INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Development of Microservices-based APIs for Smart Energy Systems

RODRIGO OLIVEIRA COSTA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Development of Microservices-based APIs for Smart Energy Systems

Rodrigo Oliveira Costa

Dissertação
Mestrado em Engenharia de Software

Supervisor: Dr. Sérgio Ramos
Co-supervisor: Dr. João André Pinto Soares

Porto, June, 2025

Statement of Integrity

I hereby declare having 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.

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

ISEP, Porto, June 29, 2025

To my family, whose unwavering support and encouragement have been fundamental throughout my academic journey.

This report was prepared in fulfillment of the requirements of the Dissertation Curricular Unit – DIMEI, within the Master's Degree in Computer Engineering – Software Engineering. The work presented herein was developed as part of the Scientific Research Project *SAtComm: Sustainable Atlantic Communities*, supported by the Interreg Atlantic Area Program 2021–2027, through the European Regional Development Fund (ERDF), under the reference EAPA 0019/2022 – SAtComm.

viii

Abstract

The transition to sustainable energy systems requires the seamless integration of diverse and distributed technologies. However, challenges related to standardization and interoperability across international platforms continue to hinder this progress. This thesis explores the development of a microservices-ready API tailored for Smart Energy Systems, addressing these integration barriers through a modular and scalable backend design. While a fully distributed microservices architecture was initially envisioned, the implemented solution adopts a monolithic architecture structured according to microservices principles, enabling future service decomposition without significant redesign.

The developed API aims to empower citizens to actively participate as prosumers within Energy Communities, supporting functionalities such as peer-to-peer energy trading, load management, and energy storage optimization. Designed with interoperability in mind, the platform promotes collaboration with international partners and facilitates integration into broader smart energy ecosystems. This work includes a comprehensive analysis of requirements, system architecture, implementation, and validation in realistic use cases. By aligning architectural flexibility with future scalability goals, the research contributes to the advancement of citizen-centric and sustainable energy systems.

Keywords: Microservices, Smart Energy Systems, APIs, Energy Communities, Prosumers, Peer-to-Peer Trading

Resumo

Os sistemas de energia inteligentes enfrentam desafios significativos devido à falta de interoperabilidade e integração entre aplicações e plataformas distribuídas. Apesar do avanço das tecnologias no setor, a ausência de padrões unificados e a dificuldade de colaboração entre parceiros internacionais restringem o potencial de inovação, sustentabilidade e eficiência energética. Essa lacuna prejudica a implementação de soluções eficazes para a transição energética e a participação ativa dos cidadãos como elementos centrais no ecossistema energético.

Este trabalho propõe o desenvolvimento de uma API modular baseada em princípios de micro-serviços, como solução para esses desafios. Embora a arquitetura inicialmente prevista fosse distribuída, a implementação adotou uma abordagem monolítica estruturada de forma a permitir futura decomposição em serviços independentes, sem necessidade de grandes reformulações. Essa escolha foi motivada pelas vantagens de modularidade, escalabilidade e flexibilidade, possibilitando integrações complexas em ecossistemas energéticos.

A API desenvolvida visa capacitar os cidadãos a atuarem como *prosumers* (produtores e consumidores simultaneamente) em Comunidades de Energia, oferecendo funcionalidades como negociação de energia ponto a ponto, gestão de carga e otimização do armazenamento de energia. Além disso, a solução foi projetada com foco na interoperabilidade com parceiros internacionais, promovendo a integração em um ecossistema global de energia inteligente.

O trabalho inclui uma análise abrangente dos requisitos técnicos, do design arquitetónico, da implementação prática e da validação da solução em cenários de uso realistas. Ao alinhar flexibilidade arquitetónica com metas de escalabilidade futura, esta investigação contribui para o avanço de sistemas energéticos mais sustentáveis, integrados e centrados no cidadão.

Palavras-chave: Micro-serviços, Arquitetura Modular, Sistemas de Energia Inteligentes, API, Comunidades de Energia, Prosumers, Negociação de Energia Ponto a Ponto.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Doctor Sérgio Ramos, and my co-supervisor, Professor Doctor João André Pinto Soares, for their continuous support, guidance, and availability throughout every stage of this project.

I am especially thankful to my family, whose unwavering support and encouragement have been vital throughout my academic journey - this achievement would not have been possible without them.

Lastly, I also extend my heartfelt thanks to my friends, who have always stood by me, offering both academic help and emotional support, and helping me manage stress along the way.

Contents

xviii

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ASP.NET	Active Server Pages for .NET
CI/CD	Continuous Integration / Continuous Deployment
DDD	Domain-Driven Design
DIMEI	
DIP	Dependency Inversion Principle
DTO	Data Transfer Object
E2E	End-to-End
EMS	Energy Management System
GECAD	Research Group on Intelligent Engineering and Computing for Advanced Innovation Development
GDPR	General Data Protection Regulation
gRPC	google Remote Procedure Call
HMR	Hot Module Replacement
HTTP	Hypertext Transfer Protocol
IEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IPP	Polytechnic Institute of Porto
ISP	Interface Segregation Principle
LSP	Liskov Substitution Principle
MILP	Mixed-Integer Linear Programming
MSA	Microservices Architecture
NPM	Node Package Manager

OCP	Open/Closed Principle
ORM	Object-Relational Mapping
P2P	Peer-to-Peer
PR	Pull Request
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
RDBMS	Relational Database Management System
REST	Representational State Transfer
RL	Reinforcement Learning
SPA	Single-Page Application
SRP	Single Responsibility Principle
UI	User Interface
WBS	Work Breakdown Structure

1 Introduction

1.1 Contextualization

The global transition toward more sustainable and decarbonized energy systems has brought increasing attention to the development and deployment of smart energy systems. These systems leverage digital technologies to optimize the generation, distribution, and consumption of energy, with a strong emphasis on integrating renewable energy sources such as solar and wind. They are key to enhancing energy efficiency, reducing environmental impact, and enabling more resilient and adaptive energy infrastructures. [1]

Despite their potential, the integration of diverse energy systems across regions and platforms remains a complex challenge. One of the primary barriers is the lack of interoperability among systems developed by different vendors and stakeholders. This fragmentation hinders data exchange, operational coordination, and the scaling of energy services across national and international contexts.

To address these challenges, microservices-based architectures offer a compelling solution. Unlike traditional monolithic systems, microservices architectures break down software into smaller, independently deployable components. This modularity allows for greater scalability, flexibility, and ease of integration with other systems and technologies. In the context of smart energy systems, this approach supports decentralized coordination and facilitates the integration of heterogeneous energy assets. [2]

Concurrently, the energy sector is undergoing a fundamental shift in which citizens are becoming active participants in energy systems, often referred to as "prosumers" - individuals who both produce and consume energy. This shift calls for digital solutions that empower citizens to engage in energy communities, manage their own energy production and consumption, and participate in peer-to-peer energy trading networks. To support this vision, robust and interoperable APIs are needed to expose energy data and enable secure, real-time interactions between users and systems. [3]

This thesis explores the design and development of an API based on microservices principles for the integration of smart energy systems. Although the research initially focused on microservices-based APIs, the implemented solution consists of a modular monolithic API, structured to enable future decomposition into independent services. The goal is to create a technical foundation that supports prosumer participation, decentralized energy exchange, and intelligent energy management. By addressing both architectural and user-centered challenges, the work contributes to the broader movement toward more open, inclusive, and sustainable energy infrastructures.

1.2 Problem Description

Despite the increasing adoption of smart energy systems, the challenge of ensuring interoperability between different applications and platforms remains a significant barrier. Numerous applications have been developed, but these solutions are often isolated, with limited integration across international boundaries. This lack of integration hinders the potential for cross-border energy trading, collaborative energy management, and efficient utilization of energy storage.

The main issue lies in the absence of flexible, scalable, and standardized APIs that can connect different systems and enable seamless data exchange. Furthermore, current systems fail to fully engage citizens as active participants in energy production and consumption, a core objective of modern energy policies. [4]

This thesis seeks to address these challenges by designing and implementing a single API based on microservices architecture. This API will enable interoperability between smart energy systems, promoting the participation of citizens as prosumers in energy communities. The API will support functionalities such as energy negotiation, load management, and energy storage, while ensuring the integration of different energy systems across borders.

1.3 Analytical, critical and ethical interpretation

This section provides an overview of the different interpretations of the research, considering the technical, critical, and ethical aspects of the work.

1.3.1 Analytical interpretation

The analytical approach involves examining the technical requirements for integrating diverse energy systems using microservices-based APIs. The study analyzes the challenges in designing APIs that can efficiently support decentralized energy systems while ensuring interoperability. Key technical considerations include the scalability of microservices, the complexity of handling distributed energy data, and the ability to manage various types of energy services (e.g., energy storage, load balancing, peer-to-peer trading).

Through the development of a modular API informed by these principles, the research aims to enhance the communication between energy systems and ensure that they can efficiently interact in a dynamic and global energy market. The analytical interpretation focuses on the identification of these technical challenges and how the proposed solution can address them effectively.

1.3.2 Critical interpretation

The critical interpretation evaluates the strengths and weaknesses of the proposed microservices-based solution. While the modularity and scalability of microservices offer clear advantages, challenges exist in terms of system complexity and the potential for performance bottlenecks. For example, the decentralized nature of smart energy systems may lead to difficulties in managing large volumes of data and ensuring the real-time processing required for dynamic energy trading and load balancing.

Furthermore, the integration of multiple international partners and energy systems introduces regulatory, technical, and business model challenges. The critical interpretation also explores how these challenges can be mitigated through careful design and implementation, ensuring that the solution remains scalable and adaptable to different contexts.

1.3.3 Ethical interpretation

Ethical considerations in this research focus on the implications of integrating citizens into the energy ecosystem as prosumers. As individuals take on roles in both consuming and producing energy,

questions of fairness, access, and privacy become paramount. The ethical interpretation examines how the proposed solution ensures equal participation opportunities for all users, regardless of socio-economic background, and how data privacy and security are maintained within the system.

The responsible handling of energy usage data is a critical ethical concern, as this data can reveal sensitive information about individuals' behavior and energy consumption patterns. The research is committed to addressing these concerns by ensuring that data privacy is maintained through secure API protocols and by promoting transparency in how data is collected and used. Furthermore, the research emphasizes the need for systems that are designed to empower citizens and avoid exacerbating inequalities in energy access.

1.4 Ethical Considerations

This thesis adheres to the ethical standards set forth by the Polytechnic Institute of Porto (IPP), including the IPP Code of Conduct for students, which ensures academic integrity and responsible research practices. Ethical considerations play a crucial role throughout this work, particularly in the context of handling data, ensuring privacy, and promoting equity in energy access. Below are the primary ethical principles observed during the course of this project:

Academic Integrity:

The work strictly follows Article 6 of the IPP Code of Conduct, specifically Statement 2.8, which mandates the proper citation of ideas, sentences, paragraphs, or texts sourced from other authors. All sources of information, research, and ideas were meticulously cited to avoid plagiarism and ensure proper credit is given. Every piece of external content used in this thesis has been referenced accurately, in accordance with academic citation standards.

Authorship and Collaboration:

Statement 2.9 prohibits presenting previously published or collaborative work without explicit acknowledgment. This principle was fully adhered to throughout the project. Any collaborative contributions, whether in code, ideas, or research, have been clearly stated, and proper attribution has been provided. Figures, data, and other collaborative work have been cited appropriately to reflect their respective contributors.

Transparency and Integrity of Results:

Statement 2.10 emphasizes the importance of presenting results truthfully, without falsification or misleading interpretations. This ethical guideline was strictly followed, ensuring that all findings, whether positive or negative, were reported accurately. No results were fabricated or altered to align with desired outcomes, ensuring the integrity of the research.

Ethical Research Practices:

In accordance with Article 8 of the IPP Code, this work was conducted with thoroughness and careful analysis. Research was carried out methodically, and all related works were cited comprehensively. Furthermore, the principles outlined in Article 10 were honored, ensuring that the research was executed with diligence, results were reproducible, and the methodology was transparent and accessible for verification.

Privacy and Data Security:

Given the nature of the work, which involves the design of a modular API for smart energy systems, data privacy and security were paramount ethical concerns. Any data collected, whether user data or energy consumption information, was treated with the utmost respect for privacy. Only anonymized data was used in the analysis, and all efforts were made to comply with relevant data protection laws, such as GDPR, to ensure that no individual's personal data was misused or disclosed without proper consent.

Equity in Energy Access:

One of the major goals of this thesis is to enable fair and equitable participation in energy communities, allowing citizens to act as prosumers. Ethical considerations related to fairness and equity in energy access were central to the design of the API and systems. This work considers the potential for marginalized groups to benefit from the developed smart energy systems and ensures that the solution does not discriminate against any socio-economic group. A commitment to inclusivity and accessibility was embedded in the design process.

Transparency in Algorithmic Decision-Making:

As the project involves the use of algorithms for energy management and peer-to-peer transactions, the transparency of these algorithms is a key ethical consideration. Efforts were made to ensure that users can understand how decisions are made within the system, especially regarding energy trading, load balancing, and storage management. The goal is to provide transparency and ensure that all users have access to clear, understandable information about how their data is being used and how energy transactions are conducted.

Environmental Responsibility:

Given the project's focus on smart energy systems, the ethical responsibility of reducing environmental impact was also considered. The project emphasizes the development of energy systems that can contribute to sustainability by enabling better energy management, optimizing resource use, and reducing carbon footprints. This aligns with broader ethical goals of promoting sustainability and reducing negative environmental impacts.

Compliance with the IEEE Code of Ethics:

In line with the IEEE (Institute of Electrical and Electronics Engineers) Code of Ethics, this work maintains high standards of honesty, integrity, and transparency. These principles were integrated into the development process, ensuring that the research and outcomes contribute positively to the field of software engineering and smart energy systems. Avoidance of conflicts of interest, both in terms of biases and personal interests, was a priority throughout the project to ensure that decisions were made objectively and with the best interests of the global energy community in mind.

By adhering to these ethical guidelines, this thesis seeks to ensure that the work is conducted with the highest standards of academic integrity, respect for privacy, and fairness, while contributing to the development of sustainable and inclusive energy solutions for the future.

1.5 Document Structure

This document is organized into seven chapters, as follows:

Chapter 1: Introduction

This chapter introduces the motivation for the project, the main research questions, and the objectives of the dissertation. It also provides an overview of the approach adopted for the development of a modular API based on microservices principles for smart energy systems and offers a brief description of the structure of the document.

Chapter 2: Project Planning

This chapter outlines the planning phase of the project, detailing the steps taken to ensure its successful execution.

Chapter 3: State of the Art

This chapter reviews the current state of the art in the field of smart energy systems, focusing on the integration of microservices-based APIs. It examines existing solutions, technologies, and methodologies, highlighting the challenges and gaps in facilitating seamless integration among international partners. It also reviews the role of APIs in enabling functionalities such as energy trading, load management, and energy storage in the context of smart grids and energy communities.

Chapter 4: Analysis and Solution Design

This chapter describes the design of the proposed microservices-based API architecture for smart energy systems. It outlines the core principles and components of the system, detailing how the API will facilitate integration, energy trading, load management, and storage management. The design process also considers scalability, security, and ease of use, aiming to create a flexible solution suitable for diverse international partners.

Chapter 5: Solution Implementation

This chapter presents the implementation details of the designed solution and the evaluation process used to test the developed API. It covers the development process of the microservices-based API, the technologies used, and the integration of different components. Challenges encountered during the implementation phase are discussed, and the final architecture is presented, highlighting how the API was developed to meet the identified needs and objectives.

Chapter 6: Conclusion

The final chapter summarizes the main findings and contributions of the thesis. It reflects on the achievements of the research, discussing how the proposed solution meets the objectives and solves the identified problems. The chapter also suggests directions for future work, exploring potential improvements to the API and its application in broader contexts of energy management and integration across different regions.

2 Project Planning

2.1 Skills Management

This chapter focuses on assessing the skills and capabilities of the individual or team involved in the project, identifying both strengths and areas for development. A comprehensive evaluation of skills is crucial to ensure that the project is executed efficiently and effectively. Understanding the competencies of those involved allows for the alignment of tasks with the right expertise, highlights areas that may require further training or support, and fosters a collaborative environment where individuals can contribute their unique strengths. The identification of these areas is essential for building a solid foundation and ensuring that the project progresses smoothly toward achieving its goals.

The skills assessment is divided into several key components:

- **Identification of Key Skills**: This involves identifying the critical skills required to successfully complete the project, such as technical expertise, problem-solving abilities, teamwork, and communication skills.
- **Evaluation of Current Capabilities**: The current capabilities of the team are evaluated through self-assessment, peer reviews, and feedback. This helps highlight strengths that can be leveraged throughout the project, as well as potential gaps in skills that could hinder progress.
- **Targeted Development**: After identifying areas for improvement, strategies are devised to address skill gaps. This may involve training sessions, collaborative learning, or seeking external expertise where necessary.
- **Ongoing Monitoring**: Regular assessments of skill development and task performance are conducted throughout the project to ensure that any emerging skill gaps are promptly addressed, and the project remains on track.

By systematically managing and developing the skills of the project team, this chapter ensures that each individual is positioned to contribute effectively, improving the likelihood of project success.

2.1.1 Skills Diagnosis

The skills diagnosis process aims to evaluate the importance of each competency in the context of the project and assess the current proficiency levels based on self-evaluation and peer evaluation. This evaluation helps to identify both the strengths that can be leveraged and the areas that need improvement to ensure the project's success.

In the context of this project, the key competencies were identified based on their relevance to the tasks and the desired outcomes. The evaluation was carried out through a self-assessment by the project team members and peer evaluations, helping to create a comprehensive understanding of the team's capabilities.

Below is the Table 1 with a list of essential skills, their importance for the project's success, and the corresponding self-evaluation and peer evaluation scores:

Table 1 - Self and Peer Evaluation of Key Skills and Their Importance

Skills	Importance	Self-evaluation	Peer Evaluation
Problem analysis and resolution	5	4	4
Lifelong learning	5	5	5
Teamwork and collaboration	5	4	5
Motivation for excellence	5	5	4
Adaptability and flexibility	5	4	5
Ethics and social responsibility	5	5	5
Foreign language proficiency	4	4	5
Technical skills in the specific knowledge area	5	5	4
Decision-making	4	4	5
Time management	5	5	4
Risk-taking ability	4	4	5
Oral communication	4	5	4
Written communication	5	4	5
Resilience	4	5	5
Active listening	4	4	4
Interpersonal relationships	4	5	5
Leadership	5	5	4
Creativity and innovation	4	4	4
Adaptation and flexibility	5	4	5
Critical thinking	5	4	4
Planning and organization	5	4	5
Emotional management	4	5	5
Negotiation	5	4	4
Empathy	5	5	5
Assertiveness	5	4	5
Stress management	4	4	5
Proactivity	5	5	5

2.1.2 Most Developed Competencies

After completing the skills diagnosis, several key competencies have been identified as well-developed, providing a strong foundation for the successful execution of the project. These competencies ensure that both technical and managerial aspects of developing a modular API based on microservices principles for smart energy systems are well-handled:

- **Lifelong learning**: A commitment to lifelong learning is crucial for staying current with the rapidly evolving technologies and methodologies in the field of microservices, APIs, and smart energy systems. As the project involves integrating cutting-edge technologies, the ability to continuously acquire and apply new knowledge ensures that the design and implementation of the system remains innovative and effective throughout the development process;
- **Motivation for excellence**: This competency drives the project to maintain high standards throughout the development lifecycle, from designing microservices to building a robust modular API. Ensuring a focus on excellence promotes a results-oriented approach, which is vital for delivering a high-quality, secure, and efficient system for the smart energy application.
- **Ethics and social responsibility**: Developing smart energy systems requires a strong ethical foundation, particularly when it comes to privacy, data security, and the responsible use of technology. Ethical considerations guide the design and implementation of the system, ensuring that it meets legal standards and promotes environmental sustainability. This competency is essential for making decisions that will positively impact the broader community and future generations;
- **Technical skills in the specific knowledge area**: A key competency for the success of the project is the technical expertise in microservices and APIs. Understanding how to design and implement scalable, secure microservices and RESTful APIs is essential for the creation of the smart energy system. Additionally, this includes proficiency in integrating these technologies within the context of the energy domain, ensuring that the API is functional, secure, and capable of handling real-time data in a dynamic environment;
- **Time management**: Efficient time management is fundamental for balancing the various phases of the project, including research, development, and testing. This competency helps to ensure that tasks are completed on time, project milestones are achieved, and resources are optimally allocated throughout the lifecycle of the project. Effective time management is essential for keeping the project on schedule and preventing delays in the development process;
- **Leadership**: Leadership is a critical competency for driving the project forward. Whether working in a team or independently, effective leadership ensures the project stays on track, deadlines are met, and any challenges encountered are addressed proactively. Leadership also helps maintain motivation and focus, ensuring that all tasks are completed effectively and within the established timeframe.

2.1.3 Competencies to Develop

While there are several well-developed competencies, there are still key areas that require further development to ensure the successful completion and implementation of the project. These areas are essential for enhancing both the technical and managerial aspects of developing microservices-based APIs for smart energy systems:

- **Problem Analysis and Resolution**: As the development of a microservices-based API for smart energy systems progresses, strong problem analysis and resolution skills will be crucial for identifying and addressing any technical challenges that arise. This includes tackling issues related to system integration, scalability, performance, and security. Developing this competency will help in analyzing complex problems, troubleshooting effectively, and finding innovative solutions that ensure the smooth functioning and continuous advancement of the project, keeping it aligned with its objectives and requirements;
- **Adaptation and Flexibility**: As the project progresses, unforeseen challenges and changes are inevitable. Strengthening the ability to adapt to new requirements or unexpected technical issues will be critical to maintaining the project's momentum. This competency will help in responding effectively to any changes in scope, adjusting to new technologies, or addressing any issues with integration in the evolving system architecture. Developing greater flexibility will allow for smoother adjustments throughout the project lifecycle;
- **Planning and Organization**: Efficient planning and organization are essential for managing multiple phases of the project, such as research, design, development, and testing. Strengthening this competency will help ensure that tasks are structured, milestones are met, and the project stays within its timeline. Developing better planning skills will also help in allocating resources effectively and maintaining clear priorities across different stages of the project;
- **Stress Management**: During the project's lifecycle, periods of high pressure or tight deadlines may arise. Strengthening stress management skills is important for maintaining productivity and making sound decisions under pressure. The ability to manage stress effectively will ensure that the project team remains focused, well-organized, and efficient, even in challenging situations. Improving this competency will help mitigate stress-related risks that could affect the project's quality and delivery.

2.1.3.1 Action Plan

To improve the competencies required for the successful completion of the microservices-based API for smart energy systems project, the following specific actions should be implemented:

- **Problem Analysis and Resolution**: Strengthen the ability to effectively analyze and resolve technical issues encountered during the development of the system and API. Regularly practice problem-solving techniques such as root cause analysis, the 5 Whys, or troubleshooting steps specific to system integration, performance optimization, and error handling. Actively engage in team problem-solving sessions to gather diverse perspectives and ensure comprehensive solutions. Document common issues and solutions to build a knowledge base that can be referenced in the future, ensuring efficient problem resolution throughout the project lifecycle. Additionally, collaborate with colleagues and mentors to review complex issues, ensuring a structured approach to identifying and addressing technical challenges promptly;
- **Adaptation and Flexibility**: Develop the ability to quickly adapt to new challenges or changes in the project scope. To strengthen this competency, regularly engage in scenario-based exercises where you must adjust the project's direction in response to new technical requirements or external constraints. Participate in team discussions focused on handling unexpected issues or incorporating new technologies, ensuring that flexibility becomes a part of the project's adaptive planning process;

- **Planning and Organization**: Use project management tools such as Trello, Asana, or Jira to create detailed project timelines, track deliverables, and allocate tasks effectively. Establish regular checkpoints for assessing the progress of various project components (e.g., microservices, API development, and system integration). Create clear, detailed action plans that break down large tasks into manageable steps. This will ensure that all aspects of the project are well-organized, that deadlines are met, and that the project is advancing according to the set priorities;
- **Stress Management**: Develop techniques for managing stress through structured time-blocking strategies and setting realistic goals for project milestones. Implement mindfulness or stress-relief techniques such as short breaks, deep breathing exercises, or team-building activities. Maintain a balanced workload to avoid burnout by distributing tasks evenly across the team and ensuring that high-pressure periods are well planned in advance. This will improve your resilience during demanding phases of the project and contribute to maintaining focus and performance under tight deadlines.

Focusing on these actions will significantly enhance the competencies of adaptation and flexibility, planning and organization, and stress management, which are essential for the success of virtually any project. Strengthening these skills will ensure that the project remains adaptable, well-organized, and capable of managing challenges that arise, ultimately helping it stay on track and aligned with its goals.

2.2 Project Planning and Control

This section outlines the phases of the project, their objectives, key activities, and the timeline for completion. The plan ensures systematic progression from problem identification to the final delivery of the dissertation.

2.2.1 Project Charter

This project aims to develop a modular microservices-based API tailored for Smart Energy Systems, addressing key challenges such as integration, scalability, and citizen engagement. The API will enable seamless communication and interoperability among diverse energy technologies, fostering innovation and sustainability.

2.2.1.1 Outcomes

- Development of a robust, scalable API architecture supporting modular integration of smart energy systems.
- Empowerment of energy consumers and prosumers with functionalities such as peer-to-peer energy trading, load management, and energy storage optimization.
- Contribution to global energy transition goals by promoting decentralized and sustainable energy solutions.
- Enhancement of personal expertise in advanced microservice architectures and energy system integrations through hands-on project execution.
- Development of high-level technical documentation and project management skills as part of the thesis deliverables.

2.2.1.2 Objectives

The core aim of this project is to investigate the development and implementation of a microservices-based API to facilitate the integration of smart energy systems and enable the active involvement of citizens as prosumers in energy communities. This goal is addressed through several key objectives (O), which guide the research process and outline the expected contributions of this work.

Objectives (O)

- **O1**: Investigate and analyze the state of the art in smart energy systems, interoperability approaches, and microservices-based architectures to support the proposed solution.

  This objective focuses on conducting a thorough scientific investigation into current technologies, frameworks, and methodologies related to smart energy systems. It includes the analysis of interoperability challenges, the role of microservices-based architectures in distributed energy environments, and existing strategies for citizen participation as prosumers. The insights gained from this analysis will serve as a foundational basis for the design and implementation phases of the project.

- **O2**: Design and implement a microservices-based architecture for APIs that enables interoperability between diverse smart energy systems.

  This objective aims to develop a modular architecture using microservices that can integrate various energy systems (e.g., solar panels, wind turbines, energy storage systems, etc.) from different vendors and regions. The architecture will focus on scalability and flexibility, ensuring that it can support diverse technologies and standards.

- **O3**: Develop APIs that allow citizens to participate as prosumers in energy communities, enabling them to manage energy production, consumption, and storage.

  This objective focuses on creating APIs that facilitate the involvement of citizens in energy communities. Prosumers will be able to generate, store, and consume energy, as well as trade energy with others through peer-to-peer (P2P) networks. These APIs will handle various aspects of energy management, from transaction logging to monitoring and control of energy usage.

- **O4**: Implement functionalities to support peer-to-peer (P2P) energy trading, energy storage management, and load balancing within smart energy systems.

  The APIs developed under this objective will include features to enable decentralized energy trading (P2P), the management of energy storage systems, and real-time load balancing. The aim is to create a decentralized energy market where users can buy, sell, and exchange energy in a flexible, efficient, and secure manner.

- **O5**: Evaluate the performance, scalability, and security of the developed APIs, ensuring they can support the integration of international smart energy systems.

  This objective involves testing the APIs in various real-world scenarios to assess their performance under different conditions. It will also focus on ensuring the security of energy data and the APIs' ability to scale as more users and systems are added. Performance metrics such as latency, throughput, and failure rates will be analyzed, and security protocols will be put in place to prevent unauthorized access and data breaches.

- **O6**: Conduct an ethical analysis of the data management practices within the developed APIs, ensuring privacy and equity for all users.

  The ethical implications of handling user data are critical to the success of this project. This objective aims to ensure that all energy usage and production data are handled in compliance with privacy laws and ethical standards. Additionally, it will ensure that energy access is equitable, promoting fair participation in energy communities, regardless of economic or social background.

- **O7**: Provide recommendations for the future development and adoption of microservices-based APIs for smart energy systems, based on the findings of the research.

  This objective focuses on offering practical guidelines for the further development and adoption of microservices-based APIs in the energy sector. Based on the research findings, the thesis will suggest best practices for improving the integration of smart energy systems and empowering citizens as prosumers.

2.2.1.3 Main Stakeholders

- **Project Investigator**: Responsible for project execution, from design to validation.
- **GECAD**: Founder of the project, providing funding, computational resources, and technical support to ensure project alignment with cutting-edge research objectives.
- **Supervisors**: Academic experts providing guidance and oversight to ensure quality.
- **Industry Partners**: Collaborators supplying real-world use cases, tools, and feedback.
- **End-Users**: Energy communities and prosumers, who will test and benefit from the API.

2.2.1.4 Risk Management

Risk Identification

- **Technical Risks**: Potential challenges in implementing microservices architecture, such as inter-service communication issues, latency, and ensuring fault tolerance.
- **Data Accessibility Risks**: Limited or delayed access to real-world energy system data for testing purposes.
- **Time Risks**: Unforeseen delays in project phases, such as design or testing, which could impact overall timelines.
- **Budget Risks**: Overruns due to unanticipated costs for tools, resources, or extended project timelines.
- **Stakeholder Risks**: Potential misalignment of expectations or delays in feedback from industry partners and end-users.

