DM

A Microservice-Based Digital Twin for Optimized Energy Management and Decarbonization of the Port of Funchal

MASTER DISSERTATION

Diogo Dominguez Paulino Reis Rodrigues
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

October | 2025

A Microservice-Based Digital Twin for Optimized Energy Management and Decarbonization of the Port of Funchal

MASTER DISSERTATION

Diogo Dominguez Paulino Reis Rodrigues
MASTER IN INFORMATICS ENGINEERING

SUPERVISOR
Amâncio Lucas de Sousa Pereira

CO-SUPERVISOR
Filipe Magno de Gouveia Quintal

Faculdade de Ciências Exatas e da Engenharia

Master of Science Degree in Informatics Engineering

A Microservice-Based Digital Twin for Optimized Energy Management and Decarbonization of the Port of Funchal

Diogo Dominguez Paulino Reis Rodrigues

Supervised by:

Dr. Amâncio Lucas de Sousa Pereira

Prof. Filipe Magno de Gouveia Quintal

Constituição do júri de provas públicas:

Dra. Karolina Baras, Presidente

Dr. Pedro Campos, Vogal

Dr. Lucas Pereira, Vogal

October, 2025

Resumo

Esta tese apresenta o desenvolvimento de uma ferramenta avançada de gestão de energia para o setor marítimo, materializada como um Digital Twin (Gémeo Digital) do Porto do Funchal. O sistema visa apoiar a transição energética do porto, nomeadamente a eletrificação e a otimização do consumo energético, contribuindo para a sua descarbonização. A solução proposta adota uma arquitetura moderna baseada em microsserviços, permitindo a integração modular de diversas componentes complexas, como a modelação energética de navios, a análise de fluxos de potência em corrente contínua (DC), e ferramentas de visualização interativa.

O Digital Twin integra dados em tempo real da infraestrutura portuária e dos navios, recorrendo a técnicas de modelação preditiva e machine learning para estimar perfis de consumo energético. São exploradas as vantagens das redes DC, incluindo o dimensionamento otimizado da rede elétrica e a avaliação de indicadores de desempenho (Key Performance Indicators (KPIs)) em termos de eficiência, custos e impacto ambiental, comparativamente com soluções tradicionais em corrente alternada (AC). A componente de visualização, desenvolvida com tecnologias web modernas, oferece aos stakeholders portuários uma interface intuitiva para análise de cenários, monitorização e apoio à decisão. Este trabalho demonstra a viabilidade e os benefícios de uma abordagem Digital Twin baseada em microsserviços para a gestão energética e sustentabilidade em infraestruturas portuárias complexas.

Palavras-chave: Digital Twin, Gestão de Energia, Descarbonização Marítima, Microsserviços, Redes DC.

Abstract

This thesis presents the development of an advanced energy management tool for the maritime sector, embodied as a Digital Twin of the Port of Funchal. The system aims to support the port's energy transition, particularly electrification and energy consumption optimization, contributing to its decarbonization. The proposed solution employs a modern microservice-based architecture, enabling the modular integration of diverse complex components, such as vessel energy modeling, Direct Current (DC) power flow analysis, and interactive visualization tools.

The Digital Twin integrates real-time data from port infrastructure and vessels, utilizing predictive modeling and machine learning techniques to estimate energy consumption profiles. The advantages of DC grids are explored, including optimized electrical network sizing and the evaluation of KPIs for efficiency, costs, and environmental impact, compared to traditional AC solutions. The visualization component, developed with modern web technologies, provides port stakeholders with an intuitive interface for scenario analysis, monitoring, and decision support. This work demonstrates the feasibility and benefits of a microservice-based Digital Twin approach for energy management and sustainability in complex port infrastructures.

Keywords: Digital Twin, Energy Management, Maritime Decarbonization, Microservices, DC Grids.

Acknowledgments

First and foremost, I extend my profound gratitude to my supervisor, Lucas Pereira, for his invaluable guidance, feedback, and support throughout this research. Your expertise and mentorship were instrumental in shaping this project and my development as a researcher.

This work was conducted within the framework of the European Union's Shift2DC project. I am grateful for the opportunity to contribute to this initiative and wish to thank all members of the project consortium for the collaborative environment. My appreciation also goes to the Universidade da Madeira and the Faculdade de Ciências Exatas e da Engenharia for providing the academic foundation for this work.

Finally, I wish to thank my family and friends for their constant encouragement, patience, and support during this journey. This thesis would not have been possible without them.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AC Alternate Current

AI Artificial Intelligence

API Application Programming Interface

DC Direct Current

DT Digital Twin

EMS Energy Management System

GUI Graphical User Interface

IoT Internet of Things

KPI Key Performance Indicator

REST Representational State Transfer

SSOT Single Source Of Truth

1 Introduction

1.1 Motivation: Port Electrification and the Transition to DC Grids

The global maritime industry, a significant contributor to greenhouse gas emissions, faces increasing pressure to transition towards cleaner operations to help fight climate change [1, 2]. Ports, as crucial parts in global trade and maritime logistics, are central to this decarbonization effort. Ports worldwide are implementing strategies to reduce their carbon footprint and improve environmental performance. Key initiatives include shore-side power (cold ironing), electrification of cargo handling equipment, and adoption of alternative fuels [3, 4]. Ports are also investing in renewable energy, infrastructure modernization, and digital technologies like Artificial Intelligence (AI)-driven analytics and blockchain for emissions tracking [5]. The Port of Funchal in Madeira, as the island's primary maritime gateway, is actively exploring these strategies to enhance its energy sustainability and reduce its environmental impact.

A promising technological branch supporting this transition is the adoption of DC distribution networks. Compared to traditional Alternate Current (AC) systems, DC grids offer higher efficiency, greater power density, and more straightforward integration with renewable energy sources, such as solar photovoltaics, and energy storage systems, which are vital components of modern energy ecosystems [6].

For ports, which are becoming complex energy hubs with diverse loads and local generation, DC technology presents a compelling case. However, this transition from well-established AC systems to new DC configurations requires careful planning, deep analysis, and advanced decision support tools to guarantee operational reliability, optimize investments, and manage the complex interactions between variable vessel loads, renewable generation, and grid stability.

1.2 Context: The Shift2DC Project and the Need for a Digital Twin

Recognizing the benefits of DC technology, the European Union's Shift2DC[1] project was established to accelerate its adoption by demonstrating its advantages through four demonstrators across Europe. These include two installations in Germany focusing on data centers and industrial applications, one in France examining building infrastructure, and a port demonstrator in Portugal. Each site is designed to test and validate medium voltage DC (MVDC) and low voltage DC (LVDC)

[1] https://shift2dc.eu/

solutions through a complete, top-down approach.

This thesis is situated within the Portuguese demonstrator of the Shift2DC project, focusing specifically on the Port of Funchal. To address the planning and analysis challenges outlined previously, the project identified the need for a DT. A DT, a virtual replica of a physical system, can provide a safe, simulated environment to analyze the port's energy ecosystem. This tool will support port authorities and stakeholders by enabling them to evaluate the economic and environmental impact of investing in DC grid infrastructure, simulate various vessel charging scenarios under different conditions, and optimize overall energy performance before committing to significant capital expenditure. The work conducted in this thesis serves as a foundational component in the development of this Port of Funchal DT.

1.3 Thesis Objectives

The primary goal of this thesis is to design, develop, and evaluate a DT for the Port of Funchal, focusing on energy management and supporting its decarbonization efforts, with a specific consideration for the potential of DC power systems. To achieve this, the following objectives have been defined:

- Implement a scalable and modular microservice-based architecture for the DT to ensure flexibility, maintainability, and the seamless integration of diverse data sources and analytical components.
- Integrate machine learning techniques to predict vessel energy consumption profiles, particularly for vessels with limited historical data.
- Integrate a model that simulates various DC power grid configurations suitable for this port application to assess the performance, operational efficiency, economic viability, and environmental impact.
- Develop an interactive visualization tool that offers stakeholders the ability to monitor real-time energy data, simulate operational scenarios, and make informed decisions regarding energy infrastructure and management strategies.

1.4 Document Structure

This thesis is organized into six chapters, each building upon the last to present the research conducted.

Chapter 2 reviews the pertinent literature, establishing the theoretical foundation for this work. It covers the core concepts of DTs, their application in the maritime sector, the principles of microservice architectures, and modern data visualization techniques for energy management. The chapter concludes by synthesizing the state-of-the-art and identifying the specific research gap this thesis addresses.

Chapter 3 details the methodology and high-level system design. It begins by defining the functional and non-functional requirements that guided the development. It then presents the overall three-tier, microservice architecture, outlining the roles of the Presentation Layer, Middleware Layer, and the individual microservices. The chapter concludes by discussing the primary system constraints.

Chapter 4 provides a detailed account of the system's practical development and implementation. It explains the integration of the core Python microservices for vessel energy modeling and DC power flow analysis via Representational State Transfer (REST) APIs. The development of the Node.js middleware is described, covering its role in request orchestration, real-time data streaming with WebSockets, and user authentication. The chapter then delves into the creation of the presentation layer, detailing the technology stack (React, Next.js, Deck.gl) and the implementation of key features, including the interactive 2.5D port visualization, data dashboards, and fault-handling mechanisms. Finally, it covers the system's deployment strategy using Docker for containerization and Nginx as a reverse proxy.

Chapter 6 is dedicated to the evaluation of the developed system. It presents the results of quantitative performance benchmarks, including latency, model accuracy, and concurrent user load tests. The system's reliability is assessed through a series of fault-tolerance tests simulating microservice and data stream failures. The chapter concludes with an in-depth discussion of these results, interpreting their significance, answering the central research questions, and acknowledging the limitations of the study.

Finally, Chapter 7 concludes the thesis. It summarizes the work, reiterates the main findings and contributions, and proposes a direction for future research, outlining a path forward for enhancing and expanding the Port of Funchal's DT.

2 Related Work

This chapter presents a review of the literature foundational to this thesis, beginning with a definition of the structured research methodology used to employ a rigorous survey of the three research fields studied. The investigation first establishes a broad understanding of Digital Twins (DTs), exploring their conceptual origins, widespread adoption across various industries, and specific applications within the maritime and port sectors. Following this, the chapter delves into the principles of microservice architectures, examining their suitability as a robust and scalable foundation for developing modular Digital Twin systems. Next, the discussion turns to effective data visualization techniques, which are critical for translating the data generated by a DT into usable information for energy management. The chapter concludes by synthesizing these distinct yet interconnected fields to pinpoint the specific research gap that this thesis aims to address: the development of an energy based Digital Twin for port environments, built upon a microservice framework.

2.1 Research Methodology

To ensure a rigorous and complete review, a structured research methodology was used, being developed through four main digital research tools:

ACM Library and *Google Scholar*, were the main research engines, where a varied range of interesting papers were found, using the different queries documented in table 1.

Research Rabbit served as the secondary discovery mechanism. This interesting web tool[2] enables the creation of paper collections and utilizes intelligent algorithmic suggestions. By inputting initial research papers, the platform generates recommendations for related works, including both antecedent and subsequent research, thereby expanding the research.

Obsidian was utilized for note taking and as a knowledge aggregator. Its unique capability to interconnect notes allowed for creating a vast knowledge network, allowing me to make conceptual connections across different research papers.

The previously mentioned selection protocol, used for the literature review, followed these steps:

1. Initial broad collection of academic papers and research articles

2. Filtering based on relevance to research objectives

[2] https://researchrabbitapp.com

Table 1: Search Queries by Research Category

Category	Search Queries
Digital Twins	"digital twin" AND (challenges OR risks OR implementation)
	"digital twin" AND (marine OR maritime OR shipping OR vessel)
	"digital twin" AND (port OR seaport OR terminal)
	"digital twin" AND (manufacturing OR industry OR smart manufacturing)
	"digital twin" AND (energy OR sustainability OR decarbonization)
	"digital twin" AND (healthcare OR medical)
	"digital twin" AND (smart city OR urban OR campus)
	"digital twin" AND (agriculture OR farming)
	"digital twin" AND (review OR survey OR "state of the art")
	"digital twin" AND (framework OR architecture OR platform)
	"digital twin" AND (IoT OR "Internet of Things" OR sensors)
	"digital twin" AND (big data OR analytics OR simulation)
Microservice Architecture	microservices AND (IoT OR "Internet of Things")
	microservices AND (architecture OR patterns OR design)
	microservices AND (migration OR monolithic OR transformation)
	microservices AND (challenges OR implementation OR adoption)
	microservices AND (smart city OR urban systems)
	microservices AND (scalability OR performance OR distributed)
	microservices AND (government OR enterprise OR industrial)
	"service oriented architecture" AND microservices
Data Visualization	"digital twin" AND (2D OR 3D)
	visualization AND (2D OR 3D)
	"3D visualization" AND (cultural heritage OR reconstruction)
	"digital twin" AND (visualization OR rendering OR display)
	"mixed reality" OR "augmented reality" AND visualization
	"data visualization" AND (techniques OR tools OR methods)
Combined Topics	"digital twin" AND microservices
	"digital twin" AND "microservice architecture"
	microservices AND "digital twin" AND platform
	"semantic microservices" AND "digital twin"
	"domain driven design" AND "digital twin"
	"service oriented" AND "digital twin" AND interoperability

3. Evaluation of each source's contribution to the research domain

4. Synthesis of a collection of high-impact, pertinent research materials

This approach ensured that the state-of-the-art review was not just a collection of sources but a carefully constructed web of knowledge that provided important insights into the research domains of digital twins and multilevel visualization.

2.2 Digital Twins in Research

The story of the concept of the Digital Twin began in recent years, and in its short lifespan, it has already altered how most industries now approach the design, monitoring, and optimization of complex systems and processes [7]. Initially proposed by Dr. Michael Grieves in 2002 as a model for Product Lifecycle Management (PLM), represented through the diagram shown in fig. 1 ,the DT was envisioned as a virtual counterpart to a physical system, containing a great number of information about its real-world twin, with both entities remaining dynamically interconnected throughout their entire lifecycle [7].

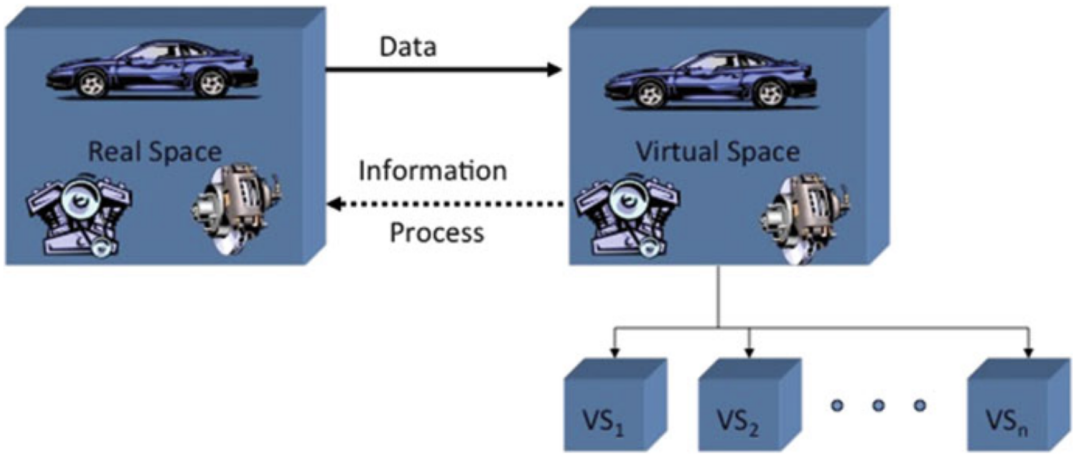


Figure. 1: Conceptual Idea for PLM by Dr. Michael Grieves, 2002.

This concept, originally termed the Mirrored Spaces Model and later the Information Mirroring Model, was formally adopted as the "Digital Twin" nomenclature with its inclusion in the book "Virtually Perfect" [7]. According to Dr. Grieves' framework, which is adopted within the Shift2DC project and this thesis, a DT is essentially a virtual representation of a physical asset that encap-

sulates all its critical information. Two primary classifications are identified: the Digital Twin Prototype, which is utilized during the design and pre-production phases to simulate and verify product functionality, and the Digital Twin Instance, which focuses on the operational lifecycle of a specific physical asset, continuously tracking its real-time conditions, historical performance, and future maintenance requirements. These DTs operate within a Digital Twin Environment, facilitating predictive analyses of future performance and interrogative assessments of current operational status [7]. This thesis will primarily focus on developing a DTI with predictive and interrogative capabilities for the Port of Funchal.

2.2.1 Adoption of The Digital Twin

The adoption of DT technology has recently been accelerated by advancements in certain technologies such as the Internet of Things (IoT) for real-time data acquisition, AI for predictive analytics and decision support, and high-performance computing (HPC) for complex simulations [8]. Consequently, DTs have found many applications across numerous sectors, including optimizing manufacturing processes [9–12], simulating urban development and infrastructure management in smart cities [13], transforming healthcare through predictive patient models [14], and improving maintenance strategies for critical infrastructure such as railways [15], among many other applications as cataloged by Attaran [16].

An early and influential exploration of DT potential was presented by Tuegel et al. [17], who laid the theoretical groundwork for a DT capable of predicting the structural life of aircraft, aiming to replace traditional, often simplified, life prediction methods with high-fidelity simulations. This conceptual work was advanced by Glaessgen et al. [18], who moved towards implementation by developing a DT for NASA and U.S. Air Force vehicles. This system integrated sensor data from onboard Integrated Vehicle Health Management (IVHM) systems, maintenance records, and historical fleet data to forecast vehicle health, remaining useful life, and mission success probability, even incorporating self-healing mechanisms and mission adjustment recommendations. NASA also prominently featured the DT concept in its technology roadmaps for future vehicle designs [19].

Despite its growing adoption, the DT field faces challenges, most notably a lack of standardization in definition and application. This inconsistency often leads to the term being confused with simpler digital models or simulations. Fuller et al. [8] attempted to address this by proposing dis-

tinct definitions for Digital Model, Digital Shadow, and Digital Twin, yet a universally accepted standardization remains an ongoing endeavor.

2.2.2 Digital Twins in the Maritime and Port Sector

Specifically within the maritime domain, the application of DTs is gaining some considerable momentum, offering a potential for enhancing operational efficiency, safety, and sustainability [20].

Research in this area is expanding, with studies exploring various applications and key technologies [21, 22]. For instance, Troupiotis-Kapeliaris et al. [23, 24] detailed the development of a DT for maritime fleet management, capable of performing both short- and long-term route forecasting and detecting events of interest for a global fleet. Their work highlights the use of machine learning techniques, historical data, and vessel-specific features to model ship movements and improve prediction accuracy.

Ports, as complex logistical and energy hubs, stand to benefit substantially from DT technology [25, 26].

Neugebauer et al. [25] noted that while full DT implementations in ports are still relatively uncommon, several leading ports are making significant advancements. Notable examples include the Ports of Hamburg [27].

These implementations often feature multi-layered systems that integrate real-time data collection from IoT sensors, modeling algorithms, and 3D visualization technologies. The DT developed at Shanghai Maritime University, for example, can mirror entire terminal operations in near real-time, facilitating monitoring, predictive maintenance, and operational optimization. The key enabling technologies for such port DTs include extensive IoT sensor integration, robust big data computing engines, and advanced rendering platforms like Unity 3D.

The work by Li et al. [28] on a DT for port facilities, utilizing multi-source data fusion and visualization such as floating stereoscopic displays, further illustrates the technological sophistication being applied, though this research also underscores the complexity involved in developing solutions for such multifaceted structures.

Despite this progress, challenges persist, particularly concerning data integration from diverse and often legacy systems, the previously mentioned lack of standardization, and ensuring the accurate and meaningful application of the DT concept to address specific port operational needs.

2.3 Microservice Architectures for Complex Systems

Microservices are essentially a software architectural style that structures a single application as a collection of small, autonomous services, each operating in its own process and communicating through lightweight mechanisms, often using HTTP resource APIs [29, 30]. This architecture first appeared as a possible solution to the inherent limitations of monolithic applications, which are large, complex software applications whose modules cannot be executed independently and rely on shared machine resources [31]. Monoliths frequently encounter challenges with maintainability, evolving complexity, prolonged downtimes for changes, sub-optimal deployment due to conflicting resource needs, restricted scalability, and technology lock-in [31].

In exact contrast, microservices emphasize cohesion and independence, where each service is designed to implement a single, strongly related functionality or business capability [32, 33].

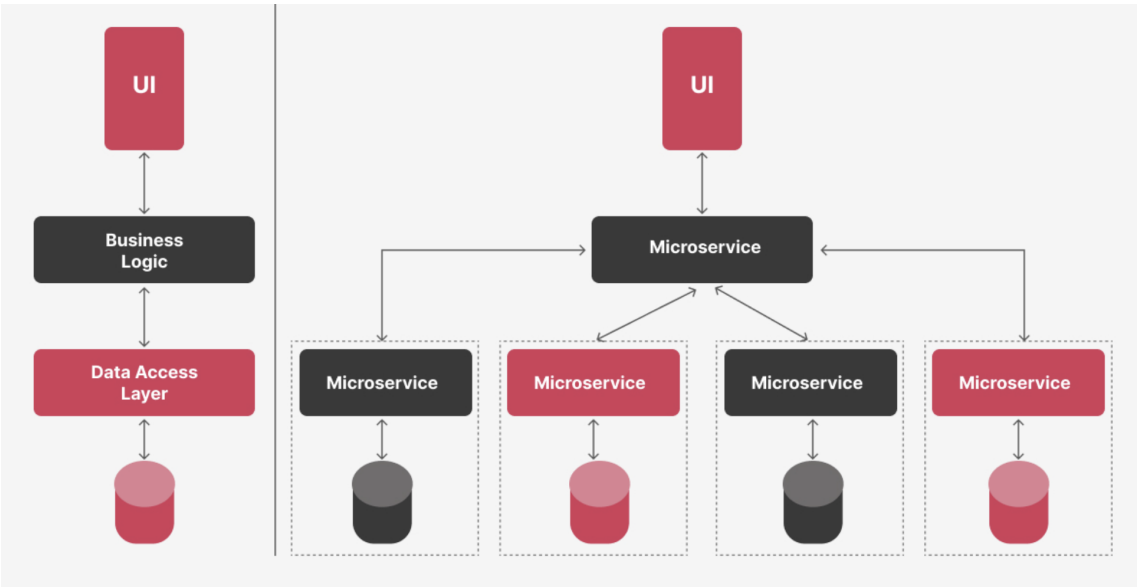


Figure. 2: Diagram on the structural differences between a Monolithic and a Microservice Architecture.

This design works well for independent deployment, scaling, and testing of individual services [29, 31]. A key characteristic is the disconnection of service interfaces from their internal implementations, ideally promoting dynamism, modularity, and reuse of services, and enabling parallel development by distinct teams [29, 31]. These teams are also typically small, and follow principles such as "you build it, you run it" [32, 34].

Microservices also push for decentralized data management, advocating that each microservice owns its data store, sometimes leading to situations where different database technologies are used across different services [30, 33], as is represented in fig. 2. Communication among microservices prioritizes "smart endpoints and dumb pipes", meaning that direct communication via simple protocols like REST, rather than complex middleware like Enterprise Service Buses are common in traditional Service-Oriented Architectures [29, 33, 34].

The adoption of microservices offers several worthwhile benefits, including significantly improved development and deployment agility, which helps with continuous integration and delivery, leading to faster updates and reduced system downtime [29, 31, 35]. It also enables optimal resource utilization by allowing individual services to be scaled based on their specific load, rather than duplicating an entire application, and provides considerable technological flexibility as developers can choose the most suitable languages and frameworks for each service [29, 31].

Despite these advantages, implementing microservices introduces its own set of challenges inherent to distributed systems. These include increased complexity in debugging, monitoring, and managing data consistency across distributed stores, as well as heightened security concerns due to a larger attack surface [31, 34]. Also, there is a recognized demand for advanced skills in distributed system development and DevOps, and the process of decomposing existing monolithic applications, particularly their data layers, can be exceptionally difficult [34–36]. It is important to note that performance can also be a concern due to the overhead of network communication compared to internal calls within a monolith [34].

Despite this, microservices are gaining traction in diverse fields, such as IoT and smart city applications, and are increasingly seen as an essential for organizations that want a quick application delivery [32, 37, 38]. Right now, ongoing research aims to address these open problems, focusing on areas like formal specification methods and enhancing security in microservice-based systems [31, 39].

2.3.1 Microservice Architecture and Digital Twins

DTs and the microservice architecture are two concepts that are usually intertwined, with microservices providing a powerful and flexible architectural option for realizing and managing complex DT implementations [40, 41]. The core connection stems from their shared principles of modularity, in-

dependent deployment, and focused functionality, enabling DTs to move beyond simple descriptive models to dynamic, behavioral representations of physical assets [40].

An important alignment exists in how both concepts deal with complexity: Digital Twins can be directly supported by the Domain-Driven Design (DDD) notion of Bounded Contexts (BCs), which are also a central abstraction in many microservice architectures [40]. This means that each part of a Digital Twin, representing a specific meaning or functionality, can be encapsulated within a Bounded Context, just as microservices are designed around such boundaries [40]. These architectural norms facilitate the design of loosely coupled DTs that can be developed and evolved independently, mirroring the benefits of microservices [40].

The Hexagonal Architecture, a style often recommended for microservices, is also particularly useful for architecting DTs, as it enforces separation of concerns and protects the integrity of the domain models within each Bounded Context [40]. The adoption of a microservices architecture for Digital Twins yields substantial benefits in terms of flexibility, agility, scalability, and maintainability [41, 42]. By breaking down the DT into small, independent services, developers can implement updates and modifications easily, without disrupting the entire system [41].

