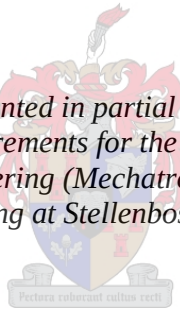
# The Development of a Service-Oriented Architecture for Digital Services on Maritime Vessels

by
Nicholas Raymond Bunn



*Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Mechatronic) in the Faculty of Engineering at Stellenbosch University*

Supervisor: Dr K Kruger

Co-supervisor: Prof A Bekker

April 2022

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily attributed to the NRF.

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2022

# Abstract

## The Development of a Service-Oriented Architecture for Digital Services on Maritime Vessels

N.R. Bunn

*Department of Mechanical and Mechatronic Engineering*
*Stellenbosch University*
*Private Bag X1, 7602 Matieland, South Africa*
Thesis: M.Eng. (Mechatronic Engineering)
April 2022

Digitalisation efforts in the maritime domain have, until now, predominantly focussed on ports and terminals. Contributions to the adoption of digitalisation technologies on vessels themselves stem mostly from industry, although the value of doing so has become increasingly apparent where digitalisation is said to present opportunities for improved vessel operation and performance. A popular approach to digitalisation, identified in the manufacturing and Industry 4.0 realms, is that of service-orientation and digital services. Here, systems are composed of discrete contributions, providing flexible and adaptable solutions to digitalisation challenges in dynamic environments with evolving needs.

This thesis details the design, development, and evaluation of a service-oriented architecture to aid in decision-making on maritime vessels. This architecture takes a microservice approach to service-orientation, employing a custom variation of the API-gateway pattern to enable a flexible and reconfigurable system. The proposed architecture includes an aggregation layer to abstract coordination activities from the service layer, negating the need for a service-mesh in the backend.

The architecture is tested and evaluated through a case study, carried out on the icebreaking polar supply and research vessel, the S.A. Agulhas II. This case study deploys information services and existing engineering models describing the vessel as microservices. Aggregating services are designed to leverage these services, providing information to aid in route planning and support more informed decision-making.

The case study details specific technology implementations to provide the specified platform functionality, with the most notable of these being gRPC as an RPC framework. The experiments indicate that RPC is a suitable communication mechanism for in-memory aggregation and real-time data delivery in this context. However, it was discovered that the gRPC interceptor functionality is not a robust

choice for all cases of rate-limiting and retry logic, and recommendations are provided for a revision of these components.

This thesis concludes that the proposed architecture is successful in providing a reconfigurable service-oriented architecture for digital service delivery on maritime vessels. Generic platform components were developed in the four programming languages used in the case study, showcasing the interoperability of services written in various languages, and by various domain experts, within the system.

# Uittreksel

## ’n Diens-georiënteerde Argitektuur vir Digitale Dienssamevoeging op Maritieme Vaartuie

N.R. Bunn

*Departement van Meganiese and Megatroniese Ingenieurswese*
*Universiteit Stellenbosch*
*Privaatsak X1, 7602 Matieland, Suid-Afrika*
Tesis: M.Ing. (Megatroniese Ingenieurswese)
April 2022

Die digitalisering van die maritieme veld fokus tans hoofsaaklik op hawens en terminale. Navorsing oor digitaliseringtegnologieë op skepe word meestal gedryf deur industrie en die waarde daarvan word beklemtoon waar hierdie digitalisering die werking en verrigting van skepe kan verbeter. ‘n Gewilde benadering in die digitalisering van die vervaardigingsindustrie, veral in die konteks van Industrie 4.0, is die van diens-oriëntering en digitale dienste. In hierdie benadering bestaan stelsels uit diskrete bydraes wat aanpasbare oplossings bied tot die uitdagings van digitalisering in dinamiese omstandighede met ontwikkelende vereistes.

Hierdie tesis bespreek die ontwerp, ontwikkeling en evaluering van ’n diens-georiënteerde argitektuur om besluitneming op maritieme vaartuie te ondersteun. Die argitektuur neem ’n mikrodienste benadering tot diens-oriëntering en gebruik ‘n pasgemaakte variasie van die toepassingprogrameringkoppelvlak-poort patroon om ’n aanpasbare en herkonfigureerbare stelsel te ontwikkel. Die argitektuur behels ’n samevoegingsvlak wat aktiwiteite van die diensvlak koördineer om sodoende die behoefte aan ’n dienstenet in die agtergrondverwerkingsvlak uit te skakel.

Die argitektuur is getoets en geëvalueer met ’n gesimuleerde gevallestudie gebaseer op ’n ysbrekende, Antarktiese navorsingskip, genaamd die S.A. Agulhas II. Die gevallestudie pas inligtingdienste en bestaande ingenieursmodelle wat die skip beskryf toe as mikrodienste. Samevattingsdienste is ontwerp om hierdie mikrodienste te benut om inligting te verskaf ter ondersteuning van roetebeplanning en besluitneming op die skip.

Die gevallestudie beskryf spesifieke tegnologie-implementerings om gespesifiseerde platformfunksionaliteit te verskaf. Die belangrikste implementering is die van gRPC as afstand-prosedure oproep raamwerk. Die eksperimente dui aan dat afstand-prosedure oproep ’n gepaste kommunikasie meganisme is vir binne-geheue samevatting en intydse data-aflewering in die konteks van digitalisering van skepe. Ten spyte hiervan, is dit egter bevind dat die gRPC onderskepper nie ‘n

robuuste keuse is vir alle tempo-beperking en herprobeer logika nie. Voorstelle om hierdie komponent te heroorweeg word verskaf.

In die gevallestudie is generiese platformkomponente ontwikkel in vier programmeringstale, wat die samewerking van dienste – ontwikkel in verskillende programmeringstale en moontlik selfs gelewer deur verskillende spesialiste – ten toon stel. Die tesis kom tot die gevolgtrekking dat die voorgestelde argitektuur slaag as 'n herkonfigureerbare diens-georiënteerde argitektuur wat digitale dienste kan lewer op skepe.

# Acknowledgements

Firstly, I would like to thank my family for their support throughout my studies and the past two years. To my parents for their unconditional love and for all the effort you have put into my education, be it through the opportunities made available to me or through your personal teachings – I would not have found my way onto this path without your guidance. To my brother, Greg, without your shared passion for creating, innovation, and life, I would not have enjoyed my work as much as I have.