Risk Mitigation Strategies

- **Technical Risks**:
    - Use robust testing frameworks to identify and resolve service communication issues early.
    - Leverage simulation environments for controlled testing of APIs.
- **Data Accessibility Risks**:
    - Develop simulated datasets to mimic real-world energy systems when actual data is unavailable.
    - Establish early agreements with stakeholders for timely access to required data.
- **Time Risks**:
    - Follow Agile sprints to address potential delays iteratively and keep tasks on track.
    - Use Gantt charts to monitor progress and adjust schedules as needed.
- **Budget Risks**:
    - Prioritize open-source tools and frameworks to minimize expenses.
    - Conduct regular budget reviews to monitor expenditures and avoid overruns.
- **Stakeholder Risks**:
    - Schedule regular meetings and feedback sessions to align expectations and address concerns proactively.
    - Maintain transparent communication with stakeholders to ensure ongoing support and engagement.

2.2.1.5 Restrictions

- **Time Constraints**: The project is scheduled to be completed within eight months, necessitating a well-structured timeline with clearly defined milestones.
- **Confidentiality**: The project involves sensitive data from stakeholders and collaborators, requiring adherence to strict confidentiality agreements. All data will be handled in compliance with privacy laws and ethical guidelines, such as GDPR where applicable. Secure data storage and transmission protocols will be implemented to prevent unauthorized access or breaches. Regular reviews will ensure that all team members and proces ses align with confidentiality requirements.
- **Budget**: Operating under a constrained budget requires efficient allocation of financial resources. Open-source tools and technologies will be prioritized to minimize costs without compromising on quality.
- **Microservices-based Architecture**: The project is bound by the technical requirements of using a microservices-based architecture.

2.2.1.6 Phases/Deadlines

The project is divided into structured phases, each essential for fulfilling its objectives effectively.

Phase 1: Problem Identification and Objective Definition

- **Objective**: Understand the limitations and challenges of integrating microservices-based APIs in smart energy systems.
- **Activities**:
    - Initial meetings with supervisors and stakeholders to collect technical and functional requirements.
    - Identification of core challenges such as scalability, interoperability, and citizen participation as prosumers.
    - Alignment of project goals with smart energy system objectives.

Phase 2: Planning and Research

- **Objective**: Explore existing approaches to microservices architectures and APIs for smart energy systems while establishing a strong foundation for the project through structured planning and research.
- **Activities**:
    - Development of the Project Charter to outline the project's objectives, scope, and deliverables.
    - Creation of the Work Breakdown Structure (WBS) to organize the project's tasks and activities.
    - Detailed Project Planning to establish timelines, resources, and dependencies.
    - Literature review focusing on relevant research questions, technologies, methodologies, and case studies related to microservices architectures and smart energy systems.
    - Analysis of key requirements for API functionalities, including load management, energy trading, and storage optimization.
    - Mapping of tools and technologies relevant to the project.
    - Drafting the "State of the Art" chapter of the thesis to summarize and analyze existing solutions and their relevance to the project.
    - Preparation of the final Presentation to summarize the planning and research outcomes.

Phase 3: Design and Implementation of Solution

- **Objective**: Develop a scalable and secure architectural model for the API and develop a prototype of the proposed API solution.
- **Activities**:
    - Design modular API architecture using microservices principles.
    - Selection of technologies and frameworks.
    - Define criteria for evaluating the solution (e.g., performance, scalability, interoperability).
    - Implementation of the microservices-based API based on the designed architecture.
    - Integration with simulated or real-world smart energy systems for initial validation.

Phase 4: Testing and Validation

- **Objective**: Evaluate the API solution in terms of performance, security, and usability.
- **Activities**:
    - Conduct unit, integration, and performance tests under different conditions.
    - Evaluate scalability using load simulations to mimic real-world energy system demands.
    - Document testing results and refine the solution as needed.

Phase 5: Documentation and Final Delivery

- **Objective**: Complete the thesis and prepare for its defense.
- **Activities**:
    - Write the final chapters of the dissertation, including findings, evaluations, and conclusions.
    - Prepare comprehensive technical documentation for the API and a user manual.
    - Rehearse the defense presentation and finalize visual aids for evaluators.

2.2.1.7 WBS

A well-organized work breakdown structure (WBS) provides a clear framework for deliverables and phases, ensuring systematic and efficient progression throughout the project lifecycle, as seen in the two images below.

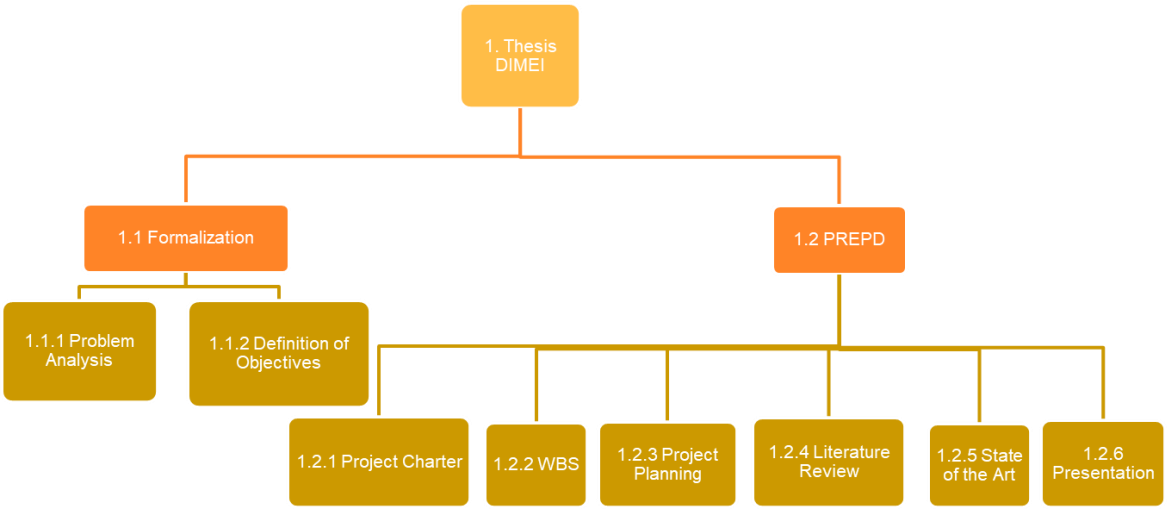

Figure 1 - WBS Part 1

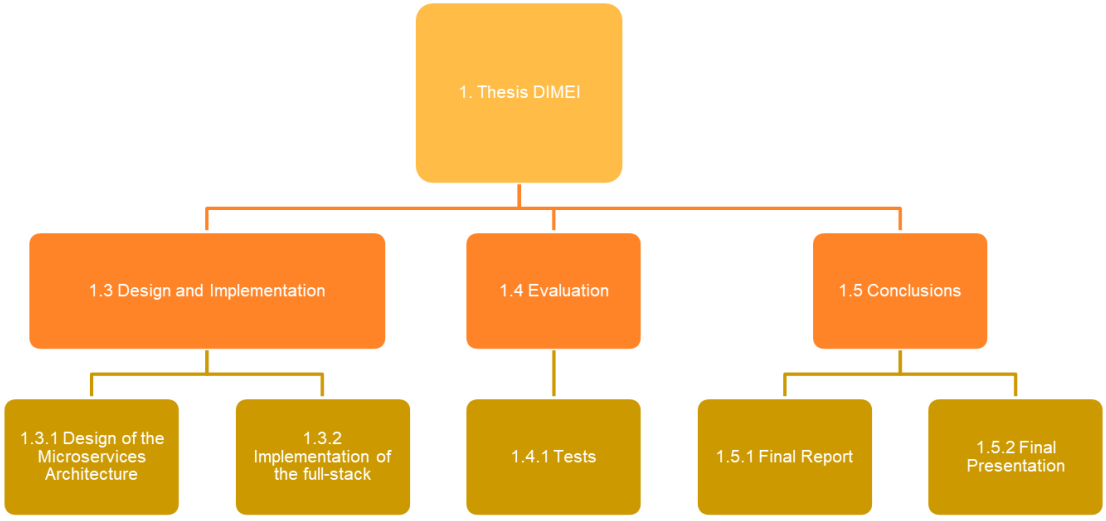

Figure 2 - WBS Part 2

2.2.1.8 Project Schedule

The project schedule is a comprehensive tool designed to ensure tasks are completed efficiently and on time, as seen in the image below. By breaking down activities into specific time frames, it provides clarity on start and end dates, assigns responsibilities to team members, and facilitates effective resource allocation. Detailed timelines, complete with milestones and deliverables, are crafted to keep the project on track.

WBS	Nome da Tarefa	Nível WBS	Duration	Start	Finish	% Complete
1	**Thesis DIMEI**	**Project**	**199 days**	**Fri 01/11/24**	**Thu 31/07/25**	**36%**
1.1	**Formalization**	**Phase**	**4 days**	**Sat 12/10/24**	**Wed 16/10/24**	**100%**
1.1.1	Problem Analysis	Deliverable	2 days	Sat 12/10/24	Mon 14/10/24	100%
1.1.2	Definition of Objectives	Deliverable	2 days	Tue 15/10/24	Wed 16/10/24	100%
1.2	**PREPD**	**Phase**	**60 days**	**Thu 17/10/24**	**Sat 04/01/25**	**100%**
1.2.1	Project Charter	Deliverable	3 days	Thu 17/10/24	Mon 21/10/24	100%
1.2.2	WBS	Deliverable	5 days	Tue 22/10/24	Mon 28/10/24	100%
1.2.3	Project Planning	Deliverable	9 days	Tue 29/10/24	Fri 08/11/24	100%
1.2.4	Literature Review	Deliverable	16 days	Sat 09/11/24	Sat 30/11/24	100%
1.2.5	State of the Art	Deliverable	23 days	Sun 01/12/24	Tue 31/12/24	100%
1.2.6	Presentation	Deliverable	3 days	Wed 01/01/25	Fri 03/01/25	100%
1.3	**Design and Implementation**	**Phase**	**61 days**	**Mon 06/01/25**	**Mon 31/03/25**	**0%**
1.3.1	Decision of technologies to use	Task	9 days	Mon 06/01/25	Thu 16/01/25	0%
1.3.2	Design of the Microservices Architecture	Deliverable	16 days	Fri 17/01/25	Fri 07/02/25	0%
1.3.3	Implementation of the full-stack	Deliverable	36 days	Mon 10/02/25	Mon 31/03/25	0%
1.4	**Evaluation**	**Phase**	**45 days**	**Tue 01/04/25**	**Sat 31/05/25**	**0%**
1.4.1	Tests	Deliverable	27 days	Tue 01/04/25	Wed 07/05/25	0%
1.4.2	Validation	Task	17 days	Thu 08/05/25	Sat 31/05/25	0%
1.5	**Conclusions**	**Phase**	**6 days**	**Sun 01/06/25**	**Fri 06/06/25**	**0%**
1.5.1	Final Report	Deliverable	4 days	Sun 01/06/25	Wed 04/06/25	0%
1.5.2	Final Presentation	Deliverable	2 days	Thu 05/06/25	Fri 06/06/25	0%

Figure 3 - Project Schedule

To visualize the project's progression, the Gantt [5] chart below below plays a pivotal role, offering a clear representation of phases, tasks, and their interdependencies. This visualization highlights overlapping tasks, enabling project managers to anticipate bottlenecks and reallocate resources as needed. By tracking task completion and progress against the schedule, the Gantt chart makes it easy to identify tasks that are on track, those requiring additional attention, and those already completed, ensuring a streamlined and transparent project management process.

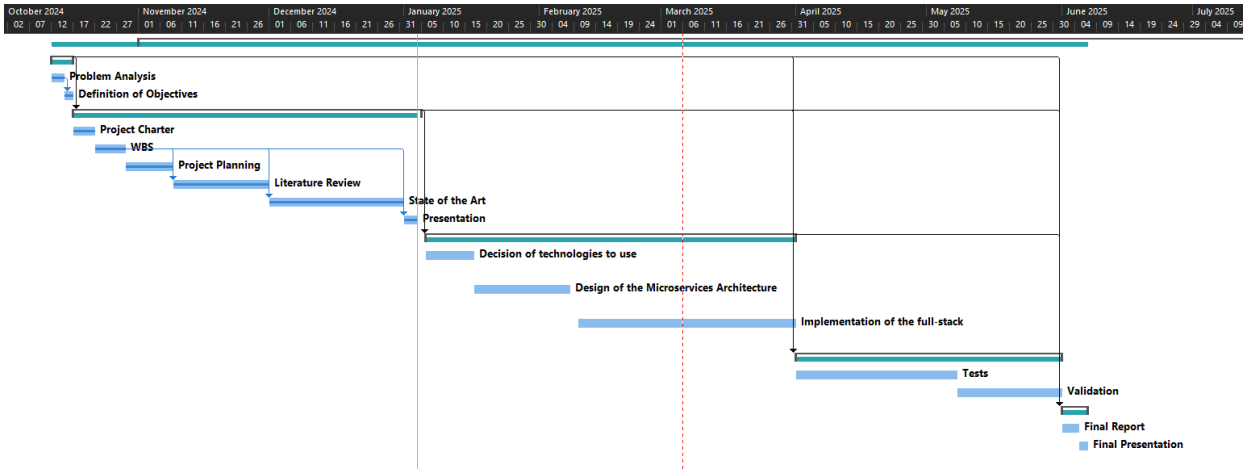

Figure 4 - Gantt Diagram

2.2.2 Planning Approach

The planning approach for this project is rooted in the integration of Agile and PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) methodologies, chosen for their adaptability, iterative processes, and systematic rigor. Agile provides a flexible framework for iterative progress, continuous feedback, and stakeholder collaboration, ensuring that the project remains responsive to dynamic requirements. Meanwhile, the PRISMA methodology introduces a structured and transparent approach to research and development, ensuring comprehensive documentation, reproducibility, and alignment with academic standards. Together, these methodologies create a robust foundation for achieving project objectives efficiently and effectively.

2.2.2.1 Agile

This project employs an Agile planning approach, emphasizing flexibility, iterative progress, and continuous feedback. Agile is particularly well-suited to this initiative due to the dynamic nature of smart energy systems and the need for adaptive development in response to stakeholder feedback. [6]

The steps involved in the Agile methodology process are visually represented in the image below, illustrating each phase that contributes to iterative development and continuous improvement.

Figure 5 - Agile Methodology [7]

Key Features of Agile in This Project

- **Iterative Development**: The project is broken into sprints, with each sprint delivering functional components of the microservices-based API. Iterative cycles ensure continuous progress and allow for incremental improvements.
- **Stakeholder Collaboration**: Regular engagement with supervisors, industry partners, and end-users ensures the solution aligns with real-world requirements and expectations. Feedback gathered during each sprint is used to refine the API and its functionalities.
- **Flexibility**: The Agile approach allows the team to adapt to changing requirements or unforeseen challenges. By prioritizing tasks based on value and feasibility, the project maintains momentum while addressing critical objectives.
- **Transparency**: Regular progress updates through sprint reviews and retrospectives foster accountability and keep stakeholders informed about the project's status.

- **Continuous Testing and Validation**: Testing is integrated into each sprint, ensuring that the API meets performance, security, and scalability criteria as it is developed.

2.2.2.2 PRISMA

The approach employed in this thesis follows a structured, systematic methodology to ensure rigorous analysis, transparent development, and reproducibility of results. In this case, the Systematic Design and Development approach is guided by the PRISMA methodology (Preferred Reporting Items for Systematic Reviews and Meta-Analyses), traditionally used for systematic reviews but adapted to structure the entire process of designing and implementing microservice-based APIs for smart energy systems. PRISMA is well-suited to this research as it offers a transparent and comprehensive framework that ensures all stages of the project, from literature review to solution evaluation, are meticulously planned and executed. [8]

The methodology can be divided into the following stages:

Identification and Selection of Relevant Literature and Technologies:

The first step is to systematically review the literature on microservices, smart energy systems, and API design, with a particular focus on current trends and best practices for energy systems and integration challenges. Literature on RESTful APIs, microservice architecture, and their usage in smart energy systems is identified and selected from academic papers, books, and industry reports. This stage also includes reviewing current tools, platforms, and frameworks used for integrating various partners (including international ones) into energy systems. This process follows PRISMA's guidelines for systematically sourcing and selecting studies that are most relevant to the objectives of the project, ensuring that the state of the art in these areas is comprehensively reviewed.

Screening and Data Extraction:

After collecting relevant studies and technologies, the next step is to evaluate and filter them based on predefined inclusion and exclusion criteria. This screening process ensures that only the most relevant and high-quality sources are used. Data extraction focuses on capturing key information on API design patterns, microservices for energy systems, and integration challenges. Emphasis is placed on approaches that address scalability, flexibility, and security in energy management systems. Additionally, case studies that illustrate the real-world application of microservice-based APIs for energy systems and related fields are scrutinized.

Data Synthesis and Analysis:

In this stage, the collected data is synthesized to identify trends, best practices, and gaps in the current research. The strengths and weaknesses of existing solutions for API integration, microservices in energy systems, and smart energy management are analyzed. A comparative analysis is conducted to assess the suitability of existing approaches for addressing the integration problem posed in this thesis. The insights derived from this synthesis help to inform the design of a more efficient and scalable API architecture for smart energy systems, offering enhancements over existing methods, particularly in the context of international integration and prosumer participation.

Solution Design and Development:

Based on the findings from the literature review and synthesis, the design phase focuses on creating a detailed architecture for the proposed solution. The design will incorporate best practices from microservice architecture, ensuring scalability, security, and flexibility in the integration of different systems. Reinforcement Learning (RL) based algorithms will be considered for enabling adaptive and efficient energy management within the API design. The goal is to design an API that supports peer-to-peer energy trading, load balancing, and energy storage functionalities while ensuring easy integration with external partners. The system will be designed to allow seamless communication between different actors (prosumers, communities, and energy providers) within the energy system.

Implementation and Evaluation:

The implementation phase involves building the microservices-based API according to the design specifications. The API will be tested in a real-world environment, using both simulated and actual smart energy data to evaluate their functionality. The PRISMA methodology will be applied during the evaluation process, focusing on a transparent comparison of the proposed solution's performance against existing methods. Key performance indicators (KPIs) such as the number of successful energy transactions, system efficiency, integration ease, and security will be used to assess the solution. Additionally, performance evaluations will consider the scalability and robustness of the API when handling large numbers of partners (both domestic and international).

Synthesis of Results and Reporting:

The final step of the approach involves synthesizing the results from the implementation and evaluation phases. This stage will include a thorough analysis of the findings, focusing on the ability of the designed API to solve the integration challenges, support prosumer participation, and improve the functionality of smart energy systems. The results will be compared with the expected outcomes and goals defined in the research objectives. This section will present the final evaluation of the solution, its advantages, and limitations, and provide concrete recommendations for future work, particularly in the context of international smart energy system integration and enhancing user engagement.

The PRISMA methodology is employed in this thesis to ensure a structured, transparent, and reproducible approach to the development of a microservices-based API for smart energy systems. This established framework, typically applied in systematic reviews, is adapted to guide the research process from the literature review through to the design and evaluation of the solution. Each stage is informed by comprehensive research and analysis, ensuring the final solution is grounded in the most current and relevant practices. This approach guarantees the academic rigor of the work, ensuring that the contributions of this thesis are both meaningful and impactful in the integration of smart energy systems.

3 State of the Art

This chapter presents an in-depth review of the current landscape of microservices-based API development, with a particular focus on smart energy systems. The review aims to explore key areas such as best practices for designing and developing RESTful APIs, tools and methodologies for automating API testing, and strategies for ensuring API security. The chapter is organized into three primary objectives: first, to examine the flexibility and scalability of microservices architectures in the context of smart energy systems; second, to identify effective patterns and approaches for optimizing the development and deployment of APIs; and third, to assess the performance of current tools and frameworks used in API development. Furthermore, we investigate how emerging technologies, such as machine learning and advanced algorithms, can be integrated to enhance API functionality, security, and efficiency.

3.1 Research Questions

The core aim of this thesis is to investigate the development and implementation of a microservices-based API to facilitate the integration of smart energy systems and enable the active involvement of citizens as prosumers in energy communities. This goal is addressed through several key research questions (RQs), which guide the research process and outline the expected contributions of this work.

Research Questions (RQs)

- **RQ1**: How can a microservices-based architecture be designed to enable interoperability between diverse smart energy systems?

  This question explores how a modular, scalable architecture can be structured to connect different energy systems, ensuring seamless data exchange and functionality across heterogeneous platforms. The answer to this question will contribute to the creation of a flexible and adaptable system that can support various international energy initiatives.

- **RQ2**: How can the design of APIs support the integration of energy communities, enabling citizens to act as prosumers in energy systems?

  This question focuses on the design of APIs that empower citizens to actively participate in the energy market by allowing them to manage energy generation, consumption, and storage, as well as engage in peer-to-peer energy trading. It aims to determine how APIs can be structured to handle decentralized transactions and ensure transparency, security, and fairness.

- **RQ3**: What functionalities and services must be supported by the APIs to facilitate peer-to-peer energy trading, load management, and energy storage within smart energy systems?

This question looks at the specific services and features that need to be provided by the APIs to enable the effective management of energy resources, including energy transactions, load balancing, and storage optimization. It aims to define the core capabilities required for seamless operation within energy communities.

- **RQ4**: How can the APIs ensure the scalability, performance, and security needed to support the integration of international smart energy systems?

  This question investigates the technical and security requirements necessary to make the APIs robust and capable of operating across a range of platforms and countries. It examines how scalability and security concerns can be addressed within the architecture of the APIs, ensuring they can handle growing demand and protect sensitive energy data.

- **RQ5**: What are the best practices for ensuring the ethical use of data within smart energy systems, particularly in terms of privacy and equity in energy access?

  This research question addresses the ethical considerations related to the handling of energy consumption and production data, especially when involving citizens as prosumers. It focuses on data privacy, security, and ensuring that the benefits of energy systems are equitably distributed among all participants, regardless of their socio-economic background.

3.2 Adopted Research Criteria

The search for academic articles was conducted using the following query across reputable platforms:

("microservices architecture" OR "modular systems" OR "distributed systems") AND ("APIs" OR "application programming interfaces" OR "RESTful APIs") AND ("smart energy systems" OR "energy integration" OR "renewable energy integration")

Platforms Used:

- **Google Scholar**: Widely recognized for its comprehensive coverage of academic articles across disciplines.
- **IEEE Xplore**: A trusted source for high-quality research in technology and engineering.
- **ACM Digital Library**: A leading repository for research in computer science and related fields.

These platforms were selected due to their reliability, academic credibility, and extensive databases covering the thesis's focus areas.

3.3 Inclusion and Exclusion Criteria

To ensure the selection of relevant and high-quality studies, the following inclusion and exclusion criteria were applied:

Inclusion Criteria

1. **Relevance**: Studies must address topics related to microservices architecture, APIs, and their application in smart energy systems;
2. **Publication Type**: Only peer-reviewed journal articles, conference papers, and high-quality academic publications were considered;
3. **Time Frame**: Preference was given to publications from 2013 to 2023 to ensure the research reflects recent developments;
4. **Practical Focus**: Studies offering insights into real-world implementations, case studies, or innovative approaches were included;
5. **Ethics and Standards**: Research discussing ethical considerations like privacy, security, or equitable access was prioritized.

Exclusion Criteria

1. **Irrelevant Topics**: Articles not directly related to microservices, APIs, or energy systems were excluded;
2. **Non-Peer-Reviewed Sources**: Publications such as blog posts, opinion pieces, or non-academic reports were not considered;
3. **Redundancy**: Duplicate studies or articles covering identical content were excluded;
4. **Outdated Works**: Publications prior to 2013 were excluded unless foundational to the topic

3.4 Collected Studies

The following studies were selected based on the predefined search query and criteria. They provide valuable insights into microservices architecture, API design, and smart energy systems, directly contributing to the thesis objectives. Below is the list of studies, each accompanied by a brief summary and its relevance to the project.

1. **"Conceptualization of Blockchain Enabled Interconnected Smart Microgrids"** [9]

   Summary: This study explores the application of blockchain technology to enable interconnected smart microgrids. It proposes a decentralized framework for transactions and database management, focusing on interoperability through inter-blockchain protocols. The framework also integrates prosumer roles and legacy grid systems for enhanced scalability and reliability.

   Relevance: Addresses RQ1 and RQ4 by providing insights into decentralized architectures and scalable solutions for smart microgrids. It also informs RQ2 by detailing frameworks for prosumer transactions.

2. **"A System Architecture for Software-Defined Industrial Internet of Things"** [10]

Summary: This paper presents a novel software-defined IIoT (SD-IIoT) architecture integrating industrial wireless sensor networks, gateways, and cloud services. The architecture ensures interoperability, scalability, and quality of service through technologies such as CoAP and SDN.
Relevance: Directly contributes to RQ1 and RQ4 by proposing scalable and interoperable designs for IIoT-based energy systems.

3. **"Toward a Practical Digital Twin Platform Tailored to the Requirements of Industrial Energy Systems"** [11]
   Summary: This study introduces a practical framework for digital twins in industrial energy systems, addressing barriers to implementation. The proposed platform supports model encapsulation, service engineering, and integration with existing systems to enhance efficiency and flexibility.
   Relevance: Relevant to RQ1 and RQ4 by demonstrating how digital twin technologies can optimize energy systems and improve their scalability.

4. **"Backend Development for E-Commerce Application"** [12]
   Summary: This research focuses on hyper-local e-commerce platforms, proposing a digital solution for community-based economic empowerment. Although not directly related to energy systems, the study emphasizes scalable, localized architectures.
   Relevance: Limited relevance to the RQs, except for potential lessons on localized system scalability.

5. **"Data Analytics and Machine Learning Methods, Techniques, and Tool for Model-Driven Engineering of Smart IoT Services"** [13]
   Summary: This dissertation enhances IoT software engineering with integrated data analytics and machine learning, enabling automated modeling and code generation for smart IoT and energy systems.
   Relevance: Supports RQ3 and RQ4 by detailing ML-based optimizations and predictive capabilities in energy systems.

6. **"DevOps Model Approach for Monitoring Smart Energy Systems"** [14]
   Summary: Proposes a hybrid modeling approach combining systemic and analytical frameworks to monitor and optimize energy systems. The paper integrates DevOps methodologies and microservices for enhanced scalability and governance.
   Relevance: Aligns with RQ1, RQ3, and RQ4 by addressing the need for scalable, modular, and governable energy systems.

7. **"Towards Software-Defined Protection, Automation, and Control in Power Systems: Concepts, State of the Art, and Future Challenges"** [15]
   Summary: This paper reviews software-defined PAC systems for power grids, emphasizing virtualization and interoperability. It highlights challenges in standardization and deployment.
   Relevance: Addresses RQ1 and RQ4 by providing insights into virtualized and scalable protection and control systems for power grids.

8. **"Approach Towards Engineering Microservice-Oriented Composable Ecosystems for Smart Industries"** [16]
   Summary: Explores microservice-based composable ecosystems for Industry 5.0, emphasizing adaptability, security, and scalability. The study demonstrates use cases in energy management and IoT-based systems.
   Relevance: Relevant to RQ1 and RQ4 by demonstrating scalable microservice architectures for energy systems.

9. **"A Cloud-Based Framework for Smart Grid Data, Communication, and Co-Simulation"** [17]
   Summary: Introduces a cloud-based framework integrating data management, communication, and co-simulation for smart grids. The framework leverages RabbitMQ, Apache Kafka, and Kubernetes for scalability and reliability.
   Relevance: Supports RQ1, RQ3, and RQ4 by addressing data integration and scalable communication solutions in smart grids.

10. **"Service-Oriented Architecture as an Enabling Technology for Sector Integration"** [18]
    Summary: Investigates the use of service-oriented architecture (SOA) to enable sector integration in energy systems, with a focus on distributed energy resources (DERs) and market participation.
    Relevance: Contributes to RQ1, RQ2, and RQ4 by exploring how SOA can facilitate system interoperability and market integration for energy systems.

11. **"A Qualitative Study on the United States Internet of Energy: A Step Towards Computational Sustainability"** [19]
    Summary: This paper explores the Internet of Energy (IoE) concept in the United States, emphasizing the integration of IoT, big data, and computational sustainability in energy systems. It discusses grid modernization, renewable energy integration, and peak load forecasting. The study provides qualitative insights into the challenges and opportunities of IoE, including interoperability, reliability, and resilience improvements.
    Relevance: Relevant to RQ1 and RQ4 by addressing the role of computational technologies and IoT in achieving scalable and reliable energy systems. It also contributes to RQ3 by analyzing renewable energy variability and demand-side management.

12. **"Digital Twins of Smart Energy Systems: A Systematic Literature Review on Enablers, Design, Management, and Computational Challenges"** [20]
    Summary: This paper conducts a systematic review of Digital Twin (DT) technologies for smart energy systems (SES). It categorizes architectures into two-layer, three-layer, four-layer, and hyper-layer frameworks and identifies key enabling technologies like IoT and edge-cloud computing. The study also highlights gaps in real-world implementations and suggests a roadmap for future DT research in energy systems.
    Relevance: Contributes to RQ1 and RQ4 by exploring scalable and modular architectures for DT in energy systems. Also supports RQ3 by detailing how DT can optimize energy management and improve SES efficiency.

13. **"SmartCityWare: A Service-Oriented Middleware for Cloud and Fog-Enabled Smart City Services"** [21]
    Summary: This paper presents SmartCityWare, a service-oriented middleware designed for the integration of the Cloud of Things (CoT) and Fog Computing in smart city applications. It emphasizes the middleware's ability to provide low-latency, scalable, and location-aware services. The platform supports applications ranging from smart energy systems to intelligent transportation.
    Relevance: Contributes to RQ1 and RQ4 by addressing challenges in scalability, interoperability, and low-latency service provisioning. Offers additional insights into middleware design, relevant to enhancing API capabilities.