Furthermore, microservices promote fault isolation, preventing issues in one service from impacting others, thereby enhancing the overall reliability and resilience of a DT platform [41, 42]. Given the diverse range of protocols and technologies used in industrial and smart systems, guaranteeing communication between different DTs, or between DTs and their physical counterparts, is a major challenge [42, 43], and microservices address this by enabling a standardized method for assembling various components from multiple vendors, allowing DTs of different parts to interact as they would in the real world [43]. This way, real-time monitoring and control are supported by ensuring that data and information can be transferred from the physical twin to the digital twin, between different digital twins, and from the digital twin back to the real assets [43].

Effective communication patterns are also a cornerstone of microservice-based DTs [42], a proposed solution is message-oriented middleware (MOM) with event-based messaging, as it supports loose coupling between services and facilitates various communication paradigms such as one-to-many (e.g., for continuous sensor data streams), many-to-one (for data fusion), and request-reply for specific information retrieval [42]. This event-driven style is crucial for achieving near real-time

capabilities and managing stateful stream processing within DTs, even when services are designed to be stateless [42].

In practical applications, designing Digital Twins with microservices allows for a generalized framework suitable for diverse domains [44]. For example, in microservice architectures themselves, DTs can be built to monitor resource utilization, detect anomalies, and predict performance in complex systems like Kubernetes clusters, ultimately improving cluster management reliability and security [45].

In smart agriculture, Digital Twins implemented as collections of microservices, as shown in fig. 3, integrating systems and semantic web technologies, can autonomously manage, maintain, and optimize physical assets like crop fields, leading to more informed decision-making [44].

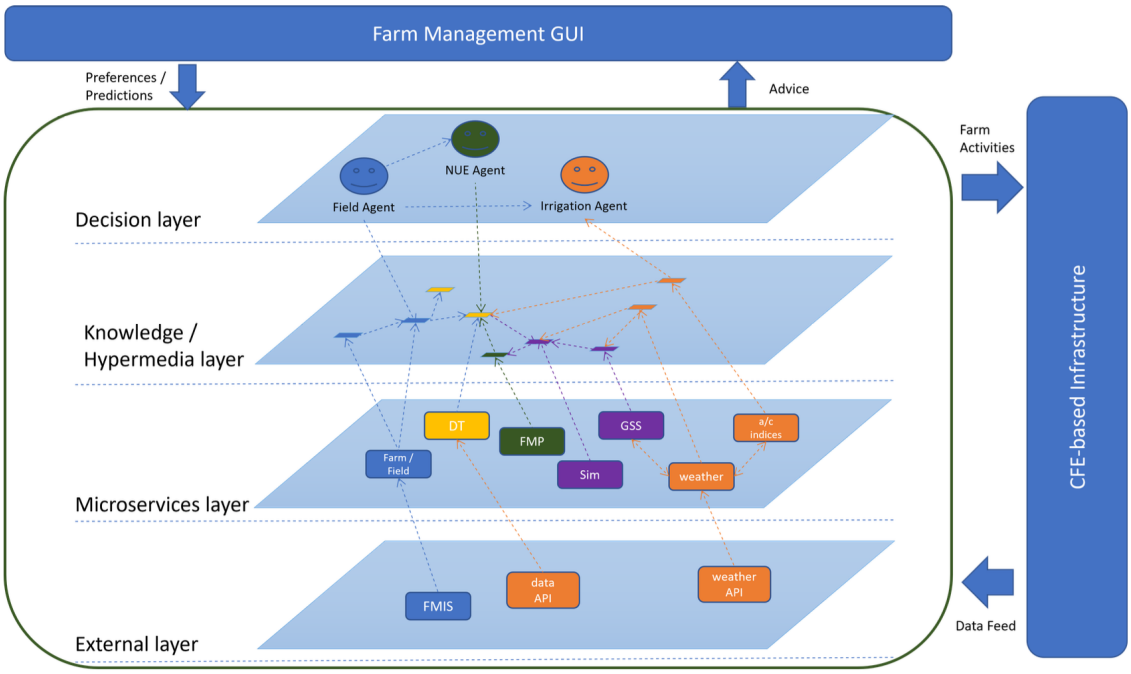


Figure. 3: Smart Agriculture DT Architecture [44]

In conclusion, from the ongoing body of work, it becomes evident that this architectural choice permits the DT platform to be adaptable, which allows for continuous updates and the incorporation of new functionalities and advanced systems with minimal disruption in future and past development cycles [41, 44].

2.4 Data Visualization for Energy Management

Providing effective data visualization options is one of the most important aspects to consider when developing a useful DT. This is the last stop between the tool and the end-user, and if this connection fails, all the previous work, as accurate or complex as it is, is of no use to take actionable insights to port authorities or stakeholders. As systems become increasingly complex, traditional methods of data presentation often fall short in conveying the aspects necessary for effective management and strategic planning [46].

Several works have explored advanced visualization techniques applicable to Digital Twins and energy systems. Lin et al. [47], for example, developed a DT using an interesting technology stack for campus visualization. The researchers utilized OpenStreetMaps for building footprints, employing the OSMnx Python package to generate geospatial data with 3D building models, using an average floor height of 3 meters. For data representation, they developed a custom web application using GitHub Pages, Danfo.js, and Plotly.js to dynamically generate popups with energy data modals. The digital twin was implemented in OpenCitiesPlanner, which allowed for color-coding buildings based on energy intensity and integrating various data sources, including building energy data. This approach enabled real-time, multi-dimensional visualization of campus infrastructure and energy performance. The result of this work can be seen in fig. 4.

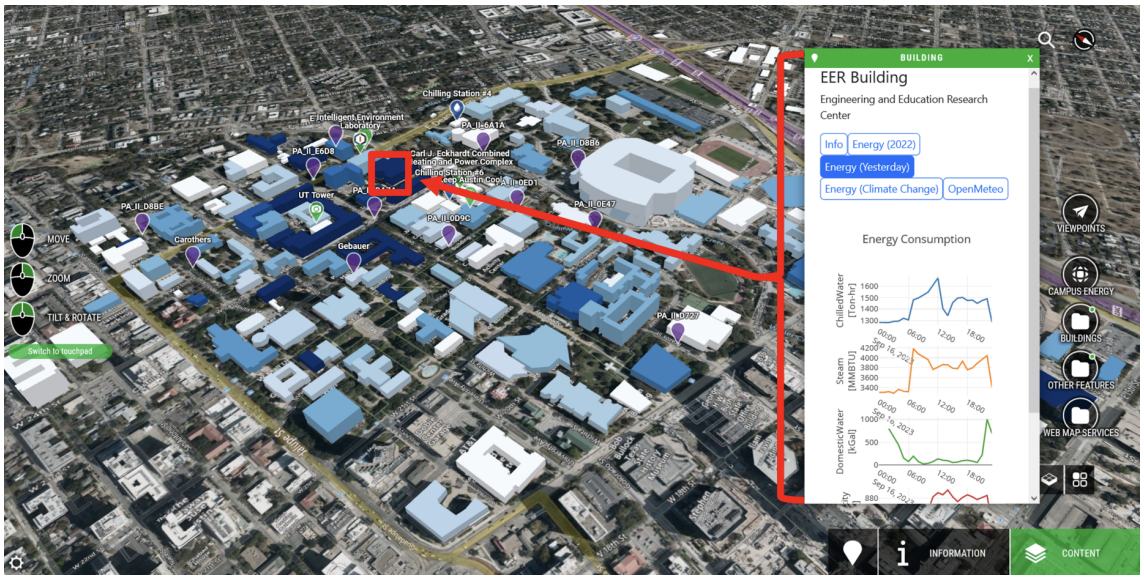


Figure. 4: Utwin Data Visualization [47].

Thelen et al. [13] created a "slim" DT platform for smart city visualization using CesiumJS for 3D geospatial rendering, CityGML for building models, and custom coloring techniques to represent energy consumption, complemented by overlay systems for detailed data and charts. Zhong et al. [48] employed Cesium with 3D Tiles for a hydraulic structure DT, incorporating particle systems for dynamic event simulation and real-time sensor data integration.

Other relevant examples include Simões et al. [49], who explored diverse web-based 3D technologies like Unreal Engine and WebRTC for collaborative DT visualization across the product lifecycle, and Pan et al. [50], who used Three.js for a virtual campus environment combined with clustering algorithms and various chart types (scatter plots, histograms) for data analysis visualization. Coppolino et al. [51] utilized OpenSearch for a geographically mapped dashboard with color-coded points indicating the status of distribution line components. Ramani et al. [52] integrated longitudinal thermal imagery with 3D urban models using deck.gl for microclimate studies. Themistocleous [53] demonstrated the use of photogrammetry from video data to create 3D models, a technique potentially useful for generating base models of port infrastructure.

Altogether, these examples highlight a trend towards interactive, multi-dimensional visualizations that combine geospatial context (2D maps and 3D models) with temporal data (time-series charts) and analytical overlays (heatmaps, network diagrams). For energy management, key visualization techniques include displaying time-series energy consumption trends, geospatial distribution of energy demand (e.g., at different berths), the topology of the electrical network with real-time status, and comparative dashboards for scenario analysis. The choice of visualization tools for this thesis (React, Next.js, Three.js/R3F, Deck.gl, Mapbox GL JS, D3.js, as detailed in Chapter 3) is informed by these advancements, aiming to provide an intuitive and powerful interface for stakeholders.

The reviewed works demonstrate interesting advancements, and although this is the case, none fully address the specific combination of DC infrastructure modeling, real-time responsiveness to dynamic energy loads, and the accessibility required for the Port of Funchal's Digital Twin. This creates a necessity for an hybrid approach, prioritizing both performance and detailed visualization.

2.5 Synthesis and Research Gap

A review of the literature reveals significant advancements in DT technology, microservice architectures, and data visualization. Monitoring technology is mature, with successful applications in the maritime sector for monitoring and operational enhancement. The microservice architecture offers a proven solution for managing complexity and interoperability in systems like a DT. Finally, data visualization techniques have evolved to support complex analytical requirements through interactive interfaces.

However, a synthesis of the reviewed literature reveals a noticeable research gap at the intersection of these domains. While port DTs for logistics are advancing, there is limited research on platforms focused specifically on energy management and decarbonization. Also, the application and validation of a microservice architecture for this specific context, integrating predictive vessel energy modeling with detailed simulations of on-shore DC power grids, remains largely unexplored. Consequently, there is a lack of, end-to-end solutions that provide port authorities with an tool to analyze the complex "ship-to-shore" energy consumption and support strategic decisions for electrification. This thesis addresses these limitations by proposing and developing a microservice-based DT for the Port of Funchal. The work provides three key research contributions:

- A new DT framework specifically designed for port energy management, which integrates real-time data, predictive analytics, and power flow simulation.
- A practical implementation of this framework using a microservice architecture that integrates heterogeneous components while maintaining responsiveness and fault tolerance.
- A real-time visualization interface that combines spatial (map-based) and tabular data displays, enabling monitoring and supporting port authorities in future decisions.

The novelty of this work lies in the development of a DT focused on energy management in a port setting, demonstrating that a microservice architecture is a valuable and viable option for this type of technology. Importantly, this research provides a tangible contribution to the Shift2DC project, as the developed system is designed to be deployed and evaluated in the scope of this European research initiative, highlighting its practical relevance.

3 Methodology and System Design

This chapter goes over the methodology used for the development of the DT for the Port of Funchal, along with a description of its system architecture design. The discussion will cover the system requirements, the overall architecture with a focus on its microservice-based approach, the different layers, key microservices, data management strategies, development technologies, and project constraints.

3.1 System Requirements

The foundation of the system design is a clear understanding of its intended capabilities and operational characteristics. These requirements were collected during meetings with the stakeholders, as well as with the consortium of the Shift2DC project, and have been categorized into functional, which describe what the system must do, and non-functional, which define how the system should perform these functions.

3.1.1 Functional Requirements

The functional requirements delineate the specific tasks and functions that the DT system must execute. These are defined as follows:

Real-time Monitoring The system must provide a continuous display of current energy consumption data alongside the operational status of the port's energy grid.

Vessel Energy Profiling The system must predict energy consumption patterns for various vessel types, considering their specific characteristics as well as their arrival and departure schedules, with a real-time connection to the physical world.

DC Power Flow Analysis The system must simulate the performance of the DC grid infrastructure, allowing for the calculation of important KPIs such as overall efficiency and energy losses within the network.

Data Visualization The system must provide users with interactive charts, geographically referenced maps, and comprehensive dashboards to present complex data and simulation results in an accessible manner. This visualization will be in 2.5D to emphasize the DT as a virtual copy of the physical asset. This approach provides better spatial context than a 2D map without the

computational overhead or potential visual clutter of a full 3D environment, making it ideal for visualizing network topology and energy flows.

3.1.2 Non-Functional Requirements

The non-functional requirements refer to the quality attributes and operational constraints that the system must adhere to in order to guarantee its effectiveness and usability.

Scalability The architecture must be designed to handle increasing volumes of data and accommodate future services that might be added to the DT without significant performance degradation.

Performance The system must ensure low-latency processing for real-time data and maintain a responsive Graphical User Interface (GUI) to provide a smooth and efficient user experience.

Modularity Individual components must be designed so they can be independently developed, deployed, updated, and maintained.

Interoperability The system must be able to integrate seamlessly with existing systems and be adaptable for future services or technological advancements.

Usability The system must feature an intuitive and user-friendly interface, designed to be easily operated by users who may not possess specialized technical expertise.

3.2 Overall System Architecture: A Microservice-Based Approach

When first prototyping the system, various approaches were considered, particularly two styles of architecture: a Monolithic architecture or a Microservice architecture. The monolithic approach was quickly discarded due to the constraints that immediately resulted when dealing with various separate stand-alone tools. The focus of the work was on integrating these tools, and joining them together in a single data layer would have created even greater constraints, possibly leading to a "dependency hell."

The architectural design chosen for the DT for the Port of Funchal was founded on the microservice architecture discussed in chapter section 2. The architecture built follows a similar structure, with the microservices serving as the foundation for the whole system, followed by a component to serve as API gateway and finally at the 'top', the client-side application. A standard micro-service architecture can be observed in fig. 5.

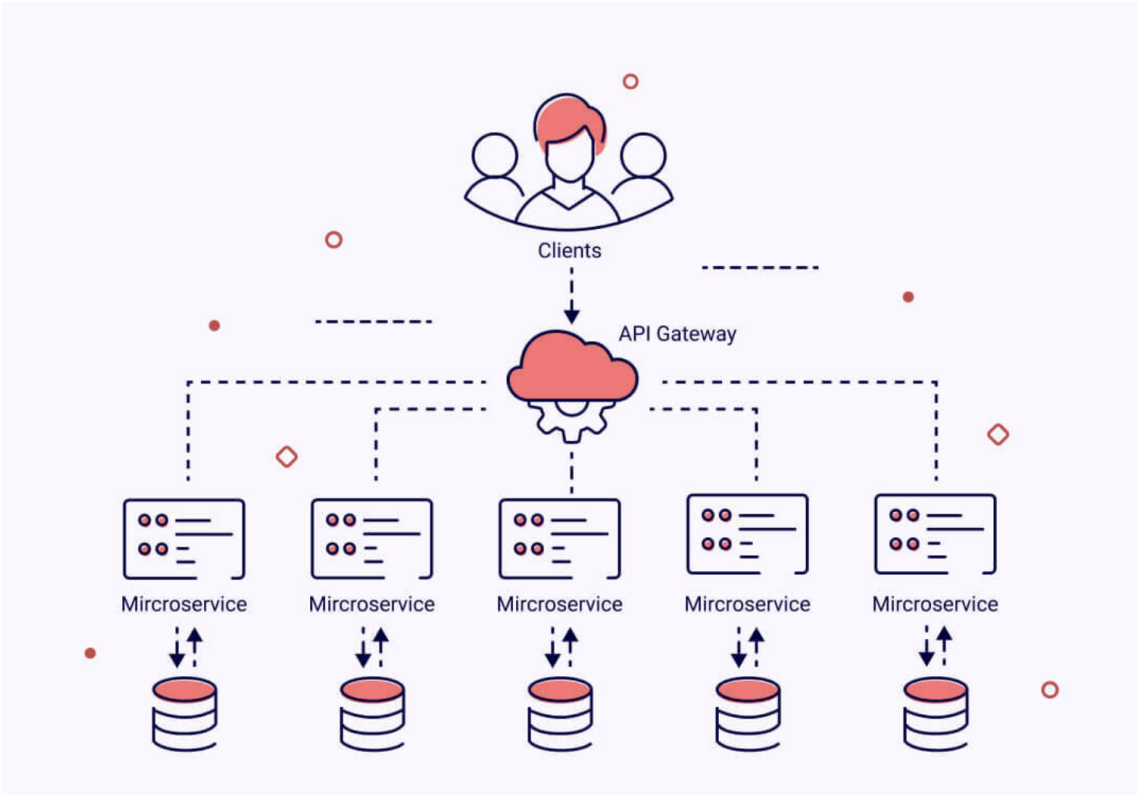


Figure. 5: Generic Micro-Service Architecture Example

This approach was selected to address the complexity and limitations inherent in integrating a diverse array of functionalities and characteristics found in each of the main services, such as vessel energy modeling, DC power flow analysis, and data management, that, as a whole, create the DTs. This modularity is what facilitates independent development, deployment, scaling, and maintenance of individual components, which proves to be especially important in an environment where each microservice uses a different set of packages, tools, and even programming languages.

3.2.1 Microservice Integration

Starting from the 'bottom' of the architecture, each microservice (e.g., Vessel Modeling Tool, DC Design Tool) operates as a standalone application with its own API server.

The primary role of this thesis, beyond the development of the Visualization component, is the integration of these disparate microservices. Communication between the middleware and the microservices primarily depends on RESTful APIs for request/response interactions and potentially WebSockets for real-time data streams where applicable. This integration ensures that data and capabilities from each specialized tool are accessible within the unified DT environment.

A standard communication strategy is essential to ensure data flows reliably from the microservices to the presentation layer for the end-user to use. Taking into consideration the importance of a robust integration, development began with an API centred design philosophy. Before implementation, a formal interface documentation was defined for each microservice.

A example of this documentation can be observed in fig. 6, which specifies the available endpoints, the expected request/response data (e.g., JSON objects for vessel simulations or power flow results), and HTTP methods (GET, POST).

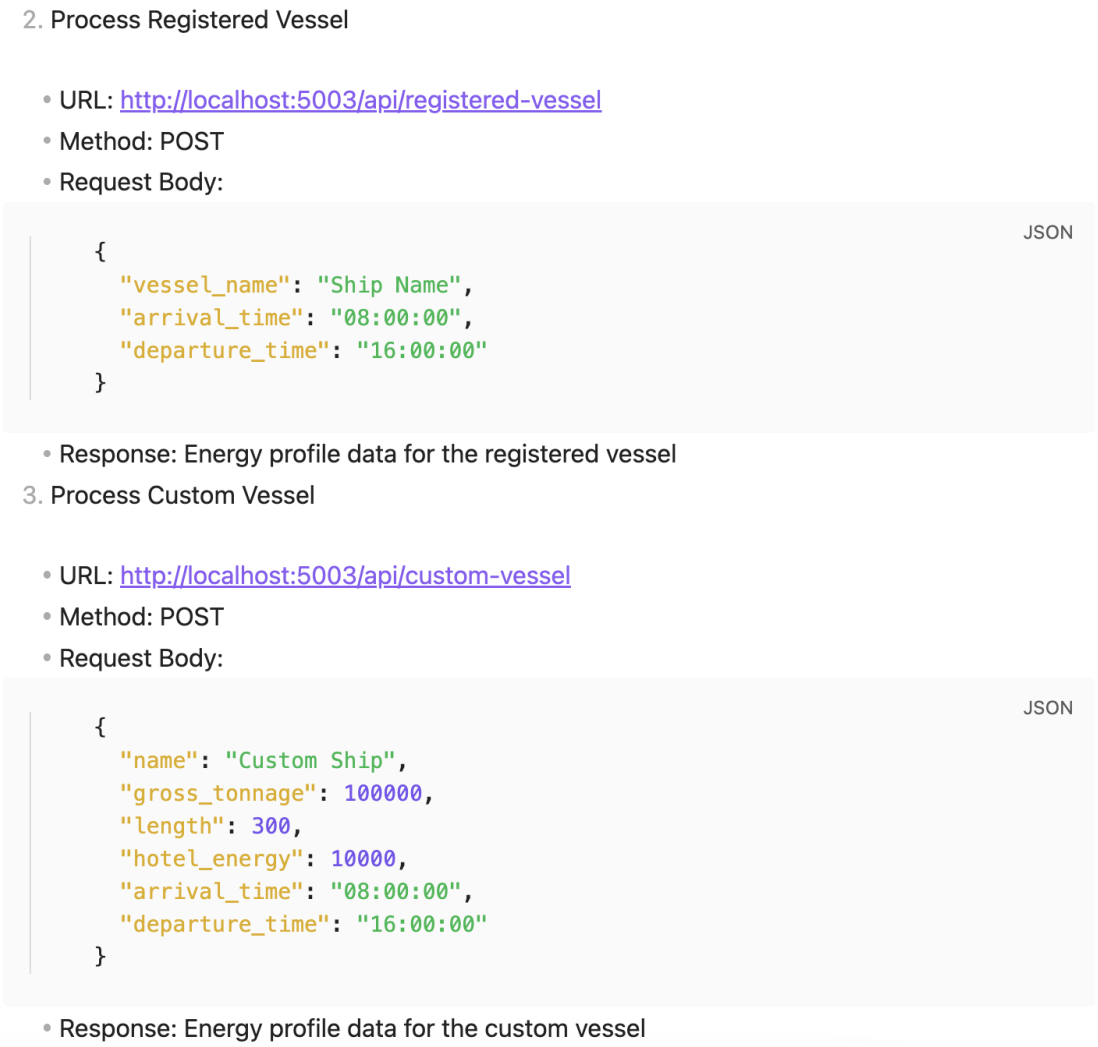


Figure. 6: Vessel Modeling APIs Documentation

This approach separates the development of the microservices from the middleware, allowing apredictable and well documented integration point.

3.2.2 Middleware Layer

The Middleware Layer is the one who has the most responsibilities out of all the other components that consist the DT, namely four main ones as showed in fig. 7.

This layer acts as an orchestrator and API gateway between the client layer and the microservices. The tasks of this layer are numerous and are all critical to ensure that the system works cohesively.

Primarily, the middleware is tasked with Request routing, which consists of intelligently directing client requests received from the client-side to the appropriate microservice that is equipped to handle them.

To handle the constant flow of real-time data from the port's physical assets (e.g., sensor readings) without overwhelming the system, asynchronous communication is used. This is implemented using WebSockets by having the Middleware Layer establish persistent WebSocket connections with the Presentation Layer. As new data arrives at the middleware from external sources (like a database poller), it is immediately pushed to all connected clients. This pattern disconnects the data producers from the consumers, using a constant flow of information that gives the system the DT title, by maintaining this constant connection with the real asset. This guarantees that updates to the 2.5D map and dashboards occur in near real-time without requiring the client to constantly poll the server for new information.

For user actions that require an immediate response, the system uses synchronous communication via the discussed RESTful APIs. When a user, for example, requests a custom vessel energy simulation, the Middleware Layer makes a direct HTTP request to the corresponding microservice's API. This is a request/response pattern, where the middleware blocks and waits for the result before sending it to the Presentation Layer. This pattern is optimal for interactions like this, where the user expects a direct result from their action.

This layer will also manage Authentication and Authorization, thereby securing access to the various services and ensuring that only legitimate users with proper permissions can interact with the system's functionalities and data.

In some situations where information from multiple sources is required to fulfil a client request, the middleware performs Data aggregation, combining data retrieved from several microservices to provide a unified response.

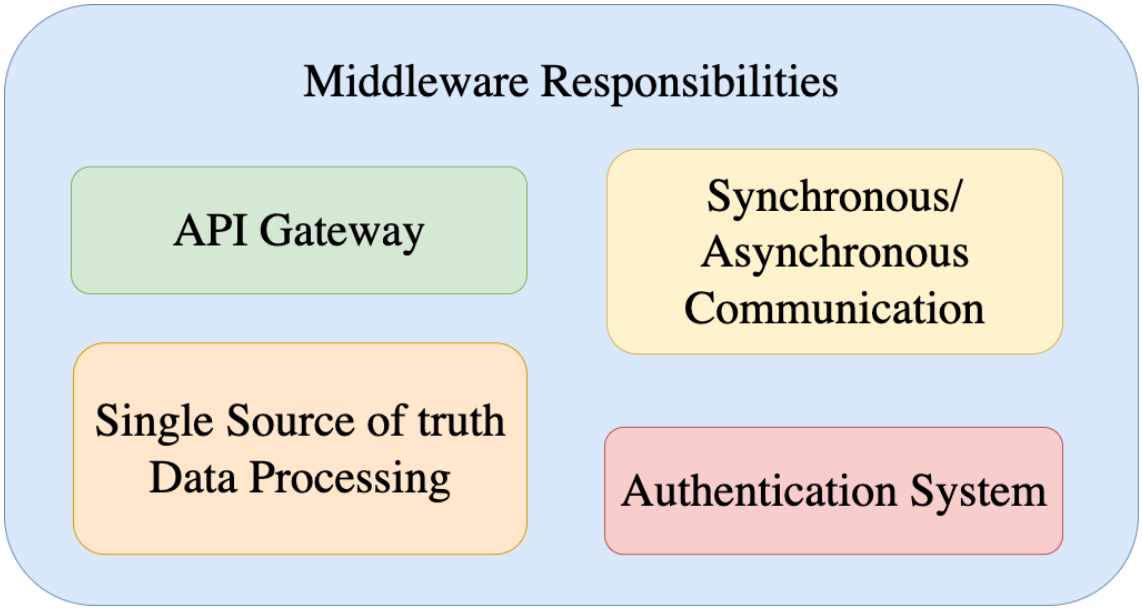


Figure. 7: The 4 main responsibility of the Middleware Layer

Another important task that is part of this layer is the connection with SSOTs. This layer simplifies the Presentation layer logic by providing a single point of interaction and decouples the client from the specifics of Middleware service discovery and communication.