Then, I would like to thank my two supervisors for their guidance and support throughout this project. Dr Kruger, for providing me with direction and introducing me to the world of software. You have helped to structure this work; keeping me on track while giving me the freedom to wander into that which interests me. You have helped me to discover a passion for software that I was unaware of before beginning this masters. To Prof Bekker for your constant motivation and for showing me the true value of having passion in work. Your love for the S.A. Agulhas II has been inspiring throughout, and I truly appreciate the environment that you have built between you and your students. The opportunities and experiences that you have provided, through our group runs, coffees, courses, our data project, and my time on the ship, will stick with me forever – I am truly grateful. The two of you have instilled a love for learning in me, and over the last two years have helped me to discover the direction which I would like to take in the future.

I would like to extend my gratitude to my fellow researchers, especially those within my two research groups. It is always comforting having others around, and being surrounded by people who share a passion for technology makes for stimulating conversation when breaks are, inevitably, needed. The value that this sense of community has had was only amplified through the pandemic, and having ears to share challenges with made it far easier to carry momentum through the slower times.

A special mention should be made of the AMSOL crew on board the S.A. Agulhas II. During my voyage, you made us feel so welcome on board that there were times when I felt more like a crew member than a researcher. Your willingness to assist us, entertain our ideas, and teach us was so greatly appreciated. The knowledge you shared throughout our time working together has helped more than you can imagine. To the DEFF personnel responsible for our voyage, you navigated difficult situations from before we had even entered quarantine and still managed to keep a smile on your faces throughout the expedition – additionally, the opportunities you unlocked for us were once in a lifetime and will forever be cherished. Thanks should also be given to all other passengers on the SANAE 60 expedition for contributing to a welcoming environment. The vast knowledge available through all of you, and the inclusive nature embodied by all, made for an interesting and exciting trip in all aspects.

Finally, gratitude is extended to the NRF for their financial support over the past two years. Your financial assistance to my research has been greatly appreciated, in addition to your support for the greater research performed through the SANAP program. The knowledge-oriented environment you support in the SANAP program made for a far more enriching post-graduate experience.

# Table of contents

# List of figures

**Page**

# List of tables

**Page**

# Glossary

**Aggregation**

The process of grouping/combining different sources of information, where the combination of them does not alter the information in any way. Aggregation simplifies accessing the information by 'packaging' it in an easy-to-manage way.

**Application Programming Interface (API)**

A computing interface that defines the interactions between software components. APIs define what kind of requests can be made, the information to be communicated, the information to be expected, and the data format to be used.

**Architecture**

A (software) architecture refers to the fundamental software components and relationships between components that comprise a specific software system. An architecture acts as a blueprint for developers to follow in implementing software, in the same way an architect/engineer develops blueprints for contractors to follow when constructing a building.

**Design Pattern**

A design pattern, in the context of software development, refers to a generic and repeatable solution that can be applied to common problems encountered during software design. Design patterns help designers to avoid common pitfalls by employing template solutions resulting from lessons learnt by more experienced designers and developers.

**Digital Service**

Describes the delivery of a service over an electronic network. Information is communicated in an automated manner with little to no human intervention. The information any given service provides is dictated by the need for the service itself, thus, each service serves a specific purpose.

**Digital Twin**

A virtual representation of a real-world entity or process that emulates the state and behaviours of the physical entity, based on sensor input from the entity and/or its environments. Digital twins encapsulate domain knowledge about some real-world entity in order to provide an interactive representation in the virtual realm.

**Digitalisation**

Not to be confused with digitisation, which describes the process of converting information from a physical format to a digital one. Digitalisation is the act of leveraging digitised information and digital technologies to improve business processes.

**Industry 4.0**

Also known as the fourth industrial revolution. Industry 4.0 describes the trend towards automation and intelligent data exchange between machines using wireless communication. This trend describes a world of connected machines and devices capable of sharing information regarding their state and behaviour for enhanced decision-making, operation, and collaboration.

**Maritime 4.0**

Refers to the implementation of Industry 4.0 technologies and ideologies within the maritime domain. Specifically, working towards smarter shipbuilding, and autonomous vessel navigation and operation using connected fleets.

**Microservice Architecture**

A modern extension of the service-oriented architecture (SOA) with a focus on domain-driven design. Microservices are finer-grained than services in an SOA and exist such that they are independently deployable and scalable. Microservices allow for continuous development, improved modularity, and highly scalable systems.

**Multi-Agent System (MAS)**

MAS' describe a software system comprising of discrete, self-organising entities (agents). The agents are considered intelligent entities, allowing for autonomous decision-making and organisation in fulfilling their goals. MAS' are an approach to modelling and building complex distributed systems in software.

**REpresentational State Transfer (REST)**

REST is an architectural style, providing 6 guiding principles, intended for use in the design of APIs. It places a focus on efficient hypermedia data transfer. A REST API provides a representation of the state of the resource that is offering the interface.

**Remote Procedure Call (RPC)**

RPCs are an approach to inter-process communication enabling a computer program to invoke a procedure on a separate machine as if it were invoking a function executing locally. The RPC hides all networking complexity such that the program requesting it does not need to navigate the network aspect of the call.

**Service-Oriented Architecture (SOA)**

A software design style where digital services are provided to other components, through a communication protocol over a network. A 'service' is a discrete, remotely-accessible unit of functionality responsible for a specific business activity.

# 1 Introduction

This introductory chapter provides the background to this research such that the context is understood and the objectives can be fully appreciated. Thereafter, it defines the objectives of this work and motivates the value that its contributions will have. Finally, this chapter describes the methodology that was followed to achieve the specified objectives.

## 1.1 Background

"Digitalisation has the potential to add wind to the sails of global seaborne trade, if leveraged effectively," – Mukhisa Kituyi, United Nations Secretary-General (UNCTAD, 2018).

The maritime industry is undoubtedly the biggest contributor to global trade, with shipping responsible for over 90% of the world's export (Geiling, 2013). However, even with this industry forming the backbone of the world's international trade sector, it is still considered one of the most conservative industries concerning technological change and digitalisation. Recently, major industry players have been investing in vessel digitalisation to propel their fleets into the digital era.

Ships require immense capital investment, having significant costs associated with operation and maintenance too. Digitalisation presents opportunities for more streamlined vessel operation with improved performance, facilitating more effective maintenance plans and reduced operation costs – these are often attributed as resulting from the implementation of digital twins. The concept of digital twins is one approach to digitalisation. In the maritime domain, limited work has been conducted on vessel digital twins within the research community. Currently, the majority of contributions on this topic lie within industry where suppliers of components, such as generators and motors, offer their products with an associated software representation (the digital twin). These 'digital twins' offer comprehensive data collection and in some cases, self-diagnoses or health monitoring.