14. **"Energy Management Systems: State of the Art and Emerging Trends"** [1]
    Summary: This article reviews state-of-the-art energy management systems (EMS), highlighting their evolution in the context of smart grids. It discusses features such as disaggregated energy data, real-time monitoring, integration with IoT devices, and advanced analytics for consumer behavior modeling. Emerging trends focus on intelligent automation, mobility, and privacy in energy systems.
    Relevance: Relevant to RQ3 and RQ5 by identifying functionalities required for effective energy management, such as real-time analytics, security, and privacy mechanisms. Also contributes to RQ2 by exploring how EMS supports consumer engagement.

3.5 Data Analysis

RQ1: How can a microservices-based architecture be designed to enable interoperability between diverse smart energy systems?

Answer:
For energy distribution between smart systems, interoperability ensures seamless data exchange and coordinated actions. Microservices provide a modular framework that allows diverse systems to connect without being tightly coupled. Specific strategies include:

1. **Blockchain for Interconnected Microgrids**:
    - The study "Conceptualization of Blockchain Enabled Interconnected Smart Microgrids" demonstrates the use of blockchain-enabled microservices to connect microgrids. Each microservice represents a distinct grid function, such as energy monitoring or peer-to-peer trading, and communicates via secure inter-blockchain protocols. This ensures that a solar microgrid, for instance, can distribute surplus energy to a wind-based system without requiring uniform technologies.
2. **Service-Oriented and Cloud-Enabled Microservices**:
    - The study "A System Architecture for Software-Defined Industrial Internet of Things" highlights how microservices can be deployed at various layers, such as edge gateways and cloud controllers, to facilitate data exchange. For example, energy sensors in a factory can share real-time load data with a cloud-based system managing regional energy distribution.

3. **Middleware as a Bridge for Energy Distribution**:
    - Middleware platforms like "SmartCityWare" integrate microservices to manage communication between urban energy grids and distributed systems. For instance, a middleware microservice can translate communication protocols between legacy energy systems and modern IoT-based grids.

Key Recommendations:

- Use microservices for modularity, with each service dedicated to a specific task like energy metering, load balancing, or trading;
- Leverage blockchain protocols, such as Hyperledger, for secure energy data exchange between distributed microservices;
- Deploy middleware microservices to bridge systems using heterogeneous communication standards, ensuring compatibility.

RQ2: How can the design of APIs support the integration of energy communities, enabling citizens to act as prosumers in energy systems?

Answer:
APIs designed for microservices architectures must empower energy communities by facilitating real-time interaction with the energy system and enabling citizen participation in energy trading and management. Specific solutions include:

1. **Peer-to-Peer Energy Trading APIs**:
    - The blockchain-based APIs described in "Conceptualization of Blockchain Enabled Interconnected Smart Microgrids" allow microservices to handle peer-to-peer transactions autonomously. For example, one microservice validates transactions, while another updates the ledger, ensuring real-time trading transparency.
2. **User-Centric Resource Management APIs**:
    - "Service-Oriented Architecture as an Enabling Technology for Sector Integration" illustrates APIs that enable users to monitor their energy generation, consumption, and storage. A microservice-based API can, for instance, predict when a homeowner's solar panels will generate excess energy and automatically schedule a sale to the local community.
3. **Engagement and Transparency Tools**:
    - APIs highlighted in "Energy Management Systems: State of the Art and Emerging Trends" provide community dashboards where prosumers can view their trading history and energy usage. This fosters trust and encourages participation by offering transparency.

Key Recommendations:

- Develop RESTful APIs to facilitate energy trading, with endpoints for listing available energy, initiating trades, and verifying transactions;

- Integrate predictive analytics into APIs using AI-driven microservices for optimizing energy generation and consumption;
- Provide citizen-facing APIs for monitoring usage and trading activity via intuitive dashboards.

RQ3: What functionalities and services must be supported by the APIs to facilitate peer-to-peer energy trading, load management, and energy storage within smart energy systems?

Answer:
To manage energy distribution effectively, APIs should support the following functionalities:

1. **Energy Resource Allocation Services**:
    - APIs in "A Cloud-Based Framework for Smart Grid Data, Communication, and Co-Simulation" allow microservices to dynamically allocate energy resources based on real-time demand. For example, a microservice can prioritize sending energy to a high-demand zone while reducing supply to areas with low consumption.
2. **Predictive Load Balancing Services**:
    - As shown in "Digital Twins of Smart Energy Systems," APIs linked to Digital Twins can simulate load conditions and suggest optimal storage or distribution actions. For instance, a microservice might forecast peak demand and preemptively route energy to prevent outages.
3. **Distributed Energy Trading Services**:
    - "Conceptualization of Blockchain Enabled Interconnected Smart Microgrids" introduces APIs for energy auctions, enabling microservices to find buyers and sellers dynamically based on real-time supply and demand.

Key Recommendations:

- Design APIs to support dynamic allocation of energy, using algorithms hosted in microservices for real-time decision-making;
- Include APIs for real-time bidding and trading, ensuring fair distribution through decentralized microservices;
- Incorporate load simulation APIs connected to Digital Twins for predictive distribution strategies.

RQ4: How can the APIs ensure the scalability, performance, and security needed to support the integration of international smart energy systems?

Answer:
Scalability, performance, and security are achieved by adopting microservice architectures that use lightweight communication protocols and secure distributed frameworks:

1. **Scalability via Containerized Microservices**:
    - "DevOps Model Approach for Monitoring Smart Energy Systems" demonstrates scaling through Kubernetes, where microservices can replicate under increased

loads. For example, as new energy systems connect, additional instances of load-balancing microservices are deployed automatically.

2. **Performance Optimization with Edge Processing**:
    - Fog-enabled microservices in "SmartCityWare" process time-sensitive energy data locally, reducing latency for energy distribution decisions. For instance, edge microservices can rapidly reroute power during a grid overload.
3. **Blockchain for Security**:
    - In "Conceptualization of Blockchain Enabled Interconnected Smart Microgrids," blockchain APIs secure all interactions between microservices by encrypting transactions. For example, trading microservices use cryptographic signatures to validate energy trades.

Key Recommendations:

- Use orchestration tools like Kubernetes to dynamically scale microservices during peak energy demand;
- Deploy edge-based microservices to handle real-time energy distribution decisions near the source;
- Incorporate blockchain for transaction security and API authentication, ensuring data integrity.

RQ5: What are the best practices for ensuring the ethical use of data within smart energy systems, particularly in terms of privacy and equity in energy access?

Answer:
Ensuring ethical data practices in microservices-based energy systems requires the following:

1. **Privacy Mechanisms**:
    - "Energy Management Systems: State of the Art and Emerging Trends" describes anonymization tools integrated into microservices APIs. For example, an energy provider can use anonymized datasets to calculate grid demand without exposing individual user data.
2. **Equitable Energy Access**:
    - The study "A Qualitative Study on the United States Internet of Energy" discusses APIs that prioritize underserved communities. For example, a subsidy distribution microservice allocates additional resources to low-income households based on pre-set criteria.
3. **Transparency and Trust**:
    - Transparent APIs, as discussed in "Digital Twins of Smart Energy Systems," provide open dashboards where all stakeholders can access real-time energy data. This ensures fairness and prevents monopolistic control over energy distribution.

Key Recommendations:

- Integrate anonymization and data masking tools into microservices to protect user privacy;
- Build APIs that include equity-focused features, such as energy credits for disadvantaged communities;
- Design open APIs with accessible dashboards, ensuring transparency for all stakeholders.

3.6 PRISMA

The PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) methodology, depicted in the image below, provides a well-defined set of guidelines designed to enhance the reporting of systematic reviews and meta-analyses. Initially created for healthcare research, PRISMA is now widely adopted in various fields to ensure transparency, rigor, and reproducibility in literature reviews. It serves as a crucial tool for researchers, enabling them to systematically assess, synthesize, and report on existing studies in a consistent manner [22].

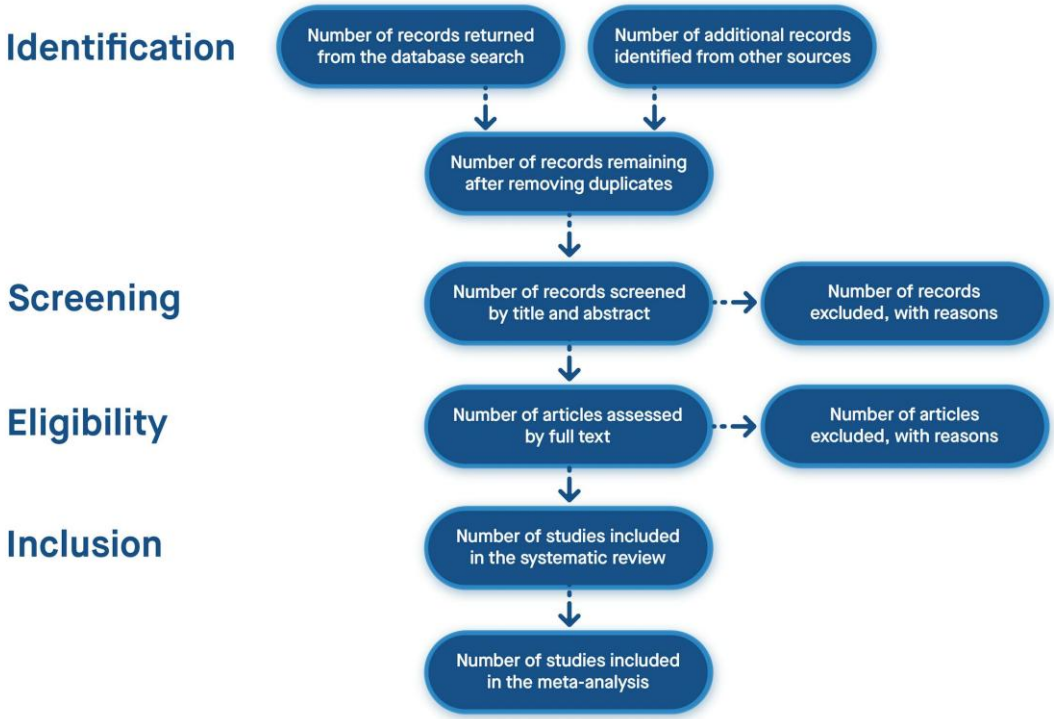

Figure 6 - The PRISMA Method [23]

3.6.1 Key Features

PRISMA offers a set of essential features that guide researchers throughout the systematic review process. These features ensure that reviews are comprehensive, transparent, and reproducible.

- **Checklist of Essential Items**: The PRISMA guidelines include a checklist of 27 items that guide researchers through every stage of the review. These items address key aspects such as:
    - Review objectives;
    - Search strategies;
    - Data extraction processes;
    - Risk of bias assessments;
    - Study selection criteria.
- **Standardized Reporting**: By adhering to PRISMA, researchers can ensure standardized reporting, which provides several benefits:
    - Reproducibility of the review process;
    - Clear and consistent reporting of methods and findings;
    - Enhanced transparency, minimizing the likelihood of bias in the analysis [24].

3.6.2 Application of PRISMA in Software Engineering and Smart Energy Systems

The PRISMA methodology, although initially designed for healthcare, is also highly applicable to fields such as software engineering and energy system integration. In particular, when reviewing topics related to smart energy systems, PRISMA helps structure the review process in a consistent and rigorous manner.

In the context of software engineering and smart energy systems, PRISMA can be used to systematically assess existing studies on topics such as:

- Microservices architectures;
- Energy integration platforms;
- APIs for energy management.

By following PRISMA guidelines, researchers can:

- Systematically identify and assess relevant studies related to microservices and energy systems;
- Synthesize findings from diverse research to uncover challenges and propose solutions for integrating smart energy systems through microservices-based APIs:
- Identify gaps in the existing literature and inform future development efforts for energy system integration across international partnerships [8].

3.6.3 Benefits of Using PRISMA in Research

The PRISMA methodology provides a number of advantages for conducting systematic reviews. In this section, we discuss the main benefits of utilizing PRISMA in research.

- **Comprehensive Mapping of Existing Work**: PRISMA facilitates the identification of research trends and emerging technologies, offering a clearer understanding of the current state of the art in areas such as smart energy systems;
- **Transparency and Bias Reduction**: The structured and transparent reporting encouraged by PRISMA minimizes bias and ensures the credibility of the findings;
- **Informed Decision-Making**: By following PRISMA's structured approach, researchers can make well-informed decisions about the development of microservices-based APIs for smart energy systems, helping to improve integration across multiple stakeholders and regions [22].

3.6.4 Example Use Case in Smart Energy Systems

To illustrate the application of PRISMA in research, this section provides a use case specifically related to smart energy systems. A systematic review on the application of microservices for smart energy systems can benefit from the PRISMA framework in several ways.

Using PRISMA, a systematic review could:

- **Define search criteria** for relevant articles on the use of microservices in energy systems;
- **Establish inclusion and exclusion criteria** for the studies considered in the review;
- **Synthesize findings** to understand the role of microservices in energy system integration;
- **Identify emerging trends** and future research opportunities in the field.

By structuring the review using PRISMA, researchers can ensure a comprehensive and transparent analysis of the challenges and solutions in integrating energy systems through microservices-based APIs [24].

3.7 Microservices Architecture for Smart Energy Systems

The image below presents the Microservices architecture (MSA) [25] is a transformative design paradigm for distributed software systems, offering modularity, scalability, and enhanced adaptability. Within smart energy systems, MSA plays a pivotal role in addressing integration challenges and improving system flexibility. This subchapter delves into the features, applications, challenges, and benefits of MSA in the context of smart energy systems.

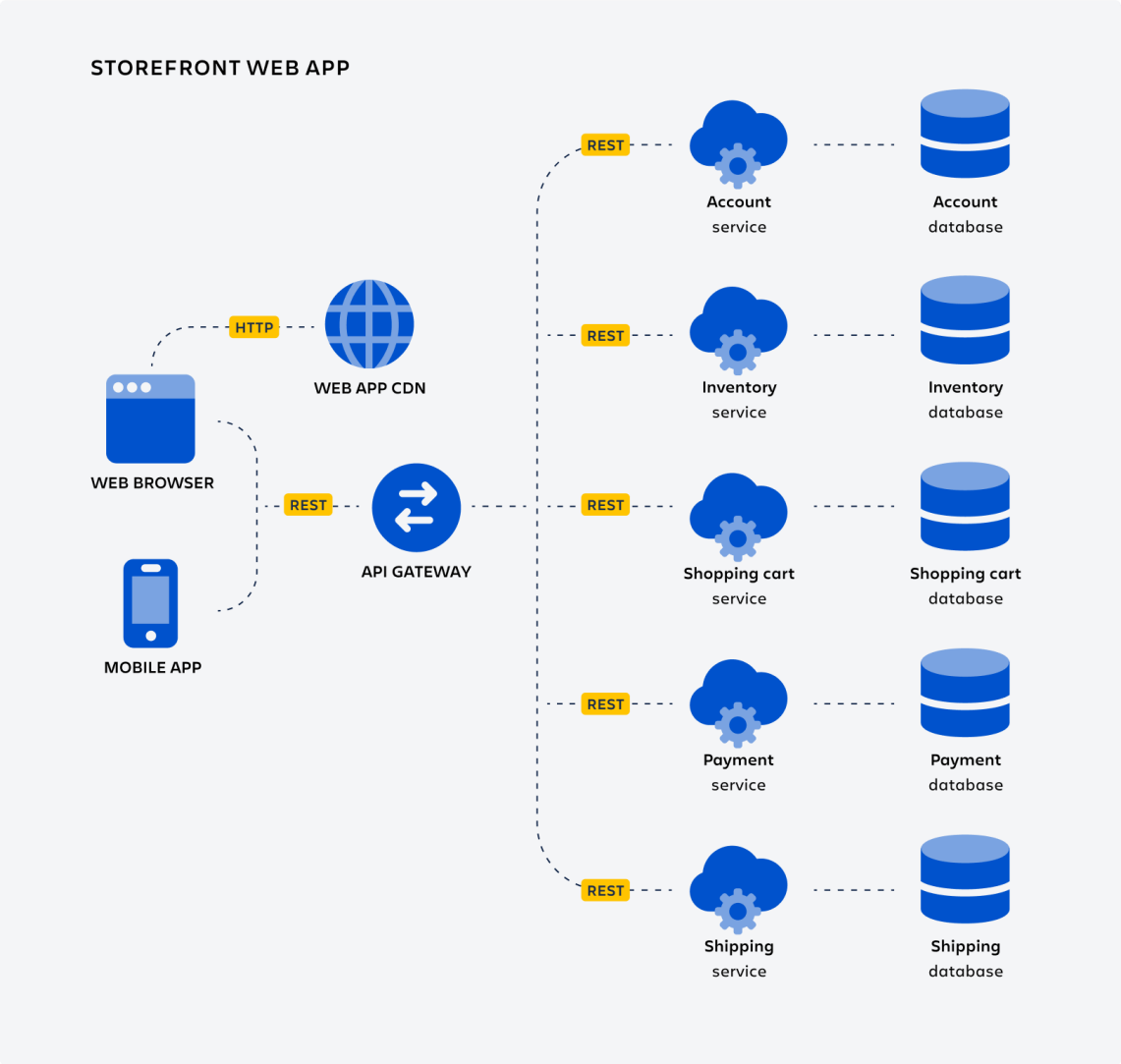

Figure 7 - Microservices Architecture [25]

3.7.1 Key Features

The microservices architecture is a highly adaptable and scalable design pattern that emphasizes modularity and independence among system components. Below are the key features that define a microservices-based architecture, focusing on how these features contribute to system robustness, flexibility, and efficiency, taking into account the example shown in Figure 7.

Modularity and Independence

- **Feature**: Each service in a microservices architecture is a self-contained module that performs a specific business function.
- **Description**:
    - Services like the Account Service, Inventory Service, and Payment Service are designed to operate independently;
    - This modular approach ensures that a failure or change in one service (e.g., the Inventory Service) does not propagate and disrupt other services.
- **Benefit**: Enhances maintainability and flexibility, allowing teams to develop, test, and deploy individual services without impacting the entire system.

API-Based Communication

- **Feature**: Microservices communicate with each other via lightweight APIs, typically RESTful APIs.
- **Description**:
    - Each service exposes its functionality through standardized API endpoints;
    - For example, the Account Service might have an API for user authentication, while the Inventory Service provides endpoints for checking stock availability.
- **Benefit**: Promotes seamless interoperability between services and enables integration with external systems or client applications.

Database Per Service

- **Feature**: Each microservice has its own dedicated database.
- **Description**:
    - For example, the Account Service only accesses the Account Database, while the Payment Service interacts with the Payment Database;
    - This architecture enforces data isolation, preventing direct dependencies between services.
- **Benefit**: Improves scalability, ensures data integrity, and allows services to use databases best suited to their specific needs (e.g., SQL, NoSQL).

Centralized API Gateway

- **Feature**: The architecture includes a central API Gateway as the single entry point for all client requests.
- **Description**:
    - The API Gateway routes incoming requests to the appropriate microservice, manages authentication and authorization, and provides caching for repeated queries;
    - For instance, a mobile app user accessing their shopping cart interacts with the Shopping Cart Service through the API Gateway.
- **Benefit**: Simplifies client-side communication by abstracting the complexity of managing multiple microservices.

Scalability

- **Feature**: Each microservice can scale independently based on demand.
- **Description**:
    - During high-traffic periods, services like the Payment Service or Inventory Service can be scaled horizontally (e.g., adding instances) without affecting others.
- **Benefit**: Optimizes resource usage and ensures system performance even under heavy load.

Fault Isolation

- **Feature**: The architecture is resilient to failures because services are isolated.
- **Description**:
    - If the Shipping Service experiences a failure, other services like the Account Service or Payment Service continue functioning normally.
- **Benefit**: Minimizes downtime and ensures high availability for users.

3.7.2 Applications in Smart Energy Systems

Microservices architecture supports a range of applications in smart energy systems by enhancing integration and efficiency:

- **Energy Monitoring**: Real-time energy consumption monitoring, facilitated by IoT devices, enables predictive modeling and analytics [26];
- **Renewable Energy Integration**: MSA improves load balancing and adaptive energy distribution, integrating renewables seamlessly into power grids [4], [26];
- **Energy Trading Platforms**: Prosumers engage in peer-to-peer energy trading through microservices-based systems that ensure scalability, security, and efficiency [4];

- **Building Energy Management Systems (BEMS)**: Distributed systems optimize energy use in smart buildings through real-time analytics and adaptive controls [26].

3.7.3 Challenges and Mitigation Strategies

Implementing MSA poses specific challenges, which can be mitigated with appropriate strategies:

- **Service Coordination Complexity**: Tools like Kubernetes and service meshes handle dependencies and orchestrate communication between services [4];
- **Performance Overheads**: The distributed nature of MSA may cause latency issues, mitigated through caching and efficient protocols like gRPC [26];
- **Security and Resilience**: Robust authentication and fault-tolerance mechanisms are essential for secure communication and system reliability [26].

3.7.4 Benefits for Smart Energy Systems

The adoption of microservices in smart energy systems brings a range of transformative benefits that address critical challenges in energy integration and management. These benefits highlight the architecture's potential to improve system interoperability, foster technological innovation, and enhance efficiency. Below, we explore the key advantages that make microservices a valuable approach in this domain.

- **Enhanced Interoperability**: MSA ensures seamless integration across diverse energy platforms and fosters collaboration across international energy networks [4];
- **Support for Innovation**: By accommodating advanced technologies like AI and blockchain, MSA expands the functional capabilities of energy systems [26];
- **Optimized Efficiency**: Granular control enabled by MSA reduces energy waste and improves grid performance [5].

3.7.5 Example Use Case

One practical example involves transitioning legacy power grids to microservices-based architectures. This shift supports real-time monitoring, predictive maintenance, and adaptive energy distribution, significantly reducing operational costs while enhancing scalability and flexibility [4].

3.8 Monolythic Architecture for Smart Energy Systems

The monolithic architecture [27], presented in the image below, is a traditional design paradigm in which an application is built as a single, unified unit. While this approach contrasts with the flexibility of microservices, it has been the backbone of many legacy systems and remains relevant in specific scenarios. This subchapter examines the key features, applications, challenges, and benefits of monolithic architectures, specifically in the context of smart energy systems, offering a comparative view of their strengths and limitations.

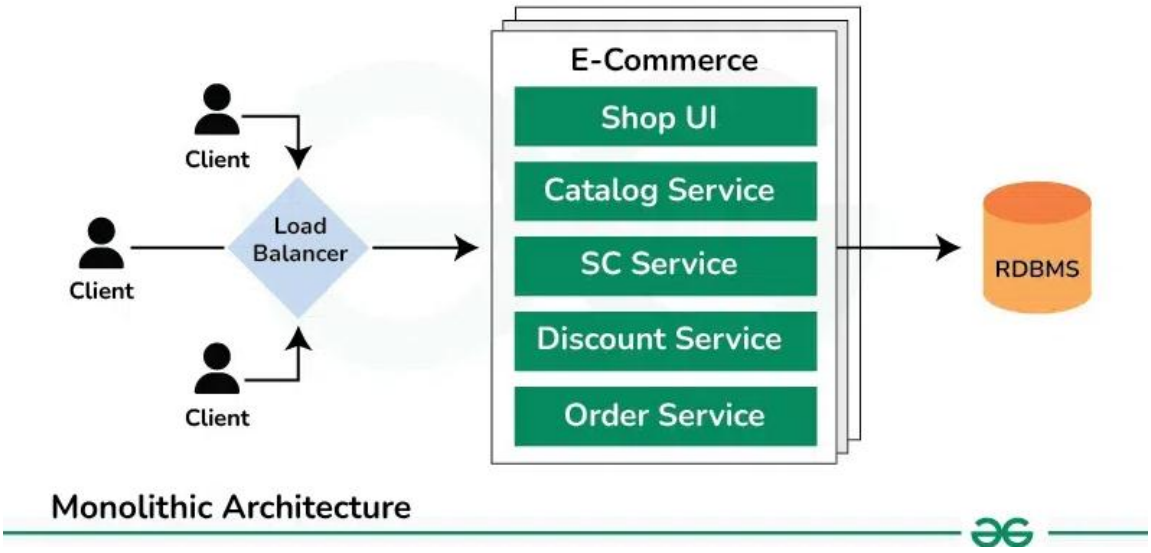

Figure 8 - Monolithic Architecture [27]

3.8.1 Key Features

Monolithic architectures are defined by their tightly coupled components and centralized nature. Below are the core characteristics:

Unified Application

- **Feature**: All components of the application - such as the user interface, business logic, and data access - are contained within a single codebase.
- **Description**:
    - This structure simplifies development and deployment as the entire system is built and maintained as one cohesive unit;
    - In the image, the Shop UI, Catalog Service, SC (Shopping Cart) Service, Discount Service, and Order Service are tightly integrated within the same application stack.
- **Benefit**: Ensures straightforward development, testing, and deployment for smaller systems.

Centralized Data Management

- **Feature**: A single relational database (RDBMS) is used to manage all data operations across the system.
- **Description**:
    - In monolithic architecture, all services directly interact with the same database, ensuring consistency but creating a single point of failure.
    - As depicted in the image, the RDBMS handles data for all services, such as catalog information, shopping cart contents, and order processing.
- **Benefit**: Simplifies data management and ensures consistency across components.

Single Deployment Unit

- **Feature**: The application is deployed as a single unit, regardless of its size or complexity.
- **Description**:
    - All components are packaged and deployed together. Any update, whether to a small feature or a major service, requires redeploying the entire application;
    - In the image, all services (Shop UI, Catalog Service, etc.) form a single deployment, tightly coupled with the database.
- **Benefit**: Reduces deployment complexity for small-scale systems.

Tight Coupling

- **Feature**: Components are tightly integrated, with direct interdependencies.
- **Description**:
    - Changes in one component can have cascading effects on others, requiring careful coordination during updates;
    - For example, in the image, if a change is made to the Catalog Service, it might impact how the Shopping Cart Service interacts with the database.
- **Benefit**: Ensures seamless communication between components but limits flexibility.

Load Balancing

- **Feature**: A single load balancer is used to distribute incoming traffic across the application.
- **Description**:
    - All client requests pass through a centralized load balancer, which routes traffic to the monolithic application;
    - As shown in the image, the Load Balancer directs requests from multiple clients to the unified application stack.

- **Benefit**: Simplifies traffic distribution but may become a bottleneck in high-load scenarios.

Scalability Challenges

- **Feature**: The application scales as a whole, even if only specific components require additional resources.
- **Description**:
    - Horizontal scaling is limited, as the entire application must be replicated to handle increased traffic;
    - For example, if the Order Service experiences high demand, the entire stack, including unused components like the Discount Service, must be scaled.
- **Benefit**: Suitable for predictable, low-demand environments but inefficient for dynamic or large-scale applications.

Simple Development and Debugging

- **Feature**: A single codebase makes development and debugging more straightforward for small teams or applications.
- **Description**:
    - Developers can work on a unified system without managing inter-service communications or complex configurations;
    - The image reflects this simplicity, as all components are part of one cohesive application.
- **Benefit**: Easier onboarding for developers and faster development cycles in smaller teams.

3.8.2 Applications in Smart Energy Systems

Monolithic architectures have been successfully implemented in various scenarios within the energy domain:

- **Legacy Systems**:
    - Many traditional energy systems rely on monolithic designs due to their simplicity;
    - For instance, older grid control systems and billing platforms often follow this model.
- **Centralized Grid Management**:
    - Centralized grids, where data flow is predictable and controlled, benefit from monolithic architectures;
    - For example, a utility company managing a single power plant might use a monolithic system for real-time monitoring and control.

- **Small-Scale Deployments**:
    - In localized contexts, such as a single building or microgrid, monolithic architectures can meet operational needs effectively;
    - These environments do not demand the scalability and modularity of microservices.

3.8.3 Challenges and Limitations

Despite their simplicity, monolithic architectures face several challenges that make them less suitable for modern, distributed energy systems:

- **Scalability Issues**:
    - Scaling requires replicating the entire application, even if only one component (e.g., load balancing) is under heavy demand;
    - For instance, during peak usage, scaling the load management function forces the entire system to consume more resources.
- **Limited Flexibility**:
    - Tightly coupled components make it challenging to implement updates or changes without risking system-wide errors;
    - For example, modifying the billing system might inadvertently impact the energy monitoring module.
- **Performance Bottlenecks**:
    - Centralized processing and data management can become bottlenecks during high loads;
    - A failure in one component, such as the data storage layer, can halt the entire system.
- **Difficult Maintenance**:
    - As the system grows, maintaining and debugging a single, large codebase becomes increasingly complex;
    - Dependencies between components can lead to cascading errors.

3.8.4 Benefits for Smart Energy Systems

Despite their limitations, monolithic architectures offer advantages that make them viable for certain use cases:

- **Simpler Development and Testing**:
    - The unified nature of the system allows for streamlined development, particularly for smaller or less dynamic applications.
- **Centralized Control**:
    - A single deployment and database provide a clear overview of the system, making it suitable for environments with minimal variability.
- **Lower Initial Costs**:
    - Development and operational costs are typically lower compared to distributed systems, making monolithic architectures ideal for startups or projects with limited resources.

3.9 Microservices vs Monolythic Architecture

In modern software development, choosing between microservices architecture and monolithic architecture [28] is a critical decision that impacts scalability, flexibility, and maintainability. Both architectures have their strengths and weaknesses, making them suitable for different scenarios. This chapter explores the key differences between these two architectural styles, with a specific focus on their applicability to smart energy systems, as relevant to this project's theme.