3.2.3 Presentation Layer

This layer is, as the name indicates, directly responsible for presenting and rendering the user interface and visualizing all forms of data presented to the user. The primary consideration in developing the user interface was to offer a powerful tool that is also minimalistic and efficient in its design. Research in the DT field discussed in chapter 2, indicates that these systems typically include a 2D or 3D component. While primarily for visual purposes, this component further strengthens the connection between the physical asset and its virtual counterpart. Incorporating such a visual element was considered a vital component for implementation in this tool.

Beyond the visual 2D/3D model, a DT is, above all, a data representation. A dashboard style was chosen to accommodate and represent all the different services. Each service has its own dedicated tab and page, allowing the port authority to explore deeply into specific aspects. Within each tab, various data representation techniques were employed, ranging from classical tables to diverse charts equipped with filtering options. Users will also have the option to input custom data into specific services, such as the vessel modeling service, enabling them to run their own simulations with different vessels. This design ensures the presentation layer remains modular and scalable,

enabling future services to be easily integrated without appearing out of place within the system. The first mock-up done for the system can be seen in fig. 8.

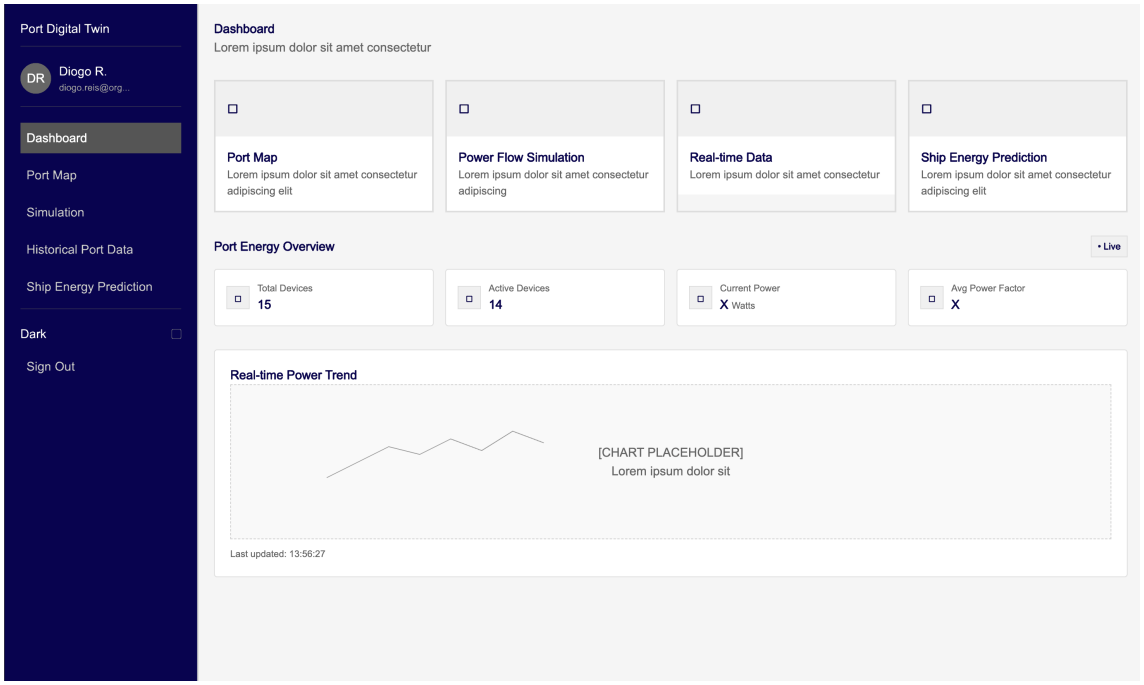


Figure. 8: First Mid-Level Mock-Up of the DT

The only exception to this tab-per-service structure is the previously discussed visual model, which also has its own tab. The objective here is to provide a more complete and simplified view of the DT as a whole, not only visually but also in its data representation. This includes an accurate connection to the real asset, with real-time data being collected and displayed in this space. It also features the information of the vessels, including their schedules and energy profiles sourced from the respective microservice, which updates daily to reflect the vessels currently stationed at the port. Finally, other important decisions were made regarding the visual component. To keep the system consistent, clean, and minimalist, a look that heavily uses maritime colours, like blue and white, was used for the primary and secondary colours throughout the system.

3.3 Key System Microservices

The DT relies on several key microservices, each dedicated to a specific domain of functionality, which operate collaboratively to provide the overall system capabilities.

3.3.1 Vessel Energy Modeling Service

The Vessel Energy Modeling Service was developed by another member of the research team, the detailed methodology and validation of its machine learning models are presented in their corresponding thesis [54]. Within the context of this work, the focus was not on creating the generating the models themselves, but on integrating this service into the broader Digital Twin ecosystem. A key architectural goal was to ensure this integration could support future updates or replacements of the modeling engine without impacting the rest of the system. The service's core function is to generate synthetic energy demand profiles for vessels by analyzing historical data, vessel characteristics, and operational schedules. To achieve this integration, a well-defined API was designed. This API exposes specific endpoints that allow the Middleware layer to request energy profiles for particular vessels or for defined future time periods.

3.3.2 DC Power Flow Optimization Service (DC Design Tool)

This microservice, known within the project as the DC Design Tool, was developed by the project consortium to facilitate the optimal design of DC power networks in the port area. The detailed implementation is documented in the project deliverable [55]. Its primary functionality centers on performing detailed DC power flow analysis, a method for modeling electricity distribution and voltage levels in a DC grid under various operating conditions. This thesis does not aim to provide an in-depth explanation of power flow theory, for a comprehensive background, the reader is referred to the established literature [55]. The tool supports network sizing for worst-case load conditions, simulates grid performance under different load profiles, and calculates KPIs related to efficiency, power loss, and voltage stability.

The contribution of this thesis lies in the integration of this service, similarly to the previous component discussed. Previously, the DC Design Tool operated as a standalone, offline application for design studies. This work transformed it into a fully integrated component of the Digital Twin, capable of running simulations using near real-time data. By developing a API wrapper, the system can now trigger power flow simulations dynamically, using live operational data as input. This elevates the tool from a complex yet static design utility to a dynamic, operational system, allowing users to assess the grid's state in response to current conditions, a major enhancement of its original purpose.

4 System Development

This chapter discusses development details, including the technologies chosen for each layer and the reasoning behind each technical decision. Figure 9 shows a top-level view of the developed system architecture. It is essentially divided into three distinct levels. The explanation will follow the flow of data from the 'lowest' level, where the microservices and the single source of truth reside, through the middleware layer, and finally to the client-facing layer of the application: the presentation layer.

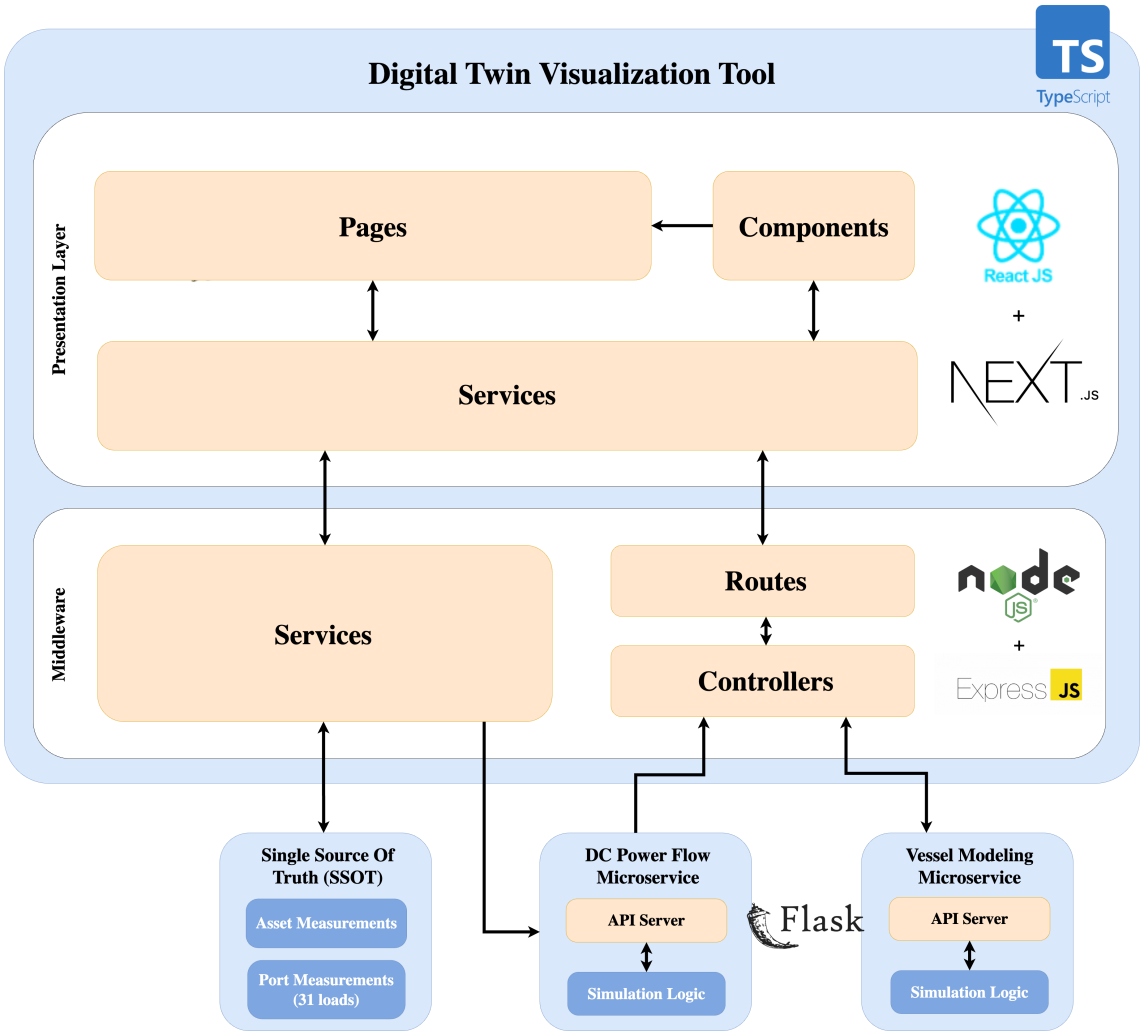


Figure. 9: Simplified Architecture of the Application.

4.1 The Data Foundation: SSOT

The foundation of any Digital Twin is its connection to the physical asset it represents. In this system, that connection is established through a data acquisition pipeline that populates a centralized

database, referred to as the SSOT. This entire physical monitoring infrastructure was designed and installed by a project consortium team to capture this energy data from the port.

As illustrated in Figure 10, the data originates from a series of eGauge[3] energy meters installed on the port's left and right electrical circuit boards. These meters monitor 33 distinct loads, capturing measurements such as voltage, current, and power consumption. The data from these eGauges is collected and aggregated by two StratoPi devices, which are Raspberry Pi industrial computers.

These StratoPi aggregators transmit the collected data over the APRAM local network to a central EMS. The EMS is responsible for processing these incoming data streams and persisting them in the SSOT database. This database serves as the definitive historical and near real-time record of the port's energy state, providing the foundational data upon which the entire Digital Twin operates.

4.2 Integration of Microservices

For DTs to function correctly, they must make use of a diverse range of tools that, when combined, create what are referred to as DTs. The problem resides in the fact that these different microservices were developed by different teams in the project consortium, and all use, in one way or another, a different set of tools. Both microservices currently integrated with the system use Python as their main language, which is directly incompatible with the language and frameworks used for both the middleware and the presentation layer, which use TypeScript.

4.2.1 API Design and Communication Protocols

To turn this disadvantage into an advantage, the previously discussed microservice architecture was employed, using HTTP APIs for communication between each layer of the system. The HTTP protocol provides the communication needed to make all the necessary transactions. In terms of organization, each API created follows the same naming convention. The APIs start with a dedicated name for each microservice, followed by a name that describes the API's purpose. This helps keep the code more organized and clean, which aids in future maintenance efforts.

4.2.2 Integrating the Vessel Energy Modeling Service

The first microservice integrated with the system was the vessel modeling. This tool, as a standalone application, worked well in the terminal by prompting the user for the necessary information to run

[3] https://www.egauge.net/

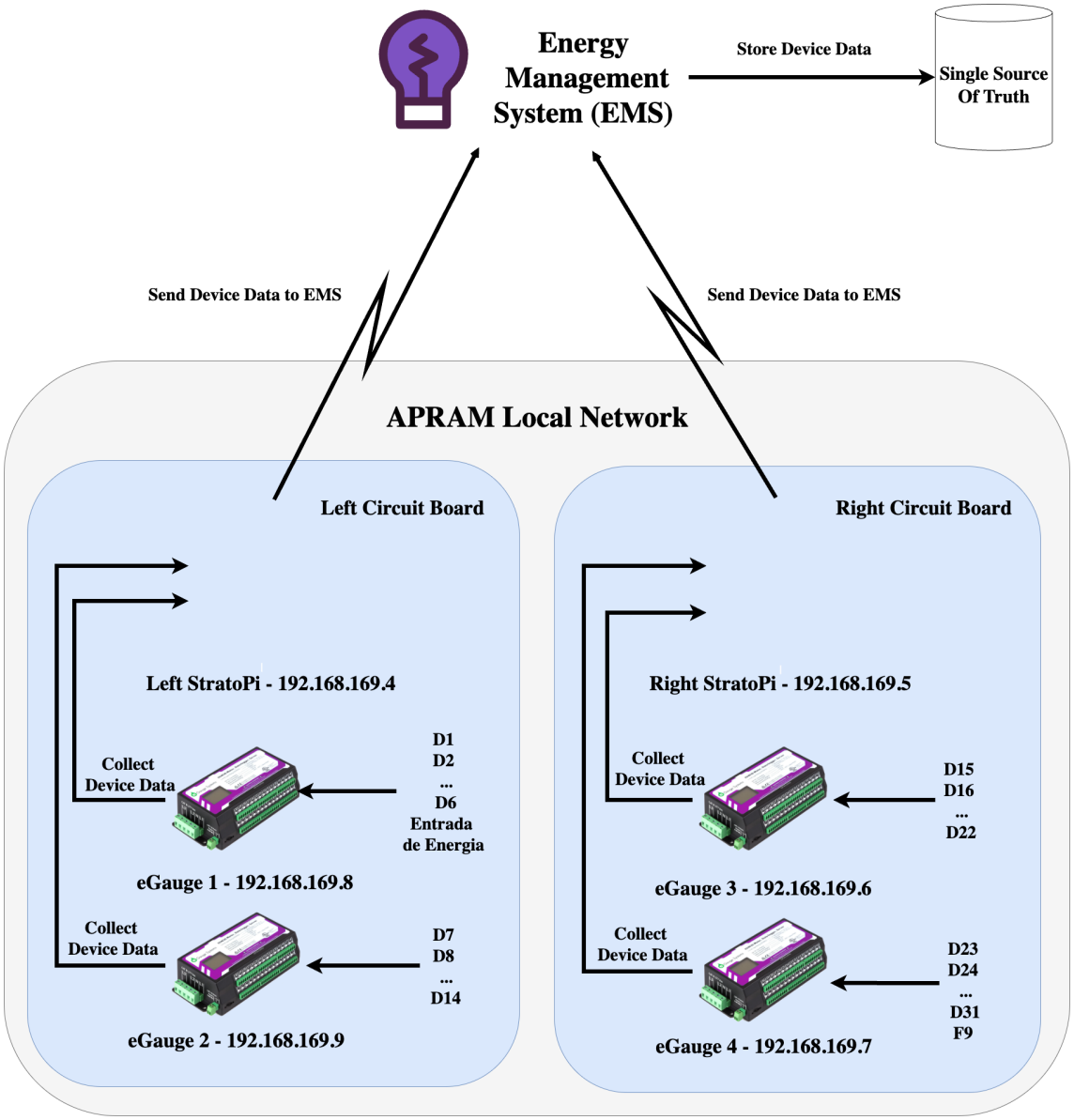


Figure. 10: Data Acquisition Pipeline for the Port of Funchal Digital Twin. Physical energy data is collected by eGauge meters, aggregated by StratoPi devices, and sent via the local network to the EMS, which stores it in the Single Source of Truth.

the simulation, such as vessel departure and arrival time, gross tonnage, vessel length, and hotel energy. The result is a JSON file, seen in listing 1, that can easily be used in HTTP communication, which facilitated the integration process.


```
{
    "closest_ship": "Ship3",
    "data": {
        "arrival_time": "06:00:00",
        "departure_time": "17:00:00",
        "energy_profile_data": [
            {
                "Chillers Power": 451.52869230769227,
                "Hotel Power": 6426.981692307695,
                "Timestamp": "2025-04-28 06:00:00"
            },
            {
                "Chillers Power": 454.0031538461538,
                "Hotel Power": 6424.252884615386,
                "Timestamp": "2025-04-28 06:05:00"
            },
            ...
            {
                "Chillers Power": 475.03492307692306,
                "Hotel Power": 6571.340692307694,
                "Timestamp": "2025-04-28 17:00:00"
            }
        ],
        "gross_tonnage": 100000.0,
        "hotel_energy": 10000.0,
        "length": 300.0,
        "vessel": "MARELLA VOYAGER"
    },
    "message": "Energy profile generated for MARELLA VOYAGER",
    "scaling_factor": 0.7692307692307693,
    "success": true
}
```


Listing 1: Vessel Modeling Simulation Result

The work involved setting up an API server for the component. To achieve this, the Flask[4] framework was used, which allows a Python-based server to be set up on a specific port. Using this tool, different GET and POST endpoints were set up. Specifically, for GET endpoints, there are three: a health check endpoint for monitoring the service status, one that retrieves simulations for a specific day, and another that returns a list of all available registered vessels in the system.

[4] https://flask.palletsprojects.com

```
@app.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({"status": "OK", "service": "vessel-modeling-api"})

@app.route('/api/simulations/<date>', methods=['GET'])
def get_simulations_by_date(date):
    """Get all vessel simulations for a specific date"""

@app.route('/api/available-vessels', methods=['GET'])
def get_available_vessels():
    """Get a list of available registered vessels"""
```

Listing 2: Vessel Modeling GET Endpoints

For POST endpoints, there are also two, relating to the different types of simulations the user can run: a custom one, where the user has to enter the full information necessary to run the simulation (detailed previously), or a simpler version, where the user can select from a list of different registered vessels in the tool's system, requiring the user only to provide the desired arrival and departure times.

```
@app.route('/api/registered-vessel', methods=['POST'])
def process_registered_vessel():
    """Process a registered vessel and generate energy profile"""
    try:
        data = request.json

        # Extract parameters
        target_ship = data.get('vessel_name')
        arrival_time = data.get('arrival_time')
        departure_time = data.get('departure_time')

        if not all([target_ship, arrival_time, departure_time]):
            return jsonify({
                "error": "Missing required parameters: vessel_name,
   arrival_time, or departure_time"
            }), 400
```

Listing 3: Vessel Modeling POST Endpoints

Finally, and most critically, a fundamental feature was developed to create a real connection between the simulation service and the real-world port schedule. This automated routine is essential to the system's function as a Digital Twin. Every night, the service scrapes the official APRAM website[5] to fetch the next day's vessel schedule. For each scheduled vessel, it automatically triggers

[5] https://apram.pt/reserva-cais

the appropriate simulation via its own API. While this nightly process is a step forward, it is recognized that a true real-time connection would be superior. This limitation and the potential for future improvement, such as integrating with live data from external sources, this will be discussed further in section 7.4.

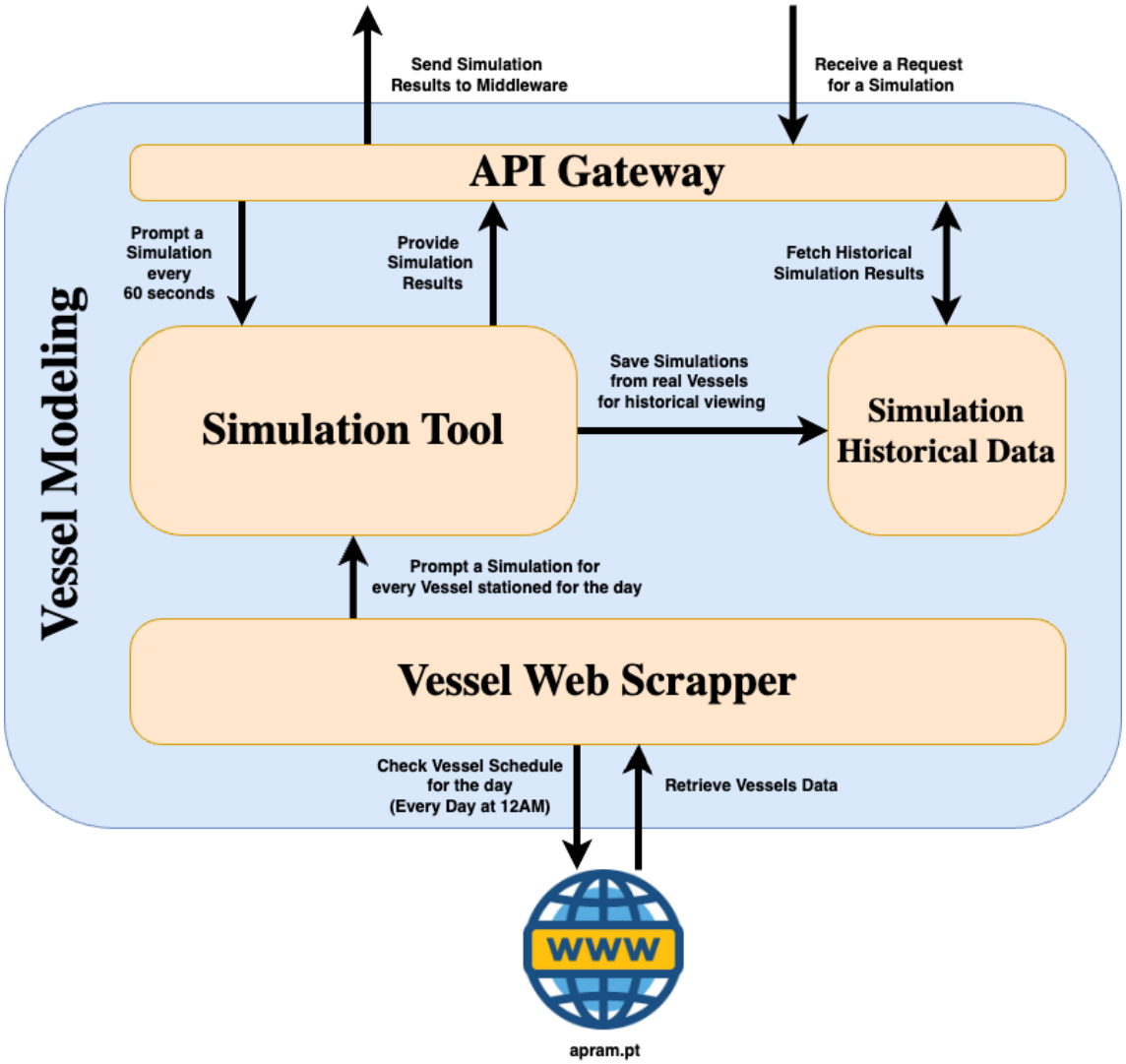


Figure. 11: Vessel Modeling Detailed Architecture.

4.2.3 Integrating the DC Power Flow Service

Integrating the DC Power Flow microservice presented a more difficult architectural challenge than the vessel modeling service. Initially conceived as a standalone, offline application, its workflow required a user to manually configure a grid topology in an Excel file, run a script, and then analyze the results in a separate output Excel file. To transform this into an automated component

of the Digital Twin, a new architecture was designed and implemented, as shown in Figure 12. This architecture, managed by a Flask API Gateway, was created to solve two primary challenges: data format incompatibility and the need for continuous, automated operation.

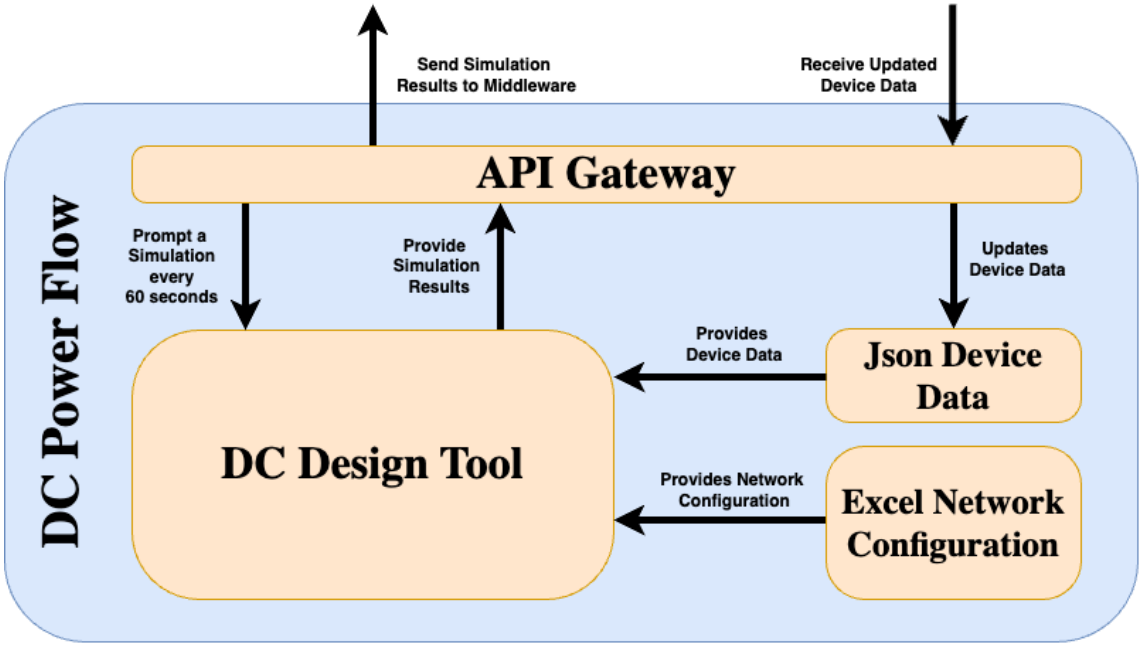


Figure. 12: Detailed architecture of the DC Power Flow microservice, showing the separation of static network configuration (Excel) from dynamic device data (JSON) and the automated simulation loop managed by the API Gateway.