The ideology of the digital twin describes a system that is all-knowing of itself, a single source of truth. A notable challenge in developing these digital twins is in the scoping of the digital twin to navigate the complexity inherent in the systems that they are applied to. Various approaches to this have been proposed, with one of the more practical being a systems-of-systems approach. Here, the digital twin of a complex system comprises sub-system digital twins that encapsulate smaller, bounded contexts within the greater system. In the context of the maritime domain, and with most vessels comprising sub-systems from various vendors, these twins typically exist as silos onboard with their data and insights existing in isolation of each other (Fonseca & Gaspar, 2020).

Returning to the core idea, digitalisation is said to offer better synchronisation in fleet operation and management, and more optimised route planning. Furthermore, digitalisation opens doors for the integration of other modern technologies, unlocking further big data capabilities to provide more valuable real-time insights to ship operators and stakeholders.

The digitalisation of a vessel, or any real-world system for that matter, relies on a sound basis to build off of. This comes in the form of a software architecture and is critical to the success of any IT system (Bergner *et al*., 2005). The digitalisation of a vessel can be achieved through developing a digital service architecture, where various ship functionalities exist as services. The term 'digital service' collectively refers to the electronic delivery of information across platforms. Here, the information is delivered and presented such that it is easy to understand and interpret for the user.

The Mechatronic, Automation, and Design (MAD) research group of Stellenbosch University's Department of Mechanical and Mechatronic Engineering have previously conducted research into the development of software architectures. Members of this group hold valuable experience relating to communication and cyber-physical system applications. Also of Stellenbosch University's Department of Mechanical and Mechatronic Engineering, the Sound and Vibration Research Group (SVRG) have a wealth of knowledge regarding the response of the S.A. Agulhas II polar research vessel to her environment, as well as on-deck experience working with their own data acquisition systems on the ship. The combined knowledge and experience of these two groups provide a strong foundation for developing a digital service architecture for use onboard maritime vessels.

## 1.2 Objectives

The objective of this thesis is to develop, deploy, and evaluate a digital service architecture for the S.A. Agulhas II, henceforth referred to as the SAAII. This system should provide the ship's operators with real-time insights into vessel performance and environmental interaction to aid in decision-making at strategic, operational, and tactical levels (Erikstad, 2019). It will comprise discrete services, acting independently while being orchestrated to serve stakeholder needs.

Considering the current state of digital twins within the maritime domain, this work considers digital twins following the digital-twin-as-a-service model. In this model, digital twins are considered as service providers, with other digital twins using the services on offer for collaboration. Regarding information usage, any authorised client is capable of querying these services to obtain information about the asset that they encapsulate. Tying into this consideration, an assumption is made on data availability on maritime vessels. In this thesis, it is assumed that where a service boasts logic operating on recorded data, that the data is stored and available in a usable format. This assumption ensures that services should not have to convert the data into a usable format before they can process it. In the case that data is stored

in a proprietary format, it is assumed that this data belongs to a digital twin that is capable of offering the data to clients in a usable format.

The objectives for this project are to design an architecture that:

- Facilitates decision-aiding services composed of discrete contributions.
- Provides a facility to run engineering models as well as information collectors to feed these models. These will exist as digital services within the system.
- Coordinates information between models and various information services, enabling the fusion of information to enhance the value provided to stakeholders.
- Supports contributions from diverse vendors, such that the system can adapt over time as the needs of the vessel and its stakeholders change. This will allow for the system to be configured on a per voyage basis, tailoring the offerings around the voyage goal. Naturally, it should simplify interfacing, encouraging vendors to contribute and enabling reconfigurability, i.e. contributors should not need an intimate understanding of the greater system in order to contribute to it.

This project, therefore, aims to provide a service-oriented architecture (SOA) for the delivery of digital services on maritime vessels. The architecture should be: high-performance to provide near real-time information flows, modular to support various services such that it is relevant for multiple applications, and enable easy contribution from multiple vendors. Note that the scope of this thesis does not include advanced service development. The research instead focuses on developing the system to facilitate services. It is also worth noting that while this architecture shares certain goals with digital twins, it is not a digital twin itself. It will, however, facilitate the integration of them through services offered.

## 1.3 Motivation

Currently, the SVRG conduct research into the SAAII's interaction with, and performance in response to, its environment; reporting their findings to the captain and vessel stakeholders annually at the SAAII Mini-Conference. At the most recent of these conferences, taking place in December of 2019, Captain Freddie Ligthelm expressed how valuable these insights would be to him if he had access to them in real-time.

At this same conference Mr Nish Devanunthan, Operations and Logistics Director for the Department of Environmental Affairs stated that there is currently a financial shortfall for the planned voyages on the SAAII, and mentioned that the current cost of operation for the ship exceeds R550 000 per day. On top of this, a challenge that

places further financial pressures on the stakeholders is the progressive increase in maintenance costs for the vessel. The successful implementation of the proposed architecture would enable the delivery of information regarding the ship's environmental interaction to the captain, such that he/she can optimise routes to reduce voyage time and/or fuel consumption. Furthermore, the information provided through this architecture can be used to direct maintenance activities more effectively, leveraging services offered by tools such as digital twins.

Erikstad (2019) argues that digital services are key enablers for improved operations in terms of efficiency, safety, and environmental impact. He further mentions how large quantities of data have been recorded over recent years without any ideas or plans to use them for decision-making support. This applies to the problem at present, in that the SVRG have gained valuable knowledge into, and data describing, the SAAII's operations through the analysis of their sensor readings, but these insights are currently only being used for reflection. This thesis aims to maximise the value of these insights by providing an architecture to deliver them in (close to) real-time in order to support the stakeholders in decision making.

Fonseca & Gaspar (2020) mention how the digital management of maritime vessels has traditionally relied on a host of software tools that produce solutions to their respective problems, going on to say how interoperability of these tools and solutions is rarely considered. Harper, *et al* (2019) speak about how, when working with digital twins, each asset vendor holds unique expertise for their equipment which makes them the best analysts of the subject matter. Thus, there is merit in composing system representations of specialised digital twins of its sub-systems. The authors advocate for digital twins and stand-alone services to exist in harmony. This approach is said to provide a separatison of concerns in an attempt to manage the complexity of modelling and representing large and composed systems.