Table 2 provides a detailed comparison of the two architectures based on critical aspects, including their impact on this project.

Table 2 - Microservices vs Monolithic Architecture Comparison for Microservices-Based Energy Systems

Aspect	Microservices Architecture	Monolithic Architecture	Relevance to This Project
Structure	Modular services, each responsible for a specific function (e.g., load management, billing).	Unified codebase where all components are tightly coupled.	Modular services enable better management of smart energy components like peer-to-peer trading and load balancing.
Scalability	Independent scalability; each service can scale separately based on demand.	Entire application must scale as a single unit, leading to resource inefficiencies.	Microservices allow specific scaling, such as for energy trading during peak hours, without impacting other services.
Fault Isolation	Failure in one service (e.g., billing) does not affect others.	Failure in one component can disrupt the entire system.	Fault isolation ensures uninterrupted operation of critical energy functions, such as load management.
Development Flexibility	Teams can use different programming languages or technologies for different services.	All components must use the same technology stack.	Flexibility to use optimized tools (e.g., AI-based services for forecasting).
Deployment	Services are deployed independently, enabling faster updates and releases.	Entire application must be redeployed, even for minor changes.	Independent deployments reduce downtime when upgrading specific energy system functionalities.

Performance	High performance under load due to independent service scaling.	May suffer from bottlenecks as all components share resources.	Better suited for energy systems that experience fluctuating demand, such as renewable energy integration.
Complexity	More complex to develop and manage due to distributed services.	Simpler to develop and manage, especially for small systems.	Microservices complexity is justified for managing distributed energy systems across large grids.
Data Management	Each service has its own database, ensuring data isolation and flexibility.	A single database for the entire application, which simplifies management but creates a bottleneck.	Independent databases allow better resource optimization, e.g., using NoSQL for energy consumption and SQL for billing.
Cost	Higher initial cost due to the complexity of setting up and managing services.	Lower initial cost due to simpler architecture.	Initial setup costs for microservices are justified for long-term benefits in large-scale smart energy systems.
Use Case Suitability	Ideal for large-scale, distributed systems that require high scalability and flexibility.	Suitable for small-scale systems with predictable workloads.	Microservices are better suited for managing dynamic, large-scale smart energy systems.

3.10 Technologies

The development of microservices-based APIs for smart energy systems relies heavily on the selection of appropriate tools, frameworks, and programming languages. These technologies ensure scalability, flexibility, and the ability to handle real-time energy data efficiently. This chapter provides an overview of the main programming languages, frameworks, and tools relevant to implementing a robust microservices architecture for smart energy systems, with a focus on their roles and benefits.

3.10.1 Programming Languages

Programming languages are the foundation of any software system. For a microservices-based architecture in smart energy systems, the choice of language depends on the specific role within the system. Typically, languages are categorized based on whether they are used for backend development, managing core services and APIs, or frontend development, managing user interfaces. Below is a breakdown of the key programming languages suitable for this project.

3.10.1.1 Backend

Backend languages power the logic, data handling, and communication between services in a microservices architecture. They ensure that core functionalities operate seamlessly and can handle the system's demands.

Java [29]:

- **Overview**: A widely used language for enterprise-grade applications due to its robustness and scalability;
- **Frameworks**: Works seamlessly with Spring Boot for developing production-ready microservices;
- **Use Case**: Ideal for handling modular services such as billing, energy load management, and user management.

C# [30]:

- **Overview**: A powerful and versatile language commonly used in enterprise and Windows-based environments;
- **Frameworks**: Paired with ASP.NET Core for building scalable, cross-platform APIs;
- **Use Case**: Suitable for energy data processing systems, IoT device integration, and grid control applications.

Python [31]:

- **Overview**: Known for its simplicity and versatility, Python excels in data-centric applications;
- **Frameworks**: Commonly paired with Flask or FastAPI for lightweight and efficient API development;
- **Use Case**: Used for energy forecasting, machine learning integration, and data analytics in smart grids.

Go (Golang) [32]:

- **Overview**: A high-performance language designed for concurrent and scalable applications;
- **Use Case**: Perfect for real-time monitoring, load balancing, and low-latency energy system operations.

3.10.1.2 Frontend

Frontend languages enable the creation of user-facing interfaces that allow energy system users to monitor, analyze, and interact with the system in real time. They are crucial for delivering an engaging and intuitive user experience.

JavaScript [33]:

- **Overview**: A core language for web development, essential for building dynamic, interactive interfaces;
- **Frameworks**: Works with tools like React.js, Vue.js, and Angular for modern frontend development;
- **Use Case**: Ideal for creating real-time energy dashboards and interactive visualizations of energy consumption.

TypeScript [34]:

- **Overview**: A statically typed superset of JavaScript, providing better code maintainability and error checking;
- **Frameworks**: Fully compatible with frameworks like React and Angular;
- **Use Case**: Best suited for complex frontend applications, such as managing peer-to-peer energy trading platforms.

3.10.1.3 Comparison

Backend

Table 3 - Backend Programming Languages Comparison for Microservices-Based Energy Systems

Criteria	Java	C#	Python	Go (Golang)	Best Choice
Performance	High performance, suitable for enterprise applications.	High performance, especially in enterprise and IoT scenarios.	Moderate performance, better for data-heavy operations.	Exceptional performance, optimized for concurrency.	Go (Golang) for performance-critical tasks.
Ease of Use	Moderate; requires familiarity with enterprise frameworks like Spring Boot.	Moderate; designed for enterprise developers familiar with .NET.	High; simple syntax, great for rapid prototyping.	Moderate; straightforward but less beginner-friendly.	Python for ease of use.
Scalability	Excellent; widely used for large-scale applications.	Good; scalable within the .NET ecosystem.	Good; suited for scaling analytics and machine learning services.	Excellent; highly concurrent and scalable.	Java for enterprise scalability.
Integration	Integrates well with enterprise frameworks like Spring Boot.	Strong integration with Microsoft and IoT systems.	Integrates easily with data analytics and ML libraries.	Limited ecosystem compared to Java/Python.	Java for enterprise and modularity.

Best Choice for the Project:

Java is the best choice for this project due to its robust frameworks, particularly Spring Boot, which simplifies the development of scalable and modular microservices. Its ability to handle large-scale, distributed systems makes it ideal for managing the complex demands of a smart energy system, such as billing, load balancing, and real-time monitoring. Java's scalability ensures that the architecture can grow seamlessly to accommodate more users, devices, or geographical areas, while its performance optimization and extensive ecosystem of tools ensure reliable, efficient, and secure operations, perfectly aligning with the project's requirements.

Frontend

Table 4 - Frontend Programming Languages Comparison for Microservices-Based Energy Systems

Criteria	JavaScript	TypeScript	Best Choice
Performance	Good; optimized for web interactions and dashboards.	Good; similar to JavaScript but with type safety.	JavaScript for web dashboards.
Ease of Use	High; widely supported and familiar to developers.	Moderate; additional learning curve due to static typing.	JavaScript for simplicity.
Complexity Handling	Moderate; can become difficult to manage in larger codebases.	High; static typing reduces bugs in large projects.	TypeScript for complex systems.
Relevance	Essential for building interactive, real-time energy dashboards.	Crucial for building scalable, maintainable frontend systems.	JavaScript/TypeScript for dashboards.

Best Choice for the Project:

JavaScript is the best choice for this project due to its versatility and extensive use in developing dynamic, real-time user interfaces. With frameworks like React and Node.js, JavaScript excels at creating responsive dashboards and asynchronous APIs, making it ideal for managing energy data and interactions in smart energy systems. Its non-blocking, event-driven architecture ensures high performance for real-time energy monitoring, while its widespread adoption and compatibility across platforms allow seamless integration with IoT devices and backend services. These qualities make JavaScript a powerful tool for delivering intuitive, interactive user experiences that align perfectly with the project's goals.

3.10.2 Frameworks

Frameworks are essential in modern software development, providing the tools and abstractions needed to streamline the development of microservices and APIs. They simplify complex tasks such as handling requests, managing databases, and integrating security, allowing developers to focus on business logic. For a microservices-based architecture in smart energy systems, the following frameworks represent the most widely used and effective solutions.

3.10.2.1 Backend

Spring Boot (Java) [35]:

- **Overview**:
  - A robust framework for creating standalone, production-ready microservices with minimal configuration;
  - Built on top of the Spring framework, it simplifies dependency management and service integration.
- **Features**:
  - Provides built-in support for RESTful APIs, database connectivity, and security;
  - Includes tools for monitoring, logging, and scaling.
- **Use Case**:
  - Suitable for developing modular energy management services, such as billing systems, grid load balancing, or user authentication.

ASP.NET Core (C#) [36]:

- **Overview**:
  - A cross-platform, open-source framework for building modern, cloud-based, and internet-connected applications;
  - Part of the Microsoft ecosystem, it integrates well with Windows-based environments.
- **Features**:
  - Built-in support for dependency injection, middleware, and RESTful APIs;
  - Offers robust tools for security and performance monitoring.
- **Use Case**:
  - Suitable for enterprise-grade energy systems that require integration with legacy or Microsoft-based technologies.

3.10.2.2 Frontend

React (JavaScript/TypeScript) [37]:

- **Overview**:
  - A JavaScript library for building user interfaces, maintained by Facebook;

    - Focuses on creating reusable UI components, making it highly efficient for frontend development.

- **Use Case**:
    - Building real-time energy dashboards or user portals for monitoring and controlling energy systems.

Angular (JavaScript/TypeScript) [38]:

- **Overview**:
    - A comprehensive TypeScript-based framework maintained by Google, designed for building dynamic single-page applications (SPAs);
    - Employs a component-based architecture, similar to React, but with additional built-in features like dependency injection and reactive programming support.
- **Use Case**:
    - Best suited for applications with extensive logic and a need for structured, maintainable codebases, such as managing energy usage, billing, and peer-to-peer trading interfaces.

3.10.2.3 Comparison

Backend

Table 5 - Backend Frameworks Comparison for Microservices-Based Energy Systems

Criteria	Spring Boot	ASP.NET Core	Best Choice
Performance	High performance for enterprise applications; optimized for JVM.	High performance, especially on Windows systems; cross-platform support.	ASP.NET Core for Windows-heavy systems.
Ease of Use	Moderate; requires familiarity with Java and Spring ecosystem.	Moderate; requires knowledge of C# and .NET framework.	Depends on Developer Expertise.
Scalability	Highly scalable; supports modular microservices architectures.	Excellent scalability, particularly in cloud environments like Azure.	Both are equally scalable.
Integration	Strong integration with Java-based tools and enterprise frameworks.	Seamless integration with Microsoft tools (e.g., Azure, SQL Server).	Depends on Ecosystem Requirements.
Relevance	Ideal for energy services requiring modularity and wide ecosystem support.	Suitable for systems using Microsoft-based technologies and IoT devices.	Spring Boot for modularity and openness.

Best Choice for the Project:

Spring Boot is the best choice for this project due to its ability to simplify the development of robust, production-ready microservices. Its modular architecture, combined with built-in tools for RESTful APIs, database integration, and security, makes it particularly suited for managing the complex operations of smart energy systems, such as billing, energy distribution, and user management. Spring Boot's scalability ensures that services can grow independently to meet increased demand, while its extensive ecosystem and compatibility with tools like Kubernetes and Docker make it ideal for deploying and managing distributed energy systems. These capabilities align perfectly with the project's goals for reliability, modularity, and efficiency.

Frontend

Table 6 - Frontend Frameworks Comparison for Microservices-Based Energy Systems

Criteria	React	Angular	Best Choice
Performance	High performance, especially for dynamic and lightweight UIs.	Excellent performance for complex, large-scale SPAs.	React for lighter UIs; Angular for complex systems.
Ease of Use	High; flexible and simple, but requires third-party libraries for many tasks.	Moderate; more opinionated but offers a complete development ecosystem.	React for simplicity.
Development Speed	Faster initial development due to its lightweight nature.	Slower for small projects, but structured for maintainability.	React for faster prototyping.
Two-Way Data Binding	Not natively supported; uses one-way data flow.	Built-in two-way data binding simplifies dynamic updates.	Angular for real-time data updates.
Relevance	Ideal for real-time energy dashboards or user interfaces.	Best for enterprise-level energy management systems with complex logic.	React for dashboards; Angular for complexity.

Best Choice for the Project:

React is the best choice for this project due to its flexibility and efficiency in building dynamic, interactive user interfaces. Its component-based architecture allows for the creation of reusable UI elements, making it ideal for developing scalable energy dashboards and user portals. React's ability to handle real-time updates and asynchronous data flows ensures that users can interact seamlessly with energy data, such as monitoring consumption or participating in peer-to-peer energy trading. Additionally, React's wide adoption and compatibility with libraries and tools for state management (like Redux) make it a reliable and adaptable choice for delivering engaging, user-friendly interfaces that align with the project’s goals.

3.10.3 Communication Between Services

In a microservices architecture, efficient communication between services is critical. Below are the most common methods:

RESTful APIs [39]:

- **Overview**: The most widely used communication protocol, leveraging HTTP methods for interaction:
- **Advantages**: Simplicity, scalability, and language-agnostic implementation;
- **Use Case**: Exchanging data between energy services, such as sending consumption metrics from a Monitoring Service to a Billing Service.

gRPC [40]:

- **Overview**: A high-performance framework using HTTP/2 and protocol buffers (Protobuf) for data serialization;
- **Advantages**: Bi-directional streaming and compact serialization make it faster than REST;
- **Use Case**: Real-time updates between distributed energy nodes, such as dynamic load balancing.

Message Brokers [41]:

- **Examples**: Apache Kafka, RabbitMQ;
- **Overview**: Enable asynchronous communication between services;
- **Advantages**: Decoupling services and ensuring reliable message delivery;
- **Use Case**: Handling events like grid failures or renewable energy surges.

GraphQL [42]:

- **Overview**: Allows services to request precise data, reducing payload size and optimizing queries;
- **Use Case**: Aggregating data for energy system dashboards or peer-to-peer energy trading platforms.

3.10.3.1 Comparison

Table 7 - Comparison of Communication Protocols for Microservices-Based Energy Systems

Criteria	RESTful APIs	GraphQL	gRPC	MQTT	Best Choice
Performance	Good; lightweight and efficient for standard use cases.	Moderate; tailored queries reduce payload size.	Excellent; highly optimized for performance-critical systems.	High; lightweight for IoT messaging.	gRPC for high-performance systems.
Flexibility	High; widely adopted and supports multiple formats.	Excellent; customizable queries provide unmatched control.	Moderate; designed for fixed contracts (Protobuf).	Moderate; limited to IoT scenarios.	GraphQL for tailored queries.
Ease of Implementation	High; simple and familiar to most developers.	Moderate; requires additional schema and tooling setup.	Moderate; requires Protobuf definitions and tools.	High; straightforward for IoT applications.	RESTful for simplicity and adoption.
Relevance	Widely used for API communication in energy systems.	Ideal for data-intensive dashboards or trading platforms.	Best for real-time updates between energy nodes.	Best for IoT-to-cloud communication.	RESTful APIs for universal compatibility.

Best Choice for the Project:

RESTful APIs are the best choice for this project due to their simplicity, flexibility, and widespread adoption for communication between microservices. Leveraging standard HTTP methods, RESTful APIs enable seamless integration across diverse components of the smart energy system, such as energy monitoring, billing, and peer-to-peer trading services. Their stateless nature ensures scalability, allowing each service to function independently, which is critical for handling the dynamic demands of energy systems. Additionally, the lightweight and language-agnostic design of RESTful APIs makes them easy to implement and maintain, aligning perfectly with the project's need for a modular and efficient communication framework.

3.10.4 Databases

Databases play a vital role in managing data for microservices. The choice of database impacts performance, scalability, and data consistency.

Relational Databases:

- **PostgreSQL** [43]: Known for its reliability and ability to handle complex queries;
- **MySQL** [44]: Popular for transactional data, such as energy billing.

NoSQL Databases:

- **MongoDB** [45]: Excellent for unstructured IoT data:
- **Cassandra** [46]: Handles large-scale time-series data, making it ideal for distributed energy systems.

In-Memory Databases:

- **Redis** [47]: Frequently used for caching and real-time data processing;
- **Use Case**: Reducing latency in energy system APIs.

Time-Series Databases:

- **InfluxDB** [48]: Purpose-built for handling high-frequency time-series data like energy consumption;
- **TimescaleDB** [49]: An extension of PostgreSQL, tailored for time-series workloads.

3.10.4.1 Comparison

Table 8 - Database Comparison for Microservices-Based Energy Systems

Criteria	PostgreSQL	MySQL	MongoDB	Redis	Best Choice
Performance	Excellent for complex queries and analytics.	Good; reliable for transactional workloads.	High performance for unstructured data.	Exceptional for real-time, in-memory processing.	PostgreSQL for analytics, Redis for real-time data.
Scalability	Highly scalable with extensive customization options.	Moderate; scales well for structured transactional data.	Excellent for handling IoT data from multiple sources.	Good; primarily used as a caching layer.	MongoDB for IoT data.
Ease of Use	Moderate; requires experience with relational databases.	High; straightforward with robust community support.	Moderate; needs familiarity with NoSQL principles.	High; simple to set up for caching.	PostgreSQL for advanced use cases.
Use Case Fit	Ideal for structured energy consumption and billing data.	Suitable for transactional workloads like energy payments.	Excellent for IoT device data and energy usage metrics.	Best for caching frequently accessed data.	PostgreSQL for structured data.

Best Choice for the Project:

PostgreSQL is the best choice for this project due to its robustness, scalability, and advanced features, making it ideal for managing the structured data required by smart energy systems. Its support for complex queries, transactional integrity, and extensive extensions allows efficient handling of critical operations like billing, user management, and energy usage analytics. PostgreSQL's scalability ensures that it can grow to accommodate the increasing demands of distributed energy systems, while its compatibility with modern tools and frameworks like Spring Boot ensures seamless integration into the project's architecture. These capabilities make PostgreSQL the perfect fit for ensuring reliable and efficient data management in this project.

3.11 Related Studies

3.11.1 Microservice-Based Architecture for an Energy Management System

The study [2] addresses the fragility, poor flexibility, and hardware dependence of traditional energy management systems (EMSs) by proposing a microservice-based architecture. Similarly, this thesis emphasizes the use of a microservices-based architecture for developing APIs tailored for smart energy systems, particularly to ensure modularity, scalability, and ease of integration. Both works align in their shared objectives of promoting efficiency, decentralization, and interoperability within energy systems.

The study demonstrates the viability of microservices in enhancing reliability and scalability through fine-grained decomposition and containerization. This thesis extends this concept to address interoperability challenges, focusing on enabling functionalities like peer-to-peer energy trading, load management, and energy storage optimization. By leveraging APIs, this work aims to foster citizen participation as prosumers within energy communities, a dimension not explicitly addressed in the study.

Key Contributions of the Related Study

1. **Fine-Grained System Decomposition**: The study highlights the benefits of fine-grained decoupling, ensuring each service is independent and highly resilient;
2. **Containerized Microservices**: It employs container and cluster technologies to enhance deployment flexibility and fault tolerance.

Differences in Focus and Implementation

While the study focuses primarily on optimizing traditional EMSs using microservices, this thesis addresses the broader scope of integrating diverse smart energy systems. The differences are outlined below:

1. **System Objectives**:
    - **Study**: Enhances reliability and cost-effectiveness within centralized EMS deployments;
    - **This Thesis**: Prioritizes interoperability and citizen empowerment within decentralized energy ecosystems.
2. **Technological Emphasis**:
    - **Study**: Leverages containerization and MILP (Mixed-Integer Linear Programming) models for resource management;
    - **This Thesis**: Incorporates RESTful APIs and microservices to enable dynamic interactions between prosumers and smart energy systems, focusing on peer-to-peer energy trading and decentralized control.
3. **Scope of Application**:
    - **Study**: Targets traditional power grid operations with containerized EMS;
    - **This Thesis**: Addresses the integration of heterogeneous energy systems, supporting international collaborations and energy communities.

4. **Citizen Participation**:
    - **Study**: Does not explicitly consider citizen engagement;
    - **This Thesis**: Actively involves citizens as prosumers, enabling them to generate, store, and trade energy while fostering equity and privacy.

3.11.2 Design of a Microservices-Based Architecture for Residential Energy Management Systems

The study [50] introduces a microservices-based architecture designed to improve the performance and stability of energy management systems within smart buildings and homes. This complements the thesis's objective of utilizing APIs to foster integration and adaptability within energy ecosystems. Both works align in advocating for modularity, scalability, and data-driven decision-making. However, the study places a stronger emphasis on predictive analytics and real-time monitoring, while this thesis explores broader functionalities such as peer-to-peer trading and decentralized energy storage management.

Key Contributions of the Related Study

1. **Integration of IoT and Mobile Technology**: The study utilizes IoT sensors and mobile applications to monitor and manage energy usage in real-time, enhancing user engagement;
2. **Hybrid Cloud Architecture**: It combines edge and cloud computing to optimize data processing and ensure high performance and reliability;
3. **Predictive Models**: AI-driven predictive analytics are employed to anticipate energy consumption patterns, enabling proactive management;
4. **Layered Microservices Design**: The architecture is structured into functional layers - IoT, processing, network and data storage, infrastructure, and microservices - to streamline operations and improve scalability.

Differences in Focus and Implementation

While the study primarily targets energy efficiency in residential settings, this thesis takes a broader perspective by addressing challenges in decentralized energy systems. Key differences include:

1. **System Objectives**:
    - **Study**: Focuses on optimizing energy efficiency within residential environments;
    - **This Thesis**: Aims to enhance interoperability and enable decentralized energy ecosystems involving diverse stakeholders.
2. **Technological Emphasis**:
    - **Study**: Leverages AI for predictive analytics and hybrid cloud computing for data processing;
    - **This Thesis**: Emphasizes the use of RESTful APIs to integrate diverse energy systems and facilitate new functionalities such as peer-to-peer trading.
3. **Application Scope**:
    - **Study**: Concentrates on residential energy management;

- **This Thesis**: Targets the broader integration of heterogeneous energy systems across various domains.

4. **Citizen Participation**:
    - **Study**: Focuses on improving user interfaces for residential consumers;
    - **This Thesis**: Advocates for empowering citizens as active prosumers within energy ecosystems.

4 Analysis and Solution Design

This chapter presents a comprehensive analysis of the system requirements and the architectural design of the proposed backend platform for smart energy systems. While the initial goal was to implement a fully distributed microservices architecture, the final implementation adopts a microservices-ready architectural style - modular, scalable, and service-oriented - designed to support future decomposition into independent services.

Building on the findings from the literature review and the challenges identified in existing energy platforms, this section outlines the rationale and engineering principles that guided the design of a scalable, interoperable, and citizen-centric backend infrastructure.

The chapter begins with a description of the overall system architecture, focusing on its layered and modular design. Each functional domain - such as energy trading, load management, user roles, and energy storage - is encapsulated in isolated modules with clear boundaries. The system supports service-level abstraction and separation of concerns, laying the groundwork for future microservice extraction if distributed deployment becomes necessary.

Although all components currently run within a single deployment, the codebase adheres to clean architecture and domain-driven design (DDD) principles. This enables independent development, testing, and potential scaling of individual modules. As such, the system is architected to evolve naturally into a true microservices architecture without requiring significant structural changes.

Subsequently, the chapter details the requirements engineering process, separating functional and non-functional requirements. Functional requirements describe the core features the system must provide - such as user management, energy profiling, optimization result handling, and data persistence. Non-functional requirements focus on qualities like performance, scalability, security, maintainability, and system interoperability with external APIs and data standards.

Special attention is given to compliance with industry protocols, data format standards, and best practices in software engineering. These considerations ensure that the system can integrate into broader smart grid ecosystems and adapt to future regulatory and technical changes.

By the end of this chapter, a clear and structured blueprint of the system is established. This architecture balances immediate implementation needs with long-term extensibility, serving as a robust foundation for the subsequent implementation, testing, and potential service decomposition phases.

4.1 System Architecture Design

The backend system for the Energy Communities Platform has been designed following a modular and scalable architecture, structured to be microservices-ready. While the current implementation is deployed as a single service, the architecture follows the principles of clean separation of concerns, well-defined component boundaries, and layered abstraction - making it suitable for future evolution into a distributed microservices-based ecosystem.

4.1.1 Domain Model

To support a well-structured and maintainable backend architecture, the domain model - illustrated in Figure 9 - was designed following Domain-Driven Design (DDD) principles. It organizes the core business logic around well-defined aggregates, each consisting of an aggregate root and relevant value objects. This structure ensures encapsulation, enforces consistency within aggregate boundaries, and promotes a clean separation of concerns. The model captures the essential entities and behaviors of the Energy Communities Platform, aligning with the real-world domain while supporting long-term flexibility and scalability.

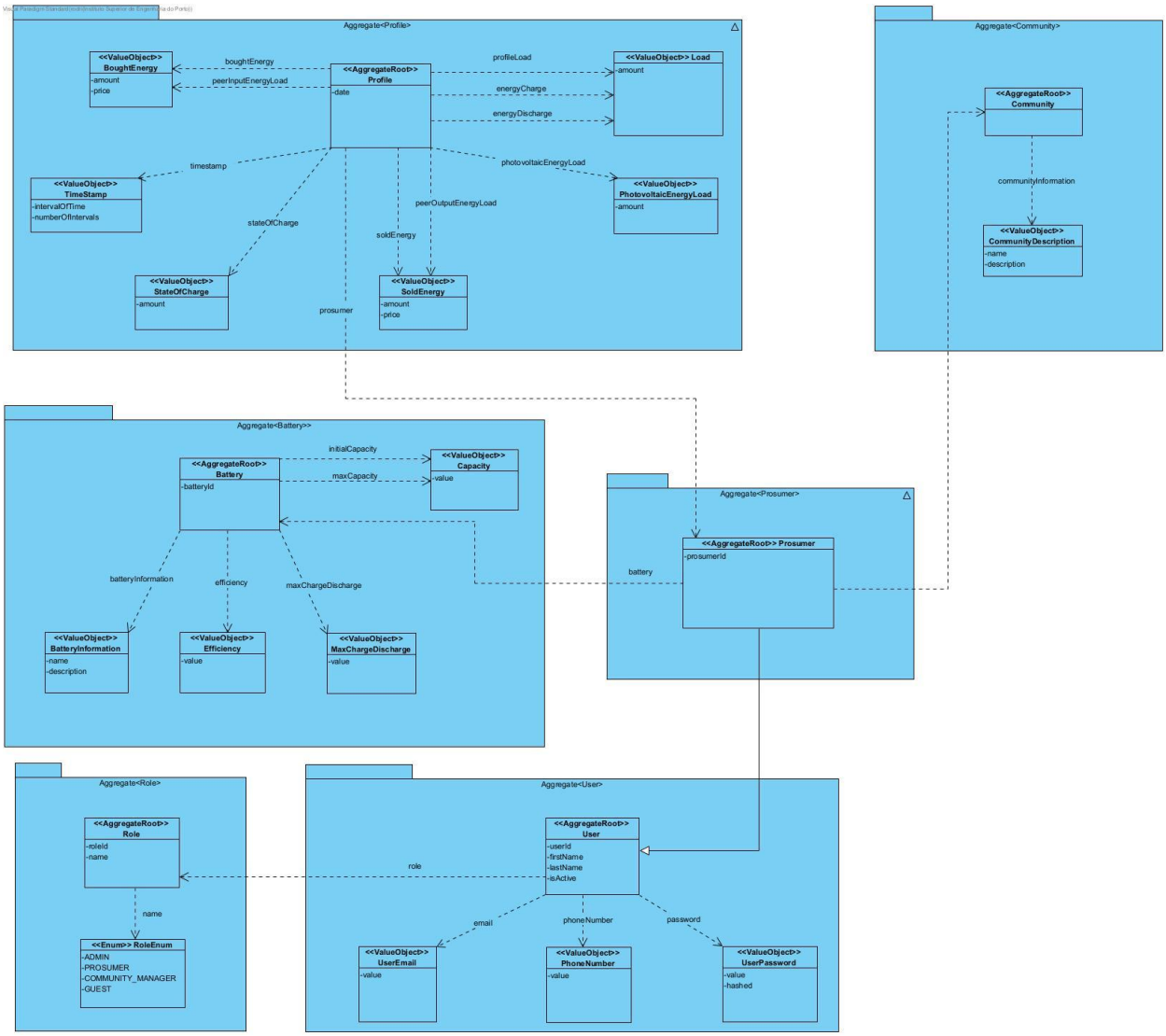

Figure 9 - EnergyCommunitiesPlatform Domain Model

The domain model is composed of several aggregates, each encapsulating a cluster of related domain objects that are treated as a single unit. At the center of each aggregate is an aggregate root, responsible for maintaining consistency and serving as the sole entry point for modifications.

At the center is the User aggregate, which represents a platform user. This aggregate root holds essential attributes such as firstName, lastName, email, phoneNumber, password, and role. It also includes domain-specific value objects like UserEmail, PhoneNumber, and UserPassword to encapsulate validation and behavior related to each concept. The user's role is modeled as a reference to the Role aggregate, which contains a unique identifier and a name drawn from a predefined enumeration of role types (e.g., ADMIN, PROSUMER, COMMUNITY_MANAGER, GUEST).

The Battery aggregate represents a physical battery unit owned by a prosumer. Its root maintains battery-specific characteristics such as batteryInformation, efficiency, maxCapacity, initialCapacity, and maxChargeDischarge. These are modeled as value objects to ensure strict type validation and maintain invariants within the domain.