The first challenge was the reliance on Excel files, which are unsuitable for inter-service communication in a microservice architecture. The output logic of the DC Design Tool was then modified to generate a structured JSON file (see Listing 4) instead of an Excel sheet. This allows simulation results to be easily transmitted via HTTP.

```
[
    {
        "timestamp": "Timestep 1",
        "bus_id": 0,
        "voltage": 1.0,
        "power": null,
        "load": -0.013247395806381956,
        "converter_X_power": 5.5,
        "converter_X_loading": 0.75
    },
    {
        "timestamp": "Timestep 1",
        "bus_id": 1,
        "voltage": 1.0,
```

```
        "power": null,
        "load": -0.013247395806381956
    }
    // ... more timestep entries
]
```

Listing 4: DC Power Flow Simulation Results

The second, more difficult challenge was automation. A Digital Twin requires a constant connection to the physical asset, meaning simulations must run periodically with fresh data. The initial approach, that was having middleware update the central Excel configuration file every minute, proved disastrous. Constant read/write operations on the same file led to instability and frequent data corruption. The final, solution is the hybrid data model depicted in the diagram. We separated the static and dynamic inputs:

1. **Static Network Configuration:** The original Excel Network Configuration file is now used only for its intended purpose: defining the static electrical grid topology (buses, lines, converters). The DC Design Tool reads this file once at startup.

2. **Dynamic Device Data:** A new Json Device Data file was introduced to handle the live data. The API Gateway receives updated device measurements from the middleware every minute and overwrites this JSON file. The DC Design Tool was modified to read its real-time load values from this file, ignoring the corresponding (and now outdated) values in the Excel sheet.

This separation guarantees data integrity and system reliability with minimal performance overhead. With this structure in place, the API Gateway orchestrates the entire real-time loop:

- It receives updated device data from the middleware.
- Every 60 seconds, it prompts the DC Design Tool to execute a new simulation.
- The DC Design Tool runs the simulation using the static Excel configuration and the latest dynamic JSON data.
- The simulation results are provided back to the API Gateway, which then sends them to the middleware for visualization in the frontend.

The main endpoints created for the API server directly support this flow: a POST endpoint to receive and update the Json Device Data, and a GET endpoint that allows the middleware to retrieve the latest simulation results.

4.3 Development of the Middleware Layer

The layer responsible for redirecting requests, as well as handling the system logic, is called the Middleware Layer. In this layer, generally speaking, two main functions can be found. The first is the previously discussed integration of microservices; in this case, the work involves redirecting all endpoints through a single point of access for the Presentation Layer. The second function is the treatment and collection of data being redirected directly from the port.

In Figure 13, it is possible to visualize these two functions reflected in three main sections inside the layer, consisting of the services, controllers, and routes.

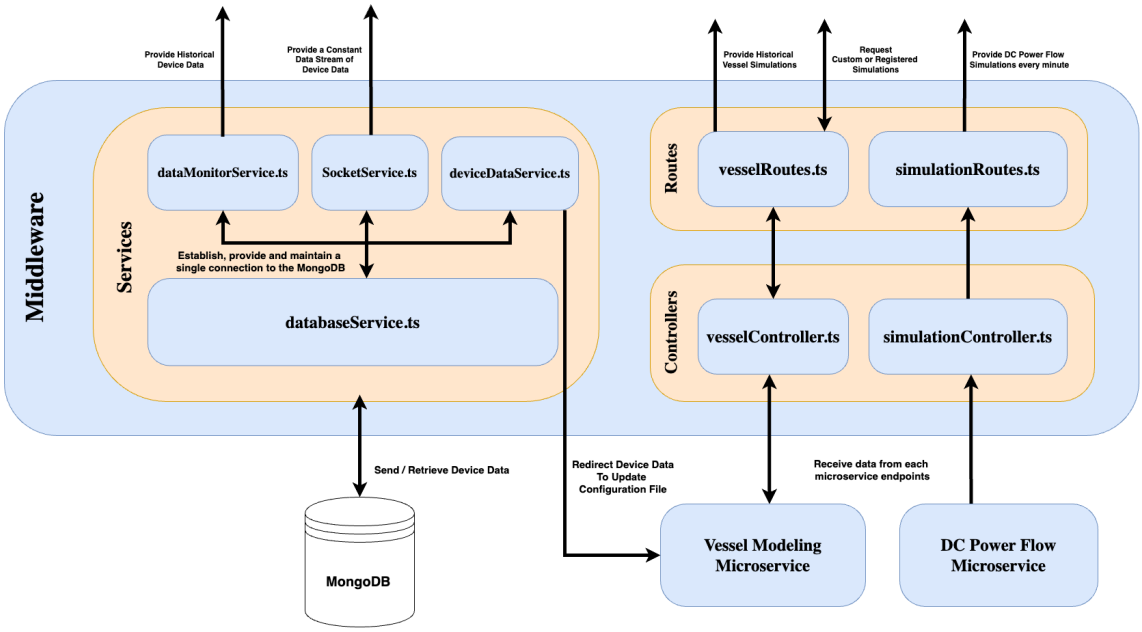


Figure. 13: Middleware Detailed Architecture.

4.3.1 Tech Stack

For this layer, Node.js[6] serves as the runtime environment. It is the foundation that allows the application to run TypeScript code on the server side.

In this application, Node.js is used to create a server that handles HTTP requests, manages WebSocket connections, and interacts with databases. The server is configured to run on port 5001 by default. Node.js's event-driven, non-blocking I/O model is particularly important here, as it efficiently handles concurrent HTTP requests and WebSocket connections, which occur frequently in such a complex system.

[6] https://nodejs.org/

Another important technology used is Express.js[7], a web application framework that provides the set of features needed for building the API server. In this application, Express is used to: create the main application instance, set up middleware for handling JSON requests, define all API routes, and handle HTTP requests and responses.

The Express application is integrated with Socket.IO by creating an HTTP server using Node's createServer function, which allows both HTTP and WebSocket connections to be handled by the same server instance.

Socket.IO[8] is implemented for real-time, bidirectional communication between the server and clients. In this layer, Socket.IO is configured with specific settings to allow connections from the frontend (defaulting to http://localhost:3000). The implementation includes a dedicated Socket.IO server instance with custom socket handlers.

This technology also plays an important role in monitoring MongoDB[9], broadcasting database changes to connected clients in real-time. The final, crucial task using Socket.IO is sending constant updates to the DC Power Flow microservice to keep its simulations constantly updated with new data, as discussed in the previous section regarding this specific microservice.

4.3.2 Requests Redirecting

The middleware implements a centralized request handling system using a modular Express.js router. All API endpoints are unified under an /api prefix, providing a single, consistent entry point for the frontend. Routes are organized by domain into separate modules (e.g., simulationRoutes.ts, vesselRoutes.ts) to ensure a clean and maintainable codebase.

4.3.3 Data Processing Logic

The core responsibility of the middleware is to manage the real-time data pipeline from the database to the client. This is handled by a set of interconnected services, with a primary focus on maintaining system availability even when the underlying data stream is unreliable, a common challenge in world IoT systems. The main services and their critical functions are:

- **Database and Socket Services:** These provide the foundational connectivity. The database service maintains a resilient singleton connection to MongoDB, while the socket service (using

[7] https://expressjs.com/
[8] https://socket.io/
[9] https://www.mongodb.com/

Socket.IO) manages real-time communication, distinguishing between clients operating in live mode versus "Historical Mode."

- **Data Monitor Service:** This is the engine of the real-time system. It employs a polling mechanism that queries the database for new sensor readings every 5 seconds. Upon finding new data, it validates and broadcasts the updates to all connected clients in live mode.
- **Device Data Service:** This service acts as a specialized adapter, formatting and providing the latest device data specifically for the DC power flow simulation, bridging the gap between the live data stream and the analytical microservice.

4.3.4 Robustness through Graceful Degradation

A critical feature of the middleware architecture is its implementation of graceful degradation to handle inevitable interruptions in the real-time data feed from the port's physical sensors. Rather than failing entirely, the system is designed to remain operational and useful during data outages. This is achieved through a specific health monitoring mechanism built into the Data Monitor Service:

1. **Outage Detection:** The service constantly monitors the timestamp of the latest data received from the Single Source of Truth. If no new data arrives within a predefined timeout period of 1 minute, it concludes that the live data stream has been interrupted.
2. **State Transition:** Upon detecting an outage, the service broadcasts a specific event to all connected clients via WebSocket.
3. **Client-Side Response:** The frontend application is designed to listen for this event. When received, it automatically switches the user interface into "Historical Mode." This involves displaying a clear visual indicator to the user and disabling UI components that rely on real-time updates.
4. **Preservation of Functionality:** In this mode, the application remains fully functional for all other purposes. Users can still browse and analyze the complete historical dataset, run custom vessel energy simulations, and trigger DC power flow analyses using historical data.
5. **Automatic Recovery:** The Data Monitor Service continues to poll the database. Once a new data point is detected, it broadcasts a new event broadcasting the recovery. The frontend

listens for this and automatically transitions back to live mode without requiring any user intervention.

This mechanism makes sure the Digital Twin provides continuous value as a tool, even when its real-time connection is temporarily severed, therefore addressing a fundamental challenge of deploying such systems in such an unpredictable environment.

4.3.5 Authentication System

The final, but equally important, part of the middle layer is the implemented authentication system. The application asks the user for an email, and a password.

Firstly, to ensure that no outsiders are able to connect and use the system without the required permissions, a simple verification was developed so that, in the registration form, the system only lets the request proceed if the email provided ends with "@apram.pt". This way, the system guarantees that only authorized personnel can create an account.

Beyond that, the rest of the process is standard, where the user chooses a password, and a verification code is sent to the user's email, to be input to verify if it matches. The Middleware uses SMTP settings provided in the environment configuration to send real emails to users. The verification code system has rate limiting, allowing users to request a new verification code only once every 60 seconds. This prevents abuse of the resend functionality and helps protect against brute-force attacks. When a user submits the verification code, the system checks its validity, ensuring that the code matches, has not expired, and has not already been used. If the code provided is correct, the user can then log in, using the same credentials, and enter the system without problems.

On the database side, the information of each user is stored in its own collection in the same MongoDB database that stores the device data. The collection structure consists of the following fields:


```
{
  "_id": {
    "$oid": "687e190f4fa84115d52bb541"
  },
  "email": "diogo.paulino10@apram.pt",
  "name": "Diogo Reis",
  "password": "$2b$12$ltJUdG7a/.Bsw3ngjlT3SeOVC/kFjUfqTNHXVODv0ifuOkU/mB6H6
    ",
  "isVerified": true,
  "createdAt": {
```

```
    "$date": "2025-07-21T10:40:15.072Z"
  },
  "verificationAttempts": 0,
  "lastLogin": {
    "$date": "2025-07-21T11:02:02.143Z"
  }
}
```

Listing 5: User Collection Structure

An important aspect, not mentioned previously, that can be noted in the collection above, is the fact that the password is encrypted to ensure that nobody can know the password contents, not even the developer, fully protecting the user.

Another part of the authentication system is the use of JWT (JSON Web Tokens)[10]. This is the mechanism used for managing user sessions and securing API endpoints. When a user successfully logs in, the middleware generates a JWT containing the user's email and name. This token is signed using a secret key safeguarded in the environment variables. The generated JWT is sent back to the client, which stores it and includes it in the Authorization header of the next API requests. All the routes are protected by requiring a valid JWT for access. This is enforced through this layer that checks for the presence of the token in the request headers, verifies its signature and expiration, and extracts the user information embedded within it. If the token is missing, invalid, or expired, the application denies access and returns an appropriate error message. JWTs are configured to expire after 24 hours, offering a balance between user convenience and security.

Figure 14 presents a flowchart of the main functioning elements of the authentication system. It illustrates with clarity the different steps the user must take to create an account or log in to their own, and from the system perspective, it focuses on the different verification steps detailed earlier that the system makes before creating an account.

4.4 Development of The Presentation Layer

The final layer, which interfaces directly with the client, is the presentation layer. Its development required extensive research into the most prominent tools and technologies in this rapidly growing field, as well as an exploration of data representation techniques for both the dashboard and the 2.5D map.

[10] https://jwt.io/

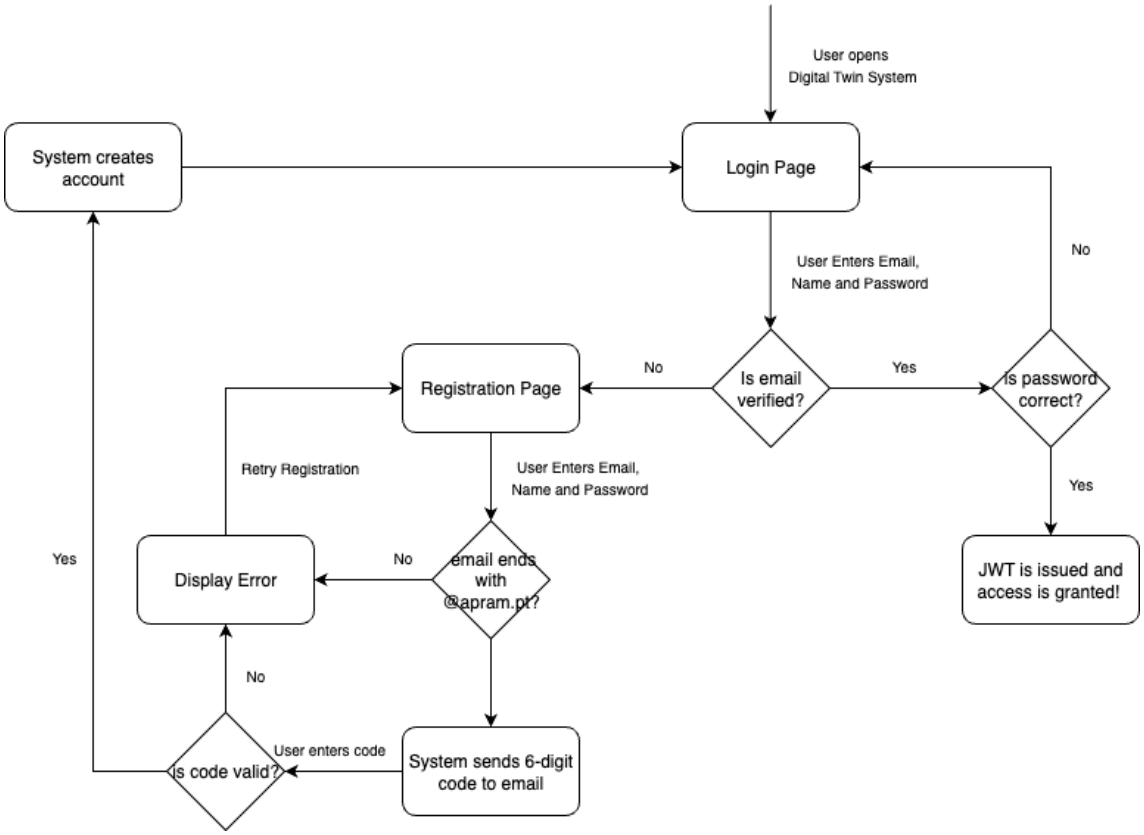


Figure. 14: Authentication System Flowchart.

As shown in Figure 15, the core of this layer is composed of four sections: services, pages, components, and CSS Files. The services are responsible for retrieving data processed on the server side and distributing it to the various visual components that are reused throughout the application, as well as directly to the pages themselves. Lastly, the CSS files offer styling to the visual components, providing a more complete, clean, and consistent look for the application as a whole.

4.4.1 Tech Stack

The tool stack that was ultimately decided to be used consisted mainly of a combination of the React[11] and NEXT.js[12] frameworks, using Typescript for the foundational language in which the layer was built.

The programming language selection was straightforward, to keep consistency between the Middleware Layer, which utilizes this same language, which also inherits its same advantages, such as type safety, compared to traditional JavaScript. This type system helps catch potential errors dur-

[11] https://react.dev/
[12] https://nextjs.org/

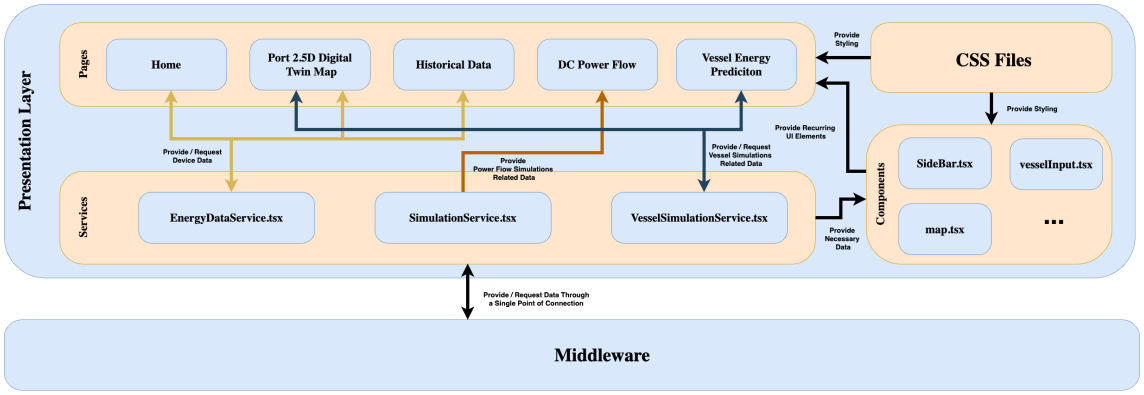


Figure. 15: Presentation Layer Detailed Architecture.

ing development and provides better code documentation through type definitions. Besides these factors, the frameworks mentioned also integrate seamlessly with this chosen language.

Regarding the main frameworks, the application uses Next.js 15.1.4, a React-based framework that provides server-side rendering, static site generation, and API routes, reducing the need for third-party libraries. While application's UI is constructed using React 18, making use of its component-based architecture.

For styling and UI design, this application uses Tailwind CSS[13], a CSS framework that facilitates fast UI development and keeps the UI design consistent across the whole layer. The styling system is enhanced with PostCSS[14], which processes the CSS with various plugins to optimize the final output.

The visualization capabilities of the application are powered by several specialized libraries. Deck.gl[15], a framework powered by WebGL, is used for high-performance visualization of large datasets, particularly useful for mapping and geospatial data, which in this case is used extensively in the 2.5D port map of the port of Funchal.

This tool also provides 3D rendering capabilities, which are used for the creation of complex 3D visualizations for the port's DT. This is used for the integration of 3D Models of different vessels that are stationed or moving around the port, with the purpose of connecting more deeply the virtual side to the physical one.

[13] https://tailwindcss.com/
[14] https://postcss.org/
[15] https://deck.gl/

Data visualization is handled through multiple charting libraries to provide various visualization options for the different microservices. Chart.js[16] and react-chartjs-2[17] are used for standard charts and graphs, offering a wide range of chart types and customization options. Recharts[18] provides additional charting options with a focus on React integration for more complex and custom visualizations.

Real-time data updates are implemented using Socket.IO client, which receives live updates from the Middleware Layer. This is important for the port's DT, where real-time data about vessels, power flow, and other metrics needs to be displayed immediately to users. Socket.io is particularly efficient here as it offers users the ability to receive updates in real-time without requiring page refreshes.

The presentation layer's technology stack was chosen to support the complex requirements of a DT application. The combination of Next.js, React, TypeScript, and the various previously discussed visualization libraries creates the optimal experience for users monitoring and interacting with the port's DT.

4.4.2 Home Page

The first page that appears when the user logs in to the application serves an important purpose: to introduce the user to the application itself. It serves as the central hub and provides a comprehensive overview of the port's current status and key metrics. The page features a modern yet simple design with a card layout that offers small introductions to each of the four main features and allows users to access all other sections of the application. Below the cards, the user can find an overview of the current status of the system regarding the real-time data arriving every minute from the port.

The main layout, seen in fig. 16, includes a dashboard interface that is consistent across the rest of the application, with the different services listed. This sidebar also shows the current logged-in user information and, in the bottom part, features two buttons: one to change the theme of the application and the other to log out of the system.

[16] https://www.chartjs.org/
[17] https://github.com/reactchartjs/react-chartjs-2
[18] https://recharts.org/

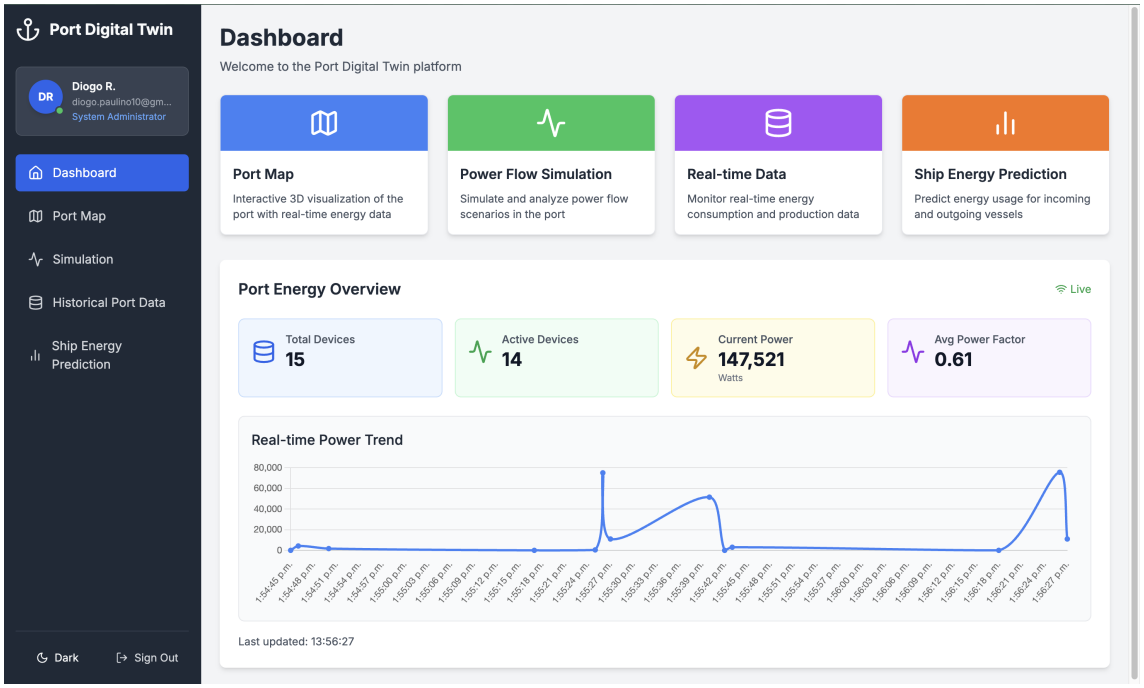


Figure. 16: Home Page View

4.4.3 Port 2.5D DT Page

The Port 2.5D DT Page, observed in fig. 17, is possibly the most important in the whole application. It offers a complete visualization of the port's infrastructure and operations. This page combines 2D and 3D elements to create a digital representation of the port. The interface features an interactive map powered by Maplibre[19] with deck.gl serving as the tool to represent the data, allowing users to explore the port's layout in detail.

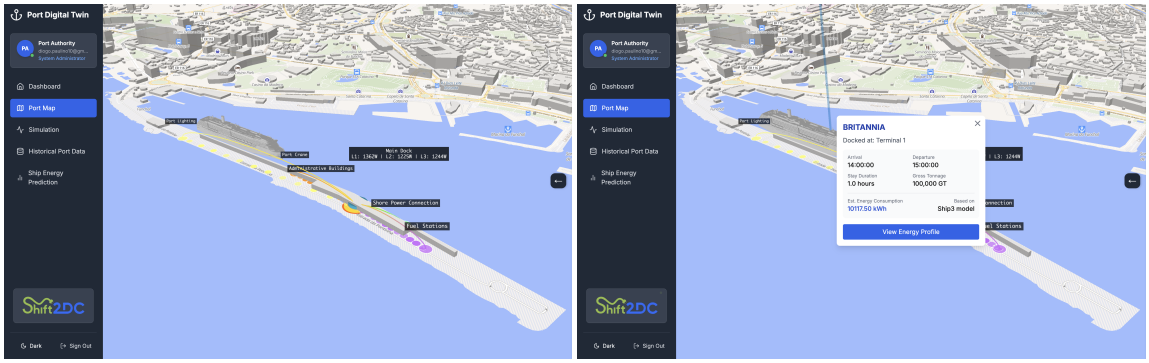


Figure. 17: Port 2.5D Map Page

Users can zoom, pan, and rotate the view to examine different aspects of the port's infrastructure. The visualization includes detailed representations of vessels, power infrastructure, and other port

[19] https://maplibre.org/

facilities. The page integrates real-time data updates through Socket.IO, ensuring that the visualization reflects the current state of the port. Users can interact with various elements to access detailed information about specific components of the port's infrastructure.