Bekker (2017) describes a decision-support concept where the SVRG's full-scale measurement system on the SAAII is updated to enable the concurrent acquisition of vessel data. Leveraging this data with efficient real-time algorithms, such as machine learning models, would enable the delivery of real-time information. Specific mention is made of the ship's hull and propulsion systems. These sub-systems of the vessel are likely candidates to boast digital twins, which can offer self-diagnosis based on current response or estimated response for specified routes. It is thus evident that a solution is required to integrate existing and future models and data describing vessel operation. Moreover, a solution is required that can integrate the information on offer by current and future digital twins and solution systems that are siloed in the current landscape of maritime vessels.

As is discovered in the literature review, service-oriented architectures (SOAs) are still new to the maritime industry. This project will not be the first application of SOAs on maritime vessels, but will certainly be early work in the use of service-based digital maritime architectures. The successful outcome of this project hopes to further bridge the gap between the current state of digital vessel management and the complete implementation of Industry 4.0 in the maritime domain.

## 1.4 Methodology

This research began by performing a review of literature, investigating relevant work done by others. This review is presented in Chapter 2, where the use of SOAs in the maritime, and other relevant domains, is investigated. This chapter presents relevant architectures and design patterns that align with the objectives of this application, considering aggregation within these architectures. Finally, middleware and security considerations that are commonplace in distributed and/or service-oriented systems were reviewed.

Following this review, a systems engineering approach was taken in the selection of a suitable architecture for this application. This approach was deemed appropriate considering that this thesis concerns the design of a system rather than an entity within a system. This selection considered what capabilities this architecture needed to deliver and entailed generating a set of requirements that it should satisfy. These needs comprise those identified in the above discussion, as well as those identified in literature.

Upon selection of an architectural style, the architecture was further developed through a detailed design process. This involved specifying the structure of the architecture, the selection of communication protocols, and the specification of generic system components. Here, important decisions were made regarding the structure of the architecture as well as the interactions and communication within. With a detailed design proposed, all generic components of an implementation were developed. These include all components not related to a specific use case. The development followed an incremental design strategy, deploying and testing each successive module as it was created to simplify the debugging process.

Subsequently, this generic architecture was implemented in the form of a case study, carried out on the SAAII. During this implementation, projects and models of the SVRG were exploited and deployed within the proposed system as services. To provide a controlled and repeatable test environment, the case study was carried out in a simulated laboratory environment using data from the SAAII, rather than on the vessel itself. Data structures and collection methodologies on the vessel were replicated in the laboratory where required. One limitation to this case study was the fact that, at the time of testing, no digital twins were available for inclusion. As digital twin development lay beyond the scope of this work, a simplified digital twin representation was used. This representation used engineering models to offer services describing sub-systems of the vessel; where these models required only the data generated by that sub-system itself. This representation was run on its own machine, as a digital twin would be on a vessel.

This thesis concluded with an evaluation of the architecture and its implementation. Here, a reflective analysis was performed on this research. This entailed the generation of evaluation criteria based on microservice characteristics and the design decisions made throughout this thesis. These criteria were assigned metrics

where appropriate. Based on the evaluation criteria, a set of experiments were designed, with their outcomes compared to the specified objectives and requirements. Through this, opportunities for improvement were identified to aid in future continuations of this, or similar, projects.

# 2 Literature Review

This chapter investigates the current state of research with regards to vessel digitalisation and maritime 4.0. The investigation begins by delving into the contributions made to the maritime domain by the previous three industrial revolutions and discusses the role that the fourth revolution currently plays in the shipping industry. Following this, an investigation is carried out into service-oriented architectures, covering their use in the maritime and relevant domains. This investigation explores the aggregation of information and services through the review of approaches to service-orientation and established design patterns. Based on the review of the state-of-the-art of service-orientation in this context, relevant middleware and popular communication mechanisms are identified. Security considerations are subsequently reviewed, placing a focus on security in distributed systems. This chapter concludes with a short discussion of the review; relating its findings to, and placing them in the context of, this thesis.

## 2.1 Industry 4.0 in the Maritime Domain

Figure 1 illustrates the four industrial revolutions and their contributions within the maritime domain. The following discussion refers to this figure in the description of the various revolutions.

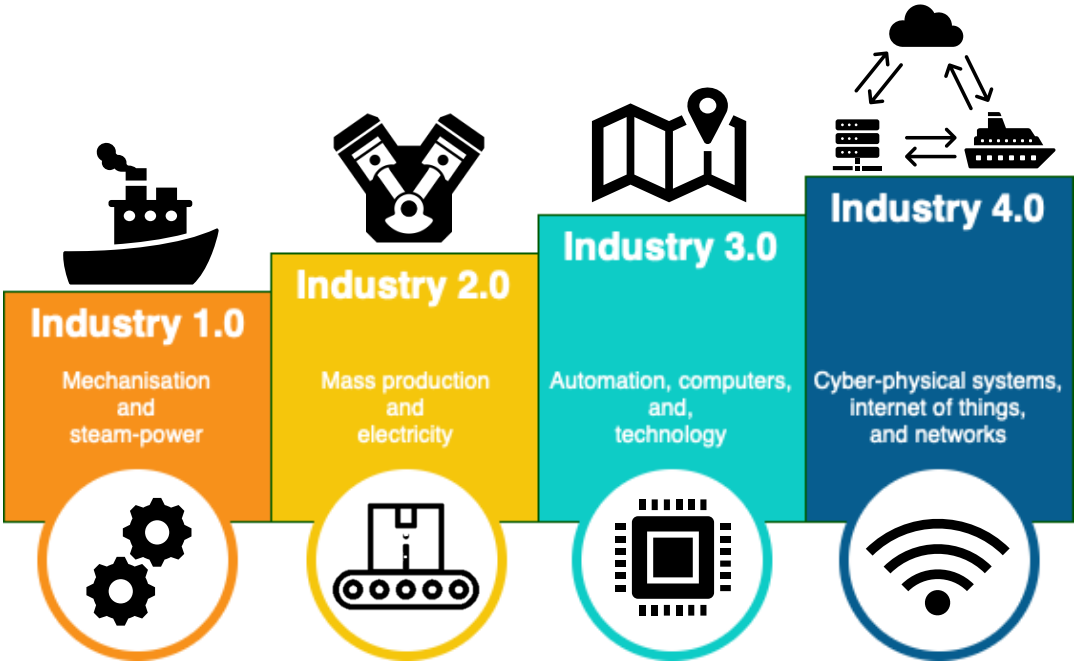



**Figure 1: Industrial revolutions in the maritime domain (adapted from Cline (2017))**

Shipping has long been an essential transportation technology, tracing back to before the 19th century where sailing ships already dominated foreign trade. While a titan in the trade industry, historians widely consider shipping to have been stagnant for hundreds of years before the first industrial revolution introduced iron steamships (North, 1968). In 1807, Robert Fulton built the first commercial steamships, kickstarting the development of the maritime industry (Kelley & O Gráda, 2018). Driven by both military and commercial industries, the advancement of shipping performance has always been of great global interest.