The Prosumer aggregate links a user to their battery and community, serving as a bridge between user actions and energy-related data. Although communities may include multiple prosumers, the model simplifies this by storing the community reference in each Prosumer. This design reduces coupling, keeps the Community aggregate lightweight, and aligns with DDD principles by maintaining clear, manageable boundaries. The Community itself uses a CommunityDescription value object to encapsulate its name and description.

The Profile aggregate models a prosumer's energy activity during a specific time interval. Each profile instance reflects energy transitions - such as production, consumption, and exchange - within a defined time period. This temporal dimension is represented by the TimeStamp value object, which contains two key attributes: intervalOfTime, usually fixed at 15 minutes, and numberOfIntervals, usually ranging from 1 to 96, corresponding to sequential 15-minute slots across a 24-hour day. The Profile also includes value objects like Load, StateOfCharge, PhotovoltaicEnergyLoad, SoldEnergy, and BoughtEnergy, capturing the detailed energy state of the prosumer at each interval.

Altogether, this model reflects real-world relationships while adhering to Domain-Driven Design (DDD) principles. Aggregates enforce consistency boundaries, value objects encapsulate domain rules, and the structure aligns tightly with the platform’s core functionality and future scalability goals.

4.1.2 Logical View Level 2

The Level 2 Logical View provides a high-level structural overview of the Energy Communities Platform system, as illustrated in Figure 10. This diagram depicts the main containers that comprise the platform, the roles they fulfill, and the relationships between them. It offers a broader perspective than the component-level breakdown, focusing on how the system is organized into independently deployable and functional units.

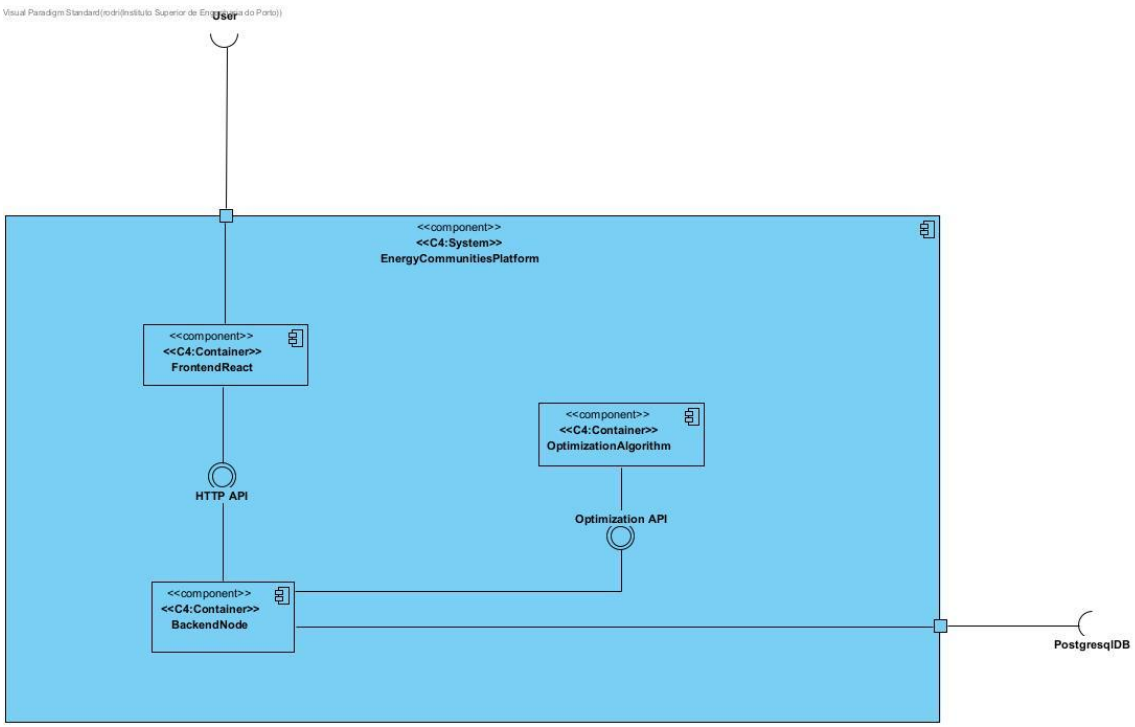

Figure 10 - Logical View Level 2

At the center of the system is the EnergyCommunitiesPlatform, which consists of two primary containers:

- **FrontendReact**: A React-based application responsible for rendering the user interface and enabling user interaction with the platform. It communicates with the backend through HTTP-based API calls, allowing users to perform operations such as authentication, user management, and energy-related actions.
- **BackendNode**: A Node.js server application that encapsulates the platform's core business logic. It handles HTTP requests, manages data persistence, and exposes a RESTful API consumed by the frontend. Internally, it adheres to clean architecture and Domain-Driven Design (DDD) principles to ensure modularity and scalability. It also communicates with the optimization algorithm over HTTP, invoking its REST API to process and distribute energy data as needed.
- **OptimizationAlgorithm**: A dedicated Python-based service, originally developed by a GECAD researcher, responsible for processing uploaded community energy data and executing the energy distribution logic. When a community manager uploads energy files, the backend invokes this component to determine how excess or deficit energy should be distributed among prosumers. If there's an energy shortage, the algorithm

assigns purchases from the external market; in the case of surplus, it handles virtual sales to the grid.

Communication between containers occurs over HTTP using well-defined API endpoints. The backend acts as the central coordinator, receiving client requests, invoking the optimization service, and managing persistent data.

On the persistence layer, the PostgreSQLDB container serves as the central relational database. It stores structured application data and is accessed by the backend through the Prisma Client - a type-safe ORM that translates high-level operations into PostgreSQL queries over TCP/IP.

The diagram also includes an external actor labeled User, representing platform users who interact with the system through the frontend interface.

4.1.3 Logical View Level 3

The Level 3 Logical View offers a detailed look into the internal structure of the BackendNode container, breaking it down into its main components and illustrating how they interact to fulfill backend responsibilities. As shown in Figure 11, this view follows the principles of clean architecture and separation of concerns, which support modularity, maintainability, and scalability of the backend application.

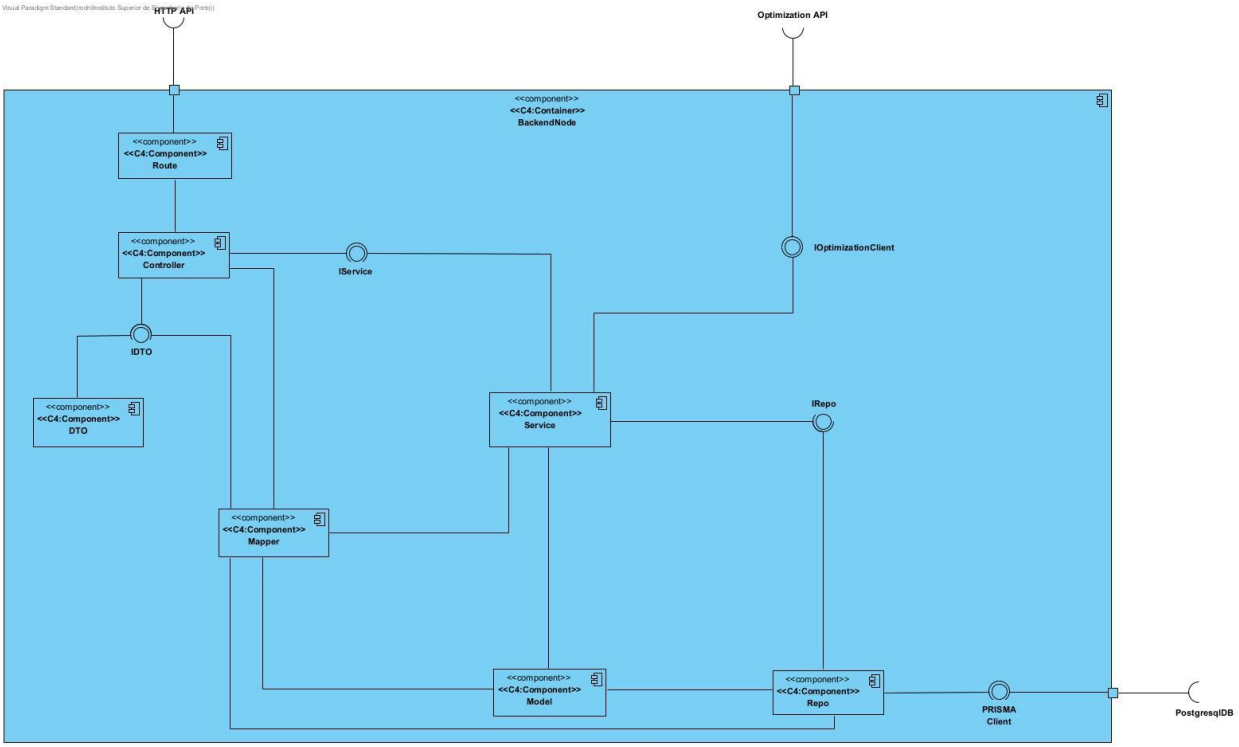

Figure 11 - Logical View Level 3

At the entry point of the system, the Route component is responsible for registering and organizing the HTTP API endpoints exposed to external clients. It defines the structure of the API by mapping HTTP methods and URL paths to corresponding controller functions. Middleware such as authentication and input validation is also applied at this level to ensure consistent preprocessing of incoming requests.

Once a request reaches a defined endpoint, the Controller component handles parsing, response formatting, and delegates execution to the business logic layer via the IService interface. This design aligns with the Separation of Concerns principle by keeping the controller free of business logic.

The Service component contains the core application logic. It coordinates operations such as invoking domain rules in the Model, transforming data through the Mapper, and persisting changes using the IRepo interface. When energy optimization is required - such as after a community energy file is uploaded - the Service layer communicates with the external Optimization API via the IOptimizationClient interface. This abstraction decouples the service from the implementation details of the external Python-based optimization engine, maintaining modularity.

To support clean data flow between layers, the system includes DTO and Mapper components. DTOs define the format of data exchanged with clients, while Mappers convert between DTOs and domain models, promoting consistency and minimizing coupling.

The Model component encapsulates domain entities and enforces business rules. These entities are managed by the Service and persisted by the Repo component, which implements the IRepo interface.

Finally, data persistence is handled through Prisma Client, a type-safe ORM that translates domain queries into SQL and communicates with the external PostgreSQLDB. This setup ensures efficient and reliable access to structured data.

4.1.4 Physical View Level 2

During the early stages of development, the Energy Communities Platform was deployed in a simplified local environment. As illustrated in Figure 12, all core components were hosted and executed on the same physical machine (localhost), allowing the development team to streamline testing, debugging, and feature iteration without the overhead of remote deployment or networking constraints.

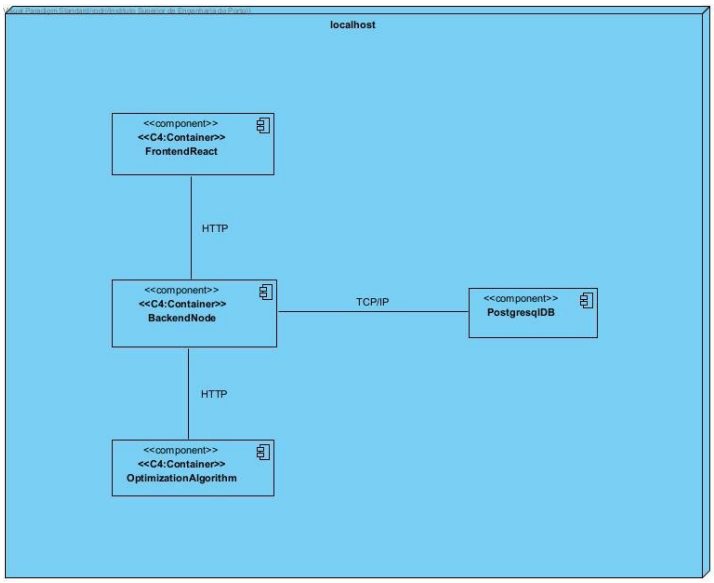

Figure 12 - Physical View Level 2

In this setup, the FrontendReact component is served locally in the user's browser via a development server on port 5137, allowing interaction with the platform through a graphical interface. It communicates with the BackendNode API server running on port 4000 over HTTP. The backend is responsible for handling all application logic, including user management, data processing, and orchestration.

To persist and retrieve data, the backend connects to a locally hosted PostgreSQLDB using TCP/IP on the default port 5432. Additionally, when energy optimization is required (e.g., after a community file upload), the backend makes HTTP requests to the OptimizationAlgorithm service - developed in Python - running on port 8000. This component exposes a REST API that processes community energy data and returns optimized energy distributions.

This localized and centralized architecture streamlines development by minimizing environmental complexity and latency, enabling faster iteration and easier debugging. However, it is not intended for production, as it lacks scalability, security isolation, and proper infrastructure management. The system is designed to evolve toward a target deployment architecture where each major component - frontend, backend, optimization service, and database - is deployed independently, as presented in the next subchapter.

4.1.4.1 Target Physical View Level 2

Figure 13 illustrates the target physical deployment of the Energy Communities Platform. In this envisioned production environment, the FrontendReact component runs directly in the user's web browser as a static client-side application. It interacts with the backend via HTTP requests to perform authentication, user management, and energy-related operations.

The BackendNode component, hosted on the GECAD Server, handles all business logic, API processing, and communication with the platform's other services. Also deployed within the same server is the OptimizationAlgorithm, which exposes an HTTP API consumed by the backend to process uploaded energy data and perform automated energy distribution between prosumers.

Hosting both the backend and optimization components on the same physical server (GECAD) simplifies deployment, minimizes network latency between services, and reduces operational overhead - especially during early or intermediate stages of development. Since both components belong to the same internal domain and are managed by the GECAD team, there is no need to isolate them across separate infrastructure units unless scaling or load-balancing requirements arise.

For data persistence, the PostgreSQL database is hosted on a Supabase Server, accessed by the backend via the TCP/IP protocol. This separation of the database into its own managed environment ensures greater security, maintainability, and scalability of data services.

This deployment strategy balances modularity with operational efficiency, preparing the system for future scaling while maintaining simplicity and performance in the current stage.

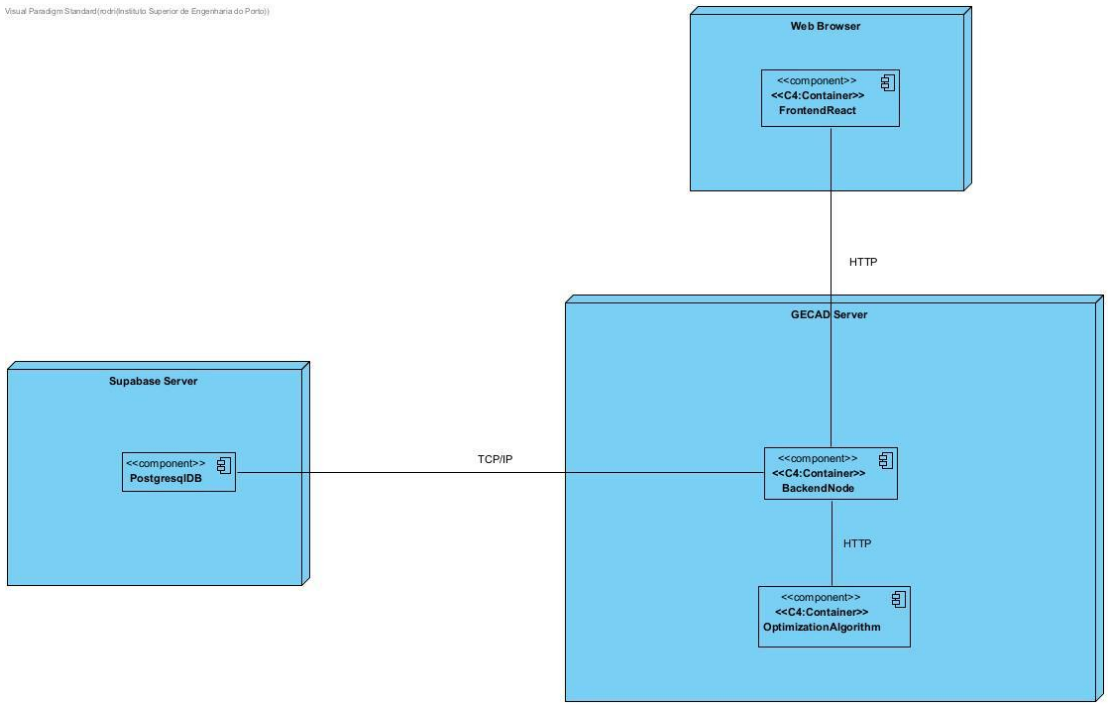

Figure 13 - Target Physical View Level 2

4.2 Requirements Engineering

This section outlines the system's functional and non-functional requirements, which were defined collaboratively with stakeholders and team members. These requirements guide the development process, ensuring that the platform delivers expected features, performance, and quality attributes aligned with real-world usage scenarios.

4.2.1 Functional Requirements

Following discussions with the GECAD supervisors and the development team, a collaborative process was established to distribute the system's functional responsibilities. Each team member was assigned a set of use cases, ensuring that the workload was balanced and the system's core functionality was effectively covered.

The table below presents the functional requirements that were assigned to me as part of the backend development effort. Each use case describes a specific functionality that the system must support, based on user roles such as Community Manager, Admin, or general User. These requirements were derived from user stories and reflect practical needs identified during platform planning.

The descriptions outline the expected behavior of the system from the user's perspective, ensuring that the implementation addresses the correct intent and delivers value to the target audience.

Use Case	Brief Description
US002 – Update existing prosumer's user profile	As a Community Manager, I want to update an existing prosumer user's profile, so that I can keep their information up to date.
US003 – Delete prosumer's user profile	As a Community Manager, I want to delete a prosumer user's profile, so that I can remove users who are deactivated.
US004 – Create user roles	As an Admin, I want to create roles, so that I can assign them to a user.
US008 – Update acount details	As a User, I want to update my account details, so that I can keep my information current.
US019 – Delete outdated/incorrect community data	As a Community Manager, I want to delete community data, so that I can remove outdated or incorrect information.
US020 – Add/Remove members from community	As a Community Manager, I want to add or remove members from the community, so that I can manage community participation.
US021 – Generate community reports	As a Community Manager, I want to generate community reports, so that I can share insights with members.

4.2.2 Non-Functional Requirements

In addition to the functional capabilities of the system, several non-functional requirements were defined to ensure the platform's quality, robustness, and maintainability. These requirements encompass key attributes such as usability, reliability, and performance, which are critical for delivering a responsive, scalable, and user-friendly experience in a real-world energy ecosystem.

4.2.2.1 Functionality

The system exposes well-defined route modules (such as userRoute, roleRoute, etc.) encapsulating specific RESTful endpoints related to its domain entity. Each route implements HTTP methods like GET, POST, PUT, and DELETE, using a consistent URI structure and clearly defined responsibilities. Input validation is performed through Joi schemas, enforced via the celebrate middleware, ensuring all request payloads conform to expected formats before reaching the controller layer. This guarantees robustness and correctness across API operations.

4.2.2.2 Usability

The system should be intuitive and user-friendly for all types of users, including community managers, prosumers, and administrators. The user interface, built in React, prioritizes accessibility and simplicity. API responses are designed to be human-readable for debugging and integration, and error messages are clear and informative.

4.2.2.3 Reliability

Reliability is ensured through extensive use of type safety (via TypeScript and Prisma) and validation mechanisms (DTOs and services). The backend must handle unexpected inputs gracefully and ensure no data corruption occurs. Repository interactions are isolated and tested, supporting consistent behavior over time.

4.2.2.4 Performance

The backend is designed for low-latency interactions with optimized queries, indexed PostgreSQL tables, and selective data fetching using Prisma. Although performance was not the primary goal, the system supports efficient request handling and scalable data access patterns suited for real-time energy monitoring.

4.2.2.5 Supportability

To facilitate long-term maintenance and evolution, the system is modular and readable. The use of interfaces (IService, IRepo, etc.) enables easy replacement and mocking. The project includes documentation, structured GitHub issues, and separation into domains that simplify onboarding, testing, and future extension.

4.2.2.6 Extras (+)

- **Security Awareness**: Basic security considerations like route protection and input sanitization are implemented or planned.
- **Testability**: Code organization supports unit testing and mocking of services and repositories.
- **Microservices Readiness**: Although implemented as a monolith, the architecture is structured to be easily modularized and deployed as microservices in future iterations.
- **Middleware Integration**: The route layer integrates reusable middleware components, such as authentication guards (middlewares.isAuth), allowing only authorized users to access sensitive routes. This modular design simplifies the enforcement of cross-cutting concerns like logging, security, and input validation.

- **Validation at Entry Point**: The use of celebrate and Joi for schema-based validation ensures that all requests are pre-validated before business logic execution. This helps maintain application integrity and reduce error propagation.
- **RESTful and Modular Route Structure**: Routes are grouped by resource (userRoute, roleRoute, among others), allowing each module to remain cohesive and independently testable.

4.2.3 Standards Used

Throughout the development of the Energy Communities Platform, several software engineering standards, best practices, and architectural conventions were adopted to ensure a maintainable, scalable, and high-quality system. These choices were guided by widely recognized industry approaches and reinforced by feedback from supervisors and peers.

The key standards and practices used include a few sections:

Architectural & Design Principles

- **Clean Architecture**: The project is structured with clearly defined layers (Controller → Service → Mapper → Model → Repository), isolating business logic from infrastructure and facilitating independent testing and deployment readiness.
- **Layered Architecture Pattern**: Each logical layer handles a specific concern, forming a well-organized backend structure. For example:
    - Controllers handle request input.
    - Services process business logic.
    - Repositories abstract persistence.
    - DTOs and Mappers ensure transformation and validation across boundaries.
- **Domain-Driven Design (DDD)**: The system revolves around meaningful domain entities such as Users, Communities, Roles, among others. These are organized using domain folders and mapped carefully through DTOs and Mappers, aligning with DDD principles.

SOLID Principles

- **Single Responsibility Principle (SRP)**: Each module, such as a controller, service, or mapper, has one well-defined responsibility.
- **Open/Closed Principle (OCP)**: Interfaces like IService, IRepo, and IMappers allow the system to be extended with new implementations without modifying existing code.
- **Liskov Substitution Principle (LSP)**: Use of interfaces allows components to be substituted polymorphically (e.g., mocking repositories in tests).
- **Interface Segregation Principle (ISP)**: Interfaces are purpose-specific (e.g., each IService only exposes necessary methods), avoiding large, generic contracts.
- **Dependency Inversion Principle (DIP)**: High-level modules depend on abstractions, not concrete implementations - the system architecture encourages injecting interfaces rather than classes directly.

Security, Validation & Code Quality

- **Middleware Layer for Cross-Cutting Concerns**: Authentication and request validation are abstracted into middleware such as isAuth, promoting reuse and clarity.
- **Input Validation with Celebrate + Joi**: Input schemas ensure that only valid data reaches the application core, improving reliability and security.
- **Type Safety with TypeScript**: Full-stack TypeScript usage enforces static typing, minimizing runtime bugs and enhancing editor tooling and refactoring safety.

Persistence and Data Access

- **Prisma ORM with PostgreSQL**: Prisma provides a type-safe and declarative approach to database access and migrations. It aligns with the repository pattern used throughout the backend.

Collaboration & Project Management

- **GitHub Repository Management**: The project was managed through a centralized GitHub repository, enabling version control, collaboration, and traceability across the development lifecycle.
- **Use of Issues and Labels**: All development tasks were tracked using GitHub Issues, each corresponding to a specific use case (e.g., US005 – View consumption and production history) and labeled accordingly (e.g., backend). This structured approach improved visibility and accountability within the team.
- **Branching Strategy**: A branching model was used to isolate development work, with individual branches created for specific features or fixes. This prevented code conflicts and ensured stable development flows.
- **Pull Requests and Code Review**: All changes were submitted through pull requests, allowing team members and supervisors to review, comment, and approve code before merging. This helped enforce coding standards and reduced the likelihood of introducing bugs.
- **Collaborative Workflow**: Team coordination was strengthened through task assignments, regular commits, and consistent integration of reviewed code into the main branch. The process supported incremental, collaborative progress on the system.

5 Solution Implementation

This chapter describes the implementation of the developed system based on the previously defined requirements and architecture. It includes an overview of the technologies used, the process of translating user stories into concrete features, the strategies adopted for testing, a visual walkthrough of the platform interface and its typical user flow, and finally, an evaluation of the resulting solution.

5.1 Development Environment and Tools

This section outlines the development environment and technologies adopted in the implementation of the Energy Communities Platform. The selected tools were chosen for their robustness, scalability, and alignment with modern web application standards. The backend and frontend components were developed using well-established technologies that support clean architecture principles, efficient data handling, and responsive user interfaces. Additionally, tools for collaboration, testing, and deployment were integrated into the workflow to streamline the development process and ensure maintainability throughout the project lifecycle.

5.1.1 Backend Stack

The backend of the Energy Communities Platform was developed using a modern, modular stack centered around Node.js and TypeScript, chosen for their performance, scalability, and strong typing support.

Although the initial comparative analysis in the State of the Art chapter (Sections 3.10.1.3 and 3.10.2.3) indicated Java with Spring Boot as the most suitable choice for scalability and enterprise-grade modularity, the final decision was to use a Node.js-based stack with TypeScript for the backend implementation. This choice was made after project planning discussions with the development team, taking into account existing team experience, development speed, and alignment with other tools in the stack. Despite not being the top-ranked choice in the theoretical evaluation, the Node.js/TypeScript solution proved to be a practical and effective fit within the project's constraints.

Key Technologies

- **Node.js & TypeScript**: Node.js provides a lightweight and efficient runtime environment, while TypeScript adds static typing and better tooling, improving code quality and maintainability.
- **Express.js**: Used as the web framework to define the RESTful API endpoints and manage routing. Route modules such as userRoute, batteryRoute, and communityRoute organize the API surface by domain.
- **Prisma ORM**: Prisma is employed to interface with a PostgreSQL database. It offers a clean and type-safe query experience, along with support for migrations and schema management.
- **Celebrate & Joi**: These libraries are used for validating incoming request payloads. Celebrate integrates Joi with Express middleware, allowing validation at the routing level.

Project Structure and Patterns

The backend follows a layered architecture, promoting separation of concerns and modularity:

- Controllers act as the entry point for each request, handling routing logic and delegating to services.
- Services encapsulate business logic and orchestrate operations between controllers, repositories, and domain models.
- Repositories (Repos) are responsible for data access and abstraction over Prisma operations.
- Mappers convert between DTOs and domain models, ensuring a clear boundary between internal logic and external representations.
- Interfaces such as IService, IRepo, IMappers, and IDTO define contracts between components, encouraging loose coupling and testability.

Design Principles

- SOLID principles were applied throughout the backend to promote clean architecture. In particular, the use of interfaces ensures adherence to the Dependency Inversion Principle, making components easier to replace and test.
- Scalability-ready: Although the project is monolithic in deployment, the architecture is structured to support a future transition to microservices, with well-isolated modules and domain-driven boundaries.

The backend of the Energy Communities Platform was developed using Node.js with TypeScript, combining performance and scalability with type safety and maintainability. The architecture was designed to be modular, separating concerns across clearly defined layers such as controllers, services, mappers, repositories, and domain models.

At the core of the application, Express.js is used to handle HTTP routing and request management. RESTful API endpoints are defined in route modules (e.g., userRoute, batteryRoute), each responsible for handling domain-specific operations. These routes incorporate middleware for cross-cutting concerns such as authentication and request validation.

To interact with the database, the project uses Prisma ORM, a type-safe and modern Object-Relational Mapping tool. Prisma facilitates efficient querying of the PostgreSQL database while ensuring data consistency and minimizing the risk of runtime errors through compile-time checks.

Celebrate and Joi were used for input validation, ensuring all incoming data adheres to expected formats before entering the core application logic. Additionally, interfaces such as IService and IRepo were used to enforce abstraction and improve testability, aligning with the SOLID principles.

The backend is structured for future scalability and microservices readiness, with components loosely coupled and clearly defined. This makes it easier to isolate functionalities if a service decomposition is required later.

5.1.2 Frontend Stack

The frontend of the Energy Communities Platform was developed using React with TypeScript, following a component-based architecture that promotes reusability, separation of concerns, and a responsive user experience.

Key Technologies

- **React & TypeScript**: React was chosen for its declarative and component-driven model, while TypeScript adds strong typing and IDE support, improving code safety and development efficiency.
- **Vite**: The project uses Vite as the frontend build tool and development server. Vite offers fast hot module replacement (HMR), optimized builds, and a lightweight configuration ideal for modern React projects.
- **Tailwind CSS**: Tailwind is used for styling the UI, providing a utility-first approach that enables consistent and maintainable design without the need for large custom CSS files.

Project Structure

The frontend codebase is organized within the src directory and structured as follows:

- **components/**
  Contains reusable UI elements such as layout containers, menus, dashboards, and breadcrumbs, ensuring consistency across pages.
- **pages/**
  Holds feature-specific views such as dashboard, batteries, products, and analytics pages. Each page corresponds to a distinct route or user-facing screen.
- **services/**
  Encapsulates API interaction logic in separate files (e.g., userService.ts, profileService.ts), abstracting HTTP requests and enabling clean communication with the backend.
- **interfaces/**
  Includes shared TypeScript interfaces for typing API responses and frontend data structures, improving type safety throughout the UI logic.
- **assets/**
  Stores static resources such as SVG files used in the UI.

This structure follows separation of concerns and enhances maintainability by grouping logic by purpose rather than technology (commonly referred to as "feature folders").

