io/patterns/data/cqrs.html. [Accessed 14 July 2024].

[46] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera and R. Sinnott, “A Performance Comparison of Cloud-Based Container Orchestration Tools,” in *2019 IEEE International Conference on Big Knowledge (ICBK)*, Beijing, 2019.

[47] H. Binani, “Kubernetes vs Docker Swarm — A Comprehensive Comparison | HackerNoon,” 4 October 2018. [Online]. Available: https://hackernoon.com/kubernetes-vs-docker-swarm-a-comprehensive-comparison-73058543771e. [Accessed 21 May 2024].

[48] Microservices.io, “Microservices Pattern: Pattern: Messaging,” [Online]. Available: http://microservices.io/patterns/communication-style/messaging.html. [Accessed 29 May 2024].

[49] Apache Software Foundation, “Apache Kafka,” [Online]. Available: https://kafka.apache.org/. [Accessed 29 May 2024].

[50] Broadcom, “RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed 29 May 2024].

[51] P. Dobbelaere and K. S. Esmaili, “Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper,” in *DEBS '17: The 11th ACM International Conference on Distributed and Event-based Systems*, Barcelona, 2017.

[52] G. Araújo, V. Barbosa, L. N. Lima, A. Sabino, C. Brito, I. Fé, P. Rego, E. Choi, D. Min, T. A. Nguyen and F. A. Silva, “Energy Consumption in Microservices Architectures: A Systematic Literature Review,” *IEEE Access,* pp. 1-1, 2024.

[53] D. Faustino, N. Gonçalves, N. Portela and A. R. Silva, “Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation,” *Performance Evaluation,* vol. 164, p. 102411, 2024.

[54] “Sonargraph Product Overview,” hello2morrow GmbH, 2025. [Online]. Available: https://www.hello2morrow.com/products/sonargraph. [Accessed 25 June 2025].

[55] “Apache JMeter - Apache JMeter™,” [Online]. Available: https://jmeter.apache.org/. [Accessed 29 May 2024].

[56] T. L. Saaty, “How to make a decision: The analytic hierarchy process,” *European Journal of Operational Research,* vol. 48, no. 1, pp. 9-26, September 1990.

[57] M. Amaral, H. Chen, T. Chiba, R. Nakazawa, S. Choochotkaew, E. K. Lee and T. Eilam, “Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications,” in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, Chicago, 2023.

[58] J. Campelo, “Release DIMEI-September-2025 · joaocampelo3/energy-consumption-monolithic-migration-decomposition,” [Online]. Available:

https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/releases/tag/DIMEI-September-2025. [Accessed 4 September 2025].

[59] E. Evans, Domain-driven design: tackling complexity in the heart of software, Boston: Addison-Wesley, 2004.

[60] R. C. Martin, “Design principles and design patterns,” vol. 1, p. 597, 2000.

[61] Microservices.io, “Microservices Pattern: Pattern: Event Sourcing,” [Online]. Available: https://microservices.io/patterns/data/event-sourcing.html. [Accessed 14 July 2024].

[62] Microservices.io, “Microservices Pattern: Pattern: Saga,” [Online]. Available: https://microservices.io/patterns/data/saga.html. [Accessed 14 July 2024].

[63] V. R. Basili and D. M. Weiss, “A Methodology for Collecting Valid Software Engineering Data,” *IEEE Transactions on Software Engineering,* Vols. SE-10, no. 6, pp. 728-738, 1984.

[64] V. R. Basili, G. Caldiera and H. D. Rombach, “The goal question metric approach,” *Encyclopedia of software engineering,* p. 528–532, 1994.

[65] “Installation Guide - Kepler,” [Online]. Available: https://sustainable-computing.io/kepler/installation/guide/. [Accessed 25 August 2025].

[66] J. Campelo, “energy-consumption-monolithic-migration-decomposition/Results/Kepler at DIMEI-September-2025 · joaocampelo3/energy-consumption-monolithic-migration-decomposition,” [Online]. Available: https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/tree/DIMEI-September-2025/Results/Kepler. [Accessed 4 September 2025].

[67] J. Campelo, “energy-consumption-monolithic-migration-decomposition/Results/JMeter at DIMEI-September-2025 · joaocampelo3/energy-consumption-monolithic-migration-decomposition,” [Online]. Available: https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/tree/DIMEI-September-2025/Results/JMeter. [Accessed 4 September 2025].

[68] J. Campelo, “energy-consumption-monolithic-migration-decomposition/Results/Sonargraph at DIMEI-September-2025 · joaocampelo3/energy-consumption-monolithic-migration-decomposition,” [Online]. Available: https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/tree/DIMEI-September-2025/Results/Sonargraph. [Accessed 4 September 2025].