Towards the end of the 19th century, the world entered into what is now considered to be the second industrial revolution. This was characterised by the introduction of mass production, assembly lines, and electricity. For the maritime industry, this brought with it the standardisation and mass production of ship engines, as well as advancements in structural engineering. These advancements were characterised mainly by the replacement of iron with steel and welded joints succeeding rivetted assemblies. Advancements in hydrodynamics and vessel stability were made by experimenting with more optimised bow types. Finally, this revolution introduced more advanced navigational equipment such as Radar and Sonar. However, navigation was still predominantly done using radio communication and techniques developed during the wars (Penobscop Marine Museum, 2012).

Starting around 1969, the third industrial revolution introduced the world to automation, computers, and electronics. Along with further advancements in manufacturing, this provided the maritime industry with the technology used for navigation and performance systems. On-board navigation systems using GPS and advanced sensor networks are implicit on modern ships. However, before their inception around 1983, or 1973 in the case of military vessels (McDuffie, 2017), ships relied mostly on traditional navigation techniques (Sid Nair, n.d.). Additionally, the third industrial revolution provided ships with digital systems capable of monitoring vessel performance such as speed and fuel consumption – with rudimentary data collection.

The world is currently entering into what is widely considered the fourth industrial revolution, entailing the introduction of cyber-physical systems (CPS), the internet of things (IoT), networks, and big data. This “Industry 4.0” is an initiative conceptualised and adopted by the German government in 2011 (Henning *et al.*, 2013). The initiative is being led by the manufacturing industry and the onset of digital factories (Weyer *et al.*, 2015). It promises a connected world full of ‘smart’ devices and machines that communicate with each other within a network, enabling real-time information transfer and autonomous control (Weyer *et al.*, 2015).

Kavallieratos *et al*. (2020) extend the existing maritime architecture framework (MAF) to include autonomous vessels. The MAF is a reference architecture used for ship-to-ship and ship-to-shore communication based on the smart-grid reference architecture. The paper focuses on extending this framework to enable fully autonomous vessels to operate without an onboard crew or human control.

de la Peña Zarzuelo *et al.* (2020) perform a literature review on Industry 4.0 in the port and maritime industry. This review describes the evolution of ports and terminals from a local scope to a point where they are fully integrated into the global supply chain. This application of Industry 4.0 technologies to ports, has been dubbed Port 4.0. Similarly, the application of Industry 4.0 technologies to the greater maritime domain, including the digitalisation of maritime vessels, has been dubbed Maritime 4.0. The current focus of studies within this Maritime 4.0 seems to be on robotic applications within shipyards. This focus on production is to be expected as the manufacturing industry is seen as the driving force behind this fourth industrial revolution. This paper concludes its review on the state of Industry 4.0 in the maritime domain by presenting emerging challenges in its adoption. The two challenges identified are those of cybersecurity, and connectivity and standardisation of systems. Cybersecurity has received the highest level of attention from port and maritime leaders recently (de la Peña Zarzuelo *et al.*, 2020), with threats growing as the initiative is adopted globally (Jones, 2014).

Ellingsen & Aasland (2019) carried out a case study on Industry 4.0 in the maritime domain, investigating enabling technologies and strategies for technology acquisition with a focus on shipbuilding. Interestingly, this paper speaks of data capture and simulation/visualisation in real-time, saying how "the benefit of being able to perform such live real-time simulations is having the human in the loop and playing around with operational experience". Here, this quote refers to real-time simulation in the manufacturing process, but it is equally applicable on board a ship. Being able to merge the operational experience of seasoned captains with real-time data and simulation provides immense opportunity for advancement in ship operation and environmental navigation. Support for this kind of simulation is discussed below, where simulations are implemented as services offered independently and through digital twins.

## 2.2 Service-Oriented Architectures

### 2.2.1 Background

Abdelhedi & Bouassidar (2020) define service-oriented architectures (SOAs) as an architectural style for building systems based on interacting services. This architectural approach describes an application that makes use of available services, where these services are provided to form applications in and of themselves. Before further investigating SOAs, it is necessary to define what a 'service' is. Each service logically represents a business activity with a specific outcome. Richardson (2018) describes a service as "a mini-application that implements narrowly focussed functionality". SOAs typically employ "smart pipes" for inter-service communication, manifesting in the form of an Enterprise Service Bus. These service busses employ heavy weight protocols, along with business and message-processing logic, to integrate services in the system. The services described in an

SOA are typically implemented such that they are a small application in themselves, with all services in the system acting on a global data model. (Richardson, 2018)

Prior to SOAs, services were understood as the end result of the application development process whereas, in an SOA, the application itself is composed of discrete services. Services are loosely-coupled and independent such that they can be delivered individually or as components of a larger, composite service. Services in an SOA interact using protocols such as REpresentational State Transfer (REST) or Simple Object Access Protocol (SOAP), often doing so over the web. However, SOAs are not limited to operation on a web-based network. While an SOA can be implemented as a web service, these two concepts are not inclusive. An important differentiation to be made is that in an SOA a service is any remotely available resource that can respond to requests. A web service is simply a service, implemented using specific web-based protocols. (Tyson, 2020)

Singh Gill (2020) describe the following nine principles of an SOA:

1. Standardised service contracts: All services within the system should share a common format with which they communicate, as well as information defining the service interface.

2. Loose service coupling: Services should minimise their dependency on one another in an attempt to minimise the scope of failure.

3. Abstracted services: Services should not expose the logic that they encapsulate. Instead, they should offer the functionality they can provide without showing how said functionality is performed.

4. Reusable services: Logic should be divided within the system such that services maximise re-use.

5. Autonomous services: Services should fully control the logic that they encapsulate, taking full ownership of the business domain they represent.