Best Practices and Patterns

- **Modular Design**: Each page and component is designed to be self-contained and reusable.
- **Service Layer for API Calls**: Centralized API interaction promotes easier debugging and testing.
- **State Management**: Although not visible from the screenshot, if needed, state could be managed via React hooks or context.
- **Responsive and Accessible UI**: Tailwind CSS enables quick prototyping while ensuring design consistency across devices.

The frontend of the Energy Communities Platform was built using React and TypeScript, combining the flexibility of a modern component-based framework with the reliability of static typing. The goal was to provide a modular, reusable, and user-friendly interface that enables community managers, prosumers, and administrators to interact with the platform's data intuitively and efficiently.

The project is bootstrapped using Vite, a fast and lightweight build tool optimized for modern frontend development. Vite offers near-instant hot module replacement and significantly faster development server startup times compared to traditional bundlers like Webpack. This results in a smoother and more productive development experience.

For the user interface layer, Tailwind CSS is used to streamline styling through utility-first classes. This approach eliminates the need for large, custom CSS files, while enabling rapid prototyping and enforcing a consistent design system across the application. Components are organized into meaningful groups within the components/ directory - for example, dashboard/, layout/, and menu/ - to promote reusability and encapsulation.

Routing and views are managed within the pages/ directory, where each folder represents a specific domain or interface screen, such as batteries, categories, dashboard, or products. This aligns with a feature-first organization that simplifies navigation and maintenance as the application grows.

API communication is handled through the services/ folder, where dedicated service modules (e.g., userService.ts, batteryService.ts) encapsulate HTTP logic for interacting with the backend. These services abstract away repetitive code and isolate concerns related to external data fetching, allowing page components to remain focused on rendering and user interaction.

Interfaces are declared in a dedicated interfaces/ folder, defining the structure of frontend data models and ensuring consistency between backend API contracts and frontend usage. This strong typing, enabled by TypeScript, reduces bugs and improves the overall robustness of the client-side application.

Overall, the frontend is designed with maintainability and scalability in mind, using clear architectural boundaries, typed contracts, and responsive design practices to ensure a seamless user experience across devices and evolving features.

5.1.3 DevOps and Collaboration Tools

Throughout the development of the Energy Communities Platform, a set of DevOps and collaboration tools was adopted to ensure consistent code quality, streamline deployment, and foster effective teamwork.

At the core of the development workflow was Git, used for version control, in conjunction with GitHub as the remote repository hosting service. Development was organized through feature branches, allowing contributors to work in isolation and minimizing conflicts with the main development line. Pull requests (PRs) were used as a review mechanism before merging into the main branch, ensuring peer review, traceability, and code quality.

To manage tasks and track progress, GitHub Issues and Labels were extensively used. Each backend-related feature or bug fix was associated with a user story and labeled accordingly (e.g., backend, bug, enhancement). This structured task management enabled clear assignment of responsibilities among team members and simplified sprint planning and reporting.

For dependency management and project configuration, npm (Node Package Manager) was employed on both the backend and frontend.

Although full CI/CD pipelines were not deployed in this iteration, the project was designed with DevOps-readiness in mind. The modular architecture and clear separation of concerns across the backend and frontend components pave the way for future integration with tools such as GitHub Actions or Docker.

Collaboration was further supported through discussions with supervisors at GECAD and team meetings where design decisions, architectural choices, and task assignments were reviewed. Documentation was maintained in the project repository using README.md files and GitHub project boards, promoting transparency and onboarding ease.

5.2 Feature Implementation: Detailed Use Case Walkthrough

This section presents a detailed walkthrough of a selected use case to illustrate the flow of information and logic across the system's layers. Specifically, we examine Use Case 002 – Update Existing Prosumer’s User Profile, showcasing the interaction between the frontend, routing, controller, service, domain, and persistence layers. This step-by-step breakdown highlights how the architectural components work together to fulfill functional requirements.

5.2.1 Step-By-Step Execution Flow

5.2.1.1 Frontend Interaction

The execution flow begins when a user interacts with the application's user interface by clicking the "Update User" button, found within his profile. This action is triggered after the user has modified one or more input fields - such as first name, email, or role - in a form. Once the button is clicked, the frontend captures the updated data and prepares it to be sent to the backend via an HTTP PUT request.

5.2.1.2 HTTP Request and Route Handling

Once the update request from the frontend reaches the backend, it is handled by the defined route in the userRoute module, as seen in Figure 14.

```
route.put(
    path: "/update/:id",
    celebrate( requestRules: {
        body: Joi.object( schema: {
            firstName: Joi.string().optional(),
            lastName: Joi.string().optional(),
            phoneNumber: Joi.string().optional(),
            email: Joi.string().email().optional(),
            role: Joi.string().optional(),
            password: Joi.string().optional(),
        }),
    }),
    middlewares.isAuth,
    (req : Request<ParamsDictionary, any, any, ParsedQs, Record<string, any>> , res : Response<any, Record<string, any>> , next : NextFunction ) => ctrl.updateUser(req, res, next)
);
```

Figure 14 - UserRoute PUT method

The PUT /update/:id route is registered and wrapped using the celebrate middleware for request validation. This validation schema, defined using Joi, ensures that all incoming fields (e.g., firstName, email, role) are optional but must conform to expected formats - such as verifying the structure of the email.

Following validation, the middlewares.isAuth middleware checks whether the request is authenticated, enforcing access control before any update logic is executed. If the request passes

both validation and authentication layers, it is forwarded to the controller method updateUser, as defined in the UserController class. At this point, the request parameters and body data are parsed and passed along to the next stage of the application: the controller logic that orchestrates service-level operations.

5.2.1.3 Controller Invocation

Once the request passes through validation and authentication middleware, it is routed to the appropriate controller method - in this case, updateUser from the UserController class.

The UserController class is marked with the @Service decorator, which registers it with the dependency injection container. Inside the constructor, the controller declares a dependency on the IUserService interface, which is injected via the @Inject decorator using a service identifier (see Figure 15). This design promotes modularity and flexibility, allowing the controller to rely on abstractions rather than concrete implementations.

```
@Service()  no usages  Rodrigo Costa +1 *
export default class UserController {  Rodrigo Costa +1 *

  constructor(  no usages  new *
      @Inject(config.services.user.name) private userService: IUserService,
  ) {
  }
```

Figure 15 - UserController class

When updateUser is invoked, it extracts the id from the request parameters and user data from the request body. It then assembles a UserDTO object and delegates the actual update logic to the updateUser method of userServiceInstance. The controller is responsible for interpreting the result and returning an appropriate HTTP response - either an error message (400) if the update fails, or a success message (200) along with the updated user data if the operation is successful.

```
public async updateUser(req: Request, res: Response, next: NextFunction) : Promise<...>  {  no usages  Rodrigo Costa *
  try {
    const { id : string  } = req.params;
    const { firstName, lastName, phoneNumber, email, role, password } = req.body;

    // Gather the data to update from the request body
    const userDTO : IUserDTO  = UserMap.toDTO(firstName,lastName,phoneNumber,email,role,password)

    // Call the userService's update method to handle the actual update logic
    const result = await this.userService.updateUser(id, userDTO);

    // If the update failed, return an error response
    if (result.isFailure) {
      return res.status( code: 400).json( body: { message: result.error });
    }

    // Successfully updated user
    return res.status( code: 200).json( body: { message: 'User updated successfully', data: result.getValue() });
  } catch (error) {
    return next(error);
  }
}
```

Figure 16 - UpdateUser method inside UserController

5.2.1.4 Business Logic in the Service Layer

Once the controller delegates the request, the UserService class takes over the business logic through its updateUser method. The service implements the IUserService interface, ensuring that all exposed functionalities are clearly defined and injectable elsewhere in the application.

```
export default interface IUserService  {  Show usages   Rodrigo Costa +1
  SignUp(userDTO: IUserDTO): Promise<Result<{userDTO: IUserDTO, token: string}>>;
  SignIn(email: string, password: string): Promise<Result<{ userDTO: IUserDTO, token: string }>>;
  SignOut(token: string): Promise<Result<void>>;
  ForgotPassword(email: string): Promise<Result<string>>;
  ResetPassword(token: string, password: string): Promise<Result<string>>;
  confirmAccount(token:any): Promise<Result<void>>;
  getUser(id:string): Promise<Result<IUserDTO>>;
  findStaff(): Promise<Result<IUserDTO[]>>;
  isAdmin(id: string): Promise<Result<boolean>>;
  getAllUsers(): Promise<Result<IUserDTO[]>>;
  updateUser(id: string, userDTO: IUserDTO);
  toogleActiveStatus(id: string): Promise<Result<void>>;
```

Figure 17 - IUserService interface

The controller communicates with the service through its interface, promoting loose coupling and adherence to the Dependency Inversion Principle. Similarly, the service interacts with the repository via its interface, ensuring that the business logic remains decoupled from the underlying data access implementation.

```
@Service()  no usages   Rodrigo Costa +1 *
export default class UserService implements IUserService {   Rodrigo Costa +1 *
  private revokedTokens : Set<string>  = new Set<string>();

  constructor(  no usages   Rodrigo Costa
      @Inject(config.repos.user.name) private userRepo: IUserRepo,
      @Inject(config.repos.role.name) private roleRepo: IRoleRepo,
      @Inject( serviceName: 'logger') private logger
  ) {
  }
```

Figure 18 - UserService class

As shown in Figure 19, the service method begins by retrieving the user through the repository. After updating the required fields - applying validations and transformations - the modified domain object is persisted using the same repository. Finally, the UserMap is used to convert the updated domain entity into a DTO, ensuring the controller remains decoupled from the core domain logic.

```
public async updateUser(id: string, userDTO: IUserDTO): Promise<Result<IUserDTO>> {  no usages  Rodrigo Costa
    // Check if the user exists
    const user : User  = await this.userRepo.findById(id);
    if (!user) {
      return Result.fail<IUserDTO>( error: 'User not found');
    }

    // Update the user properties based on the provided userDTO
    if (userDTO.firstName) {
      user.updateFirstName(userDTO.firstName)
    }
    if (userDTO.lastName) {
      user.updateLastName(userDTO.lastName)
    }
    if (userDTO.email) {
      const email : UserEmail  = await UserEmail.create(userDTO.email).getValue();
      user.updateEmail(email)
    }
    if (userDTO.phoneNumber) {
      const phoneNumber : PhoneNumber  = await PhoneNumber.create(userDTO.phoneNumber).getValue();
      user.updatePhoneNumber(phoneNumber)
    }
    if (userDTO.role) {
      const role : Role  = await this.roleRepo.findByName(userDTO.role);
      if (!role) {
        return Result.fail<IUserDTO>( error: `Role ${userDTO.role} not found`);
      }
      user.role = role;
    }
    if (userDTO.password) {
      const hashedPassword : string  = await argon2.hash(userDTO.password);
      const password : UserPassword  = await UserPassword.create({ value: hashedPassword, hashed: true }).getValue();
      user.updatePassword(password)
    }

    // Save the updated user to the database
    await this.userRepo.save(user);

    // Return the updated userDTO
    const userDTOResult : IUserDTO  = UserMap.toDTO(user) as IUserDTO;
    return Result.ok<IUserDTO>(userDTOResult);
  } catch (error) {
    this.logger.error(error);
    return Result.fail<IUserDTO>( error: 'Error updating user: ' + error.message);
  }
}
```

Figure 19 - UpdateUser method inside UserService

5.2.1.5 Domain Logic and Validation

Figure 20 illustrates the updateFirstName() method within the User aggregate, which encapsulates domain-specific rules for modifying a user's first name. Rather than allowing external layers (such as the service or controller) to directly manipulate the firstName property, this method ensures that only valid data can enter the domain model. It enforces business constraints - such as rejecting empty or excessively long names - thereby maintaining the integrity of the entity. This approach exemplifies one of the core principles of Domain-Driven Design: keeping business rules close to the data they govern, and ensuring consistency through behavior-rich domain models rather than relying solely on external validation.

```
public updateFirstName(firstName: string): Result<void> {  Show usages  Rodrigo Costa
  if (!firstName || firstName.trim().length === 0) {
    return Result.fail<void>( error: 'First name cannot be empty');
  }

  if (firstName.length > 50) {
    return Result.fail<void>( error: 'First name must be less than 50 characters');
  }

  this.props.firstName = firstName;
  return Result.ok<void>();
}
```

Figure 20 - UpdateFirstName validation logic inside the User class

5.2.1.6 Repository Access

The persistence layer of the application is handled by the UserRepo class, which implements the IUserRepo interface to promote decoupling and testability. The repository interacts with the database through Prisma, a type-safe ORM injected via the class constructor as shown in Figure 21.

```
@Service()  no usages  Rodrigo Costa +1 *
export default class UserRepo implements IUserRepo {  Rodrigo Costa +1 *
  constructor(  no usages  Rodrigo Costa
      @Inject( serviceName: 'prisma') private prisma: PrismaClient
  ) {
  }
```

Figure 21 - UserRepo class

5.2.1.7 Mapper

The UserMap class is responsible for converting domain entities into data transfer objects (DTOs), and vice versa. This mapping layer is crucial to ensure a clean separation between the domain and external layers, such as the controller and database.

```
export class UserMap extends Mapper<User> {  Show usages  Rodrigo Costa +1

  public static toDTO( user: User): IUserDTO {  Show usages  Rodrigo Costa +1
    return {
      id: user.id.toString(),
      firstName: user.firstName,
      lastName: user.lastName,
      email: user.email.value,
      phoneNumber: user.phoneNumber.value,
      password: "",
      role: user.role.id.toString(),
      isActive: user.isActive
    } as IUserDTO;
  }
```

Figure 22 - UserMap class

As shown in Figure 22, the toDTO static method inside UserMap receives a User domain object and returns a plain IUserDTO. It extracts only the necessary fields, such as id, firstName, lastName, email, phoneNumber, role, and isActive.

5.2.1.8 Returning the Response

After the user is successfully updated, the service returns a DTO representation of the user, which is then passed back to the controller. The controller formats the HTTP response accordingly and sends it to the client with a success message and a 200 OK status code. In case of failure during the update process, an appropriate error message and status code (400 Bad Request) are returned instead, ensuring clear feedback to the client.

5.3 Testing Approach

To ensure the correctness, reliability, and maintainability of the developed system, multiple levels of testing were applied, each targeting specific aspects of the architecture. This layered testing strategy aligns with the principles of clean and modular backend design, where components are tested both in isolation and in integration with others.

Unit testing was employed to verify the functionality of individual components such as domain entities, services, and utility classes. End-to-End and Acceptance Testing were used to validate complete system flows, ensuring that user interactions translated into correct backend behavior across multiple layers, from API endpoints to persistence. Additionally, API Testing with Postman served as a practical mechanism to verify the system's public interface through scripted HTTP requests, enabling rapid validation of core functionality and response consistency.

5.3.1 Unit Testing

Unit testing was applied to verify the behavior of individual components in isolation, with a particular focus on the core service logic. The updateUser method in the UserService class served as a representative example of this approach. This method encapsulates business rules such as

updating user details, validating roles, and persisting changes, making it ideal for targeted unit testing.

To ensure robustness, a dedicated test suite was implemented that covered a variety of realistic and edge-case scenarios. These included successful updates, cases where the user was not found, role validation failures, handling of missing input fields, and the system's behavior under unexpected errors. The service was tested in isolation by mocking dependencies such as the user repository, role repository, and domain value objects (e.g., UserEmail, UserPassword, and PhoneNumber).

The unit test suite was executed using Jest and consisted of six distinct test cases. Each case verified specific paths through the updateUser logic, ensuring that the correct methods were called and that the return values reflected the expected outcomes. This level of granularity made it possible to identify and correct logical errors early in the development process.

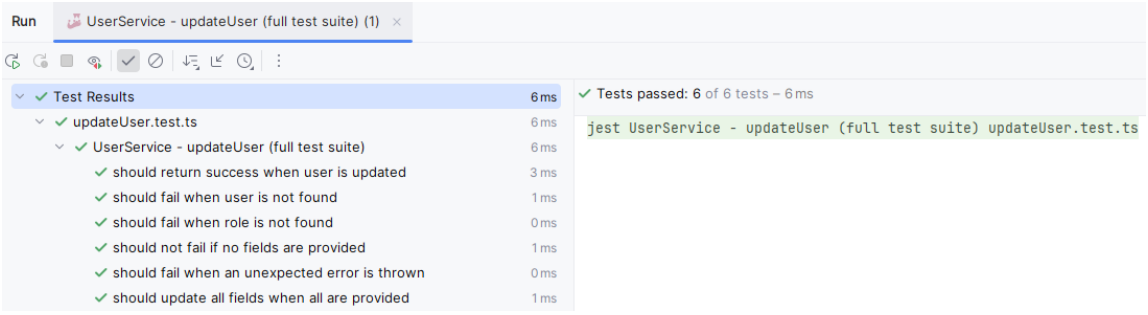

Figure 23 - Results of the Unit Test Suite for the updateUser method in the UserService

The successful execution of the unit tests for this method, as shown in Figure 23, illustrates how specific service logic was verified under multiple conditions, helping to reinforce confidence in the reliability of this component in isolation.

5.3.2 End-to-End and Acceptance Testing

To validate the system as a whole and ensure that it functioned correctly from the user's perspective, End-to-End (E2E) and Acceptance Testing were conducted through manual interaction with the application. This approach aimed to verify that all components - from the user interface to the backend services and database - worked together as expected.

The development team performed typical user flows within the web application, including actions such as user registration, authentication, profile updates, and role-based access scenarios. These tests provided a high-level confirmation that the application fulfilled its functional requirements and adhered to the business rules defined during the design phase.

This form of testing served both as an informal acceptance test and a practical E2E validation. By simulating real-world usage scenarios through the interface, the team ensured that all integrated parts of the system operated in harmony, with correct data persistence and feedback mechanisms.

Although automated E2E tests were not implemented, this manual testing approach was effective in detecting integration issues and validating overall system behavior during development and prior to delivery.

5.3.3 API Functional Test

To verify the correctness and robustness of the system's public-facing endpoints, a series of API functional tests were created using Postman. These tests were designed to simulate real client interactions with the backend and validate that the HTTP responses matched the expected behavior under various conditions.

The focus was placed on the /api/users/update/:id endpoint, which handles updates to user records. Each test case targeted a specific usage scenario or edge case and was implemented as an independent request within a structured Postman collection. These scenarios included:

- **Update User – Success**: Verifies that a valid request with proper authentication and data results in a successful update (HTTP 200 or 201).
- **Update User – User Not Found**: Tests how the API responds when the provided user ID does not exist in the database, expecting a 404 Not Found error.
- **Update User – Invalid User ID Format**: Checks that the API handles malformed user IDs correctly, returning a 400 or 422 status.
- **Update User – Missing Required Fields**: Simulates an attempt to update a user without mandatory fields, expecting the API to reject the request with a 400 Bad Request.
- **Update User – Unauthorized Request**: Validates that the endpoint is protected, and requests without valid authentication return a 401 Unauthorized.

Each request included a JavaScript-based assertion in the Postman "Tests" tab, allowing automated validation of response codes and message content.

The structure of the Postman collection, as shown in Figure 24, provided a clean and reusable way to manually or automatically validate API functionality.

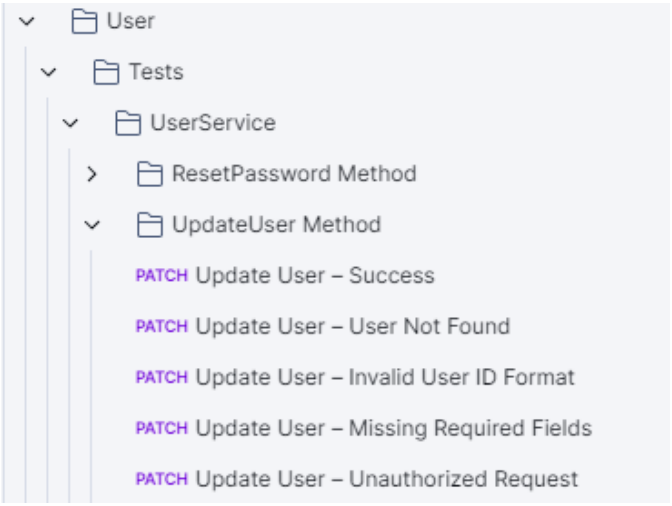

Figure 24 - Structure of the Postman test collection for the updateUser endpoint

Among the functional tests defined for the Energy Communities Platform, one verifies the successful execution of a user update operation using the PATCH HTTP method. This test validates the system's ability to correctly handle user data modifications when provided with valid input and proper authorization.

As shown in Figure 25, the request is made to the /users/update/:id endpoint, where the user identifier is dynamically set using the userId environment variable. The request body also uses

variables to allow flexible test input. This approach supports repeatable testing and easier adjustments across test environments.

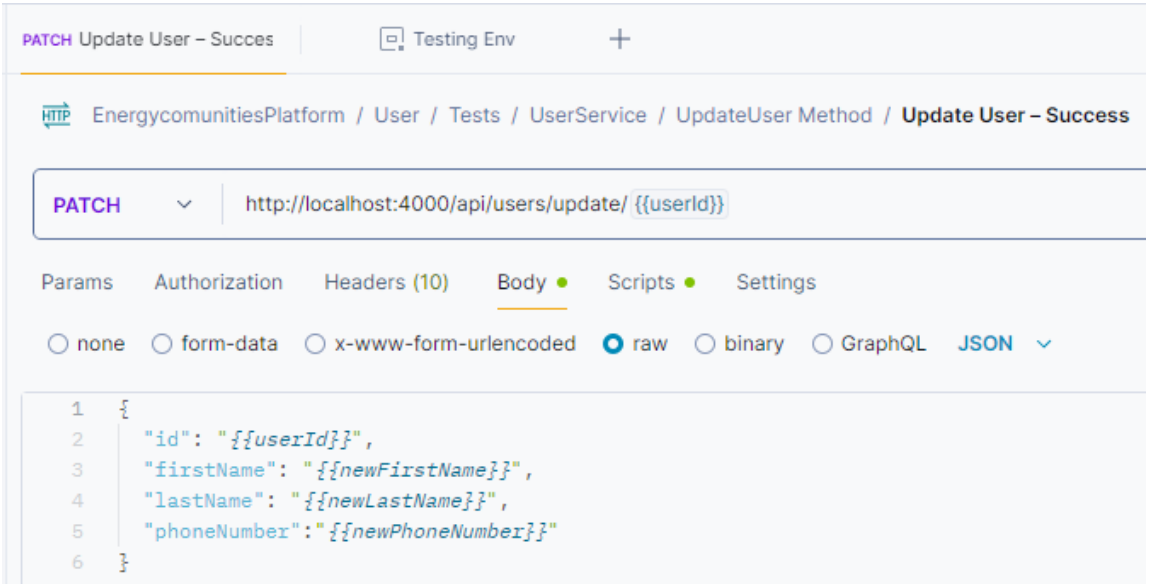

Figure 25 - PATCH request with dynamic variables in URL and body

To manage these variables, a dedicated Postman environment was used. As illustrated in Figure 26, variables like userId, newFirstName, and newPhoneNumber are defined to inject values into the request dynamically. This setup simplifies test maintenance and reduces hardcoding.

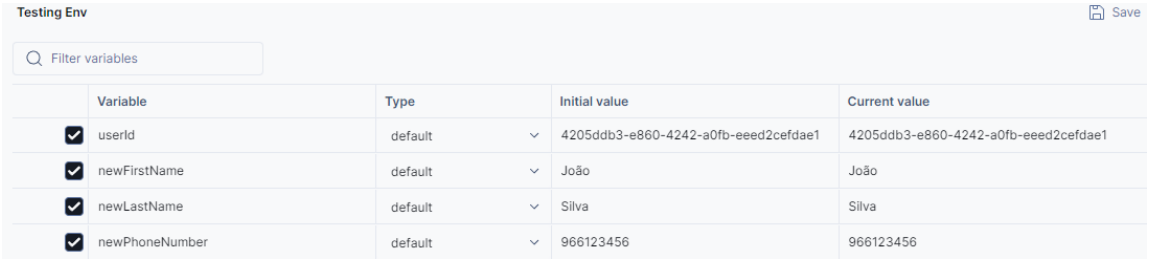

Figure 26 - Testing environment with defined Postman variables

To verify the correctness of the response, several automated test scripts were written in Postman. These scripts check whether the request returned a successful status code, whether the updated fields match the input values, and whether authorization and performance conditions are met. These checks are visible in Figure 27.

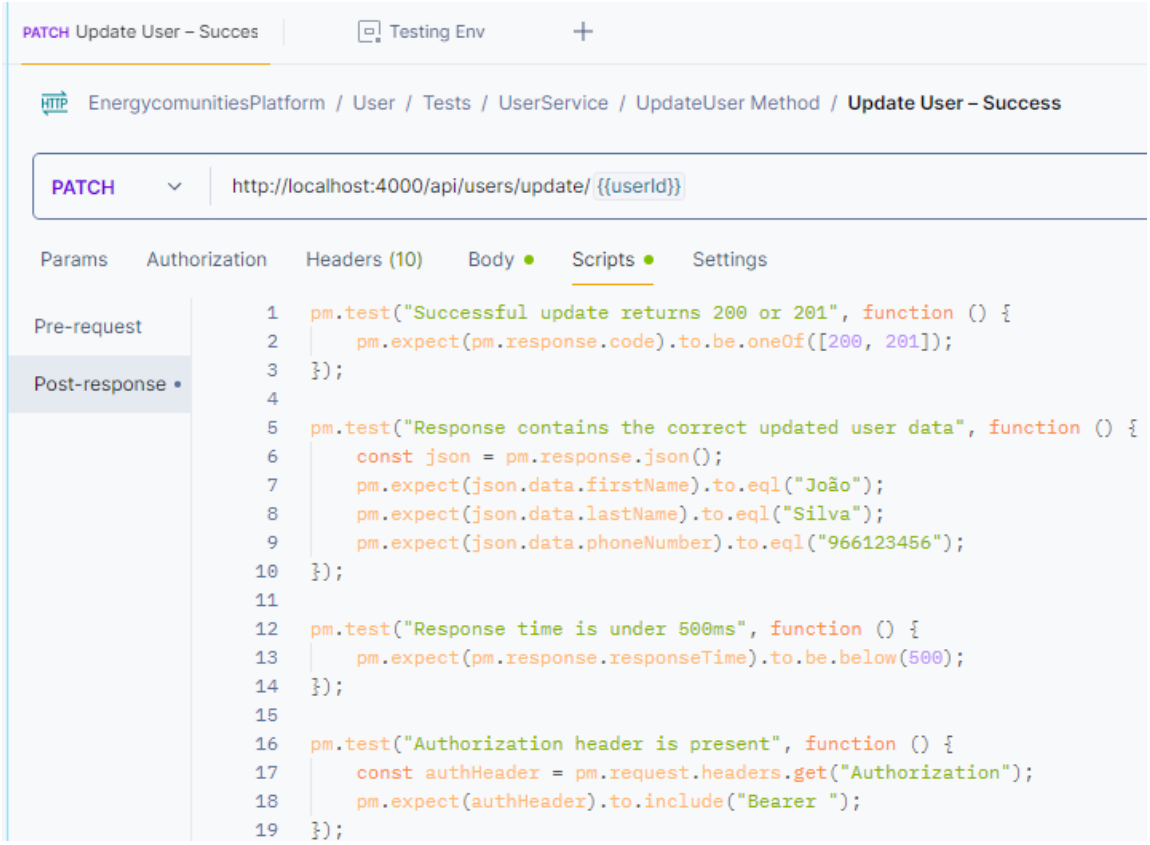

Figure 27 - Postman test scripts validating the Update User operation

When executed, the test passed all conditions, confirming that the endpoint worked as expected. The result, shown in Figure 28, includes a success message and a data object reflecting the updated user information.

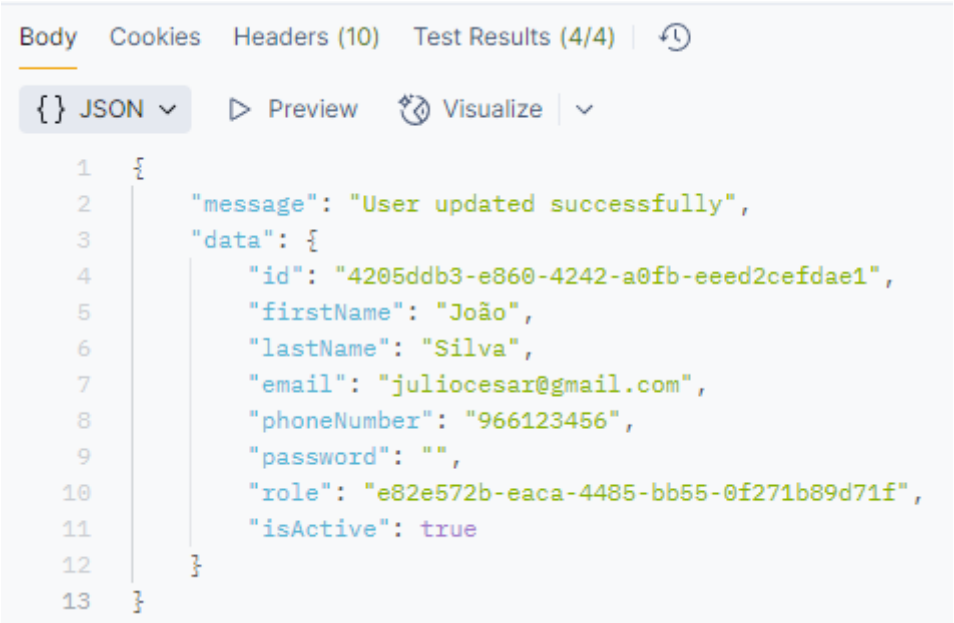

Figure 28 - Response body with updated user data

Lastly, Figure 29 shows the summary of all test results, where each validation ran successfully, confirming that the PATCH request was executed under valid, expected conditions.