[69] J. Campelo, “energy-consumption-monolithic-migration-decomposition/Grafana Dashboard at DIMEI-September-2025 · joaocampelo3/energy-consumption-monolithic-

migration-decomposition," [Online]. Available: https://github.com/joaocampelo3/energy-consumption-monolithic-migration-decomposition/tree/DIMEI-September-2025/Grafana%20Dashboard. [Accessed 4 September 2025].

Appendix A: Project plan

In order to manage the project effectively, it is crucial to clearly understand its different phases, their sequence, and the estimated effort.

As mentioned, this work will compare and analyse different migration and decomposition patterns to microservices, considering energy consumption and various quality attributes to determine greener alternatives. So, a Gantt chart was created outlining the project's expected phases and estimated timeframes.

This project plan is divided into eleven parts, and these are referred to as activities(A):

A1. **Literature Research and Review:**
This step is intended to develop an idea and define the research methodology. This initial phase's primary objective is to comprehend software development's influence on energy consumption.

A2. **Evaluate Projects:**
At this stage, the purpose is to evaluate different open-source projects and choose one monolith project for this work in order to respect ethical standards.

A3. **Evaluate tools:**
In parallel with A2, all the state-of-the-art tools described for measuring energy consumption will be analysed, and then one of the tools and the measurement method will be chosen. In addition, tools will be chosen to collect metrics for the quality attributes chosen later.

A4. **Design methodology for migration and decomposition:**
After choosing an open-source monolithic project, a wide range of possibilities will be considered, including database and containerisation technologies, migration and decomposition approaches, and communication between services. As well as the design of a roadmap in which these patterns will be applied.

A5. **Implement solution:**
This task will be a complementary task to activity A2 and seeks to enrich the initial application if needed to be ready for the experimentation phase. This enrichment will mainly be the addition of unit tests so that the application's functionality remains integral throughout the experimentation process.

A6. **Develop evaluation criteria and metrics:**
The GQM method [64] presented will define energy and various quality attribute metrics for evaluating migration and decomposition patterns concerning energy consumption changes during software development.

A7. **Conduct initial testing and validation:**

As a starting point for the experimentation phase, a set of tests and validations will be carried out on the monolithic application in order to gather information on the application's initial state using the metrics defined in the GQM.

A8. **Application of migration and decomposition methodology defined:**

This phase of experimentation consists of applying the methodology designed in the A4.

A9. **Conduct testing and validation:**

This activity will be similar to A7, aiming to collect GQM metrics for each migration and decomposition pattern carried out by activity A8.

A10. **Analyse the results and conclude:**

After the experiment phase, the results will be analysed to answer the questions using the GQM approach with all data collected. It will identify limitations and analyse workarounds/mitigation steps to overcome them.

It will compare the benefits and drawbacks of migrating monolithic architectures to microservices.

A11. **Thesis Writing**

During the whole project, every phase will be described in the thesis.

Given that some of the activities have already been carried out in the previous year, this plan will take into account the activities A1, A8, A9, A10 and A11, as can be seen in Figure 28.

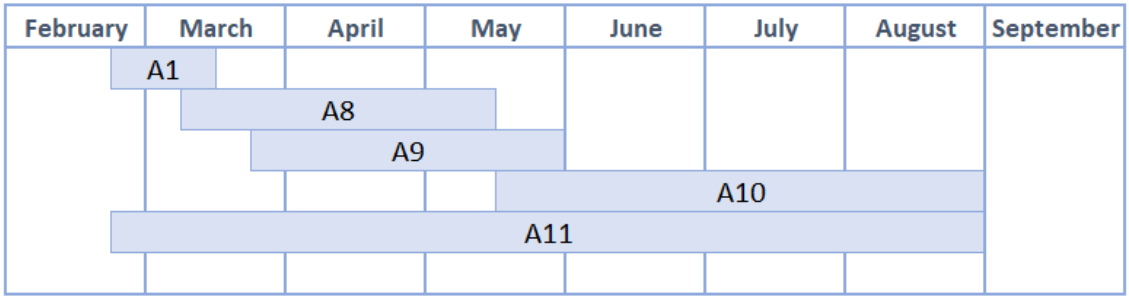

Figure 28 – Project plan

Appendix B: AHP method to choose a tool

After the developing the decision hierarchic tree which appears in Figure 7, the Saaty scale [56] was then applied to compare the importance of each criterion. The pairwise comparison matrix for the criteria is shown in Table 21.

Table 21 – Criteria Comparison Matrix – Tool

Criteria	**Simplicity**	**Reliability**	**Functionalities**	**Cost**
Simplicity	1	1/5	1/3	1/7
Reliability	5	1	2	1/3
Functionalities	3	1/2	1	1/7
Cost	7	3	7	1
Sum	16	33/7	31/3	13/8

The following step requires the criteria comparison matrix to be normalised. The matrix values need to be divided by their respective column totals for the normalisation process. After normalising this matrix, the priority vector becomes available through row mean calculations. The resulting vector shows the relative weight of each criterion. The relative priorities were determined in Table 22.