4.4.4 Historical Data Page

The Historical Data Page, in fig. 18, provides admin access to the port's historical operational data and real-time incoming data. This page features data visualization components powered by Chart.js, giving users the ability to analyze trends and patterns over time. The interface includes filtering and search capabilities, allowing users to focus on specific time periods or devices.

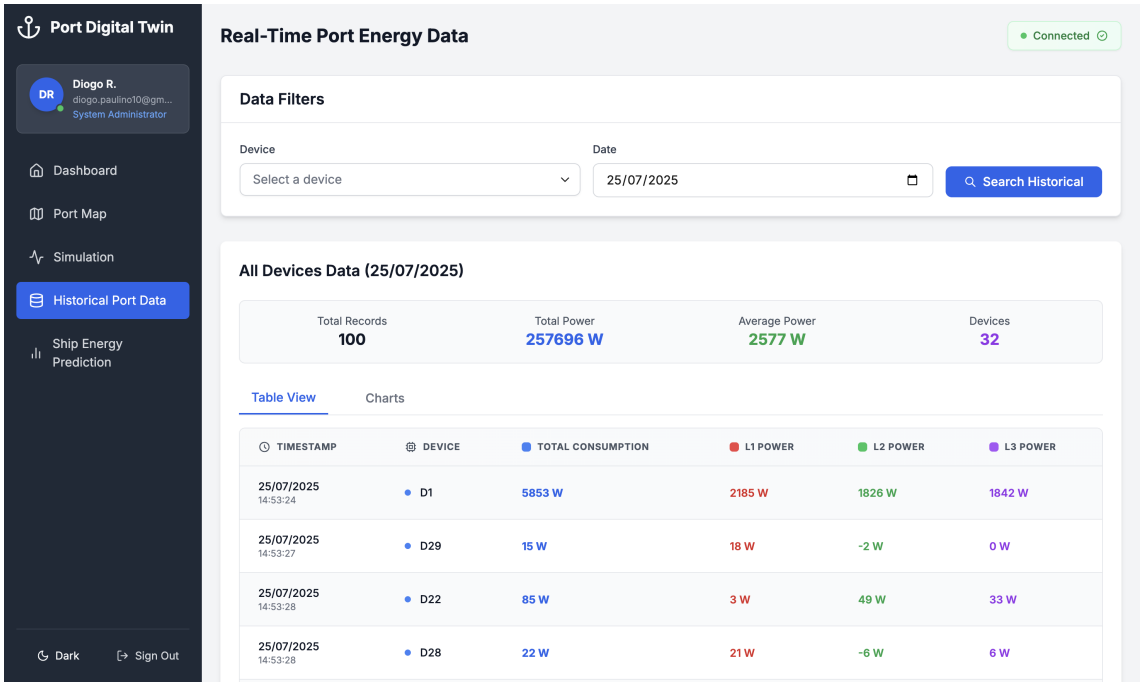


Figure. 18: Historical Data View

This service essentially offers two functioning modes: real-time data and historical data. By default, the page operates in the former mode as a mechanism to improve performance. When a user first enters the application, a query is quickly made to gather the 100 most recent data entries, and from there, it keeps updating each time a new entry is entered into the database. This approach was chosen to keep the system as connected and fast as possible with the real physical asset. If a user queries the system for all the data for a specific day or device, the number of records collected can exceed 10,000, significantly straining the system's performance.

The user can still make a more detailed query using the filtering options. In the example provided by fig. 19 and fig. 20, it is possible to observe the variation of the overall power consumption during a full day. Just from these two graphs, it is possible to infer that this device is probably related to some function regarding the port staff, since the power consumption severely diminishes during the night. From this, a port authority can monitor the daily consumption of this specific device and possibly find anomalies and prevent the misuse of resources.

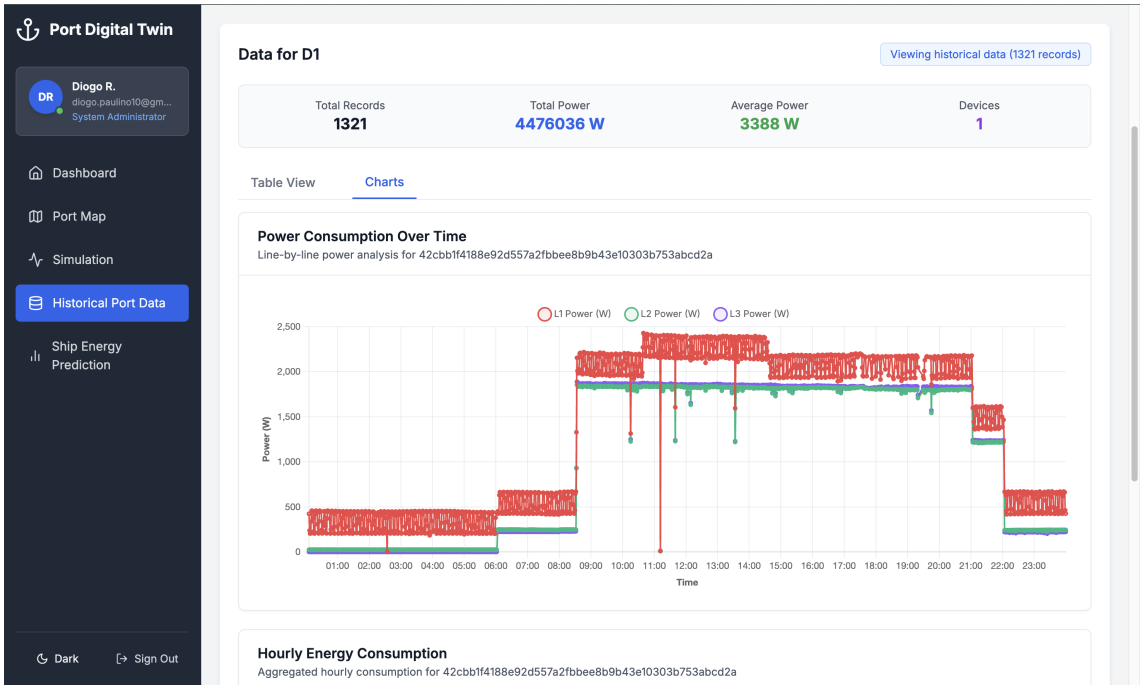


Figure. 19: Historical data for device D1 with line graph representation.

4.4.5 Power Flow Page

The Power Flow page is dedicated to simulating and analysing the port's power distribution system as if it worked as a DC network. This page provides a real-time visualization of power flow through the port's electrical infrastructure. Users can track power consumption, distribution, and efficiency metrics across different parts of the port. This page is mostly divided into two separate parts: the classic table style visualization, as seen in fig. 21, and a more interactive option for a data visualization style that uses graphs and interactive circuit diagrams presented in fig. 22. The visualization is updated in real-time through Socket.IO connections, ensuring that users have access to the most current power flow data. This simulation is updated every minute with the data coming directly from the port.

Figure. 20: Historical data for device D1 with column graph representation.

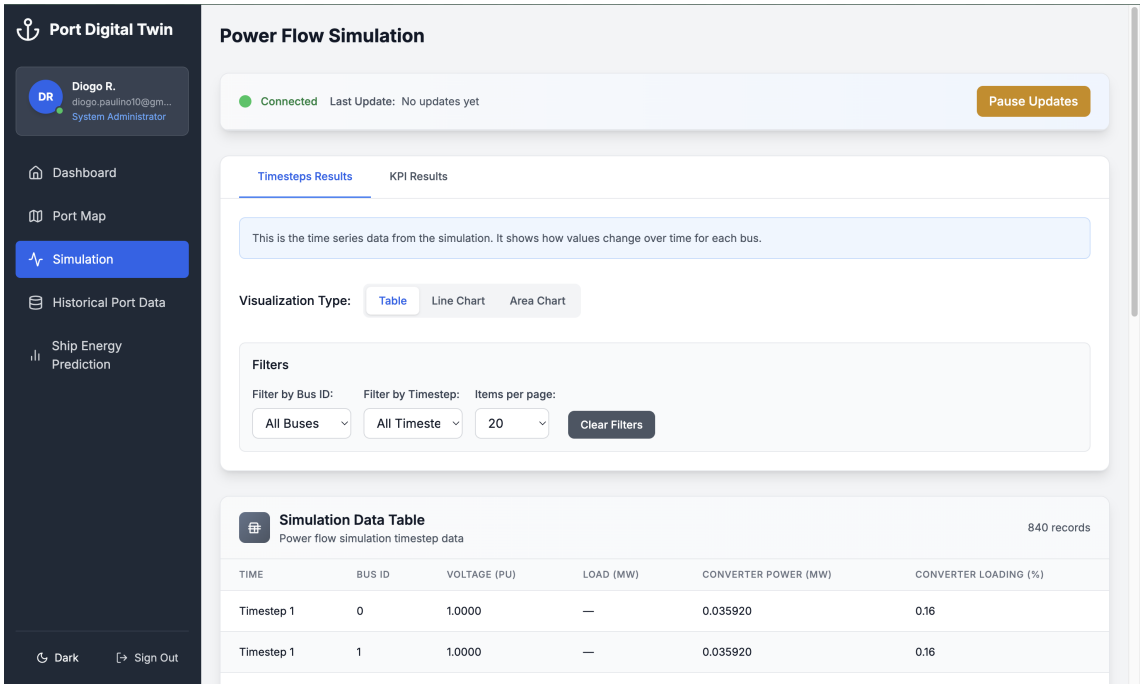


Figure. 21: Power Flow Results in Table Format

While the current table and graph formats provide precise quantitative data, an important area identified for future improvement is the creation of a more intuitive, schematic representation of the grid. A visualization in the form of an interactive single line diagram, representing the port's network topology with color coded nodes for voltage levels and animated lines for power

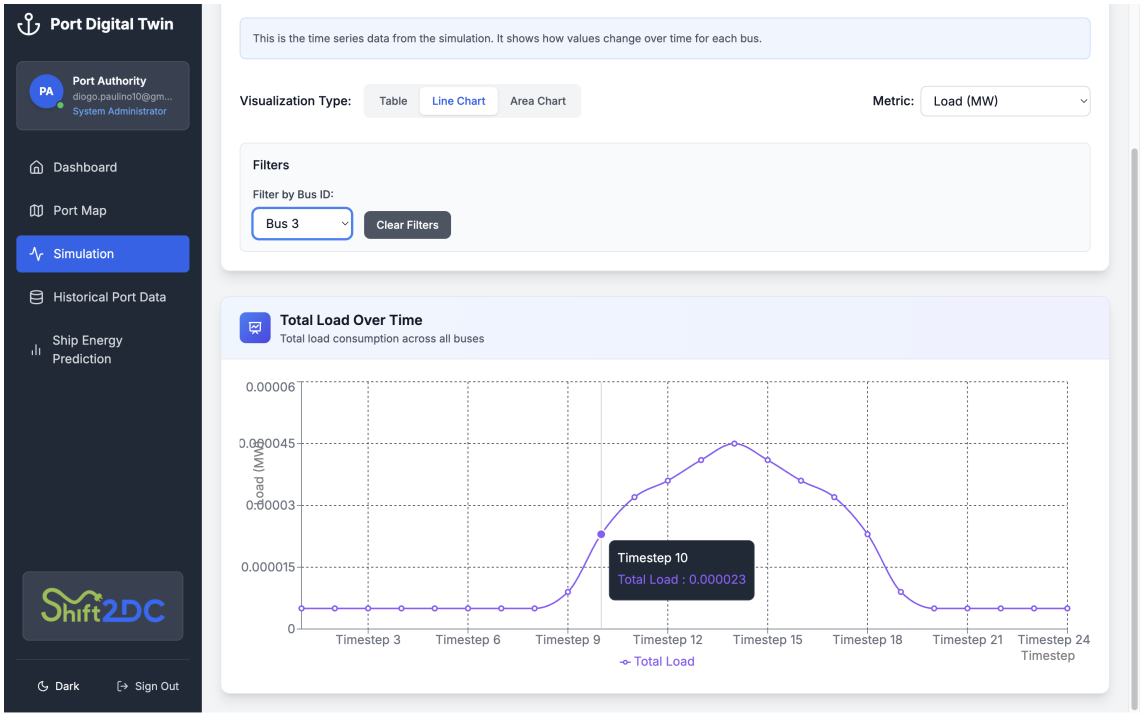


Figure. 22: Power Flow Results in Graph Format

flow—would make the results more accessible to port operators without a background in power systems engineering. This planned enhancement is a key point that will be discussed further in the Future Work section (Section 7.4).

4.4.6 Vessel Energy Modelling Page

The Vessel Energy Modelling Page provides two distinct but complementary functions: a "what-if" scenario for analytical purposes, and an operational forecast that directly reflects the real-world state of the port, serving as the core Digital Twin functionality.

The first capability is the simulation tool, designed to help port operators conduct planning and analysis. It allows them to understand the potential energy impact of any vessel, whether it is a frequent visitor or a new, hypothetical one. As shown in Figure 23, the user can run a custom simulation by either providing all the necessary vessel parameters manually (name, gross tonnage, etc.) or by selecting a pre-registered vessel and simply providing arrival and departure times.

The second, and more integral, component is the operational forecast, which embodies the page's Digital Twin purpose. This section, seen in Figure 24, displays a historical and future view of simulations that were automatically generated by the system based on the official APRAM vessel

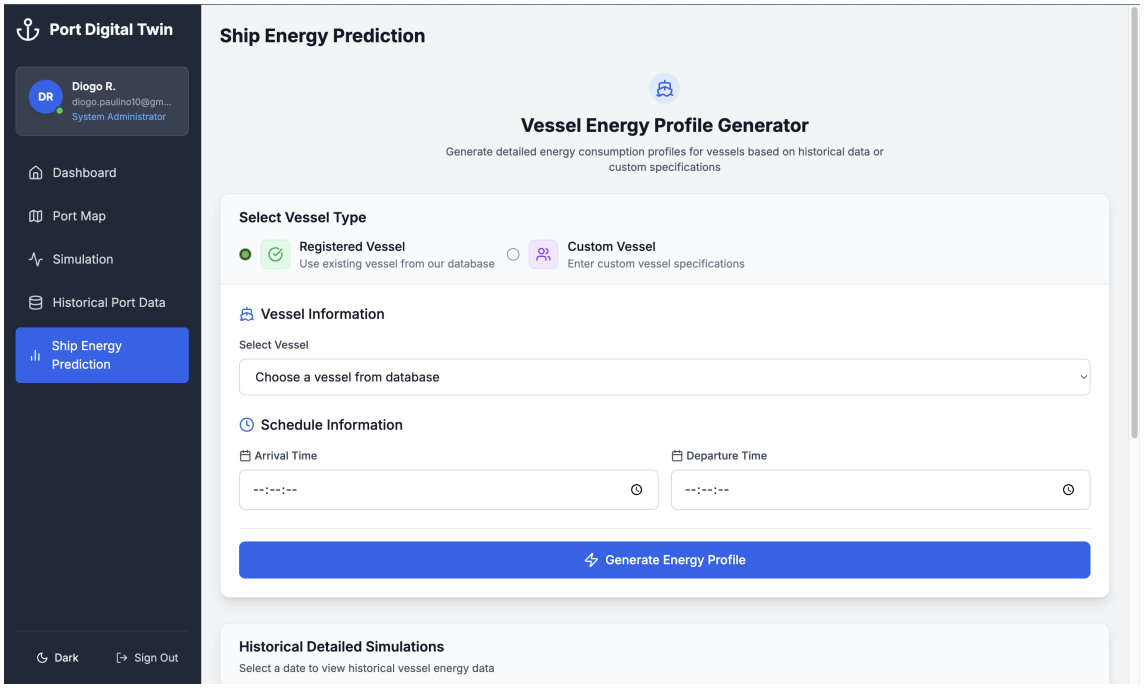


Figure. 23: The custom simulation form, serving as a "what-if" scenario planning tool.

schedule. These are not random simulations, they represent the generated energy demand profile for the specific vessels that were, or will be, physically present in the port on a given day.

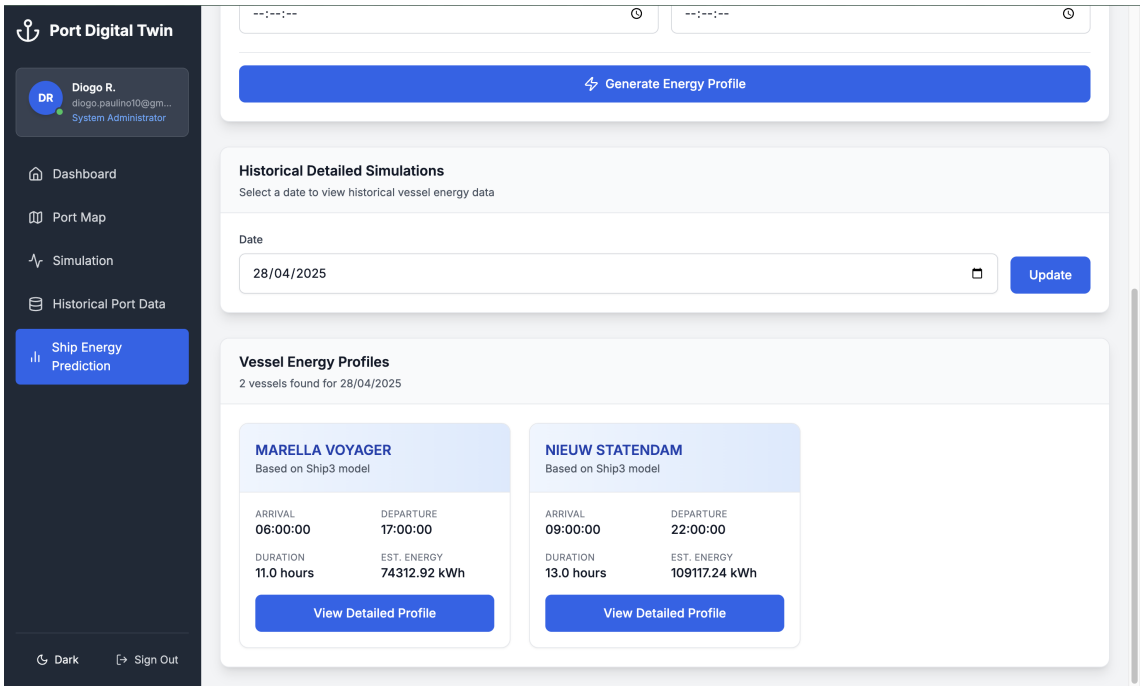


Figure. 24: The list of automatically generated simulations based on the official port schedule, representing the Digital Twin's operational forecast.

Although these two functions serve different needs, (proactive planning versus operational monitoring) they both lead to the same detailed analysis page. Whether a user runs a custom simulation or selects one from the operational forecast, they are redirected to a sub-page where the vessel's detailed energy profile can be explored through charts and data tables, as will be discussed in the next section.

4.4.7 Vessel Simulation Details Page

This is a sub-page of the one discussed in section 4.4.6, to which a user is redirected after performing a custom simulation or selecting a previously run simulation of a vessel stationed at the port.

The page starts with a profile header that clearly identifies the vessel by name, in this case, the MARELLA VOYAGER, and notes the simulation model used. A key metric, the "Scaling Factor," is displayed prominently to the right. Below this header, three cards provide a summary of the simulation:

- Vessel Details: Shows static information about the ship, such as Gross Tonnage and Length.
- Port Visit: Details the specific stay, including arrival/departure times and total duration.
- Energy Summary: Presents key calculated metrics from the simulation, such as Total Energy consumed, the number of Data Points, and the average Energy Rate.

The central feature of this page is the Power Consumption Chart, observed in fig. 26. This graph visualizes the vessel's energy usage patterns over the duration of its stay, plotting different power components like "Chillers Power" and "Hotel Power" alongside the "Total Power" consumption. This allows for an immediate visual analysis of energy demand fluctuations. Further down the page, a Detailed Energy Data table provides the raw, timestamped power consumption readings that underpin the chart. This table allows for a simpler inspection of the data, with columns for the timestamp and the corresponding power values for each recorded interval.

4.4.8 Login/Register Page

This page is the first point that the user encounters when entering the application. The purpose of this page can be divided into two main functions: introduction and validation.

As can be observed in fig. 27, the UI is essentially divided into two parts: the left side focuses on explaining and presenting the application, while the right side focuses on user validation. It follows

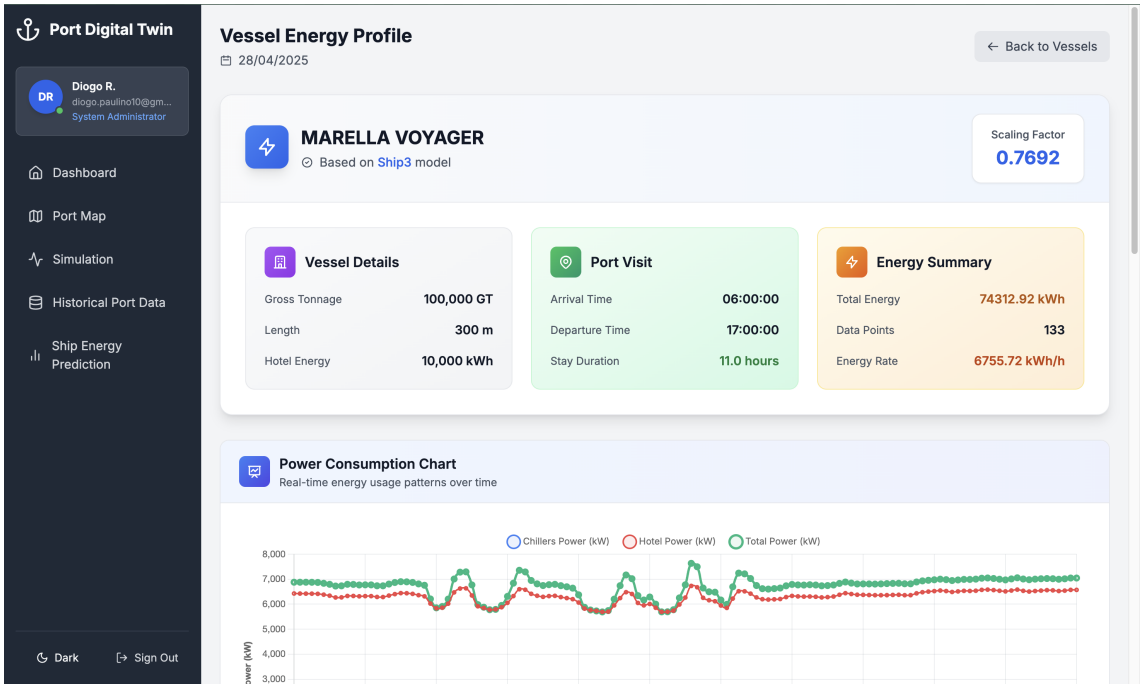


Figure. 25: Vessel Details Page Part 1

Figure. 26: Vessel Details Page Part 2

a standard login form design, with the email field at the top, followed by the name the user wishes to be called, and finally the password.

If the user does not yet have an account, the system prompts them to move on to the registration form shown in fig. 28. This form again follows the standard format found in modern applications,

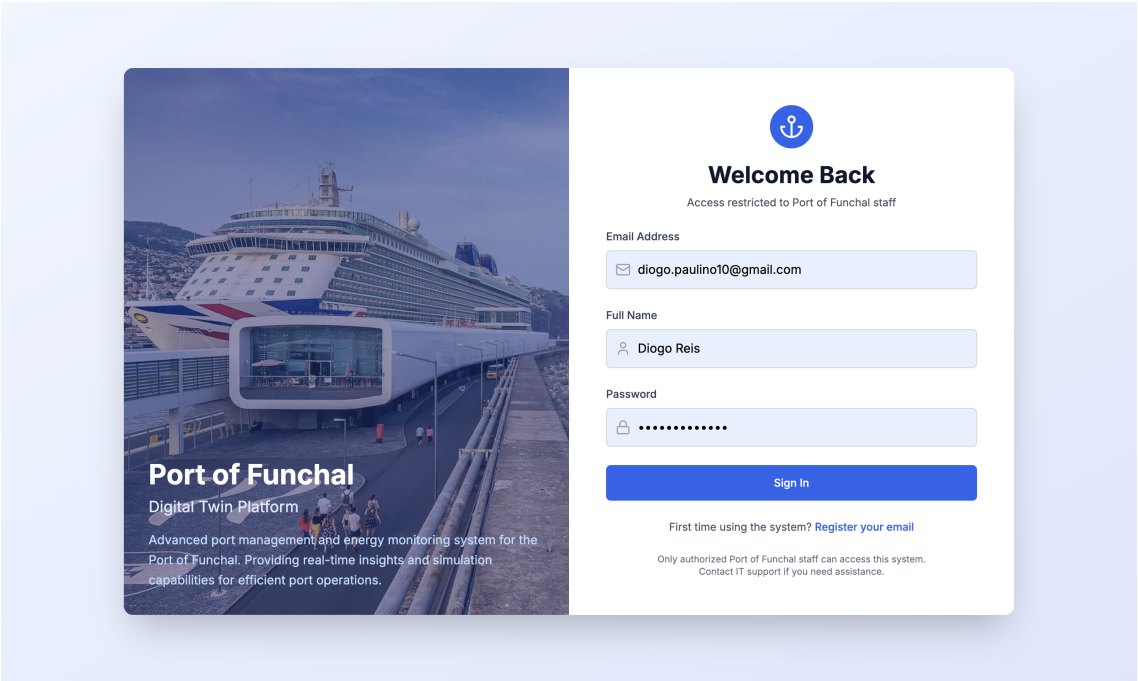


Figure. 27: Login Form

with the same input fields as the login form, with the exception that the password needs to be input twice to ensure that the user has not typed it incorrectly.