6. Stateless services: Services within an SOA should be stateless meaning that they should not need to store data regarding program states.

7. Discoverable services: Services should be visible within the system, with their location being discoverable by interested clients.

8. Composable services: Encapsulating discrete business logic into services allows one to break larger problems into a series of smaller, independent problems.

9. Interoperable services: Services should leverage standards to provide support to any users of said service.

### 2.2.2 Service-Oriented Architectures in the Maritime Domain

SOAs have been used by the military because of the benefit realised from the loose coupling of services (Russell *et al.*, 2008). Zoughbiy *et al.* (2011) consider the architecture design of SOAs within the military environment, stating that many global military organisations have adopted SOAs due to their flexibility and information capabilities. More specifically to the maritime domain, Meyer (2007) speaks of how naval fleets have adopted the SOA as it is "the best technical approach to integration of processes, functionality, and data in heterogeneous, cross-organisational, technical environments". This paper further speaks of how SOAs help with producing and integrating information from sensor systems into decision processes.

Microservices are an approach to service-orientation whereby a system is composed of discrete, lightweight components that each focus on doing a single task. They are discussed in more detail in the section on Microservices. He *et al.* (2019) designed a microservice-based information system for an inland river ship, hosted using Spring Cloud. In this application, microservices offer information through a RESTful interface using JSON message format on top of the HTTP protocol. One of the requirements of this application was for the architecture to have "strong horizontal expansion capabilities" which translates to a modular and scalable system. This application focuses on using microservices to facilitate data regarding the waterway and traffic information, rather than onboard measurements. The microservice architecture implemented here uses API gateways for service aggregation, such that it encapsulates the internal structure of the application and the client only needs to interact through said gateway. The API Gateway pattern is discussed further in Section 2.2.4.2.

### 2.2.3 Relevant Service-Oriented Architecture Applications

Berger *et al.* (2017) document their experiences in using containerised development to deploy a microservice architecture for self-driving vehicles. Containerisation is an approach to deploying microservices, whereby each service is assigned its own lightweight operating system to ensure service independence. In their application, communication is carried out using a publish-subscribe/event-driven approach. The authors mention the reasons a microservice architecture was used, how containerised development assisted with the deployment of the architecture, and the advantages that a microservices architecture offers to their use case – being quick onboarding for new developers, scalability, and ease of component addition and modification (flexibility). In this application, various vehicle tasks and/or functions exist as their own microservice, making use of the Docker ecosystem for containerised development and monitoring of microservices. Containerised development was used for the development and deployment of software components that interface with hardware components such as sensors. Their reflection on the use of microservices provides suggestions for improvement pertaining to the use of publish-subscribe communication and the reliance on the

Docker ecosystem as a limitation. No mention is given of issues or regrets in using microservices for this application.

Building on the idea of containerisation, Borodulin *et al.* (2017) investigate the use of containerisation for the development of digital twins in smart factories. To provide execution of the digital twins, this project designs a "Digital Twins Cloud Platform" that provides an Application Programming Interface (API) to present each digital twin as a microservice. This platform, therefore, provides digital twins as a service. Here, information that the digital twin is able to provide to consumers is implemented as a service offering with no specific examples provided.

Kruger *et al.* (2021) present an architecture for integrating digital twins within a service network. This work describes a service mesh being deployed alongside digital twins. In this architecture, digital twin instances (DTIs) encapsulate and offer services that can be interacted with in the same way as services in the service mesh. The differentiation is that services encapsulated by a DTI are constrained to using the data captured or generated within that digital twin – they are entirely introspective. This allows a strong separation of concerns for digital twins, allowing one to define and follow a strict scope during development. In the proposed architecture, services in the service network are aligned to user requirements, instead of to detailed domain knowledge of a physical counterpart. This approach hinges on the idea of a digital-twin-as-a-service. This allows for the servitisation of digital twins (and their associated physical assets), promoting integration through customised service platforms. The paper mentions service-oriented architectures for the service mesh, with emphasis placed on the more modern trend towards microservice architectures.

Remaining within the manufacturing domain, Ciavotta *et al.* (2019) looked at creating a microservice-based middleware for the digital factory. This application focused on enabling the interoperability of enterprise applications and CPS's, paying special attention to simulation tools. The developed architecture provides support for digital twins but does not provide a digital twin itself. This architecture acts as a framework to support the digital twins of CPS's using REST APIs for the aggregation of twins. Aggregation within service-oriented architectures and more specifically, microservice architectures, is discussed further in section 2.2.4.2.

Gamboa *et al.* (2015) present a framework for modelling manufacturing processes through the use of service-oriented holonic manufacturing systems. The authors state that holonic architectures and SOAs are the two most studied solutions to provide flexible and responsive systems for rapidly changing environments. Further mention is given of how these approaches provide agile environments for "next-generation manufacturing systems". This work takes the approach of combining these two systems, specifying what it is that services represent in these manufacturing systems.

Derigent *et al.* (2021) considers the contribution that holonic manufacturing systems have had to Industry 4.0. This work includes multi-agent systems as an approach to implementing holonic control architectures (HCA). The authors attribute the adoption of HCAs to their cooperative, adaptable, autonomous, and decision-making properties. These yield systems that can be easily and dynamically reconfigured such that they remain relevant when modelling complex systems boasting dynamic relationships – as is the case in large-scale manufacturing control systems.

Egert *et al.* (2021) investigate holarchies as an architectural pattern for smart grid applications. This work states how these holarchies provide isolation and self-maintained operation of their subparts (holons). Further mention is made of the support that holons, specifically, provide for the dynamic reconfiguration of systems. Holonic architectures (holarchies) are discussed in Section 2.2.5.

### 2.2.4 Microservices

Whether microservices are in fact a type of SOA, or an entirely new architectural style on their own, is a point of contention in literature. However, for the purpose of this study, and considering the end goal of creating discrete services to break down domain logic, they are considered as a sub-category of SOAs. Newman (2014) describes microservices as autonomous services that work together, further stating that they are small, lightweight, and focussed on doing one task well. Here, autonomy refers to the services communicating via network calls. This is specified to enforce the separation of services and avoid tight coupling.