Table 22 – Normalised Criteria Matrix – Tool

Criteria	**Simplicity**	**Reliability**	**Functionalities**	**Cost**	**Relative Priority**
Simplicity	1	1/5	1/3	1/7	0,06
Reliability	5	1	2	1/3	0,23
Functionalities	3	1/2	1	1/7	0,12
Cost	7	3	7	1	0,59

In order to evaluate the result, consistency depends on calculating both the Consistency Index (CI) and the Consistency Ratio (RC). The consistency index (CI) calculation uses the following formula:

$$CI = \frac{(\lambda_{max} - n)}{(n-1)}$$

The number of criteria is represented by n. In this case, $n = 4$. The maximum eigenvalue (λ_{max}) was calculated by multiplying the normalised comparison matrix with the priority vector and

then averaging the resulting values relative to the original priorities. The approximate value of $\lambda_{max} \approx 4{,}119$ led to a Consistency Index of $CI = 0{,}039$.

The Consistency Ratio (RC) was determined through the following calculation:

$$RC = \frac{CI}{IR}$$

Where the index of a random matrix (IR) was calculated for square matrices of order n by the Oak Ridge National Laboratory in the USA. The following Table 23 defines the IR values as a function of the number of criteria:

Table 23 – IR values for square matrices of order n

n	1	2	3	4	5	6	7	8	9	10
IR	0	0	0,58	0,90	1,12	1,24	1,32	1,41	1,45	1,49

Based on this, the consistency ratio calculation results in: $RC = \frac{CI}{IR} = \frac{0.040}{0.90} = 0.044$. It can be concluded that the relative priorities are consistent due to the $CR < 0.1$ value.

Next up, for each criterion defined, a matrix of comparison was created, considering each of the alternatives identified. Additionally, the matrix and the respective priority for each alternative under each criterion were computed as follows:

Table 24 – Comparison matrix and priorities vector – Simplicity

Simplicity	**Alternative 1**	**Alternative 2**	**Alternative 3**	**Alternative 4**	**Alternative 5**	**Alternative 6**	**Relative Priority**
Alternative 1	1	1/3	1/3	1/4	3	2	0,1
Alternative 2	3	1	1	1	5	5	0,26
Alternative 3	3	1	1	1	5	5	0,26
Alternative 4	4	1	1	1	5	5	0,28
Alternative 5	1/3	1/5	1/5	1/5	1	2	0,06
Alternative 6	1/2	1/5	1/5	1/5	1/2	1	0,05

Table 25 – Comparison matrix and priorities vector – Reliability

Reliability	Alternative 1	Alternative 2	Alternative 3	Alternative 4	Alternative 5	Alternative 6	Relative Priority
Alternative 1	1	5	5	5	4	1/3	0,29
Alternative 2	1/5	1	1	1	1/3	1/5	0,06
Alternative 3	1/5	1	1	1	1/3	1/5	0,06
Alternative 4	1/5	1	1	1	1/3	1/5	0,06
Alternative 5	1/4	3	3	3	1	1/3	0,15
Alternative 6	3	5	5	5	3	1	0,39

Table 26 – Comparison matrix and priorities vector – Functionalities

Functionalities	Alternative 1	Alternative 2	Alternative 3	Alternative 4	Alternative 5	Alternative 6	Relative Priority
Alternative 1	1	1/6	1/6	1/6	1/7	1/3	0,03
Alternative 2	6	1	1	1	1/2	2	0,18
Alternative 3	6	1	1	1	1/2	2	0,18
Alternative 4	6	1	1	1	1/2	2	0,18
Alternative 5	7	2	2	2	1	3	0,32
Alternative 6	3	1/2	1/2	1/2	1/3	1	0,10

Table 27 – Comparison matrix and priorities vector – Cost

Cost	Alternative 1	Alternative 2	Alternative 3	Alternative 4	Alternative 5	Alternative 6	Relative Priority
Alternative 1	1	1/7	1/7	1/7	1/5	1	0,03
Alternative 2	7	1	1	1	3	7	0,27
Alternative 3	7	1	1	1	3	7	0,27
Alternative 4	7	1	1	1	3	7	0,27
Alternative 5	5	1/3	1/3	1/3	1	5	0,12
Alternative 6	1	1/7	1/7	1/7	1/5	1	0,03

Appendix C: AHP method for the project selection

Thereafter creating a decision hierarchy tree presented in Figure 8, the decision hierarchy tree's result should be mapped with the criteria comparison matrix presented in Table 28:

Table 28 – Criteria Comparison Matrix – Project

Criteria	**Complexity and Scalability**	**Modularity and Architecture**	**Implementation**	**Documentation**
Complexity and Scalability	1	3	1/2	2
Modularity and Architecture	1/3	1	1/4	2
Implementation	2	4	1	2
Documentation	1/2	1/2	1/2	1
Sum	23/6	17/2	9/4	7

The next step is to normalise the criteria comparison matrix. Once the normalisation is complete, the priority vector can be obtained and indicates the relative importance per criterion (Table 29).