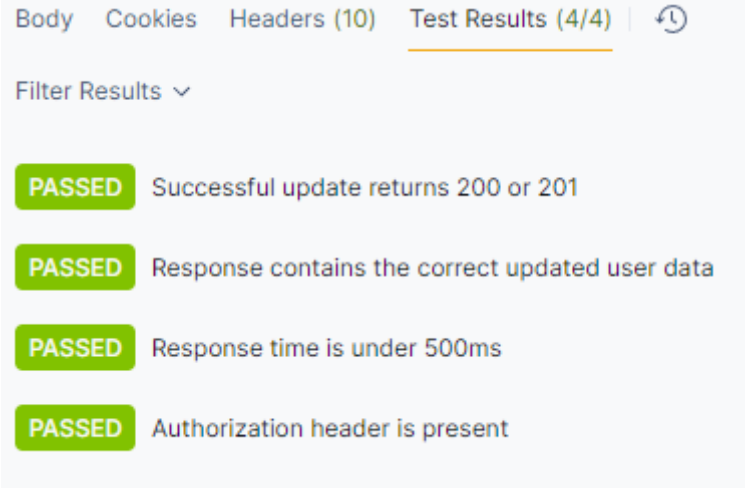

Figure 29 - Passed functional test results for the Update User endpoint

5.4 Platform Overview and User Flow

In addition to the backend-focused technical design, this section offers a visual and functional overview of how the Energy Communities Platform is experienced by end users. It highlights the key interface areas and typical interaction flows, especially from the perspective of a Community Manager. Through this role, users can manage core entities such as communities, prosumers, and batteries, while also accessing energy analytics and dashboard summaries. The following subchapters outline the main tabs available to a community manager, illustrating how different parts of the system come together to support real-world energy coordination and decision-making.

5.4.1 User

The community manager can access their user section to update personal information, manage settings, or toggle between light and dark themes, as shown in Figure 30.

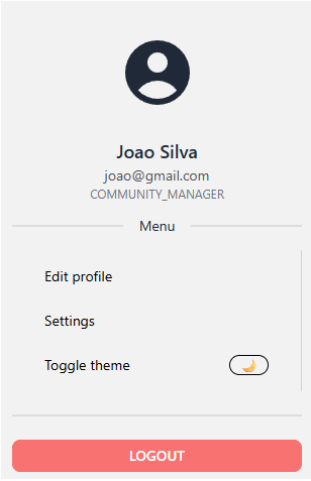

Figure 30 - User Tab: Settings

The community manager can also update their personal information, as shown in Figure 31. Upon clicking the 'Save' button, the frontend triggers the updateUser method, previously detailed in this document.

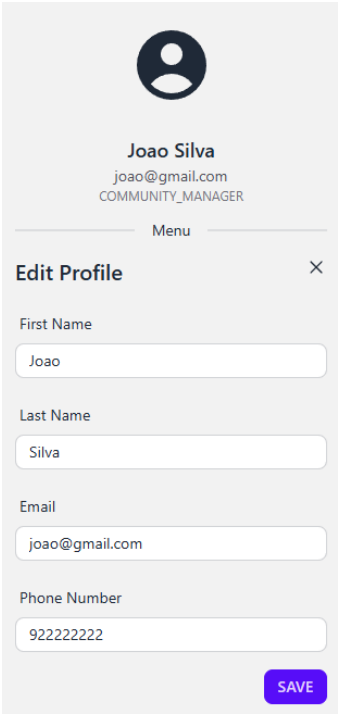

Figure 31 - User Tab: Edit user information

5.4.2 Communities

The community manager can view their assigned community, including its list of members, and has the ability to manage them - such as adding or removing prosumers - as illustrated in Figure 32.

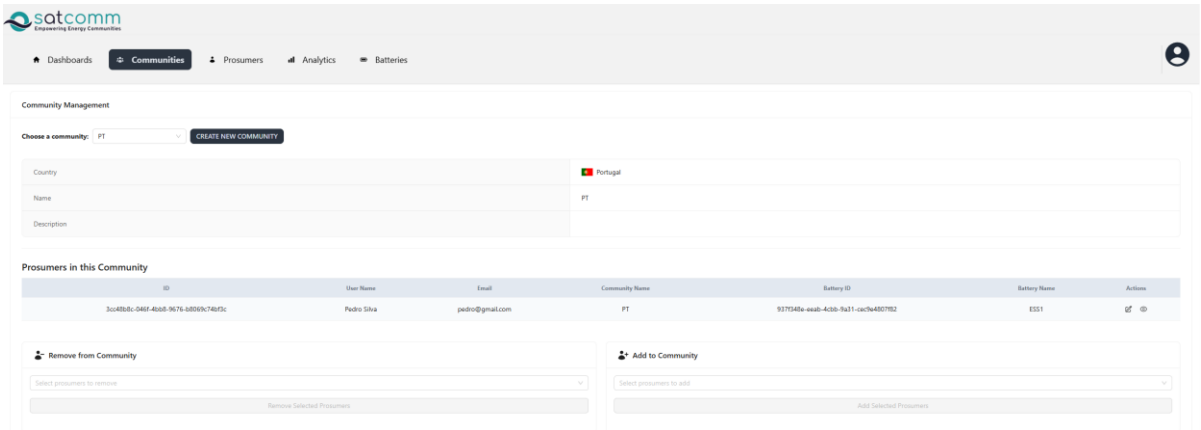

Figure 32 - Communities Tab: Overview and Member Management

The community manager can also upload community energy data in Excel format, apply the optimization algorithm, and filter the results by selecting a start and end date, as illustrated in Figure 33.

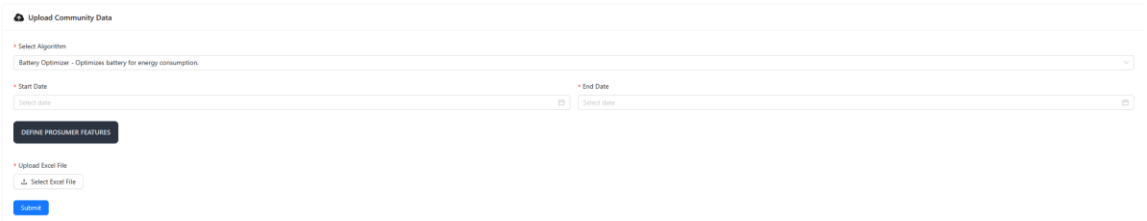

Figure 33 - Communities Tab: Upload community data

The file to be submitted must follow the structure illustrated in Figure 34. Each prosumer is represented by 96 rows per day, corresponding to 15-minute intervals. The data is organized sequentially - first all entries for Prosumer 1 on Day 1, then Prosumer 2 on Day 1, and so on, followed by Prosumer 1 on Day 2, Prosumer 2 on Day 2, etc.

Day	Time_Step	Prosumer	P_buy	P_sell	SOC	P_ESS_ch	P_ESS_dch	P_PV_load	P_PV_ESS	P_Peer_out	P_Peer_in	P_Load
1	1	1	1.889	0	0.42	0	0	0	0	0	0	1.889
1	2	1	3.7265	0	0.42	0	0	0	0	0	0	3.7265
1	3	1	3.8255	0	0.42	0	0	0	0	0	0	3.8255
1	4	1	3.805	0	0.42	0	0	0	0	0	0	3.805
1	5	1	2.821	0	0.42	0	0	0	0	0	0	2.821
1	6	1	3.224	0	0.42	0	0	0	0	0	0	3.224
1	7	1	3.271	0	0.42	0	0	0	0	0	0	3.271
1	8	1	3.3605	0	0.42	0	0	0	0	0	0	3.3605
1	9	1	3.9845	0	0.42	0	0	0	0	0	0	3.9845
1	10	1	4.365	0	0.42	0	0	0	0	0	0	4.365
1	11	1	3.702	0	0.42	0	0	0	0	0	0	3.702
1	12	1	2.95	0	0.42	0	0	0	0	0	0	2.95
1	13	1	3.429	0	0.42	0	0	0	0	0	0	3.429
1	14	1	2.823	0	0.42	0	0	0	0	0	0	2.823
1	15	1	3.301	0	0.42	0	0	0	0	0	0	3.301
1	16	1	3.0945	0	0.42	0	0	0	0	0	0	3.0945
1	17	1	2.8995	0	0.42	0	0	0	0	0	0	2.8995
1	18	1	3.335	0	0.42	0	0	0	0	0	0	3.335
1	19	1	2.853	0	0.42	0	0	0	0	0	0	2.853
1	20	1	3.542	0	0.42	0	0	0	0	0	0	3.542
1	21	1	3.168	0	0.42	0	0	0	0	0	0	3.168
1	22	1	3.2105	0	0.42	0	0	0	0	0	0	3.2105
1	23	1	3.0455	0	0.42	0	0	0	0	0	0	3.0455
1	24	1	3.101	0	0.42	0	0	0	0	0	0	3.101
1	25	1	3.921	0	0.42	0	0	0	0	0	0	3.921
1	26	1	3.289	0	0.42	0	0	0	0	0	0	3.289
1	27	1	2.895	0	0.42	0	0	0	0	0	0	2.895
1	28	1	3.9155	0	0.42	0	0	0	0	0	0	3.9155
1	29	1	4.0405	0	0.42	0	0	0	0	0	0	4.0405
1	30	1	4.071	0	0.42	0	0	0	0	0	0	4.071
1	31	1	2.982	0	0.42	0	0	0	0	0	0	2.982
1	32	1	3.2535	0	0.42	0	0	0	0	0	0	3.2535
1	33	1	3.9905	0	0.42	0	0	0	0	0	0	3.9905
1	34	1	4.5165	0	0.42	0	0	0	0	0	0	4.5165
1	35	1	3.8705	0	0.42	0	0	0	0	0	0	3.8705
1	36	1	3.5865	0	0.42	0	0	0	0	0	0	3.5865
1	37	1	2.31633	0	0.42	0	0	0.04217	0	0	0	2.3585
1	38	1	2.79492	0	0.42	0	0	0.20758	0	0	0	3.0025
1	39	1	3.36883	0	0.42	0	0	0.39067	0	0	0	3.7595
1	40	1	2.14492	0	0.42	0	0	0.58258	0	0	0	2.7275

Figure 34 - Community data file example

5.4.3 Prosumers

In the Prosumers tab, the community manager can view all existing prosumers and add new ones to the system, as shown in Figure 35.

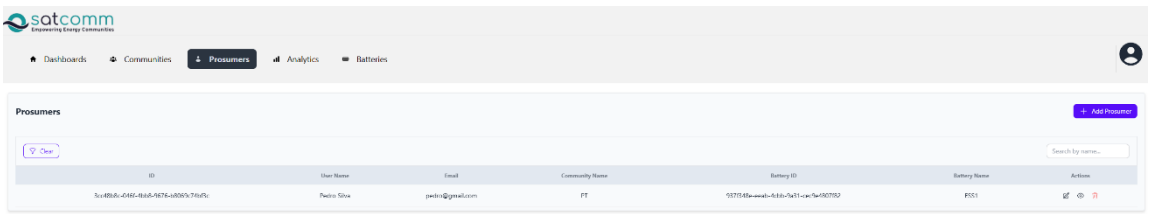

Figure 35 - Prosumers Tab

5.4.4 Batteries

In the Batteries tab, the community manager can view all existing batteries and add new ones to the system, as shown in Figure 36.

Figure 36 - Batteries Tab

5.4.5 Analytics

In the Analytics tab, the community manager can select a prosumer and view a simulation of their energy transitions over time, as illustrated in Figure 37. The chart supports three levels of granularity: 15-minute intervals (each point represents one profile), 1-hour intervals (averaging four 15-minute profiles per hour), and 1-day intervals (averaging 96 profiles across a full day). Additionally, users can filter the results by selecting a start and end date; if left empty, the system displays all available data.

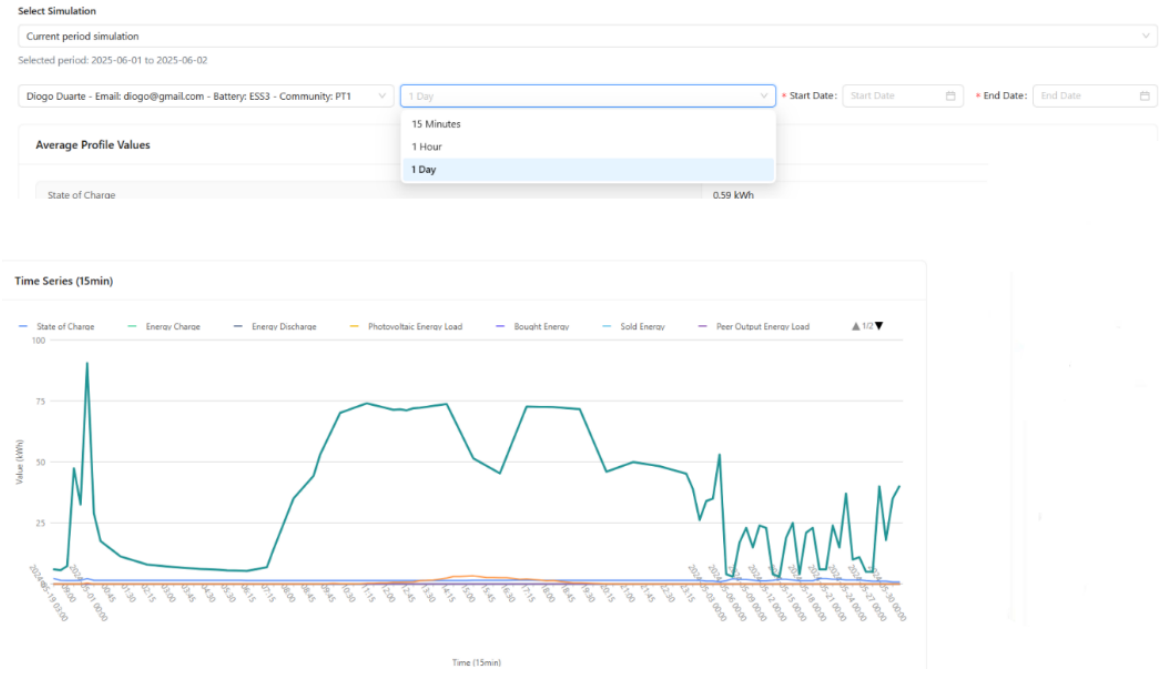

Figure 37 - Analytics Tab: Simulation of energy transition over time

The community manager can also analyze the community's energy distribution through multiple visual perspectives, as shown in Figure 38. The available breakdowns include:

- **All Energy Distribution**: A global overview of every energy transaction within the prosumer, including peer-to-peer exchanges (Peer Input and Output), market interactions (Bought and Sold Energy), photovoltaic generation, battery usage (Charge and Discharge), and overall consumption (Profile Load).

- **External Transactions**: Focuses solely on market interactions, detailing how much energy was bought from or sold to the grid.
- **Generation vs Consumption**: Compares energy produced locally through photovoltaic generation with the total consumed energy (Profile Load), providing insights into self-sufficiency.
- **Battery Charge/Discharge**: Displays how much energy was stored in and retrieved from batteries, supporting an evaluation of battery usage patterns and efficiency.

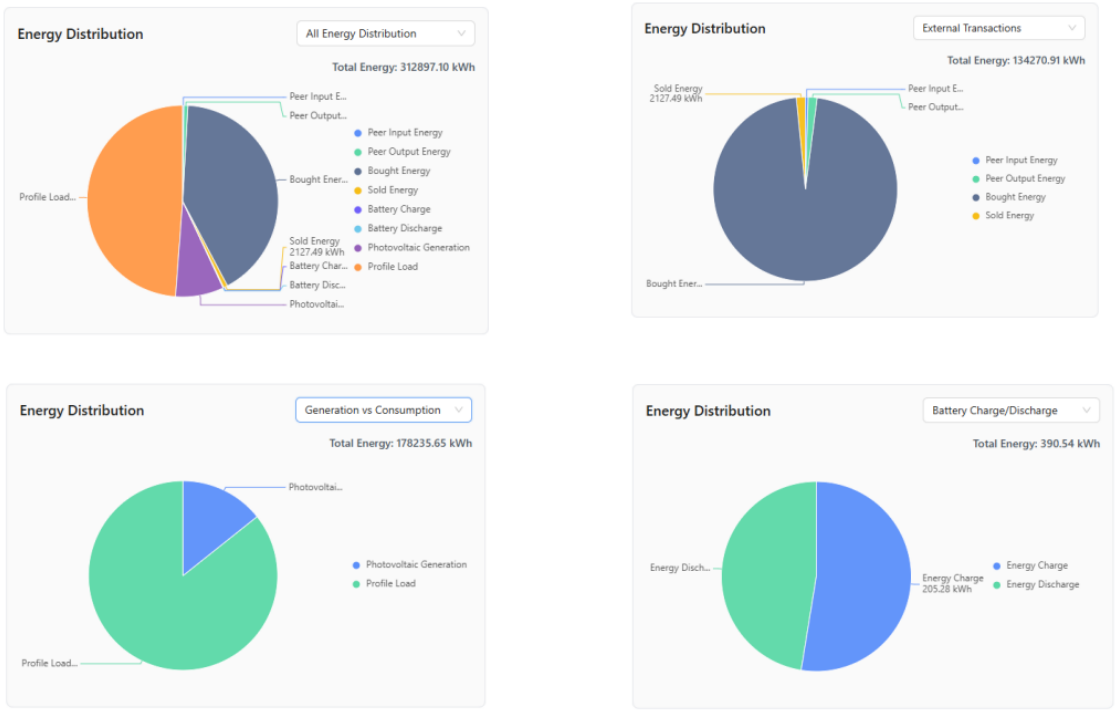

Figure 38 - Analytics Tab: Energy distribution

5.4.6 Dashboards

In the Dashboards tab, the community manager can access a high-level summary of the community's energy activity over time, as illustrated in Figure 39. The view includes key performance indicators such as total energy consumption, photovoltaic generation, energy bought from and sold to the grid, along with a line graph that tracks metrics like energy efficiency, storage behavior, and transaction trends across selected periods.

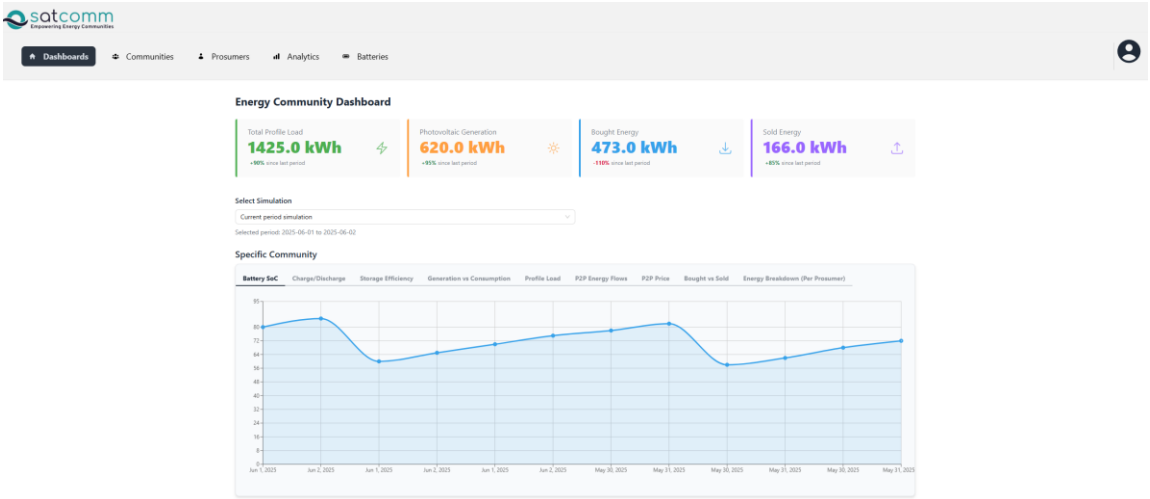

Figure 39 - Dashboards Tab

5.5 Solution Evaluation

The implemented solution meets all functional requirements established during the planning and analysis phases. Core operations such as user registration, authentication, role management, and profile updates were successfully developed and validated through a combination of unit tests and API-level functional tests. The updateUser functionality, in particular, was used as a representative case for verifying service correctness across various input conditions, with comprehensive coverage of success and failure scenarios.

Although the initial intention was to adopt a fully microservices-based architecture, the final implementation was developed as a microservices-ready monolith. This means that the system was structured using modular principles - such as domain-driven design (DDD), separation of concerns, and dependency injection - so that individual modules (e.g., user, role) can be extracted into separate services in the future with minimal effort. This architectural approach supports scalability and aligns with microservices principles while maintaining the simplicity and efficiency of a unified codebase during early development stages.

From a non-functional perspective, the solution was evaluated using the FURPS+ model:

- **Functionality** was supported through the implementation of input validation, authentication, and role-based authorization across key endpoints, ensuring secure and correct behavior. These features were verified through unit tests and Postman-based functional tests.
- **Usability** was not a central focus due to the backend nature of the project. Nonetheless, a clean and consistent API design was achieved, supported by Postman collections for ease of testing and integration with potential frontend clients.
- **Reliability** was addressed by implementing error handling, layered validation (DTO, service, domain), and tests that confirmed system behavior in edge cases such as invalid input, unauthorized access, or non-existent user IDs.
- **Performance** was assessed informally during development. Endpoints demonstrated responsive behavior under normal conditions, aided by straightforward data access patterns and efficient service logic.
- **Supportability** was emphasized through a modular and microservices-ready architecture using TypeScript, clear separation of concerns, and domain-driven design principles. The presence of automated tests and organized code structure supports maintainability and future scalability.

In summary, the solution delivers a solid backend foundation that not only fulfills current requirements but is also architected for future evolution into a distributed microservices environment. This balance between immediate functionality and long-term scalability reflects a deliberate and practical approach to backend system design.

6 Conclusions

This chapter summarizes the key outcomes of the work carried out during the development of the platform, reflecting on the objectives that were achieved, the limitations encountered, and the possible directions for future work. It also provides a final assessment of the solution's relevance, technical soundness, and potential for continued evolution. The reflections presented here are based on both the implementation process and the testing and evaluation phases described in earlier chapters.

6.1 Achieved Objectives

The development of this project led to the full or partial achievement of the thesis objectives, with some areas strategically scoped for future work due to complexity and resource constraints. The following summary outlines the extent to which each objective was fulfilled and the reasoning behind these outcomes.

O1: Investigate and analyze the state of the art in smart energy systems, interoperability approaches, and microservices-based architectures to support the proposed solution.

Fully achieved.

A comprehensive review of the literature and current technological landscape was conducted to understand the challenges and opportunities in smart energy systems. The investigation focused on interoperability, the use of microservices-based architectures in distributed environments, and strategies for citizen participation as prosumers. This foundational analysis directly informed the architectural and design decisions made during the project, ensuring that development choices were grounded in current scientific knowledge and best practices.

O2: Design and implement a microservices-based architecture for APIs that enables interoperability between diverse smart energy systems.

Partially achieved.

While a fully distributed microservices architecture was originally envisioned, the final solution consists of a modular, *microservices-ready* monolithic backend. This approach was chosen to prioritize maintainability, testability, and clear domain logic within the time and scope of the project. The backend was architected following modern design principles - such as separation of concerns, domain-driven design, and layered architecture - ensuring that it can be decomposed into microservices with minimal refactoring in future iterations.

O3: Develop APIs that allow citizens to participate as prosumers in energy communities, enabling them to manage energy production, consumption, and storage.

Partially achieved.

Core API functionalities were successfully implemented to support citizen participation as prosumers. Features include endpoints for uploading energy data, viewing consumption and production metrics, and interacting with community-related elements. While full support for energy storage management was not implemented, the groundwork for prosumer engagement is well established, aligning with the project's focus on enabling user interaction and decentralized participation.

O4: Implement functionalities to support peer-to-peer (P2P) energy trading, energy storage management, and load balancing within smart energy systems.

Not achieved.

Due to the technical and operational complexity of real-time trading, storage orchestration, and load balancing, these advanced functionalities were not implemented in this phase.

O5: Evaluate the performance, scalability, and security of the developed APIs, ensuring they can support the integration of international smart energy systems.

Partially achieved.

Throughout development, functional and unit tests were applied to ensure the correctness and reliability of the core logic. While scalability and security were considered in architectural decisions - such as stateless design and modularity - formal performance benchmarking and penetration testing were beyond the project's scope. These remain important areas for future investigation to fully meet this objective.

O6: Conduct an ethical analysis of the data management practices within the developed APIs, ensuring privacy and equity for all users.

Partially achieved.

Ethical concerns were addressed primarily through the implementation of secure password handling, authentication, and role-based access control. These measures ensure baseline user privacy and data protection. However, a deeper analysis of equity and long-term data ethics - such as algorithmic fairness or differential access - was not conducted in this phase and warrants further study.

O7: Provide recommendations for the future development and adoption of microservices-based APIs for smart energy systems, based on the findings of the research.

Fully achieved.

Drawing from the design and implementation experience, the project presents actionable recommendations for future development and adoption of microservices-based APIs. These include suggestions for gradually evolving the current monolithic architecture, enhancing prosumer engagement, and integrating advanced features such as P2P trading and dynamic load balancing. These insights aim to support continued innovation in citizen-centric, interoperable smart energy systems.

6.2 Contributions

Throughout the development of this project, the student made significant contributions across multiple dimensions - including literature research, system design, architectural and technological decision-making, practical implementation, and consistent communication with both the development team and project stakeholders.

The initial phase began with an in-depth exploration of the **state of the art**, where the student formulated key research questions to guide the investigation. These questions helped define the project's scope and expected outcomes. A structured literature review was conducted using reputable academic databases such as IEEE Xplore, Google Scholar, and the ACM Digital Library, focusing on backend architectures, smart energy platforms, and interoperability standards. The analysis of these studies enabled the student to compare architectural paradigms - primarily monolithic versus microservices-based systems - and assess their suitability within the context of the project. This led to the collaborative decision to adopt a **microservices-ready monolithic architecture**, balancing scalability with simplicity given the project's constraints and timeline. These architectural choices were validated in consultation with the project supervisor and the SATCOMM research team to ensure alignment with broader project goals.

In terms of **project coordination**, the student was actively involved in meetings with the supervisor and SATCOMM stakeholders - both onsite at the **GECAD research center** and remotely. These regular interactions helped clarify functional requirements and facilitated continuous alignment among team members. Early collaboration with a GECAD researcher who had previously worked on a similar platform provided valuable context, including technical challenges, architectural decisions, and implementation strategies. The student participated in defining the technology stack, selecting frameworks, and creating a structured, task-oriented development roadmap using GitHub Issues. These tasks were estimated, tracked, and organized to support an agile development process, enabling frequent progress reviews and iterative delivery.

From a **design perspective**, the student contributed to the development of several key diagrams, including logical and physical architecture diagrams, the domain model, and other relevant visual artifacts. These diagrams were essential for visualizing the system's structure, flow, and data interactions, allowing the team to solidify their implementation approach. This well-defined design phase helped streamline the implementation by serving as a clear reference point, reducing uncertainty and minimizing rework during development.

Regarding **implementation**, the student was responsible for several core backend tasks, particularly those related to the API and service layer. The implementation followed a layered structure, with the student actively contributing across all major components - including routing, controllers, services, repositories, domain logic, and object mappers. This hands-on experience provided a deep understanding of the backend architecture, system behavior, inter-layer communication, and design rationale.

In order to validate the developed features, the student also took a proactive role in **testing**, implementing a comprehensive suite of unit tests, end-to-end tests, and API functional tests. These tests were designed to verify both isolated components and complete user flows, ensuring the correctness, reliability, and expected behavior of the system. By covering key use cases across the backend, this testing effort minimized unexpected behavior and significantly improved the robustness of the overall solution. The tests not only supported debugging during development but also laid the foundation for future automation and continuous integration.

In addition, the student contributed to the development of robust **documentation**, which extended beyond inline code comments using Javadoc-style conventions. Centralized documentation was created to assist the development team in onboarding, troubleshooting, and maintaining consistency across modules. This proved particularly useful for supporting team communication and ensuring knowledge transfer throughout the project.

The student also played an active role in the **writing of this dissertation**, collaborating closely with the advisor to present the research process, technical implementation, and results in a clear and structured manner. This involved organizing the content across all chapters, articulating the theoretical foundations and practical work, and ensuring the coherence and academic rigor of the final document. Through this process, the student developed not only technical and research skills but also improved scientific communication, which is essential for effectively conveying complex ideas in academic and professional contexts.

In summary, the student played a central role in the project's success by contributing to research, system design, architectural decisions, backend implementation, team coordination, and documentation - delivering a well-structured and scalable solution aligned with the strategic goals of the SATCOMM initiative.

6.3 Limitations

Despite the significant progress made throughout the development of this project, it is important to acknowledge certain limitations that shaped its trajectory. These challenges, however, did not prevent the successful implementation of a solid foundation for the platform and served as learning opportunities that reinforced the value of adaptability and perseverance.

One of the main constraints was the limited availability of the student, who balanced the demands of this academic work with professional responsibilities in a full-time job. While this naturally reduced the time available for deep technical exploration or extended feature development, it also demonstrated the student’s ability to manage time effectively, prioritize tasks, and focus on delivering key architectural and functional elements.

The complexity of the platform also introduced challenges, particularly given its ambition to offer a well-structured backend, RESTful APIs, and frontend integration - all in line with modern architectural practices. Rather than scaling down the scope entirely, the project adopted a phased approach, focusing first on delivering a microservices-ready monolithic backend, while ensuring that future evolution into a distributed system would be seamless.