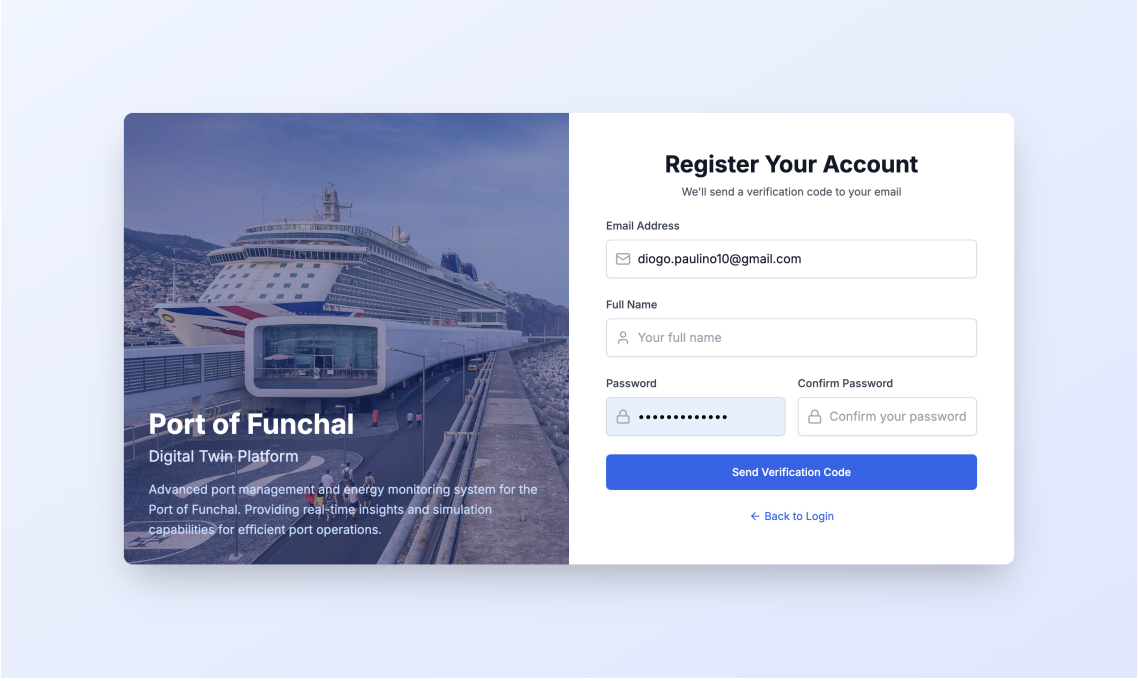


Figure. 28: Registration Form

After submitting the information in the registration form, the user is redirected to the third and final form, as shown in fig. 29, where they must input the code sent to the email they used to create the account. This form also provides the user with the ability to resend the code if they have not received it yet and informs them of the possibility of the email ending up in the spam folder, which is a common occurrence when dealing with automated emailing systems.

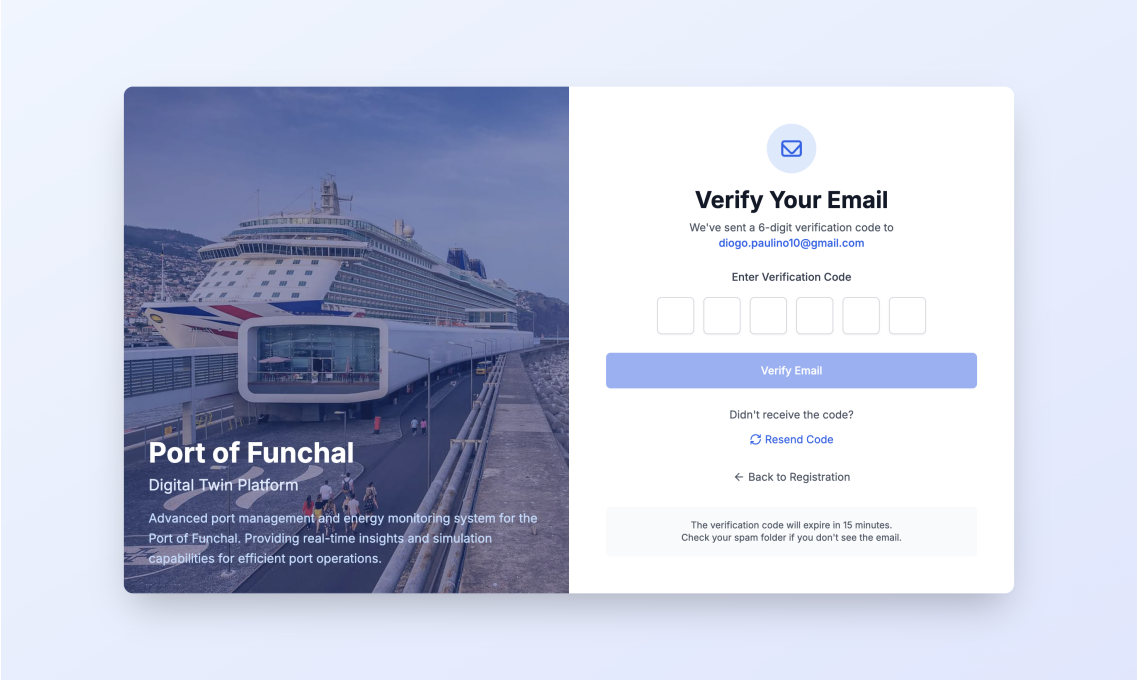


Figure. 29: Verification Code Form.

The email received by the user is shown in fig. 30. It consists of a header clearly stating that this email is from the Port DT system regarding email verification, followed by a small message directed to the user. Right below that, highlighted by a blue background, is the code needed to activate the account. Finally, the message concludes with some important information regarding the validity of the code and security measures.

From here, the user inputs the code and is then redirected back to the login form, where they can now log in to their newly created account. Furthermore, another important aspect not yet mentioned is that all forms also provide clear and consistent error messages for various scenarios, such as invalid email formats, unauthorized domains, expired or incorrect verification codes, and rate-limiting violations.

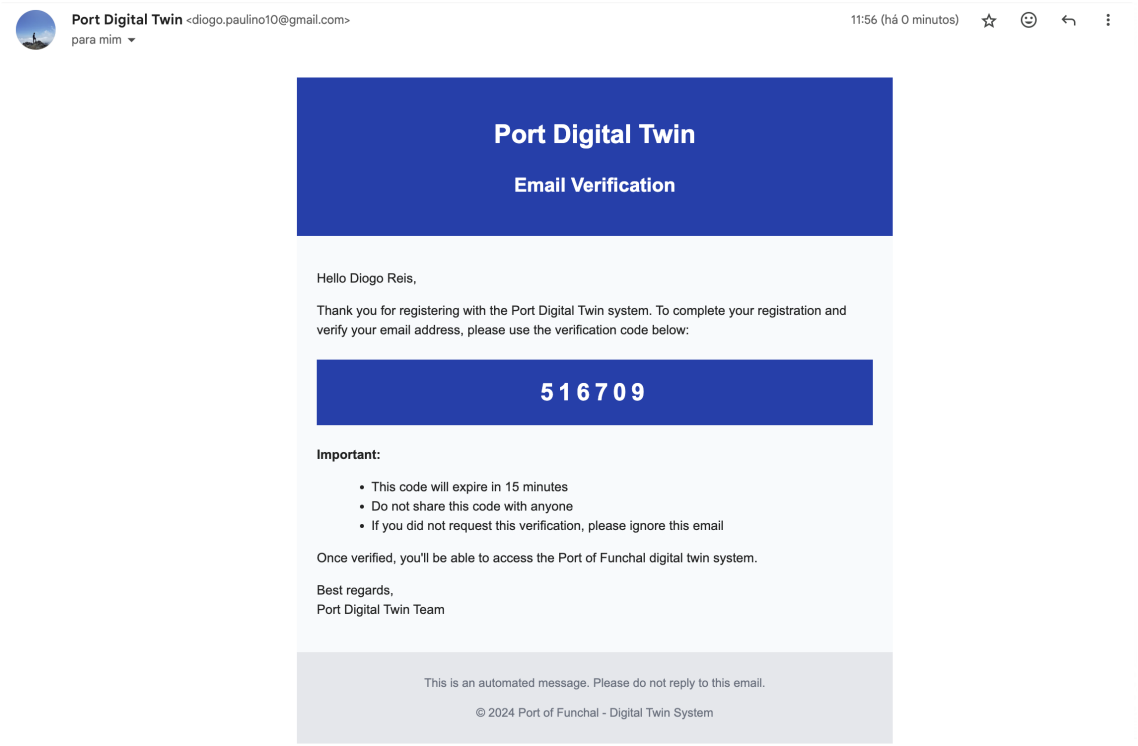


Figure. 30: Email sent to user with the Verification Code.

4.4.9 Dark/Light Mode

An addition that was retroactively added to the system was the dark mode. This feature is commonly found on modern apps, as a change of colour from a more colourful, vibrant tone to a darker and relaxing version of the same system.

Although this feature was not deemed crucial, during the development, the previous decisions made, especially in the tech stack of the presentation layer discussed in chapter 4.4.1, permitted the fast and easy integration of this feature into the current system. Specifically, the use of the CSS framework Tailwind accelerated this procedure greatly with the addition of a dynamic feature to an otherwise mainly static markup language, by using the 'data-theme' property as seen in listing 6.

```
[data-theme='dark'] {
  --background: #0a0a0a;
  --foreground: #ededed;
  --port-blue: #3b82f6;
  --sidebar-bg: #111827; /* Darker gray for sidebar */
  --sidebar-text: #f9fafb;
  --sidebar-highlight: #60a5fa;
  --card-bg: #1f2937;
  --card-shadow: rgba(0, 0, 0, 0.3);
```

```
  --hover-bg: #374151;
  --border-color: #4b5563;
}
```

Listing 6: Tailwind data-theme for Dark Mode.

The results can be observed in fig. 31, and compared with fig. 16. Every page was similarly changed to match the same tone. To change between each version, a small button was positioned in the bottom left corner that says "Light" or "Dark", opposite to the current mode the system finds itself in.

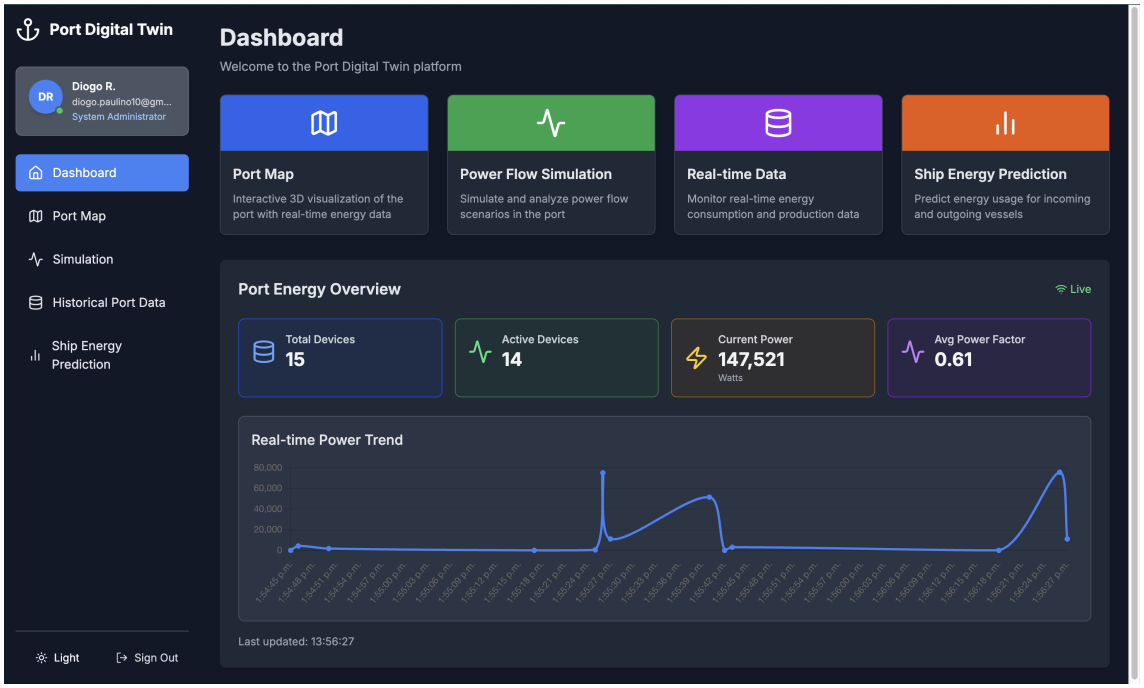
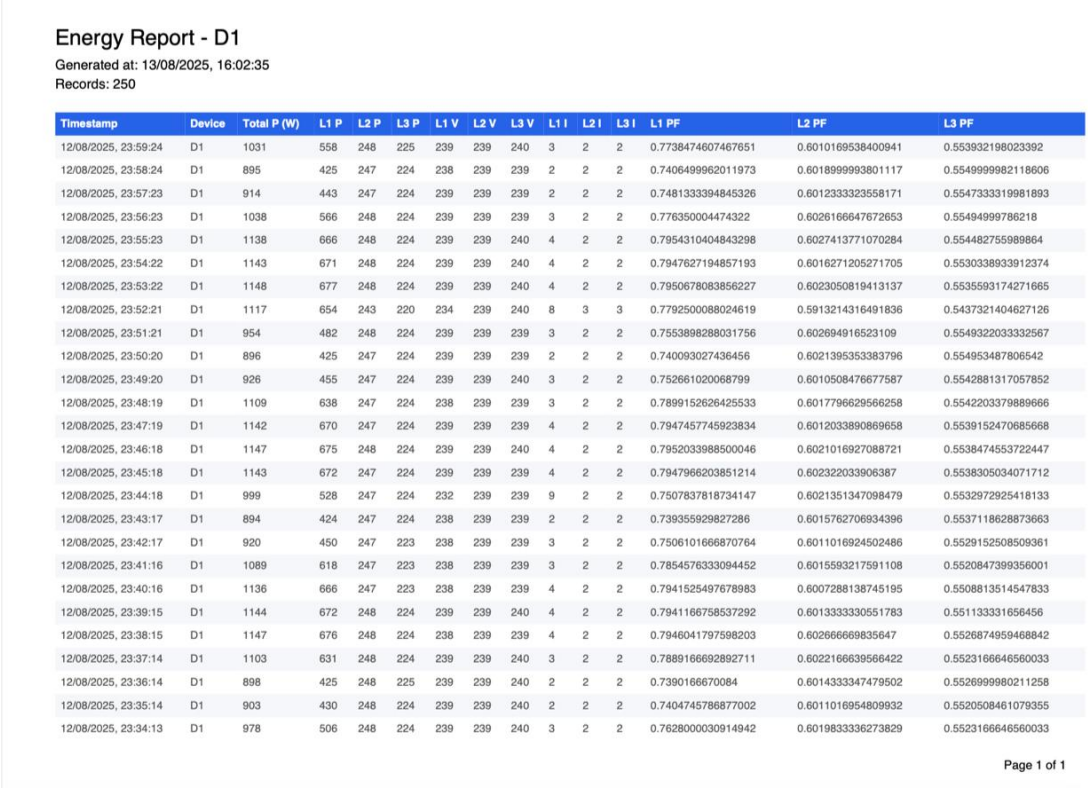


Figure. 31: Home Page With Dark Mode

4.4.10 Export Functionality

To further enhance the utility of the historical data analysis tools, a PDF export feature was implemented. Recognizing that often, there is a need to generate formal reports, archive data snapshots, or share findings with other personnel, this function allows users to create a printable, portable document of the filtered dataset directly from the interface. After a user performs a query—whether for a specific device, a time range, or both—an "Export" button becomes active. To achieve this, the application leverages the jsPDF and jspdf-autotable libraries on the client-side. When the button is clicked, the frontend: Gathers the currently displayed data from its

state. Dynamically generates a new PDF document in the browser. Adds a title to the document, including the query parameters (e.g., "Energy Report - D1", "Generated at 25/07/2025", as can be observed in fig. 32).



Energy Report - D1

Generated at: 13/08/2025, 16:02:35

Records: 250

Timestamp	Device	Total P (W)	L1 P	L2 P	L3 P	L1 V	L2 V	L3 V	L1 I	L2 I	L3 I	L1 PF	L2 PF	L3 PF
12/08/2025, 23:59:24	D1	1031	558	248	225	239	239	240	3	2	2	0.7738474607467651	0.6010169538400941	0.553932198023392
12/08/2025, 23:58:24	D1	895	425	247	224	238	239	239	2	2	2	0.7406499962011973	0.6018999993801117	0.5549999982118606
12/08/2025, 23:57:23	D1	914	443	247	224	239	239	239	2	2	2	0.7481333394845326	0.6012333323558171	0.5547333319981893
12/08/2025, 23:56:23	D1	1038	566	248	224	239	239	239	3	2	2	0.776350004474322	0.6026166647672653	0.55494999786218
12/08/2025, 23:55:23	D1	1138	666	248	224	239	239	240	4	2	2	0.7954310404843298	0.6027413771070284	0.554482755989864
12/08/2025, 23:54:22	D1	1143	671	248	224	239	239	240	4	2	2	0.7947627194857193	0.6016271205271705	0.5530338933912374
12/08/2025, 23:53:22	D1	1148	677	248	224	239	239	240	4	2	2	0.7950678083856227	0.6023050819413137	0.5535593174271665
12/08/2025, 23:52:21	D1	1117	654	243	220	234	239	240	8	3	3	0.7792500088024619	0.5913214316491836	0.5437321404627126
12/08/2025, 23:51:21	D1	954	482	248	224	239	239	239	3	2	2	0.7553898288031756	0.602694916523109	0.5549322033332567
12/08/2025, 23:50:20	D1	896	425	247	224	239	239	239	2	2	2	0.740093027436456	0.6021395353383796	0.554953487806542
12/08/2025, 23:49:20	D1	926	455	247	224	239	239	240	3	2	2	0.752661020068799	0.6010508476677587	0.5542881317057852
12/08/2025, 23:48:19	D1	1109	638	247	224	238	239	239	3	2	2	0.7899152626425533	0.6017796629566258	0.5542203379889666
12/08/2025, 23:47:19	D1	1142	670	247	224	239	239	239	4	2	2	0.7947457745923834	0.6012033890869658	0.5539152470685668
12/08/2025, 23:46:18	D1	1147	675	248	224	239	239	240	4	2	2	0.7952033988500046	0.6021016927088721	0.5538474553722447
12/08/2025, 23:45:18	D1	1143	672	247	224	239	239	239	4	2	2	0.7947966203851214	0.602322033906387	0.5538305034071712
12/08/2025, 23:44:18	D1	999	528	247	224	232	239	239	9	2	2	0.7507837818734147	0.6021351347098479	0.5532972925418133
12/08/2025, 23:43:17	D1	894	424	247	224	238	239	239	2	2	2	0.739355929827286	0.6015762706934396	0.5537118628873663
12/08/2025, 23:42:17	D1	920	450	247	223	238	239	239	3	2	2	0.7506101666870764	0.6011016924502486	0.5529152508509361
12/08/2025, 23:41:16	D1	1089	618	247	223	238	239	239	3	2	2	0.7854576333094452	0.6015593217591108	0.5520847399356001
12/08/2025, 23:40:16	D1	1136	666	247	223	238	239	239	4	2	2	0.7941525497678983	0.6007288138745195	0.5508813514547833
12/08/2025, 23:39:15	D1	1144	672	248	224	239	239	240	4	2	2	0.7941166758537292	0.6013333330551783	0.551133331656456
12/08/2025, 23:38:15	D1	1147	676	248	224	238	239	239	4	2	2	0.7946041797598203	0.602666669835647	0.5526874959468842
12/08/2025, 23:37:14	D1	1103	631	248	224	239	239	240	3	2	2	0.7889166692892711	0.6022166639566422	0.5523166646560033
12/08/2025, 23:36:14	D1	898	425	248	225	239	239	240	2	2	2	0.7390166670084	0.6014333347479502	0.5526999980211258
12/08/2025, 23:35:14	D1	903	430	248	224	239	239	240	2	2	2	0.7404745786877002	0.6011016954809932	0.5520508461079355
12/08/2025, 23:34:13	D1	978	506	248	224	239	239	240	3	2	2	0.7628000030914942	0.6019833336273829	0.5523166646560033

Page 1 of 1

Figure. 32: Energy Report Generated of Data from Device D1

Renders the filtered data into a well-formatted table within the PDF. Triggers a file download of the generated report. This client-side process offers a fast and responsive user experience without requiring additional server-side processing. The resulting PDF is a clean, professional-looking report that preserves the data in a read-only format, making it ideal for official documentation and sharing. This provides a strong function for port authorities to archive and communicate their operational data, supporting a wide range of analytical and reporting workflows.

4.4.11 Layer Logic

Although the system logic is mainly delegated to the Middleware and Microservices, some of it still exists in the presentation layer to keep everything functioning as it should.

The first component to discuss is the **DeviceDataHealthMonitor.ts**. One of the most prominent constraints, inherent to any DT is the possibility of a problem in the process of delivering data

from the IoT devices to the DT system. This was an expected issue that occurred frequently during the development process, usually due to external factors. To minimize the damage caused by this disconnection from the real world, an alternative was developed. This is called the Historical Mode. Since the DT essentially becomes unable to visualize current real-time data, the focus of the system shifts to the visualization of previously collected historical data.

The **DeviceDataHealthMonitor.ts** was implemented to specifically monitor the websocket that serves as the pipeline for real-time data for the DT. The Health Monitor system performs a health check every 15 seconds, and after three consecutive failed checks, it notifies the user of an abnormality in the data flow. This notification, as can be seen in fig. 33, informs the user of this problem and indicates that the DT's functions are now limited to Historical Mode.

Figure. 33: Data Flow Interruption Notification

A second reminder was added to the historical data page to further clarify this problem. It comes in the form of a simple UI element with a light yellow background, as can be observed in fig. 34, with the sole objective of clarifying the current limitations of the system. Another notable change in the UI relates to the top left element that changes from connected green to a red element stating that the DT is currently disconnected. When the flow of Data resumes to normal, this element returns to the connected green, and the yellow warning disappears.

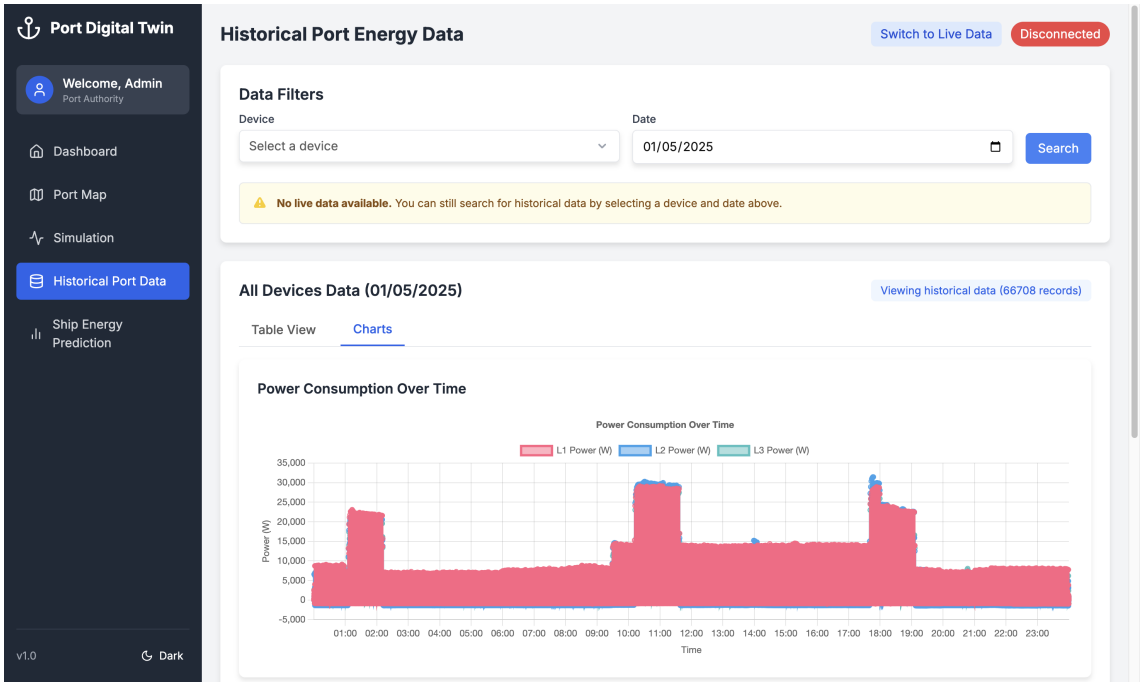


Figure. 34: Data Flow Interruption Warning

The rest of the logic mainly concentrates on connecting the websockets to the presentation layer (**DataReceiver.ts**), and fetching the required endpoints to complete the user requests (**EnergyDataService.ts, SimulationService.ts, and VesselSimulationService.ts**)

5 System Deployment

The final step in the development phase consisted on the deployment of the system in one of the team's servers. Until this point the system was run locally on a development environment, but to fully release the first version of the DT system, the application should be easily accessible, in any device with an web connection.

To achieve this, two main tools were used: Docker[20] for containerizing each service for a easier deployment, and Nginx[21] to serve as a reverse proxy.

5.1 Service Containerization and Orchestration

Docker was chosen to solve the classic "it works on my machine" problem and facilitating deployment. By creating a container for each of the system's components, each service runs in an isolated environment with all its dependencies, totally independent of the host machine's configuration. To integrate Docker into the system, two types of files were created: Dockerfiles for defining the image of each service, and a docker-compose.yml file for manage the multi-container application.