Table 29 – Normalised Criteria Matrix – Project

Criteria	**Complexity and Scalability**	**Modularity and Architecture**	**Implementation**	**Documentation**	**Relative Priority**
Complexity and Scalability	1	3	1/2	2	0,28
Modularity and Architecture	1/3	1	1/4	2	0,15
Implementation	2	4	1	2	0,43
Documentation	1/2	1/2	1/2	1	0,14

After it proceeds to determine the Consistency Index (IC) and the Consistency Ratio (RC) to evaluate the consistency of the results. The calculation of the consistency index (IC) uses the following formula:

$$IC = (\lambda_{max} - n)/(n - 1)$$

In this case, the n corresponds to the number of criteria considered ($n = 4$).

The calculation of λ_{max} involves multiplying the normalised matrix by the priorities and, after this, calculating the mean between the result obtained and the values from the priorities vector. This leads to a $\lambda_{max} \approx 4{,}249$ and a $IC = 0{,}083$.

The Consistency Ratio (RC) is calculated using the formula:

$$RC = \frac{IC}{IR}$$

Based on the Table 23, the IR values as a function of the number of criteria and the consistency ratio calculation results in: $RC = \frac{IC}{IR} = \frac{0.083}{0.90} = 0.092$. Considering that $CR < 0.1$, it can be concluded that the relative priorities are consistent.

Next up, for each criterion defined, a matrix of comparison was created, considering each of the alternatives identified. Additionally, the matrix and the respective priorities vector for each of the criteria are in Table 30, Table 31, Table 32 and Table 33.

Table 30 – Comparison matrix and priorities vector – Complexity and Scalability

Complexity and Scalability	**Alternative 1**	**Alternative 2**	**Relative Priority**
Alternative 1	1	1/4	0,2
Alternative 2	4	1	0,8

Table 31 – Comparison matrix and priorities vector – Modularity and Architecture

Modularity and Architecture	**Alternative 1**	**Alternative 2**	**Relative Priority**
Alternative 1	1	1/4	0,2
Alternative 2	4	1	0,8

Table 32 – Comparison matrix and priorities vector – Implementation

Implementation	**Alternative 1**	**Alternative 2**	**Relative Priority**
Alternative 1	1	2	0,67
Alternative 2	1/2	1	0,33

Table 33 – Comparison matrix and priorities vector – Documentation

Documentation	**Alternative 1**	**Alternative 2**	**Relative Priority**
Alternative 1	1	4	0,8
Alternative 2	1/4	1	0,2

Appendix D: Kepler Dashboard setup

The Kepler dashboard in Grafana enables visualization of energy consumption data from containerized workloads. By importing a preconfigured dashboard from the Kepler repository, users can quickly monitor and analyse energy usage. The following description outlines the process of setting up and using this dashboard.

To begin, open Grafana and navigate to the main menu, where the Dashboards section can be accessed as shown in Figure 29.