Contrasting his description of an SOA, Richardson (2018) describes microservices as using “dumb pipes”, such as message brokers or direct service-to-service communication. “Dumb pipes” refer to communication support without containing business logic, placing all business logic within the services themselves. Complimenting these “dumb pipes” are lightweight messaging protocols such as REST or gRPC. In addition to the difference in communication, microservices most often work with a single data model per service, instead of a single data model for the entire system. The final and most obvious variation to an SOA is in the service size. Each service in a microservice is considerably smaller than that of an SOA. A microservice typically encapsulates a single function within the system, whereas a service in an SOA encapsulates a single application within the system. Considering this difference, an SOA’s service is almost always larger and more complex than a microservice as its purpose is to integrate monolithic applications.

#### 2.2.4.1 Characteristics of Microservices

Extrapolating from Newman (2014) and Richardson (2018), a microservice architecture’s core characteristics can be identified as:

1. Allowing for technology heterogeneity and easy technological adoption: Through strict service boundaries and standardised interfaces, each microservice is free to use any technology that suits its logic. This encourages the adoption of new technologies in the system and allows the most appropriate technology to be used for every function within.

2. Providing resilience and fault isolation: Performing a similar role as the bulkhead of a ship, service boundaries provide resilience to failures. Failures in a service cannot spread beyond its interface, isolating the failure to that service where it originated.

3. Providing a scalable system: In a traditional monolithic application, the entire application would need to scale together when the load required it. However, in microservices, each service is able to scale independently as required.

4. Enabling independent service deployment: Any changes to a service will only affect that service. When updating this service during deployment, only the updated service needs to be relaunched, instead of the entire system.

5. Allowing for organisational alignment: With each service existing independently of others within the system, teams can be assigned to services without them having to deal with the work of other teams. This allows for services to be decomposed based on domain teams, allowing domain experts to own and focus on their service without concerning themselves with the work of others in the system.

6. Supporting composability: Having discrete services responsible for specific functions and logic opens up opportunities for reuse of functionality instead of replication of it.

7. Enhanced maintainability: By keeping services small in size, replacing and refactoring services is a less intimidating task than that in monolithic applications. In a traditional application, legacy systems are often left as is because they are too complicated and have too large of a codebase to be practically (and safely) updated. Microservices ensure that this is avoided so that all components of the system can be kept up to date, enabling the system to grow along with technological change.

#### 2.2.4.2 Aggregation within Microservices

Malik *et al.* (2019) propose a solution to integrate ‘virtual objects’ based on contextual information in order to provide an IoT service. The virtual objects spoken about here are described as digital counterparts of physical objects. This paper considers the orchestration of microservices, with each ‘virtual object’ associated to its own microservice. The IoT services are created by representing real-world objects with virtual-world objects, using the collected real-world sensor

data. Orchestration is left as a responsibility of the user in this system; giving them the option to select all possible virtual objects and observe all possible service combinations, or allowing them to select the desired services and customise the scope of aggregation.

Damyanov (2019) investigated data aggregation within a microservice architecture. In microservices, data aggregation is performed in memory, by services themselves. This differs from centralised applications where a relational database is typically employed to execute this aggregation logic. Building on this fact, this paper investigates the use of the JavaScript Object Notation (JSON) format with an API for aggregation. Language INtegrated Queries (LINQ) defines the API used to enable the querying of data collection. Aggregation is performed in-memory in this application by using advanced LINQ queries that select and join JSON data fields to serve user requests. LINQ offers an SQL-like query syntax allowing the querier to filter (where), map (select), sort (order), and bind/group information. Leveraging this, the API can be defined to perform aggregation of in-memory data sources instead of relying on a relational database and SQL-queries to provide aggregation.

#### 2.2.4.3 Microservice Design Patterns

With microservices gaining popularity in recent times through their adoption by larger institutions such as Netflix, Uber, and Amazon, some form of standardisation has emerged. This standardisation comes in the form of design patterns. Patterns are not a new concept but have only recently been introduced to the world of microservices. The concept was first introduced by Alexander *et al.* (1977) as describing a problem and the core of a solution that can be reused in different ways within a field of expertise. This definition was originally put forward in the context of architecture but has since been adopted in the world of software development. Here, patterns refer to generic and reusable approaches or blueprints to building a codebase where different ‘patterns’ can be used depending on the problem or goal at hand. Patterns are developed through lessons learnt in taking various approaches to implementing applications. As such, a designer can avoid certain shortfalls and avoid common mistakes through the selection of a suitable pattern for their application.

The nature of a microservice architecture is that it comprises a suite of loosely-coupled, fine-grained services with each running in its own process (Fowler & Lewis, 2014). As a result of how independent each service is, one of the most common design patterns used for this architectural style is the 'aggregator' design pattern (Basu, 2018). In this pattern, a single aggregator service exists to invoke the required services to serve the requested information to the user.

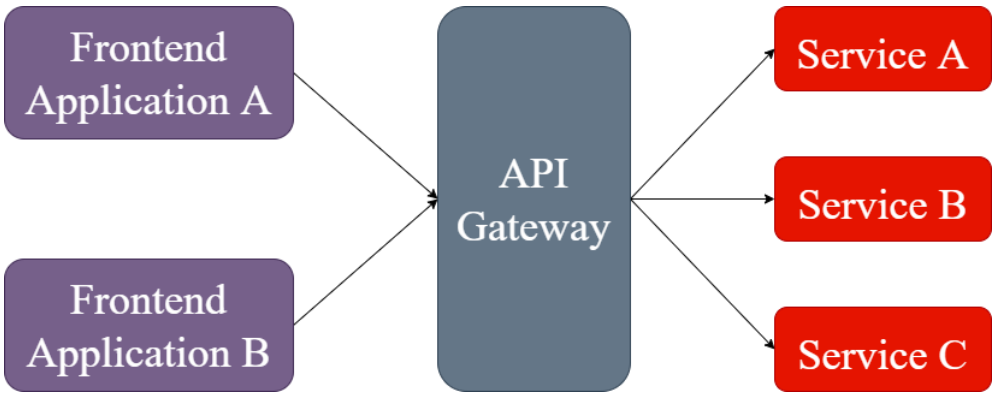



**Figure 2: API gateway pattern**

A popular variation of the aggregator pattern is the 'API Gateway' design pattern, shown in Figure 2. These two patterns look nearly identical with a single service sending requests to multiple downstream services. The difference between the two is apparent in the function of the aggregator/gateway. Aggregators are responsible for collecting information from various services and returning an information aggregate to their clients. The gateway, on the other hand, is a single entry point to the system with the primary responsibility of request routing. This API gateway pattern arose because microservices are often too fine-grained for client (user) needs and, as such, the client requires information from multiple services to serve their application (Richardson, 2018). Additionally, this pattern makes provision for different clients requiring different data (for instance, where one client exists on a mobile platform and another exists on the web).