For each component of the system, a Dockerfile was written to provide Docker with a set of instructions on how to build a container image. While each file is made for to its specific service, they all follow best practices for security and efficiency. Each service starts from an official, lightweight base image (python:3.11-slim or node:18-alpine) to minimize the final image size.

Dependencies are copied and installed in an early layer. This uses Docker's layer caching, so dependencies are only re-installed if the requirements file (requirements.txt or package.json) changes, speeding up subsequent builds. For enhanced security, a dedicated non-root user is created and used to run the application. This prevents processes inside the container from having root privileges on the container's filesystem, mitigating potential security vulnerabilities.

The HEALTHCHECK instruction is also implemented in all services. Docker periodically runs this command to check if the container is still operating correctly. If a health check fails repeatedly, Docker can automatically restart the container, helping to guarantee high availability.

A good example of optimization is the Presentation layer Dockerfile, which uses a multi-stage build, as presented in listing 7.

[20] https://www.docker.com/
[21] https://nginx.org/

```
# Frontend Dockerfile - Multi-stage build
# Stage 1: Build
FROM node:18-alpine AS builder

WORKDIR /app

# Copy package files
COPY package*.json ./

# Accept build-time backend URL for Next.js public env
ARG NEXT_PUBLIC_BACKEND_URL
ENV NEXT_PUBLIC_BACKEND_URL=${NEXT_PUBLIC_BACKEND_URL}

# Install dependencies including dev dependencies for building
RUN npm ci && npm cache clean --force

# Copy source code
COPY . .

# Build the Next.js application
RUN npm run build

# Stage 2: Production
FROM node:18-alpine AS runner

# Install wget for health checks
RUN apk add --no-cache wget

WORKDIR /app

# Create non-root user
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001

# Copy built application from builder stage
COPY --from=builder --chown=nextjs:nodejs /app/public ./public
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs

# Expose port
EXPOSE 3000

# Set environment
ENV NODE_ENV=production
ENV PORT=3000
ENV HOSTNAME=0.0.0.0

```

```
# Health check (simplified since we don't have a health endpoint)
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD wget --no-verbose --tries=1 --spider http://localhost:3000/ || exit 1

# Start the application
CMD ["node", "server.js"]
```

Listing 7: Frontend Dockerfile using a multi-stage build.

This approach uses a first stage (builder) with the full Node.js development environment to build the Next.js application. The second, final stage (runner) then copies only the compiled, production-ready artifacts from the builder stage. The result is a much smaller and more secure production image, as it does not contain build tools, source code, or development dependencies.

While Dockerfiles define individual services, docker-compose.yml defines how they run and interact together. It is the central configuration file for orchestrating the entire application stack with a single command (docker-compose up). The docker compose file used manage the application can be seen in listing 8.

```
services:
  # Backend API
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: port-backend
    restart: unless-stopped
    environment:
      NODE_ENV: production
      PORT: 5001
      MONGODB_URI: *********
      JWT_SECRET: ************
      DC_POWER_FLOW_API: http://dc-power-flow:5002
      VESSEL_API: http://vessel-modelling:5003
      VESSEL_API_URL: http://vessel-modelling:5003
      FRONTEND_URL: https://portdt.prsma.com
      SMTP_HOST: smtp.gmail.com
      SMTP_PORT: 587
      SMTP_USER: diogo.paulino10@gmail.com
      SMTP_PASS: ************
    networks:
      - port-network
    ports:
      - "127.0.0.1:5001:5001"
    volumes:
      - ./backend/logs:/app/logs
```

```
    extra_hosts:
      - "host.docker.internal:host-gateway"

  # DC Power Flow Service
  dc-power-flow:
    build:
      context: ./dc_power_flow
      dockerfile: Dockerfile
    container_name: port-dc-power-flow
    restart: unless-stopped
    environment:
      FLASK_ENV: production
      PORT: 5002
    networks:
      - port-network
    ports:
      - "127.0.0.1:5002:5002"
    volumes:
      - ./dc_power_flow/data:/app/data
      - ./dc_power_flow/logs:/app/logs
      - ./dc_power_flow/output:/app/output
      # Mount source code for development (optional)
      - ./dc_power_flow/main.py:/app/main.py
      - ./dc_power_flow/api_server.py:/app/api_server.py
      - ./dc_power_flow/run_simulation.sh:/app/run_simulation.sh

  # Vessel Modelling Service
  vessel-modelling:
    build:
      context: ./vessel_modelling
      dockerfile: Dockerfile
    container_name: port-vessel-modelling
    restart: unless-stopped
    environment:
      FLASK_ENV: production
      PORT: 5003
    networks:
      - port-network
    ports:
      - "127.0.0.1:5003:5003"
    volumes:
      - ./vessel_modelling/data:/app/data
      - ./vessel_modelling/logs:/app/logs
      - ./vessel_modelling/output:/app/output
      - ./vessel_modelling/Data:/app/Data

  # Daily automation to run vessel_automation.py at 02:00
  vessel-automation:
    build:
```

```
77      context: ./vessel_modelling
78      dockerfile: Dockerfile.automation
79    container_name: port-vessel-automation
80    environment:
81      API_BASE_URL: http://vessel-modelling:5003
82      TZ: UTC
83    depends_on:
84      - vessel-modelling
85    networks:
86      - port-network
87    restart: unless-stopped
88    volumes:
89      - ./vessel_modelling/logs:/app/logs
90      - ./vessel_modelling/output:/app/output
91
92  # Frontend
93  frontend:
94    build:
95      context: ./frontend
96      dockerfile: Dockerfile
97      args:
98        NEXT_PUBLIC_BACKEND_URL: https://portdt.prsma.com
99      shm_size: 1gb
100   container_name: port-frontend
101   restart: unless-stopped
102   mem_limit: 2gb
103   memswap_limit: 2gb
104   environment:
105     NODE_ENV: production
106     NEXT_PUBLIC_BACKEND_URL: https://portdt.prsma.com
107   depends_on:
108     - backend
109     - dc-power-flow
110     - vessel-modelling
111   networks:
112     - port-network
113   ports:
114     - "127.0.0.1:3000:3000"
115
116 # Nginx reverse proxy
117 nginx:
118   image: nginx:alpine
119   container_name: nginx-proxy
120   ports:
121     - "80:80"
122     - "443:443"
123     - "8080:8080"  # Development port
124   volumes:
125     - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
```

```
126       - ./nginx/ssl:/etc/letsencrypt:ro
127       - ./nginx/logs:/var/log/nginx
128       - certbot-webroot:/var/www/certbot:ro
129
130     networks:
131       - port-network
132     depends_on:
133       - frontend
134       - backend
135       - dc-power-flow
136       - vessel-modelling
137     restart: unless-stopped
138
139   # Certbot for SSL certificates (production)
140   certbot:
141     image: certbot/certbot
142     container_name: certbot
143     volumes:
144       - ./nginx/ssl:/etc/letsencrypt
145       - certbot-webroot:/var/www/certbot
146     command: certonly --webroot --webroot-path=/var/www/certbot --email
      diogo.paulino10@gmail.com --agree-tos --no-eff-email -d portdt.prsma.
      com
147     profiles:
148       - ssl-setup
149
150 volumes:
151   certbot-webroot:
152
153 networks:
154   port-network:
155     driver: bridge
```

Listing 8: Final Docker Compose file for managing all application services.

This final configuration defines seven services. In addition to the four main application components: frontend (Presentation Layer), backend (Middleware Layer), dc-power-flow, vessel-modelling, it also includes the vessel-automation for running scheduled tasks, nginx as a reverse proxy, and certbot for managing SSL certificates.

The ports are bound to 127.0.0.1, meaning they are not exposed directly to the public internet. Instead, all external traffic is routed through the Nginx proxy, which enhances security.

A custom bridge network named port-network is created. All services are attached to this network, allowing them to communicate with each other using their service names as hostnames (e.g., the backend (middleware) can reach the DC Power Flow service at http://dc-power-flow:5002). This

provides a secure and isolated network for the application.

The depends_on directive ensures that services are started in a logical order. For instance, the frontend depends on the backend (middleware), and nginx depends on all application services, ensuring that the APIs and web server are available when the proxy starts.

5.2 Request Routing and SSL Termination with a Reverse Proxy

With all services running in Docker containers, a single entry point was needed to manage and direct incoming web traffic. Nginx was configured as a reverse proxy to fulfil this role. Instead of users accessing each service on its specific port (e.g., :3000, :5001), they access a single domain, and Nginx routes their requests to the correct internal service based on the URL path. This provides a clean and unified entry point for the entire application. Adding to this, Nginx handles SSL termination, encrypting all traffic between the client and the server, and adds security headers to protect the backend services from direct exposure. To implement this, an nginx service was added to the docker-compose.yml file, and a configuration file (nginx.conf), seen in listing 9, was created to define the routing rules. The following configuration demonstrates how Nginx routes requests to the different services.

```
1
2  # Nginx reverse proxy configuration for Port Digital Twin
3  # Domain: portdt.prsma.com
4
5  events {
6      worker_connections 1024;
7  }
8
9  http {
10     # Upstream definitions for load balancing
11     upstream frontend {
12         server frontend:3000;
13     }
14
15     upstream backend {
16         server backend:5001;
17     }
18
19     upstream dc_power_flow {
20         server dc-power-flow:5002;
21     }
22
23     upstream vessel_modelling {
```

```
        server vessel-modelling:5003;
    }

    # HTTP to HTTPS redirect (for production)
    server {
        listen 80;
        server_name portdt.prsma.com www.portdt.prsma.com;

        # For Let's Encrypt challenges
        location /.well-known/acme-challenge/ {
            root /var/www/certbot;
            allow all;
        }

        # Redirect all other traffic to HTTPS
        location / {
            return 301 https://$server_name$request_uri;
        }
    }

    # Main HTTPS server block
    server {
        listen 443 ssl http2;
        server_name portdt.prsma.com www.portdt.prsma.com;

        # SSL configuration
        ssl_certificate /etc/letsencrypt/live/portdt.prsma.com/fullchain.
    pem;
        ssl_certificate_key /etc/letsencrypt/live/portdt.prsma.com/privkey.
    pem;
        ssl_session_timeout 1d;
        ssl_session_cache shared:SSL:50m;
        ssl_session_tickets off;

        # Modern SSL configuration
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers
    ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-R
    ;
        ssl_prefer_server_ciphers off;

        # HSTS (optional, enable after testing)
        # add_header Strict-Transport-Security "max-age=63072000" always;

        # Security headers
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;
        add_header X-XSS-Protection "1; mode=block";
        add_header Referrer-Policy "strict-origin-when-cross-origin";
```

```
69
70          # Frontend - Main application
71          location / {
72              proxy_pass http://frontend;
73              proxy_http_version 1.1;
74              proxy_set_header Upgrade $http_upgrade;
75              proxy_set_header Connection 'upgrade';
76              proxy_set_header Host $host;
77              proxy_set_header X-Real-IP $remote_addr;
78              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
79              proxy_set_header X-Forwarded-Proto $scheme;
80              proxy_cache_bypass $http_upgrade;
81              # Rate limiting for static content
82              limit_req zone=static burst=50 nodelay;
83          }
84
85          # Backend API
86          location /api/ {
87              proxy_pass http://backend;
88              proxy_http_version 1.1;
89              proxy_set_header Host $host;
90              proxy_set_header X-Real-IP $remote_addr;
91              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
92              proxy_set_header X-Forwarded-Proto $scheme;
93
94              # CORS headers for API
95              add_header Access-Control-Allow-Origin *;
96              add_header Access-Control-Allow-Methods "GET, POST, PUT, DELETE
    , OPTIONS";
97              add_header Access-Control-Allow-Headers "DNT,User-Agent,
    X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range,
    Authorization";
98
99              # Handle preflight requests
100             if ($request_method = 'OPTIONS') {
101                 add_header Access-Control-Allow-Origin *;
102                 add_header Access-Control-Allow-Methods "GET, POST, PUT,
    DELETE, OPTIONS";
103                 add_header Access-Control-Allow-Headers "DNT,User-Agent,
    X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range,
    Authorization";
104                 add_header Access-Control-Max-Age 1728000;
105                 add_header Content-Type 'text/plain; charset=utf-8';
106                 add_header Content-Length 0;
107                 return 204;
108             }
109             # Rate limiting for API
110             limit_req zone=api burst=20 nodelay;
111         }
```

```
112
113         # DC Power Flow API
114         location /dc-api/ {
115             rewrite ^/dc-api/(.*) /$1 break;
116             proxy_pass http://dc_power_flow;
117             proxy_http_version 1.1;
118             proxy_set_header Host $host;
119             proxy_set_header X-Real-IP $remote_addr;
120             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
121             proxy_set_header X-Forwarded-Proto $scheme;
122             # Rate limiting for API
123             limit_req zone=api burst=10 nodelay;
124         }
125
126         # Vessel Modelling API
127         location /vessel-api/ {
128             rewrite ^/vessel-api/(.*) /$1 break;
129             proxy_pass http://vessel_modelling;
130             proxy_http_version 1.1;
131             proxy_set_header Host $host;
132             proxy_set_header X-Real-IP $remote_addr;
133             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
134             proxy_set_header X-Forwarded-Proto $scheme;
135             # Rate limiting for API
136             limit_req zone=api burst=10 nodelay;
137         }
138
139         # WebSocket support for real-time features
140         location /socket.io/ {
141             proxy_pass http://backend;
142             proxy_http_version 1.1;
143             proxy_set_header Upgrade $http_upgrade;
144             proxy_set_header Connection "upgrade";
145             proxy_set_header Host $host;
146             proxy_set_header X-Real-IP $remote_addr;
147             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
148             proxy_set_header X-Forwarded-Proto $scheme;
149         }
150
151         # Static file caching
152         location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$
    {
153             proxy_pass http://frontend;
154             proxy_http_version 1.1;
155             proxy_set_header Host $host;
156             proxy_set_header X-Real-IP $remote_addr;
157             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
158             proxy_set_header X-Forwarded-Proto $scheme;
159             # Caching for static assets
```

```
            expires 1y;
            add_header Cache-Control "public, immutable";
            add_header X-Cache-Status "STATIC";
        }
    }
    # ... other headers
}
```

Listing 9: Nginx configuration for reverse proxy and SSL.

This configuration defines two server blocks. The first, listening on port 80, has two purposes: serving challenges for Let's Encrypt SSL certificate renewal and redirecting all other HTTP traffic to HTTPS. The main server block listens on port 443 for HTTPS traffic. It is configured with the SSL certificates managed by certbot. The location blocks define the routing logic:

- Requests to the root URL (/) are proxied to the frontend service.
- Requests prefixed with /api/ are routed to the backend (middleware) service.
- Requests prefixed with /dc-api/ or /vessel-api/ are routed to their respective microservices. For these, a rewrite rule strips the prefix from the URL path before passing the request to the upstream service, ensuring the microservice receives the expected path.

This architecture centralizes concerns like SSL encryption and request routing in Nginx, allowing each microservice to focus solely on its own business logic.

6 System Evaluation

This chapter goes over the evaluation of the DT system. Given its intended application within the Shift2DC project for port infrastructure management, the evaluation focuses on the non-functional requirements that guarantee its operational viability. The primary objective was to validate the architectural design choices, particularly the microservice approach, by assessing three key quality attributes: system performance under realistic conditions, reliability and fault tolerance in the event of component failure, and data pipeline latency for near real-time monitoring. The following tests were conducted on the fully deployed system, which operates using a Docker containerization system discussed on the previous section 5, to simulate a production environment.

6.1 System Performance Evaluation

A series of benchmarks were conducted to quantify the system's performance under realistic conditions. The primary goal was to measure the latency of key operations and ensure the architecture could support the demands of a production system. Client-side measurements were taken using the Network tab in Google Chrome's Developer Tools to precisely record the time from when a request was initiated to when a response was fully received.

The latency of the simulations was measured to account for the full round-trip time. This metric captures the total time from a user's click in the browser, through the Nginx proxy, to the appropriate microservice, and back to the browser with the final result.

6.1.1 Vessel Modeling Latency

For the Vessel Energy Modeling service, direct measurements from 10 test runs were recorded for each type of simulation. The detailed test results can be observed in table 2. The average latency for a registered vessel (requiring a database lookup) was 2550 ms. For a custom vessel (requiring a Python model execution), the average latency was 3160 ms. The 600 ms difference between the two cases can be directly attributed to the computational overhead of the prediction model. While these response times are in the 2-4 second range, they are considered acceptable for a detailed planning and simulation tool where the user's primary goal is analytical insight rather than instantaneous feedback.

Table 2: Vessel Modeling Latency Measurements (10 Test Runs)

Vessel Type	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**Avg. Latency (s)**
Registered Vessel (Database Lookup)	2.500 (s)	2.600 (s)	2.550 (s)	2.510 (s)	2.590 (s)	2.520 (s)	2.580 (s)	2.530 (s)	2.570 (s)	2.550 (s)	**2.550** (2550 ms)
Custom Vessel (Python Model)	3.110 (s)	3.210 (s)	3.160 (s)	3.120 (s)	3.200 (s)	3.130 (s)	3.190 (s)	3.140 (s)	3.180 (s)	3.160 (s)	**3.160** (3160 ms)

Data retrieval requests that do not require computational modeling demonstrated significantly faster performance. For example, accessing historical vessel movements or simulation records achieved near-instant results. Based on 10 test runs, the average latency for this type of operation was 234 ms.

Table 3: Latency for Simulations Retrieval Operations

Operation Type	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**Avg. Latency**
Historical Retrieval (Database Access)	228 (ms)	238 (ms)	233 (ms)	229 (ms)	237 (ms)	243 (ms)	223 (ms)	226 (ms)	240 (ms)	223 (ms)	**234** (ms)

6.1.2 DC Power Flow Latency:

For the DC Power Flow service, a complete simulation cycle, including data retrieval from the database, the pandapower simulation itself, and result processing, took an average of 31 seconds as can be seen in table 4. This complex calculation is the most computationally intensive part of the system. Successfully executing it well within the 60-second operational requirement validates the performance of the simulation microservice and its suitability for near real-time grid monitoring.

Table 4: DC Power Flow Simulation Latency (Complete Cycle)

Service	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**Avg. Latency (s)**
DC Power Flow (pandapower Model)	30.5 (s)	32.1 (s)	31.0 (s)	30.2 (s)	31.5 (s)	30.8 (s)	31.4 (s)	30.9 (s)	31.6 (s)	30.0 (s)	**31.0**

6.1.3 Historical Data Latency

The performance of the historical data retrieval functionalities was also evaluated. Two distinct query types were executed ten times each. The first query was a broad request where no specific device filter was applied, retrieving all device data for an entire operational day, resulting in an average latency of 4640 ms. The second test utilized a more selective filter to retrieve data for a

single device over the same period, yielding a significantly lower average latency of 2150 ms. The results of each test can be observed in table 5.

Table 5: Historical Data Retrieval Latency

Query Type	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**Avg. Latency (ms)**
Wide Query (All Device Data)	4600 (ms)	4700 (ms)	4650 (ms)	4580 (ms)	4720 (ms)	4610 (ms)	4660 (ms)	4590 (ms)	4700 (ms)	4590 (ms)	**4640**
Filtered Query (Single Device Data)	2100 (ms)	2200 (ms)	2150 (ms)	2120 (ms)	2180 (ms)	2110 (ms)	2170 (ms)	2130 (ms)	2190 (ms)	2150 (ms)	**2150**

6.2 System Scalability

Beyond the baseline performance for a single user, the system was evaluated under load to verify scalability when handling an increasing number of active users.

6.2.1 Test Environment and Methodology

To ensure the reproducibility of these tests, all evaluations were performed in a controlled environment with the following specifications:

- **Host Machine:**
  - **CPU:** ARM Neoverse-N1 (2 cores @ 2.0 GHz)
  - **Memory:** 4GB RAM
  - **OS:** Ubuntu 24.04.1 LTS
  - **Docker Engine:** version 28.0.1
- **Test Protocol:**
  - All system components (except the SSOT which runs on a separate machine) were run as Docker containers on the same host machine to minimize network latency variability.
  - The load-generating client scripts were run directly on the host machine.
  - For each load level (e.g., 1, 10, 50, 100 clients), the system was allowed to run for 60 seconds to stabilize before a 5-minute measurement period began.
  - Resource usage (CPU, Memory) for the relevant Docker containers was captured every second using the docker stats command, and the average values over the measurement period were recorded.

This standardized methodology ensures that the following results are both reliable and comparable across different test scenarios.

6.2.2 Scalability of the Real-Time Simulation Broadcast

A design decision of the DC Power Flow system is its decoupling of the computationally expensive simulation from the data delivery to users. Instead of each user triggering a new simulation, a single, periodic background task runs the simulation once every 60 seconds. The result is then broadcast to all concurrently connected users via websockets.

Considering this, the traditional load test of hitting a request endpoint is not applicable. Instead, the testing focused on the system's ability to handle a large number of concurrent client connections, as this is the true measure of its scalability. The key question is not "how does response time degrade?" but more of "how much resource overhead does each additional user add to the system?"

For this test, a Python script using the websockets[22] library was created to simulate an increasing number of clients connecting to the server and listening for simulation results. This script can be seen in Listing 10 The server's CPU and Memory consumption were monitored during the test to measure the impact of these connections.

```
# --- Configuration ---
BACKEND_URL = "http://localhost:5001"
NUM_CLIENTS = int(sys.argv[1]) if len(sys.argv) > 1 else 1
# Test duration in seconds
TEST_DURATION = int(sys.argv[2]) if len(sys.argv) > 2 else 60

# Statistics
messages_received = 0
connection_errors = 0
clients_connected = 0

async def connect_and_listen(client_id: int):
    """
    Simulates a single client connecting to the Socket.IO server and
    passively listening for simulation updates.
    """
    global messages_received, connection_errors, clients_connected

    sio = socketio.AsyncClient(
        reconnection=False,
        logger=False,
        engineio_logger=False
```

[22] https://websockets.readthedocs.io

```
    )

    @sio.on('simulation_update')
    def on_simulation_update(data):
        global messages_received
        messages_received += 1

    @sio.on('connect')
    def on_connect():
        global clients_connected
        clients_connected += 1

    @sio.on('connect_error')
    def on_connect_error(data):
        global connection_errors
        connection_errors += 1

    try:
        # Connect to the server
        await sio.connect(BACKEND_URL, transports=['websocket', 'polling'])

        await asyncio.sleep(TEST_DURATION)
        await sio.disconnect()

    except Exception as e:
        connection_errors += 1
        print(f"Client {client_id} error: {e}", file=sys.stderr)

async def main():
    """
    Main function to orchestrate the test by launching all clients.
    """
    global messages_received, connection_errors, clients_connected
    print(f"Starting test...\n")

    start_time = time.time()
    tasks = [connect_and_listen(i) for i in range(NUM_CLIENTS)]

    # Run all clients concurrently
    await asyncio.gather(*tasks, return_exceptions=True)

    end_time = time.time()
    duration = end_time - start_time
    print(f"\n=== Test Results ===")
    print(f"Total Duration: {duration:.2f} seconds")
    print(f"Clients Successfully Connected: {clients_connected}/{
    NUM_CLIENTS}")
    print(f"Connection Errors: {connection_errors}")
    print(f"Total Messages Received: {messages_received}")
```

```
71      if clients_connected > 0:
72          print(f"Average Messages per Client: {messages_received/
        clients_connected:.2f}")
73      print(f"\n{'='*50}")
74
75  if __name__ == "__main__":
76      try:
77          asyncio.run(main())
78      except KeyboardInterrupt:
79          print("\nTest interrupted by user")
80          sys.exit(0)
```

Listing 10: Python script used for WebSocket scalability testing.

The results, presented in Table 6, are very positive and validate the efficiency of the system. CPU usage remained essentially constant, increasing from a baseline of 0.47% to only 0.50% with 100 concurrent clients. This negligible increase confirms that the computational cost is fixed, regardless of the number of observers. Memory consumption grew modestly from 69.43 MB to 71.89 MB, indicating a low overhead of approximately 25 KB for each client. This proves the design can scale to hundreds, or even thousands, of users without significant resource strain on the specified hardware.

Table 6: Middleware Resource Usage for DC Power Flow Broadcast Scalability

Concurrent Clients	**Average CPU Usage (%)**	**Average Memory (MB)**
1 (Baseline)	0.47%	69.43
10	0.54%	98.95
50	0.45%	69.70
100	0.50%	71.89

6.2.3 Scalability of the Real-Time Database Monitoring

This test was designed to test the system's performance under a growing number of concurrent users, focusing on two main aspects: resource scalability and message delivery success rate. While the latency was measured, the primary goal was to verify that the middleware could handle increasing client loads without significant CPU/memory strain or message loss. The system uses a 5 second polling interval to query the database, a design choice, discussed in section 4.3.3, intended to minimize database load.

The results, presented in Table 7, show the architecture's efficiency. The primary indicators of scalability, CPU and Memory usage, showed excellent performance. Average CPU consumption remained below 1% even at 100 concurrent users, a d memory growth was minimal and predictable.

The system also achieved a 100% message delivery success rate across all load levels, proving its reliability. No updates were dropped, confirming that the Socket.IO broadcast mechanism is adequate under these conditions. The measured latency, as expected, fluctuated in line with the 5 second polling window, validating that the mechanism was behaving as designed but not serving as a direct measure of load induced delay.

Table 7: Real-Time Update Performance Under Concurrent Load

Concurrent Users	Avg. Latency (s)	Avg. CPU (%)	Avg. Memory (MB)
1 (Baseline)	3.34	0.33%	91.82
5	10.13	0.60%	98.56
10	12.14	0.88%	92.73
50	4.65	0.29%	94.95
100	12.13	0.83%	98.35

6.3 System Reliability and Fault Tolerance

A series of fault injection tests were performed to verify the system's resilience, a core aspect of the microservice architecture.