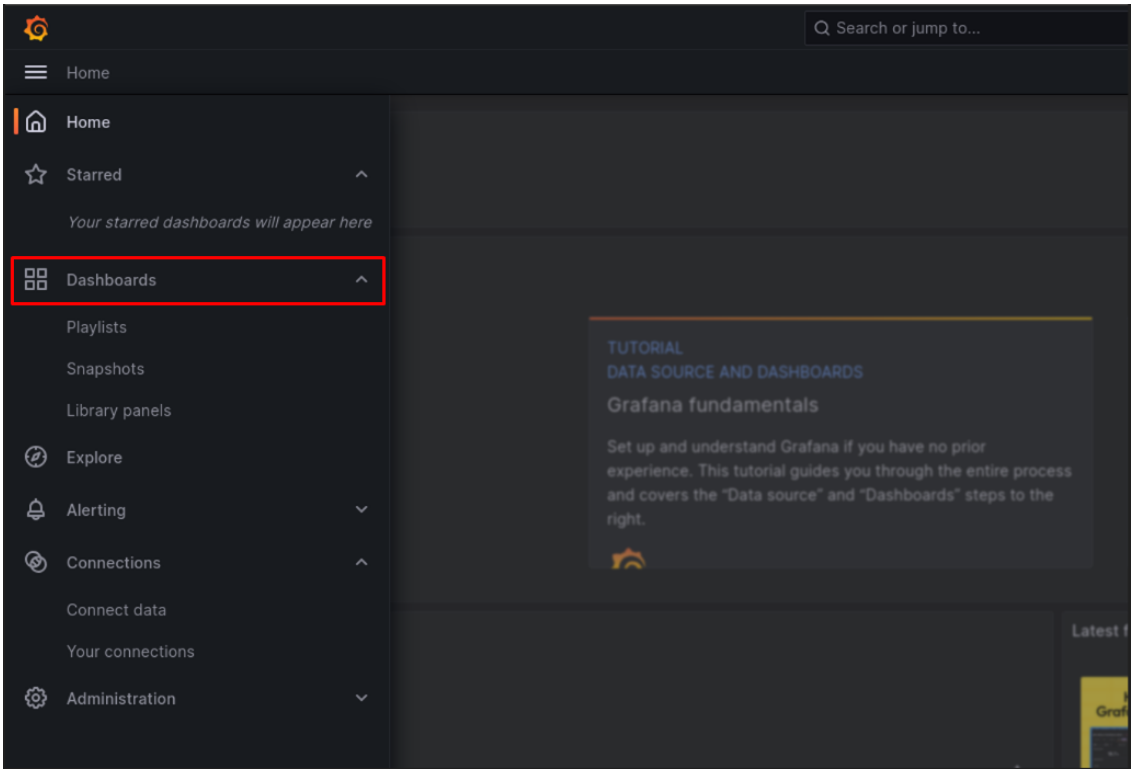

Figure 29 – Grafana's Dashboard menu

Within this section, a new dashboard can be created by selecting the Import option from the dashboard creation menu (Figure 30).

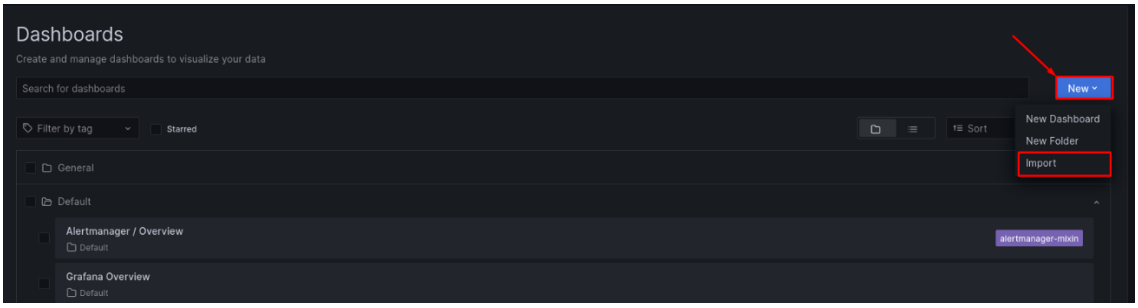

Figure 30 – Import Grafana dashboard

This allows the integration of a predefined configuration such as the one provided in the project repository [69]. After uploading the configuration file, a name is assigned, the Prometheus data source is chosen, and the dashboard is imported (Figure 31).