Additionally, team collaboration was occasionally impacted by availability mismatches, as members were in different stages of their academic and professional lives. While this limited the frequency of synchronous meetings, it encouraged independent contributions, asynchronous coordination, and the use of clear documentation to maintain alignment across the development process.

6.4 Future Work

While this project successfully delivered a solid and scalable foundation for a backend platform serving smart energy communities, several areas remain open for future enhancement and development.

A natural next step is the evolution from a microservices-ready monolith to a fully distributed microservices architecture. Although the current structure follows domain-driven design and clear module boundaries, splitting individual domains (such as user management, role management, and energy operations) into independently deployable services would further improve scalability, fault tolerance, and team autonomy.

Another important direction is the implementation of advanced energy-related features, including peer-to-peer (P2P) energy trading, energy storage management, and load balancing. These functionalities were envisioned in the original objectives but were not prioritized due to the project's time constraints. Their implementation would elevate the platform from a user and access management system to a fully functional smart energy management solution.

From a security and compliance standpoint, enhancing data protection mechanisms - such as detailed audit logging, fine-grained access control, and full GDPR compliance - would further strengthen the platform's readiness for production environments involving sensitive user and energy data.

Additionally, the introduction of automated testing and CI/CD pipelines would improve development agility and deployment reliability. While unit and functional tests have been implemented, expanding the test coverage to include integration and load testing, and integrating these into an automated pipeline, would bring greater confidence in system quality over time.

On the integration side, connecting the platform to real-world smart energy devices or external APIs (such as IoT platforms or energy grid data sources) would validate the system's interoperability and further demonstrate its practical applicability.

6.5 Scientific Research Article

As part of the work developed in this thesis - and with the aim of consolidating knowledge, promoting scientific dissemination, and gaining experience in sharing technical research - a scientific article was produced, titled "Development of Microservices-Based APIs for Smart Energy Systems". This article was submitted, peer-reviewed, and accepted for publication at the International Conference on Business and Technology (ICBT'Cairo'2025), a reputable event that fosters the intersection of technological innovation and real-world applications across global contexts.

The article presents the core contributions of this project, including the architectural approach, API design, and the modular, microservices-ready backend developed to support interoperability and citizen participation in energy communities. A copy of the published article is included in Appendix A of this document.

6.6 Final Assessment

The experience of developing a platform aimed at empowering smart energy communities through modern backend architecture and API integration proved to be both technically enriching and personally rewarding. This project represented not only a culmination of academic knowledge, but also a real opportunity to apply professional best practices in a complex, realistic scenario.

Despite the inherent challenges - such as managing a demanding work schedule alongside the academic requirements of the thesis - the process was marked by perseverance and continuous learning. Building a microservices-ready backend solution, integrating layered abstractions, applying domain-driven design, and ensuring testability and extensibility provided a comprehensive and hands-on learning experience in modern backend development.

Collaboration with the team, though occasionally limited by differing availabilities, was constructive and encouraging. Each contribution helped shape a system that is technically solid and architecturally prepared for future expansion, even if not all initial ambitions could be fully realized within the available timeframe.

The work carried out fostered a deeper understanding of software design principles, architectural scalability, and the real-world complexities of integrating technology with the energy sector. It also demonstrated the value of structured planning, modular thinking, and thoughtful prioritization in achieving impactful results with limited resources.

This project marked a significant milestone in the student's growth as a software engineer. It combined academic rigor with practical experience, resulting in a solution that is both functionally meaningful and technically robust - ready to evolve further in the service of future smart energy ecosystems.

Bibliography

[1] S. Aman, Y. Simmhan, and V. Prasanna, “Energy Management Systems: State of the Art and Emerging Trends,” 2013.

[2] Z. Lyu, H. Wei, X. Bai, and C. Lian, “Microservice-Based Architecture for an Energy Management System,” *IEEE Syst J*, vol. 14, no. 4, pp. 5061–5072, Dec. 2020, doi: 10.1109/JSYST.2020.2981095.

[3] A. Taik, B. Nour, and S. Cherkaoui, “Empowering Prosumer Communities in Smart Grid with Wireless Communications and Federated Edge Learning,” Apr. 2021, doi: 10.1109/MWC.017.2100187.

[4] M. Söylemez, B. Tekinerdogan, and A. K. Tarhan, “Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review,” Jun. 01, 2022, *MDPI*. doi: 10.3390/app12115507.

[5] Gantt.com, “What is a Gantt Chart?” Accessed: Jan. 04, 2025. [Online]. Available: https://www.gantt.com/

[6] Atlassian, “The Agile Coach.” Accessed: Dec. 29, 2024. [Online]. Available: https://www.atlassian.com/agile

[7] S. Laoyan, “What is Agile methodology? (A beginner’s guide).” Accessed: Dec. 29, 2024. [Online]. Available: https://asana.com/pt/resources/agile-methodology

[8] M. J. Page *et al.*, “The PRISMA 2020 statement: An updated guideline for reporting systematic reviews,” *PLoS Med*, vol. 18, no. 3, pp. e1003583-, Mar. 2021, [Online]. Available: https://doi.org/10.1371/journal.pmed.1003583

[9] D. L. Dinesha and P. Balachandra, “Conceptualization of blockchain enabled interconnected smart microgrids,” Oct. 01, 2022, *Elsevier Ltd*. doi: 10.1016/j.rser.2022.112848.

[10] *A System Architecture for Software-Defined Industrial Internet of Things*. 2015.

[11] L. Kasper, F. Birkelbach, P. Schwarzmayr, G. Steindl, D. Ramsauer, and R. Hofmann, “Toward a Practical Digital Twin Platform Tailored to the Requirements of Industrial Energy Systems,” *Applied Sciences (Switzerland)*, vol. 12, no. 14, Jul. 2022, doi: 10.3390/app12146981.

[12] A. Ghodake, F. Shaikh, D. Shukla, A. Shaha, and S. Sadvilkar, “Backend Development for E-Commerce Application,” 2024, doi: 10.1109/ICCUBEA61740.2024.10774978.

[13] A. Moin, “Data Analytics and Machine Learning Methods, Techniques and Tool for Model-Driven Engineering of Smart IoT Services,” in *Proceedings - International Conference on Software Engineering*, IEEE Computer Society, May 2021, pp. 287–292. doi: 10.1109/ICSE-Companion52605.2021.00130.

[14] L. N. Lévy, J. Bosom, G. Guerard, S. Ben Amor, M. Bui, and H. Tran, “DevOps Model Appproach for Monitoring Smart Energy Systems,” *Energies (Basel)*, vol. 15, no. 15, Aug. 2022, doi: 10.3390/en15155516.

[15] N. Kabbara *et al.*, “Towards Software-Defined Protection, Automation, and Control in Power Systems: Concepts, State of the Art, and Future Challenges,” Dec. 01, 2022, *MDPI*. doi: 10.3390/en15249362.

[16] S. Javed, “Approach Towards Engineering Microservice-Oriented Composable Ecosystems for Smart Industries,” 2023.

[17] G. Adeyemo, “A Cloud-based Framework for Smart Grid Data, Communication and Co-simulation,” 2021.

[18] J. Lukkarinen, “Service-Oriented Architecture as an Enabling Technology for Sector Integration,” 2023.

[19] N. Sakib, E. Hossain, and S. I. Ahamed, “A Qualitative Study on the United States Internet of Energy: A Step towards Computational Sustainability,” *IEEE Access*, vol. 8, pp. 69003–69037, 2020, doi: 10.1109/ACCESS.2020.2986317.

[20] A. Aghazadeh Ardebili, M. Zappatore, A. I. H. A. Ramadan, A. Longo, and A. Ficarella, “Digital Twins of smart energy systems: a systematic literature review on enablers, design, management and computational challenges,” Dec. 01, 2024, *Springer Nature*. doi: 10.1186/s42162-024-00385-5.

[21] N. Mohamed, J. Al-Jaroodi, I. Jawhar, S. Lazarova-Molnar, and S. Mahmoud, “SmartCityWare: A service-oriented middleware for cloud and fog enabled smart city services,” *IEEE Access*, vol. 5, pp. 17576–17588, Jul. 2017, doi: 10.1109/ACCESS.2017.2731382.

[22] A. Liberati *et al.*, “The PRISMA Statement for Reporting Systematic Reviews and Meta-Analyses of Studies That Evaluate Health Care Interventions: Explanation and Elaboration,” *PLoS Med*, vol. 6, no. 7, pp. e1000100-, Jul. 2009, [Online]. Available: https://doi.org/10.1371/journal.pmed.1000100

[23] American Journal Experts, “How to Create a PRISMA Flow Diagram.” Accessed: Dec. 23, 2024. [Online]. Available: https://www.aje.com/arc/how-to-create-prisma-flow-diagram/

[24] D. Moher, A. Liberati, J. Tetzlaff, D. G. Altman, and T. P. Group, “Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement,” *PLoS Med*, vol. 6, no. 7, pp. e1000097-, Jul. 2009, [Online]. Available: https://doi.org/10.1371/journal.pmed.1000097

[25] Atlassian, “Microservices Architecture.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture

[26] I. Nuñez, C. Rovetto, E. Cruz, A. Smolarz, D. Concepcion, and E. E. Cano, “Design of a microservices-based architecture for residential energy efficiency monitoring,” *International Journal of Electronics and Telecommunications*, vol. 70, no. 4, pp. 1089–1098, 2024, doi: 10.24425/ijet.2024.152511.

[27] GeeksforGeeks, “Monolithic Architecture - System Design.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.geeksforgeeks.org/monolithic-architecture-system-design/

[28] Atlassian, “Microservices vs. monolithic architecture.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[29] Amazon Web Services (AWS), “What is Java?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/what-is/java/?nc1=h_ls

[30] Coursera, “C# Programming: What It Is, How It’s Used + How to Learn It,” 2024, Accessed: Dec. 24, 2024. [Online]. Available: https://www.coursera.org/articles/c-sharp

[31] Amazon Web Services (AWS), “What is Python?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/what-is/python/

[32] Coursera, “Go Programming Language.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.coursera.org/articles/go-programming-language

[33] Amazon Web Services (AWS), “What is Javascript (JS)?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/what-is/javascript/

[34] GeeksforGeeks, “Introduction to TypeScript.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-typescript/

[35] VMware Tanzu, “Sprint Boot.” Accessed: Dec. 24, 2024. [Online]. Available: https://spring.io/projects/spring-framework

[36] Microsoft, “What is ASP.NET Core?” Accessed: Dec. 24, 2024. [Online]. Available: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet-core

[37] Sanity.io, “React.js overview.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.sanity.io/glossary/react-js

[38] Angular Documentation, “What is Angular?” Accessed: Dec. 24, 2024. [Online]. Available: https://v17.angular.io/guide/what-is-angular

[39] Amazon Web Services (AWS), “What is a RESTful API?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/what-is/restful-api/?nc1=h_ls

[40] gRPC Documentation, “Introduction to gRPC.” Accessed: Dec. 24, 2024. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/

[41] VMware, “What is a Message Broker?” Accessed: Dec. 24, 2024. [Online]. Available: https://www.vmware.com/topics/message-brokers

[42] Hygraph, “What is GraphQL?”, Accessed: Dec. 24, 2024. [Online]. Available: https://hygraph.com/learn/graphql

[43] Amazon Web Services (AWS), “What is PostgreSQL?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/rds/postgresql/what-is-postgresql/

[44] J. Erickson, “MySQL: Understanding What It Is and How It’s Used.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.oracle.com/mysql/what-is-mysql/

[45] J. Erickson, “What Is MongoDB? An Expert Guide.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.oracle.com/pt/database/mongodb/

[46] Apache Cassandra, “Cassandra Basics.” Accessed: Dec. 24, 2024. [Online]. Available: https://cassandra.apache.org/_/cassandra-basics.html

[47] IBM, “What is Redis?” Accessed: Dec. 24, 2024. [Online]. Available: https://www.ibm.com/topics/redis

[48] IONOS, “What is InfluxDB?” Accessed: Dec. 24, 2024. [Online]. Available: https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-influxdb/?srsltid=AfmBOooeatDUp-gxur4QwKkRA4sY932UkHAbTPz8uZ3_fTOXASjMplAa

[49] Timescale, “Timescale is Postgres made powerful.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.timescale.com/

[50] I. Nuñez, C. Rovetto, E. Cruz, A. Smolarz, D. Concepcion, and E. E. Cano, “Design of a microservices-based architecture for residential energy efficiency monitoring,” *International Journal of Electronics and Telecommunications*, vol. 70, no. 4, pp. 1089–1098, 2024, doi: 10.24425/ijet.2024.152511.

Appendix A – Scientific Research Article

Development of Microservices-based APIs for Smart Energy Communities

Costa, R.
Engineering School of the Porto Polytechnic, Portugal
rodrigesio333@gmail.com

Ramos, S.
Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development, Engineering School of the Porto Polytechnic, Portugal
scr@isep.ipp.pt

Soares, J.
Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development, Engineering School of the Porto Polytechnic, Portugal
jan@isep.ipp.pt

Abstract. The energy transition toward decentralized, sustainable communities demands platforms capable of managing distributed energy data and user roles effectively. This paper presents the development of a microservices-ready backend architecture for the Energy Communities Platform—a web-based tool designed to support community energy management. Using Domain-Driven Design and Clean Architecture principles, the backend integrates with a React frontend, a PostgreSQL database, and an external Python-based optimization algorithm to automate energy distribution among prosumers. The architecture emphasizes modularity, scalability, and maintainability, while functional and structural tests validate its robustness. The resulting system enables secure user management, energy tracking, and optimized distribution workflows, laying the groundwork for future deployment in real-world smart grid contexts.

Keywords: Smart energy systems, microservices, clean architecture, domain-driven design, backend development, prosumer energy optimization

1 Introduction

As the world shifts toward decentralized and sustainable energy models, digital platforms must evolve to support community-level energy management. Prosumers—users who both produce and consume energy—require infrastructure to monitor, share, and optimize energy usage collectively. The Energy Communities Platform (ECP) was developed to meet this demand by providing a scalable backend system tailored for smart energy communities. This article presents the design and implementation of the backend, which integrates multiple technologies and follows a modular, future-proof architecture.

Figure 40 - Page 1 of the published article

2 Background and Motivation

Despite growing adoption of smart energy systems, interoperability across platforms remains a major obstacle. Many existing applications are isolated and lack integration, particularly across international boundaries, limiting the potential for cross-border energy trading, coordinated energy management, and optimal use of energy storage.
A key gap lies in the absence of standardized, flexible, and scalable APIs capable of enabling seamless communication between heterogeneous systems. Additionally, most current solutions do not effectively engage citizens as prosumers—active participants in both energy production and consumption—which is central to modern energy policy.
This project addresses these challenges by developing a **monolithic, microservices-ready backend platform** designed to support interoperable APIs for smart energy systems. The system enables integration with external optimization tools and supports core functionalities such as energy negotiation, load management, and storage coordination. By promoting modularity and scalability, the solution lays a foundation for future microservices deployment and broader citizen participation in decentralized energy ecosystems.

3 Objectives

This project aims to design and implement **microservices-based APIs** to enhance interoperability in smart energy systems and enable citizen participation as prosumers. The main objectives are:

- **O1:** Develop a modular, microservices-ready API architecture to integrate diverse energy systems across regions and technologies.
- **O2:** Create APIs that empower citizens to manage their energy production, consumption, and storage within energy communities.
- **O3:** Implement features for peer-to-peer energy trading, storage management, and load balancing to support decentralized energy markets.
- **O4:** Evaluate the APIs for performance, scalability, and security in realistic scenarios, ensuring robustness for international deployment.
- **O5:** Ensure ethical data management practices, focusing on privacy, security, and equitable access to energy services.
- **O6:** Provide recommendations and best practices for future adoption and evolution of microservices-based APIs in the energy sector.

4 State of the Art

Smart energy platforms are increasingly complex due to distributed infrastructures, diverse devices, and real-time responsiveness demands. This has amplified the architectural debate between monolithic and microservices approaches. While monolithic systems offer simplicity in early development, they often lack scalability and fault isolation. Microservices, though more scalable and modular, introduce

Figure 41 - Page 2 of the published article

significant operational overhead. As a compromise, **microservices-ready monoliths** have emerged—modular architectures designed for future decomposition, offering a balance between maintainability and scalability. [1]

RESTful APIs have become essential in exposing backend functionality for energy systems, supporting integration with devices, clients, and third-party services. They follow web standards (HTTP, JSON) and promote interoperability. Tools like Swagger and Postman enhance API development and testing, though careful design is needed to ensure security and performance in critical systems. [2]

Modern practices like **Domain-Driven Design (DDD)** and **clean architecture** further improve modularity, aligning backend logic with energy domain boundaries. Frameworks such as Express.js and Prisma simplify the development of modular, testable APIs.

Database selection also plays a key role. **MySQL** [3] and **PostgreSQL** [4] are strong choices for transactional and structured data. **MongoDB** [5] offers schema flexibility for dynamic datasets, while **Cassandra** [6] excels at handling high-throughput time-series data at scale.

Despite ongoing innovation, many platforms still lack standardized, extensible APIs and fall short in enabling prosumer participation and interoperability. This project addresses these gaps through a modular, RESTful backend architecture that is monolithic yet designed for future microservices deployment—supporting integration, scalability, and citizen-centric energy management.[7][8]

5 System Requirements

5.1 Functional Requirements

The system supports key features necessary for managing smart energy communities:

- **User and Role Management**: Registration, authentication, profile updates, and role assignment.
- **Community and Prosumer Operations**: Creation and management of communities and their members.
- **Energy Data Handling**: Upload and processing of structured energy data through an external optimization algorithm.
- **Battery Configuration**: Viewing and managing prosumer battery parameters.
- **Analytics and Reporting**: Generating energy usage and trading reports.
- **RESTful API Access**: Consistent, validated endpoints for all operations.

5.2 Non-Functional Requirements

The platform was designed with the following non-functional qualities:

- **Scalability**: Modular, microservices-ready architecture.
- **Maintainability**: Clean code structure using DDD and interface-driven layers.
- **Security**: Role-based access control and request validation.

Figure 42 - Page 3 of the published article

- **Performance**: Low-latency responses and optimized database queries.
- **Usability**: Intuitive API design, with testing via Postman and unit tests.
- **Interoperability**: Standards-compliant REST APIs and JSON formats.

6 Design

6.1 Domain Model

To ensure a maintainable backend, the domain model (Figure 1) follows Domain-Driven Design (DDD) principles. It structures core logic around aggregates with clear boundaries, promoting encapsulation, consistency, and separation of concerns. This approach aligns with real-world energy community operations while supporting future scalability.

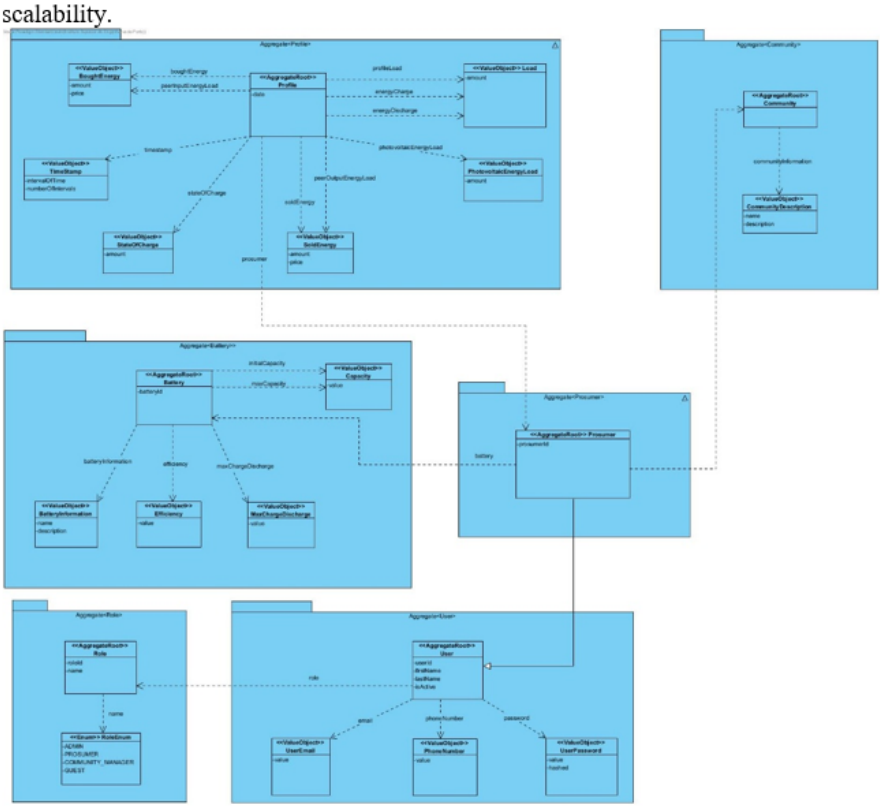

Figure 1 – Domain Model

Figure 43 - Page 4 of the published article

6.2 Architecture Overview

The system architecture consists of three main containers:

- **Frontend** (React) – provides user interface and API interaction.
- **Backend** (Node.js) – handles business logic and system orchestration.
- **Optimization Algorithm** – Python-based module for energy redistribution logic.

The backend exposes a REST API, interacts with a PostgreSQL database, and invokes the optimization service via HTTP.

The Level 2 Logical View provides a high-level structural overview of the Energy Communities Platform system, as illustrated in Figure 2.

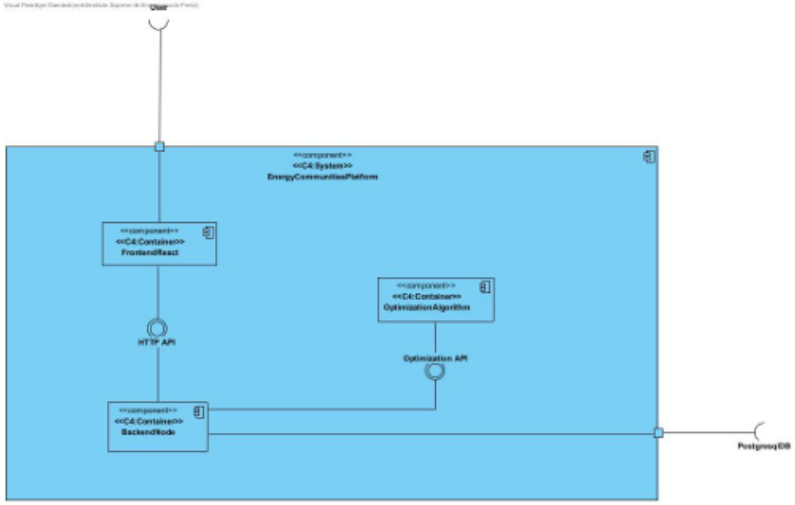

Figure 2 - Logical View Level 2

The Level 3 Logical View offers a detailed look into the internal structure of the BackendNode container, breaking it down into its main components and illustrating how they interact to fulfill backend responsibilities, as shown in Figure 3.

Figure 44 - Page 5 of the published article

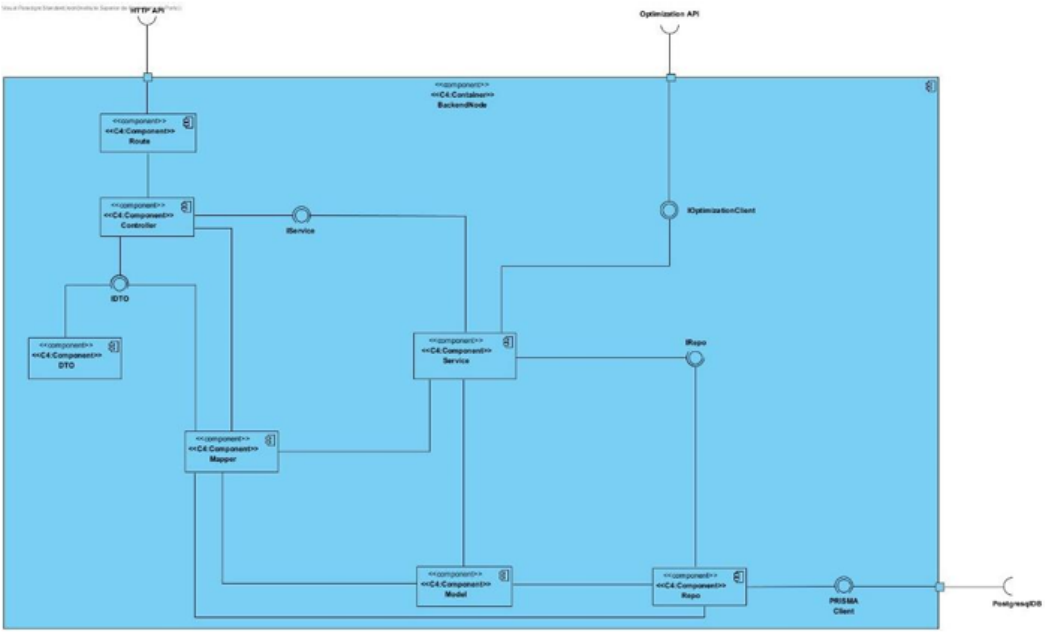

Figure 3 – Logical View Level 3

7 Implementation

7.1 Backend Architecture

The backend follows Clean Architecture and Domain-Driven Design (DDD). It separates concerns into layers: Route → Controller → Service → Domain Model → Repository. Each component adheres to an interface-driven design, supporting future microservice decomposition.

7.2 Integration with OptimizationAlgorithm

An external Python algorithm developed by a GECAD (Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development) researcher is integrated through a REST API. The backend invokes this component upon file upload to compute optimal energy exchanges within the community. Results inform storage usage and market transactions (buy/sell decisions).

Figure 45 - Page 6 of the published article

8 Platform Walkthrough

A community manager interacts with the platform through the following features:

- User Management – create, edit, and assign roles (admin, prosumer, etc.).
- Community & Prosumer Management – manage members and their batteries.
- Optimization File Upload – upload structured Excel files (96 intervals per day per prosumer) to trigger energy distribution.
- Analytics – view consumption, generation, and storage activity in different scopes (15-minute, hourly, daily).
- Dashboards – access summary KPIs and trends across energy usage, grid interactions, and storage behavior, as seen in Figure 4.

Figure 4 – Platform Dashboard Tab

9 Testing and Evaluation

To ensure system reliability and maintainability, a layered testing approach was used, validating components both in isolation and in integration—aligned with clean, modular backend design principles.

9.1 Unit Tests

Unit testing was applied to verify the behavior of individual components in isolation, with a particular focus on the core service logic.

9.2 API Functional Tests

To verify the correctness and robustness of the system's public-facing endpoints, a series of API functional tests were created using Postman. These tests were designed to simulate real client interactions with the backend and validate that the HTTP responses matched the expected behavior under various conditions.

Figure 46 - Page 7 of the published article

9.3 End to End Tests

End-to-End and Acceptance Testing were performed manually to validate full system functionality from a user perspective. A team member tested key flows—registration, authentication, profile updates, and role-based access—confirming that the system met its functional and business requirements.

9.4 Solution Evaluation

The implemented solution fulfills all planned functional requirements, including user registration, authentication, role management, and profile updates—validated through unit and API-level tests. The system was developed as a microservices-ready monolith, using modular design, DDD, and separation of concerns to enable future service extraction. This approach balances the simplicity of a monolith with the scalability and flexibility of microservices architecture.
Using the FURPS+ model, the solution satisfies:

- **Functionality** – through validated routes and role-based access.
- **Usability** – via a clean API and testable frontend.
- **Reliability** – ensured through type safety, DTOs, and service validations.
- **Performance** – supported by lightweight queries and efficient routing.
- **Supportability** – aided by modular structure and clear boundaries.

10 Conclusion

The project successfully met its core objectives, delivering a modular, microservices-ready monolithic backend focused on maintainability and domain logic. While full microservices deployment and advanced features like peer-to-peer trading were deferred, the architecture was designed to support them in future iterations. Key functionalities—such as user and role management, prosumer data processing, and community interaction—were implemented and validated through unit and API testing. Ethical aspects, including privacy and access control, were addressed at a foundational level. The work provides a scalable foundation and practical guidance for future development of citizen-centric energy platforms.

11 Future Work

Future work includes evolving the current microservices-ready monolith into a fully distributed architecture, enhancing scalability and service autonomy. Key features like peer-to-peer energy trading, load balancing, and storage management should be implemented to expand platform capabilities. Security can be improved through audit logging, fine-grained access control, and GDPR compliance. Introducing CI/CD pipelines and broader test coverage will boost deployment reliability. Lastly, integrating real-world IoT devices and external APIs will validate system interoperability and practical use.

Figure 47 - Page 8 of the published article

References

[1] Atlassian, “Microservices vs. monolithic architecture.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[2] Amazon Web Services (AWS), “What is a RESTful API?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/what-is/restful-api/?nc1=h_ls

[3] J. Erickson, “MySQL: Understanding What It Is and How It’s Used.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.oracle.com/mysql/what-is-mysql/

[4] Amazon Web Services (AWS), “What is PostgreSQL?” Accessed: Dec. 24, 2024. [Online]. Available: https://aws.amazon.com/rds/postgresql/what-is-postgresql/

[5] J. Erickson, “What Is MongoDB? An Expert Guide.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.oracle.com/pt/database/mongodb/

[6] Apache Cassandra, “Cassandra Basics.” Accessed: Dec. 24, 2024. [Online]. Available: https://cassandra.apache.org/_/cassandra-basics.html

[7] GeeksforGeeks, “Monolithic Architecture - System Design.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.geeksforgeeks.org/monolithic-architecture-system-design/

[8] Atlassian, “Microservices Architecture.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture

Figure 48 - Page 9 of the published article