6.3.1 Failure Within the Microservices

To test fault isolation, the Docker containers for the Vessel Modeling and DC Power Flow services were individually stopped. In both cases, the failure was successfully isolated to the specific feature, with the rest of the application remaining fully functional. The UI displayed clear error messages instead of crashing, as detailed in Table 8.

When the Vessel Energy Modeling container was stopped, any API request from the frontend to its endpoint (/api/vessel-modeling/...) was intercepted by the Nginx API Gateway, which returned a 503 Service Unavailable HTTP status code. The frontend is designed to catch this specific error, preventing a crash and instead triggering a notification indicating that the "Vessel Modeling service is temporarily unavailable". All other functionalities, which route to different microservices, continued to operate without interruption, confirming successful fault isolation.

Table 8: System Behavior During Microservice Failure

Failed Service	**Affected Functionality**	**Observed System Behavior**
Vessel Energy Modeling	"Ship Energy Prediction" page.	The rest of the application remained fully functional. The affected page displayed an error notification.
DC Power Flow	"Power Flow Simulation" page.	The fault was isolated to the specific page. All other application features operated normally.

A similar logic was observed in the DC Power Flow component, the failure mechanism, however, was different due to its reliance on a persistent WebSocket connection rather than a simple HTTP request. When the DC Power Flow container was terminated, the WebSocket connection was abruptly closed. The Middleware application is designed to handle this event gracefully. Instead of crashing, it degrades the functionality of the "Power Flow Simulation" page by displaying a message stating "Connection to the simulation service lost. Attempting to reconnect..." The "Last Updated" timestamp froze, clearly indicating to the user that the data being displayed was stale. This failure was perfectly isolated, allowing all other parts of the application to remain fully operational.

6.3.2 Failure of the Data Stream

To simulate an interruption in the live data pipeline, the data ingestion service was halted. The system responded exactly as designed:

1. After 45 seconds, the stale data stream was detected.
2. A notification banner appeared, informing the user the system was now in "Historical Mode."
3. The UI status indicator changed from "Connected" to "Disconnected".
4. The application remained stable, with all historical data and simulation features fully accessible.
5. Upon data stream restoration, the system automatically returned to "Live Mode".

This test validated the system's ability to handle a critical data failure gracefully, maintaining its utility and preserving user trust.

6.4 Discussion of Results

This chapter provides an interpretation of the results presented in the previous chapter. The findings are analyzed in the context of the thesis objectives, the significance of the contributions is highlighted, the work is compared against the established related work, and the inherent limitations of the study are acknowledged.

The system evaluation yielded positive results across all three pillars: performance, scalability, and reliability. These findings collectively validate the architectural and technological choices made throughout the development of the Port of Funchal's DT.

6.4.1 Performance

The latency benchmarks demonstrate that the microservice architecture, despite the inherent network overhead of inter-service communication, is highly performant. End-to-end response times for operations, such as generating custom vessel energy profiles (3160 ms) and running full DC power flow simulations (31 seconds), are well within acceptable limits for an interactive decision support tool. Furthermore, the latency of 4 seconds for real-time data updates confirms the system's ability to provide a accurate enough "live" view of port operations. This proves that a distributed system, when properly designed, can meet and exceed the performance demands of a complex DT.

6.4.2 Scalability

The scalability tests revealed important conclusions into the system's architectural efficiency. The DC Power Flow broadcast system showed excellent scalability, with average CPU usage remaining effectively flat and under 1% even when scaling from a single client to 100 concurrent clients. Memory consumption grew predictably with a minimal overhead of approximately 25 KB per connection. This validates the decision to separate the single, periodic simulation execution from the result distribution, proving the design is capable of serving a very large number of users with negligible resource strain.

The real-time database monitoring system exhibited different but equally positive results. While the latency measurements varied (ranging from 3s to 12s), this was an expected outcome dominated by the system's 5 second polling interval rather than a sign of performance degradation. The more telling metrics—CPU and memory usage, remained remarkably low, scaling modestly and proving the system was not under pressure.

Collectively, these results demonstrate that the current architecture is not only ready for the Port of Funchal's expected operational scale but also possesses significant space for future growth. The minimal resource footprint (under 1% CPU and 100 MB of RAM under full test load) indicates that future enhancements would be new functional requirements rather than performance necessities. The current system comfortably exceeds the project's requirements for a scalable platform.

6.4.3 Reliability

The fault tolerance tests also provided a practical demonstration of the theoretical benefits of a microservice architecture. The ability of the system to isolate failures and degrade gracefully, remaining operational even when the Vessel Modeling or DC Power Flow services were offline, is a testament to its resilience. Similarly, the system's automated transition into "Historical Mode" during a data stream interruption ensures it remains a useful tool for analysis and prevents user frustration. This robustness is paramount for a system intended for critical infrastructure management.

For a system intended to manage energy infrastructure under the shift2dc initiative, which uses constant innovative approaches, reliability is not a feature but a prerequisite. A system failure could have significant operational consequences. The fault tolerance tests directly demonstrated the most significant benefit of the microservice architecture in this context.

6.4.4 Limitations of the Study

Despite the successful outcomes, it is important to acknowledge the limitations of this study, which also present opportunities for future research.

1. **Scope of Simulation Models:** The vessel energy and DC power flow models, while its somewhat accurate, it is mostly simplifications of reality. They do not yet account for all possible variables, such as the impact of weather conditions on a vessel's hoteling power requirements or complex effects within the electrical grid. These models also needed to relay on estimations (e.g. the cable size for the port infrastructure), since no detailed measurement were made of this equipment as of this moment. As a result, It did not make explicit sense to verify the accuracy of the models considering the aspects discussed. Although, in future iterations that is a characteristic that should be explored and thoroughly tested.

2. **Long-Term Stability:** The performance evaluation was conducted through short-term benchmarks and load tests. The long-term stability and performance of the system over months or years of continuous operation, including potential memory leaks or database growth issues, have not been assessed.

7 Conclusion

7.1 Summary of Work

This thesis details the design, implementation, and deployment of a DT for the Port of Funchal, developed to address the challenge of managing energy infrastructure during the transition to sustainable port operations. The work translates the theoretical concept of a DT into a real tool.

The system is built on a microservice architecture that integrates three core capabilities: generating energy demand profiles for arriving vessels, DC power flow simulation for grid analysis, and real-time monitoring of 33 physical circuits. These services are unified through a central middleware layer and presented to users via an interactive web platform. This platform offers port operators with both real-time operational information, visualized on a 2.5D geospatial map, and powerful analytical tools for planning and historical analysis.

Containerized with Docker and deployed to production behind an Nginx reverse proxy, the Digital Twin is currently an operational tool used by the Port of Funchal authorities [23]. This demonstrates a complete development lifecycle, from initial design to a functional system in a real-world environment.

7.2 Main Contributions

This work makes two primary contributions to the field of Digital Twins for maritime infrastructure:

1. **A Reference Architecture for Port Digital Twins with a focus on Energy Management:** This thesis provides a documented, validated architecture for port DT systems with energy management focus. The integration patterns, technology choices (Next.js/React for presentation, Node.js/Express for middleware, Python/Flask for domain services, Socket.IO for real-time streaming), and API design principles are documented with sufficient detail to serve as a reference implementation for similar projects. Unlike purely theoretical frameworks, this architecture has been tested under real operational conditions with actual sensor deployments.

2. **An Operational Tool for Maritime Decarbonization:** The platform itself represents a new contribution. By combining vessel specific energy profile generation, DC grid simulation, and real-time monitoring in a single integrated interface, it provides capabilities not available in

[23] https://portdt.prsma.com

existing commercial port management systems. The tool itself can support difficult decisions for electrification projects, shore power implementation, and renewable energy integration, which are all critical to maritime decarbonization goals.

7.3 Implementation Challenges and Solutions

The development process revealed several significant challenges, each requiring specific architectural or implementation responses:

7.3.1 Data Reliability and Availability

The most fundamental challenge proved to be the inconsistent availability of real-time data from physical port infrastructure. Analysis of historical data streams (Figure 35) revealed device availability ranging from only 52.4% to 81.8% over a nine month period. These gaps result from sensor failures, network connectivity issues, and power interruptions at the remote monitoring devices.

For a Digital Twin system, this represents an existential challenge, the real-time connection to the physical asset is the defining characteristic that distinguishes a DT from a static simulation or historical analysis tool. Complete system failure during data outages would severely limit operational utility.

Figure. 35: Device data availability from December 2024 to September 2025. Each horizontal line represents a different device. The green segments indicate periods when data was successfully received, while the gaps represent sensor or network outages. The overall data availability percentages, ranging from 52.4% to 81.8%, highlight the inconsistent data stream that the system must tolerate.

The implemented solution employs graceful degradation through a WebSocket health monitoring system. The presentation layer monitors connection health every 15 seconds, and after three consecutive failures, automatically switches to "Historical Mode." Clear visual indicators inform users of the limitation, and the system remains fully functional for historical analysis and custom simulations. When real-time data resumes, the system automatically reconnects without user intervention.

This design decision prioritizes continuous availability over perfect real-time accuracy, recognizing that a Digital Twin with 80% real-time coverage plus robust historical capabilities provides more value than a system that becomes unusable during outages.

7.3.2 Technological Heterogeneity

Another challenge is the integration complexity and the future maintainability of the whole system. This is a classical limitation, discussed previously in chapter 2, of the microservice architecture [31,34]. Due to the fact that there is a heterogeneous set of technologies, and data formats employed in the different microservices, combined with the consideration that these components might be developed by different members and teams within the overall Shift2DC consortium, the complexity of the system is expected to get exponentially higher as development advances.

The design solution proposed for this challenging aspect is effectively a protocol to be followed. In this case, to mitigate the inherent problems of the microservice architecture, clear and strict documentation must be written for each layer and service to ensure future maintainability and scalability

7.3.3 Computational Performance

A significant design challenge was guaranteeing real-time results without adding computational costs to the point its detrimental to the overall system. The decision to decouple computationally intensive simulations into asynchronous, containerized services proved to be one of the most crucial. By running the DC power flow on a fixed 60 second interval, the system balanced the need for new data against resource constraints, validating that a microservice approach can manage performance bottlenecks in DT systems.

7.4 Future Work

Despite the successful deployment, several aspects of the system remain that represent opportunities for future improvements:

7.4.1 Vessel Schedule Integration

The current system relies on overnight scraping of the APRAM website for vessel schedules, providing daily updates rather than true real-time tracking. Future work should integrate with live vessel tracking APIs (data from MarineTraffic, VesselFinder) or the Single Logistics Window platform (Janela Única Logística). This would enable the Digital Twin to update vessel positions and energy profiles dynamically as ships arrive and depart, rather than on a fixed daily schedule.

7.4.2 DC Grid Visualization

The DC power flow results are currently presented in tabular and chart formats. A more intuitive representation would be a circuit diagram visualization showing the actual network topology with color-coded nodes representing voltage levels and animated flows showing power distribution. This would make the results more accessible to non-technical port operators and facilitate identification of grid bottlenecks or inefficiencies.

7.5 Closing Remarks

The transition to sustainable maritime operations requires not just new technologies, like shore power, electrified equipment, renewable energy, but also the tools to plan, optimize, and operate these systems effectively. Digital Twin technology is a turning point for this transition, providing the visibility needed to make informed decisions about complex, interconnected energy systems.

The Port of Funchal Digital Twin is now operational, providing daily information into energy consumption and supporting planning for future electrification projects. This system will serve as both a practical tool for port operations and a research example for advancing Digital Twin methodologies in the maritime world.

This thesis demonstrates that Digital Twin systems for port energy management are technically feasible and can be built using established software engineering patterns and open-source technologies. The microservice architecture, while introducing integration complexity, provides the flexibility and scalability necessary for systems that must evolve alongside fast changing technologies.

References

[1] J. Uchechukwu and J. U. Mba, "Advancing sustainability and efficiency in maritime operations: Integrating green technologies and autonomous systems in global shipping," *International Journal of Science and Research Archive*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:274700410

[2] V. Koilo, "Decarbonization in the maritime industry: Factors to create an efficient transition strategy," *Environmental Economics*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271450824

[3] E. Kostidi and D. Lyridis, "Decarbonizing ports for a sustainable future: Challenges and strategies," *Technical Annals*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:277880207

[4] L. Zhang, Q. liang Zeng, and L. Wang, "How to achieve comprehensive carbon emission reduction in ports? a systematic review," *Journal of Marine Science and Engineering*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269425692

[5] D. R. Cunha, C. B. M. Oliveira, M. de Santana Porte, S. S. Cutrim, and N. N. Pereira, "Strategies for decarbonisation in the port and maritime sector: Key challenges and leading ports," *Revista de Gestão Social e Ambiental*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277834012

[6] M. Fotopoulou, D. C. Rakopoulos, D. Trigkas, F. Stergiopoulos, O. Blanas, and S. Voutetakis, "State of the Art of Low and Medium Voltage Direct Current (DC) Microgrids," *Energies*, vol. 14, no. 18, p. 5595, 2021. [Online]. Available: https://doi.org/10.3390/en14185595

[7] M. W. Grieves and J. Vickers, "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems," in *Transdisciplinary Perspectives on Complex Systems*. Springer, 2017, pp. 85–113. [Online]. Available: https://doi.org/10.1007/978-3-319-38756-7_4

[8] A. Fuller, Z. Fan, and C. Day, "Digital Twin: Enabling Technologies, Challenges and Open Research," *IEEE Access*, 2020. [Online]. Available: https://doi.org/10.1109/

access.2020.2998358

[9] J. Vachálek, L. Bartalský, O. Rovný, D. Šišmišová, M. Morháč, and M. Lokšík, "The digital twin of an industrial production line within the industry 4.0 concept," *IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pp. 258–262, 2017. [Online]. Available: https://doi.org/10.1109/pc.2017.7976223

[10] C. Cimino, E. Negri, and L. Fumagalli, "Review of digital twin applications in manufacturing," *Computers in Industry*, vol. 113, p. 103130, 2019. [Online]. Available: https://doi.org/10.1016/j.compind.2019.103130

[11] F. Tao and M. Zhang, "Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing," *IEEE Access*, vol. 5, pp. 20 418–20 427, 2017. [Online]. Available: https://doi.org/10.1109/access.2017.2756069

[12] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, "Shaping the digital twin for design and production engineering," *Cirp Annals-manufacturing Technology*, vol. 66, no. 1, pp. 141–144, 2017. [Online]. Available: https://doi.org/10.1016/j.cirp.2017.04.040

[13] S. Thelen, F. Eder, M. Melzer, D. W. Nunes, M. Stadler, C. Rechenauer, M. Obergrießer, R. Jubeh, K. Volbert, and J. Dünnweber, "A Slim Digital Twin for a Smart City and Its Residents," *Symposium on Information and Communication Technology*, 2023. [Online]. Available: https://doi.org/10.1145/3628797.3628936

[14] K. Bruynseels, F. S. de Sio, and J. van den Hoven, "Digital Twins in Health Care : Ethical Implications of an Emerging Engineering Paradigm," *Frontiers in Genetics*, vol. 9, p. 31, 2018. [Online]. Available: https://doi.org/10.3389/fgene.2018.00031

[15] W. Ahmad, M. Mutz, and D. Werth, "Digital Twin of Rail for Defect Analysis," *International Conference Virtual and Augmented Reality Simulations*, 2024. [Online]. Available: https://doi.org/10.1145/3657547.3657549

[16] M. Attaran and B. G. Çelik, "Digital Twin: Benefits, use cases, challenges, and opportunities," *Decision Analytics Journal*, vol. 6, p. 100165, 2023. [Online]. Available: https://doi.org/10.1016/j.dajour.2023.100165

[17] E. J. Tuegel, A. R. Ingraffea, T. Eason, and S. M. Spottswood, "Reengineering Aircraft Structural Life Prediction Using a Digital Twin," *International Journal of Aerospace Engineering*, vol. 2011, pp. 1–14, 2011. [Online]. Available: https://doi.org/10.1155/2011/154798

[18] E. H. Glaessgen and D. Stargel, "The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles," *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Apr. 2012. [Online]. Available: https://doi.org/10.2514/6.2012-1818

[19] E. J. Tuegel, "The Airframe Digital Twin: Some Challenges to Realization," *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, 2012. [Online]. Available: https://doi.org/10.2514/6.2012-1812

[20] R. Klar, A. Fredriksson, and V. Angelakis, "Digital Twins for Ports: Derived From Smart City and Supply Chain Twinning Experience," *IEEE Access*, vol. 11, pp. 71 777–71 799, 2023. [Online]. Available: https://doi.org/10.1109/access.2023.3295495

[21] X. Liu, J. Zhao, and Y. She, "A Research on the Application of Digital Twin in Ship Industry," *ICBAR*, 2023. [Online]. Available: https://doi.org/10.1145/3656766.3656932

[22] P. Yu, W. Zhaoyu, G. Yifen, T. Nengling, and W. Jun, "Application prospect and key technologies of digital twin technology in the integrated port energy system," *Frontiers in Energy Research*, vol. 10, p. 1044978, 2023. [Online]. Available: https://doi.org/10.3389/fenrg.2022.1044978

[23] A. Troupiotis-Kapeliaris, N. Zygouras, M. Kaliorakis, S. Mouzakitis, G. Tsapelas, A. Artikis, E. Chondrodima, Y. Theodoridis, and D. Zissis, "Data Driven Digital Twins for the Maritime Domain," *Progress in marine science and technology*, 2022. [Online]. Available: https://doi.org/10.3233/pmst220087

[24] A. Troupiotis-Kapeliaris, G. Spiliopoulos, G. Grigoropoulos, E. Filippou, I. Chamatidis, M. Vodas, M. Kaliorakis, and D. Zissis, "A Digital Twin for Maritime Situational Awareness," *BDCAT*, 2023. [Online]. Available: https://doi.org/10.1145/3632366.3632378

[25] J. Neugebauer, L. Heilig, and S. Voß, "Digital Twins in the Context of Seaports and Terminal Facilities," *Flexible Services and Manufacturing Journal*, 2024. [Online]. Available:

https://doi.org/10.1007/s10696-023-09515-9

[26] J.-O. Eom, J.-H. Yoon, J.-H. Yeon, and S.-W. Kim, "Port Digital Twin Development for Decarbonization: A Case Study Using the Pusan Newport International Terminal," *Journal of Marine Science and Engineering*, vol. 11, no. 9, pp. 1777–1777, 2023. [Online]. Available: https://doi.org/10.3390/jmse11091777

[27] J. Ali-Tolppa and M. Kajo, "Mobility and QoS Prediction for Dynamic Coverage Optimization," *IEEE/IFIP Network Operations and Management Symposium*, pp. 1–2, 2020. [Online]. Available: https://doi.org/10.1109/noms47738.2020.9110396

[28] Z. Li, P. Wang, X. Han, and D. Shang, "Research on Digital Twin Technology of Port Structure Safety based on Floating 3D Display," in *2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*. Chongqing, China: IEEE, 2023, pp. 976–981. [Online]. Available: https://doi.org/10.1109/ICIBA56860.2023.10165263

[29] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *Cybersecurity and Cyberforensics Conference*, pp. 583–590, 2015. [Online]. Available: https://doi.org/10.1109/columbiancc.2015.7333476

[30] K. Bakshi, "Microservices-based software architecture and approaches," in *IEEE Aerospace Conference*, 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1109/aero.2017.7943959

[31] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[32] M. Mazzara, A. Bucchiarone, N. Dragoni, and V. M. Rivera, "Size Matters: Microservices Research and Applications," in *Microservices, Science and Engineering*. Springer, 2019, pp. 29–42. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_2

[33] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," *International Conference on Automation and Computing*, pp. 1–6, 2017. [Online]. Available:

https://doi.org/10.23919/iconac.2017.8082018

[34] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting Microservices: Practical Opportunities and Challenges," *Journal of Computer Information Systems*, vol. 60, no. 5, pp. 428–436, 2020. [Online]. Available: https://doi.org/10.1080/08874417.2018.1520056

[35] W. P. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three Brazilian government institutions," *Brazilian Symposium on Software Engineering*, pp. 32–41, 2018. [Online]. Available: https://doi.org/10.1145/3266237.3266262

[36] P. D. Francesco, P. Lago, and I. Malavolta, "Migrating Towards Microservice Architectures: An Industrial Survey," *International Conference on Software Architecture*, pp. 29–38, 2018. [Online]. Available: https://doi.org/10.1109/icsa.2018.00012

[37] C. Lai, F. Boi, A. Buschettu, and R. Caboni, "IoT and Microservice Architecture for Multimobility in a Smart City," *International Conference on Future Internet of Things and Cloud*, pp. 238–242, 2019. [Online]. Available: https://doi.org/10.1109/ficloud.2019.00040

[38] C. J. L. de Santana, B. de Mello Alencar, and C. Prazeres, "Microservices: A Mapping Study for Internet of Things Solutions," *IEEE International Symposium on Network Computing and Applications*, pp. 1–4, 2018. [Online]. Available: https://doi.org/10.1109/nca.2018.8548331

[39] G. Márquez, F. Osses, and H. Astudillo, "Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature," *IEEE Latin America Transactions*, vol. 16, no. 9, pp. 2321–2327, 2018. [Online]. Available: https://doi.org/10.1109/tla.2018.8789551

[40] A. Macías, E. Navarro, C. E. Cuesta, and U. Zdun, "Architecting Digital Twins Using a Domain-Driven Design-Based Approach*," *International Conference on Software Architecture*, 2023. [Online]. Available: https://doi.org/10.1109/icsa56044.2023.00022

[41] C. Rodríguez-Alonso, I. Peña-González, and Ó. García, "Digital Twin Platform for Water Treatment Plants Using Microservices Architecture," *Italian National Conference on Sensors*, 2024. [Online]. Available: https://doi.org/10.3390/s24051568

[42] G. Steindl and W. Kästner, "Semantic Microservice Framework for Digital Twins," *Applied Sciences*, vol. 11, no. 12, p. 5633, 2021. [Online]. Available: https://doi.org/10.3390/app11125633

[43] S. Acharya, O. Wintercorn, A. Tripathy, M. Hanif, J. V. Deventer, and T. Päivärinta, "Twins Interoperability through Service Oriented Architecture: A use-case of Industry 4.0," *TKTP*, 2023.

[44] Y. Kalyani and R. W. Collier, "Hypermedia Multi-Agents, Semantic Web, and Microservices to Enhance Smart Agriculture Digital Twin*," *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2023. [Online]. Available: https://doi.org/10.1109/percomworkshops56833.2023.10150413

[45] A. Raghunandan, D. Kalasapura, and M. Caesar, "Digital Twinning for Microservice Architectures," *ICC 2023 - IEEE International Conference on Communications*, 2023. [Online]. Available: https://doi.org/10.1109/icc45041.2023.10279802

[46] A. Ashraf, R. Belleman, and Z. Zhao, "Visualization Techniques and Tools for Developing Digital Twins of Ecosystems: State-of-the-Art and Selection," *2023 IEEE Smart World Congress (SWC)*, 2023. [Online]. Available: https://doi.org/10.1109/swc57546.2023.10448753

[47] C. Lin, T.-Y. Dai, A. D. Dilsiz, D. B. Crawley, D. Niyogi, and Z. Nagy, "UTwin: A digital twin of the UT Austin campus," *BuildSys@SenSys*, 2023. [Online]. Available: https://doi.org/10.1145/3600100.3626261

[48] T. Zhong, J. Lin, H. Wang, J. He, L. Zhong, and L. Chen, "An Initial Exploration of Digital Twin Technology for Hydraulic Structures Based on Cesium," in *Proceedings of the 2024 Guangdong-Hong Kong-Macao Greater Bay Area International Conference on Digital Economy and Artificial Intelligence*, 2024. [Online]. Available: https://doi.org/10.1145/3675417.3675545

[49] B. Simões, M. D. P. Carretero, J. Sanchez, C. Toro, and J. Posada, "Digital Twin and 3D Web-based Use Cases in Industry," *International Conference on 3D Technologies for the World Wide Web*, 2022. [Online]. Available: https://doi.org/10.1145/3564533.3565808

[50] F. Pan, Z. Chen, T. Wu, and Y. Li, "Campus Information Visualization Management and Design Using the K-means Data Analysis Method," *ICIEAI*, 2023. [Online]. Available:

https://doi.org/10.1145/3660043.3660086

[51] L. Coppolino, R. Nardone, A. Petruolo, L. Romano, and A. Souvent, "Exploiting Digital Twin technology for Cybersecurity Monitoring in Smart Grids," *ARES*, 2023. [Online]. Available: https://doi.org/10.1145/3600160.3605043

[52] V. Ramani, M. Ignatius, J. Lim, F. Biljecki, and C. Miller, "A Dynamic Urban Digital Twin Integrating Longitudinal Thermal Imagery for Microclimate Studies," *BuildSys@SenSys*, 2023. [Online]. Available: https://doi.org/10.1145/3600100.3626345

[53] K. Themistocleous, "Model reconstruction for 3d vizualization of cultural heritage sites using open data from social media: The case study of Soli, Cyprus," *Journal of Archaeological Science: Reports*, vol. 14, pp. 774–781, 2017. [Online]. Available: https://doi.org/10.1016/j.jasrep.2016.08.045

[54] L. A. G. Nunes, "Data-driven methodology for estimation of cruise ships and port energy profiles in small data settings," M.S. thesis, Universidade da Madeira, Funchal, Portugal, 2025.

[55] E. E. Sayeh, A. Addala, R. Lembeye, P. Costa, M. P. Neto, M. Ayiad, J. Iglesias, N. Rousselet, and K. Hetzenecker, "Network design tool for dc solutions," Shift2DC Project, Horizon Europe, Project Deliverable D2.1, 2025, grant agreement no. 101136131.