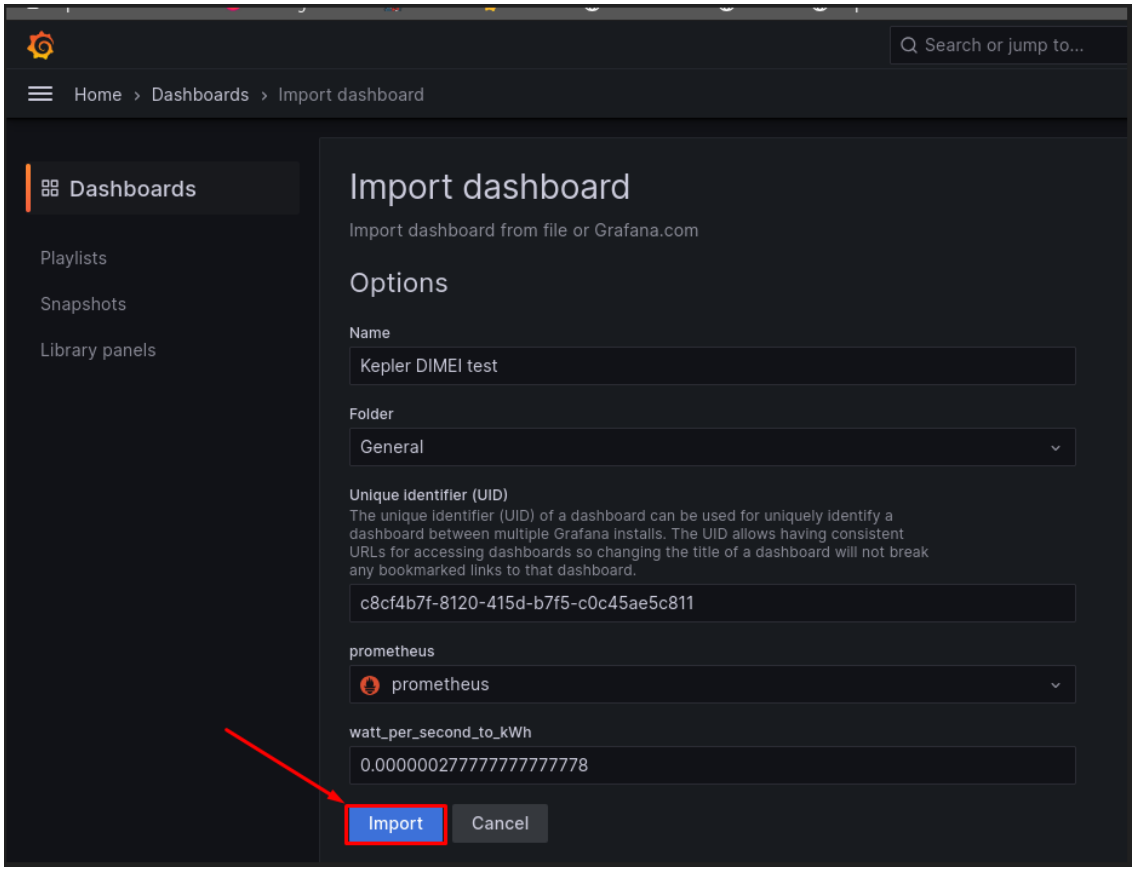

Figure 31 – Import Kepler dashboard

Once the import is complete, the Kepler dashboard becomes available and ready for use. It displays detailed metrics on power consumption and emissions, allowing tests to be executed and observed in real time (Figure 32).

Figure 32 – Import Kepler dashboard overview

To ensure results are relevant to a given experiment, the time range can be restricted to a specific interval, either using relative or absolute values (Figure 33).

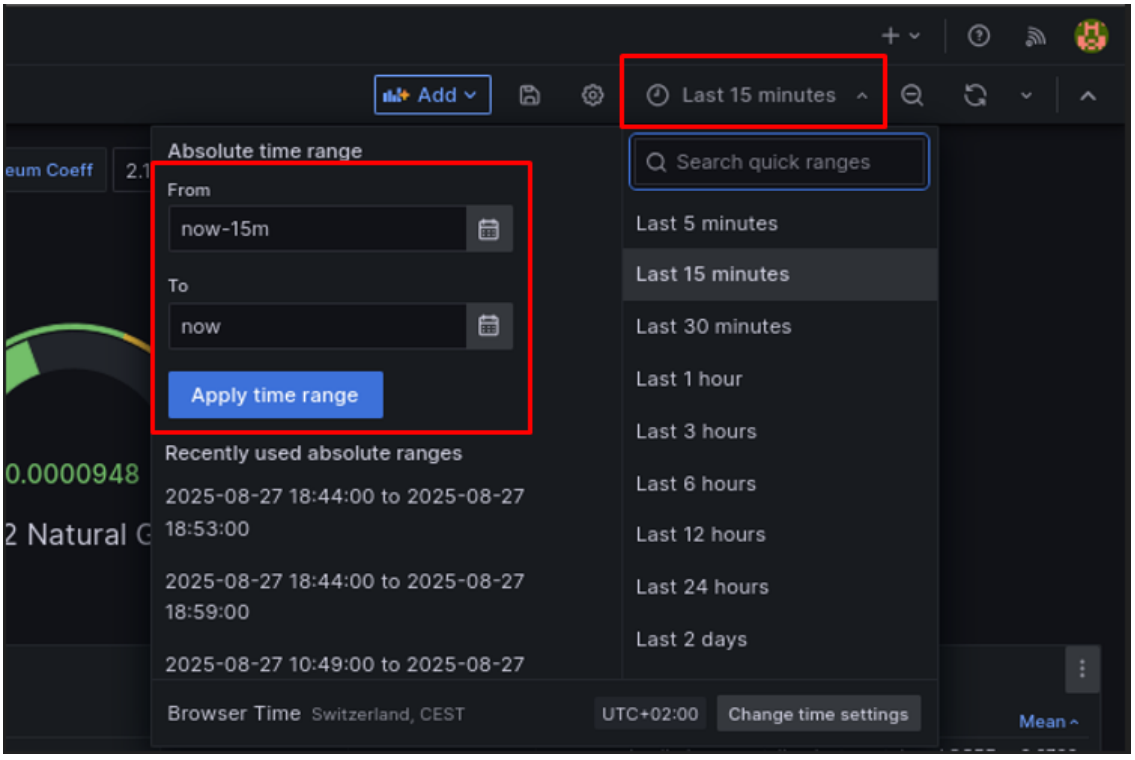

Figure 33 – Define Kepler time range

Grafana also makes it possible to filter the data by container name. This feature enables the isolation of results for specific workloads, providing more fine-grained insights into their energy consumption (Figure 34).

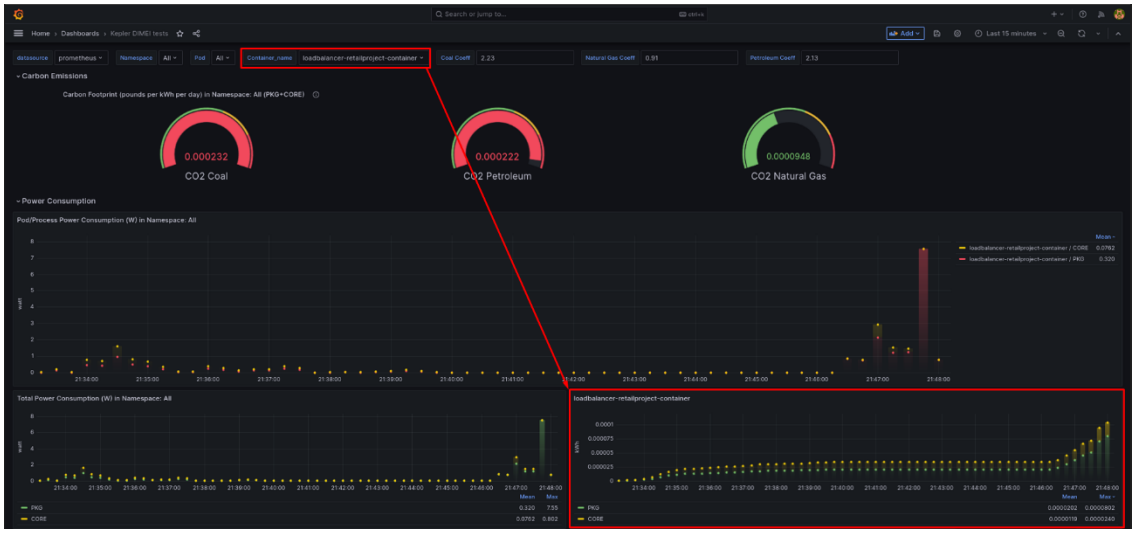

Figure 34 – Container’s energy consumption graph

For further analysis, results can be extracted by inspecting the relevant panel and downloading the data as a CSV file, which can then be processed or visualized with external tools as presented in the Figure 35 and Figure 36.

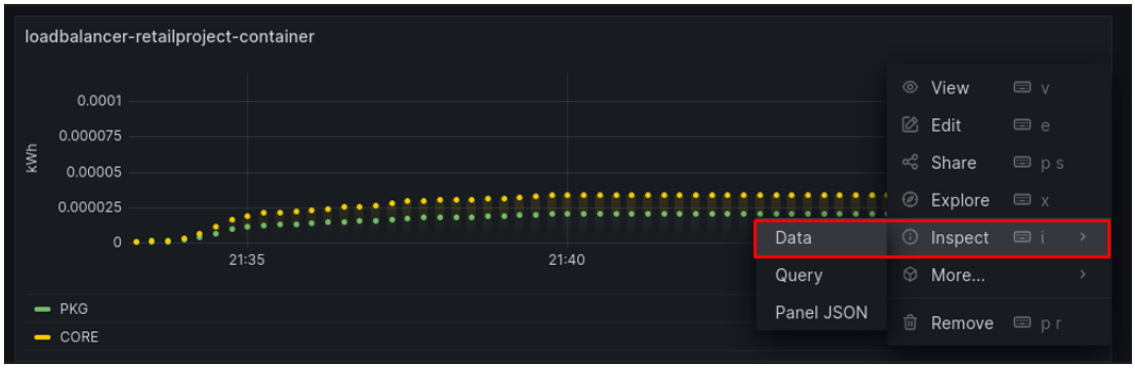

Figure 35 – Export the container's data

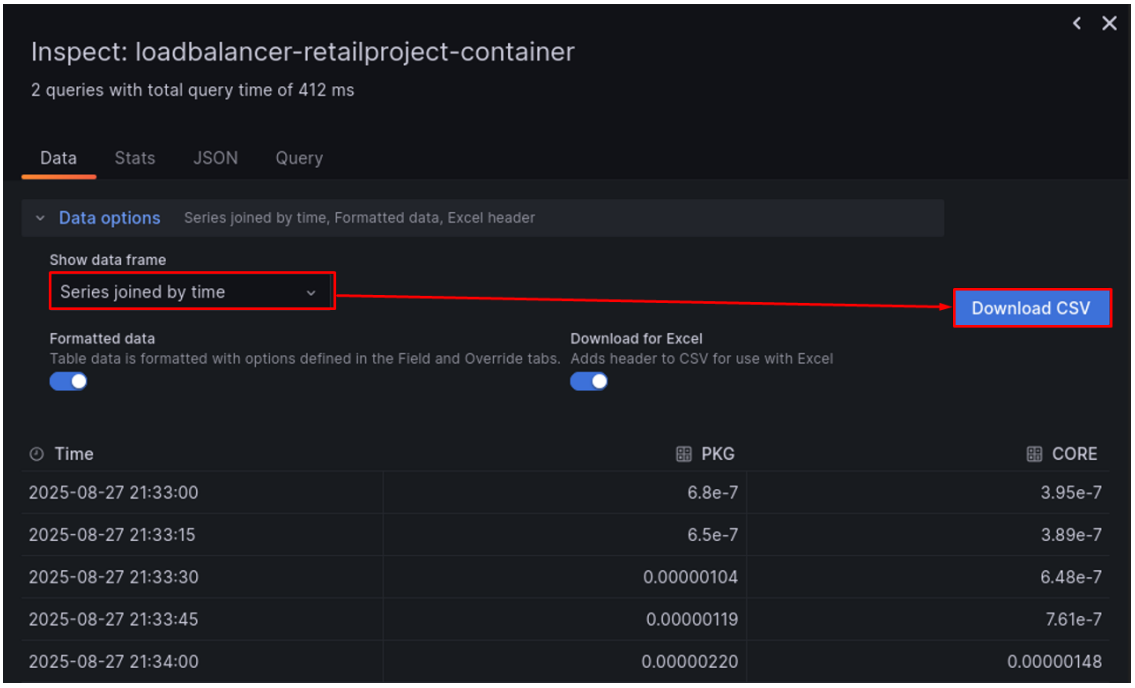

Figure 36 – Export the container's data in a csv

Appendix E: Energy consumption results

This appendix presents the energy consumption results recorded during the experimental phases. The values in Table 34 represent the data obtained from Grafana, which measures in kilowatt-hours (kWh). The results from Table 35 show the same data converted to Joules (J) for additional evaluation purposes.

Table 34 – Energy Consumption Results (kWh)

Phase	Thread Group	BZM	JP@GC
0	0.00025534	0.00071561	0.00187346
1	0.00027840	0.00070731	0.00379886
2	0.01053269	0.01979027	0.02258436
3	0.02083617	0.02796202	0.03020998
4	0.00000000	0.04637414	0.05156304
5	0.06263858	0.07553214	0.09079095

Table 35 – Energy Consumption Results (Joules)

Phase	Thread Group	BZM	JP@GC
0	919.22	2576.20	6744.45
1	1002.24	2546.30	13675.90
2	37917.69	71244.96	81303.69
3	75010.20	100663.26	108755.93
4	282172.87	166946.92	185626.95
5	225498.88	271915.71	326847.41

Appendix F: Performance results

This appendix provides the results of the performance testing conducted across different phases of the experiments. Performance is reported in terms of three primary indicators: average response time (milliseconds), error rate, and throughput. The results are presented separately for the Thread Group (Table 36), BZM (Table 37), and JP@GC (Table 38).

Table 36 – Thread Group Performance tests results

Phase	Average response time (ms)	Error %	Throughput
0	5738	19.09%	7.3849
1	5738	19.09%	7.3849
2	6849	3.93%	4.9527
3	9147	26.45%	3.8750
4	9287	23.33%	3.7910
5	10071	31.25%	3.3663

Table 37 – BZM Performance tests results

Phase	Average response time (ms)	Error %	Throughput
0	255	70.65%	33.7755
1	237	89.55%	20.4794
2	441	84.22%	11.8763
3	854	79.90%	6.8890
4	1007	79.84%	6.4062
5	1445	77.03%	4.1972

Table 38 – JP@GC Performance tests results

Phase	Average response time (ms)	Error %	Throughput
0	220	70.84%	29.9258
1	223	90.45%	19.3349
2	390	85.47%	11.2838
3	766	79.83%	6.4171
4	949	77.15%	5.7613
5	1352	80.10%	8.8843

Appendix G: Maintainability results

The appendix contains results from maintainability assessments that ran throughout all experimental phases. The analysis examines three main indicators which include Maintainability Level, Propagation Cost and Average Complexity, presented in Table 39.

Table 39 – Maintainability Results

Phase	**0**	**1**	**2**	**3**	**4**	**5**
Maintainability Level	81.12	84.89	84.9	84.28	82.65	82.18
Propagation Cost	11.38	4.16	3.02	2.20	1.79	1.06
Average Complexity	1.27	1.30	1.34	1.34	1.38	1.40