In this pattern, the API Gateway can be considered as a proxy service for routing requests to the relevant microservices. Here, the gateway acts as an entry point for all services and caters to different types of clients creating a more versatile system. With this design pattern, the client sends a request to the API gateway, with the gateway forwarding the client request to the appropriate backend service. Before forwarding the request, the gateway can additionally leverage load balancing and/or rate-limiting logic to control information at a system level. Being a variation of the aggregator service, the gateway can still send requests to multiple services and aggregate the results back to the consumer service (Kappagantula, 2019). It is worth noting, however, that it is uncommon to use the gateway in this manner as this requires a mature or custom piece of software for the gateway components, instead

of a simple proxy implementation. Additionally, one risks bloating the gateway and making it less manageable by forcing this logic into it. The more common approach is to send a request to a single service and to allow that service to invoke subsequent services as it requires. The subsequent service would then do the same, building a chain of services, until all the required information to serve the request has been obtained (Richardson, 2018).

### 2.2.5 Service-Oriented Holonic Systems

Holonic systems are characterised as software systems that are based on the concepts of holons, where holons are simultaneously part of a larger system(s) and a whole system in themselves (Egert *et al.*, 2021). This approach can be thought of as a system-of-systems, similar to how service-oriented architectures are applications built up of applications. Rodriguez *et al.* (2011) define a holon as self-similar structures composed of holons as sub-structures. The authors further state how the hierarchical structure composed of holons is referred to as a holarchy. This definition emphasises the system-of-system nature where the holon is defined in itself. Derigent *et al.* (2021) define a holon as a communicating decisional entity composed of sub-level holons while, at the same time, being part of a wider organisation composed of higher-level holons. This work considers the decision making capability and adaptability of holons as fundamental properties.

Holonic systems have strong ties to the manufacturing domain, often being employed as a solution to model flexible manufacturing systems (Rodriguez *et al.*, 2005). This is due to the nature of holons where entities can comprise more complex entities, and allows for complex relationships to be modelled. Rodriguez *et al.* (2005) make mention of the Product-Resource-Order-Staff architecture (PROSA) as an example of holonic manufacturing systems. Derigent *et al.* (2021) mention how holonic control systems have been used to tackle the problem of complex control systems of manufacturing floors. This work suggests that agents are one approach to implementing holons. Again, mention is made of PROSA in the intersection of multi-agent systems and holarchies.

#### 2.2.5.1 Multi-Agent Systems

Multi-agent systems have become a prominent technology within the manufacturing domain, maintaining relevance through the Industry 4.0 and IoT movements. Similar to microservices, agents are developed such that they act independently of other system components. However, they differ in that agents tend to have a level of intelligence about them – often striving towards achieving a goal rather than just completing a task.

Rodriguez *et al.* (2005) detail a holonic multi-agent system (HMAS) as a domain-neutral solution to self-organising entities. Here, agents create holons where an agent is unable to fulfil a task on its own. The authors recognise multi-agent systems as "useful abstractions and technologies for modelling and building complex

distributed systems". Later works consider HMAS where MAS are used to implement holarchies, with agents as the holons (Rodriguez *et al.*, 2011). Derigent *et al.* (2021) describe the purpose of MAS as providing a decentralised architecture comprising autonomous, modular, cooperative, and intelligent components.

A MAS pattern mentioned in the work of Derigent *et al.* (2021) is that of the delegate multi-agent system (D-MAS). A D-MAS describes a group of computationally-lightweight agents that more complex agents can delegate tasks to. These D-MAS agents support the more complex agents in achieving their functions (Maoudj *et al.*, 2019). In the final revision of PROSA, Valckenaers (2019) introduces a D-MAS to provide a separation of concerns in the system. Here, the original PROSA holons are allowed to focus on reflecting reality, while the D-MAS agents are responsible for all decision-making in the system. This idea of utilising discrete, computationally lightweight components to perform tasks is akin to the approach taken with microservices. Recall, however, that agents tend to be goal-seeking entities instead of task-fulfilling ones. This comparison is mirrored by Valckenaers (2019) in his comparison of beings and agents – where beings are considered satisfiers whereas agents are considered optimisers.

Rodriguez *et al. (*2005) state how, in many multi-agent system (MAS) applications, an agent may appear as a single entity while they are in fact composed of multiple agents – as is the case in holarchies. It is thus evident that aggregation is inherent in multi-agent systems due to the nature of the holarchy. During the second revision of PROSA, a "staff" holon was introduced to provide dynamic aggregation (holarchies) at a more granular level. This was introduced in an attempt to provide more optimised and reliable performance in the system. This holon type was specified as being entirely optional, even after its inclusion. While the removal of a staff holon will likely result in a reduction of optimality, it should not break the system. Thus, the manner in which this aggregation component is introduced should not create a dependency between it and the holons that it is aggregating. (Valckenaers, 2019)

#### 2.2.5.2 Characteristics of Multi-Agent Systems

Botti & Giret (2008) provide a characteristic comparison between holons and agents. Agents are said to be autonomous and flexible computational systems that are able to act in an environment, where flexible refers to being:

- reactive: reacting to changes in an agents environment;
- proactive: attempting to fulfil its goals; and
- social: being able to communicate with other agents.

Additionally, further properties of agents are specified as:

- autonomous: operating without direct intervention;

- rational: capable of reasoning about perceived data;
- adaptable: referring to an agent's capability to change its behaviour based on what it has learnt;
- mobile: the ability of an agent to move within a specified network;
- truthful: an agent's inability to deliberately provide false information; and
- benevolent: an agent is only willing to help other agents so long as it does not contradict its own goals.

The characteristics of holons share the same merit but are described in different terms. The most notable difference is in the explicit specification of the recursive nature of holons – referring to their ability to aggregate. This is not considered an explicit characteristic of agents, although it is implicit in their ability to help, and request help from, other agents.

## 2.3 Middleware

IBM (2021) define middleware as "software that enables one or more kinds of communication or connectivity between two or more applications or application components in a distributed network". Middleware streamlines application development by providing the functionality to connect applications that were not explicitly designed to connect to each other. When investigating middleware, definitions can be vague as the scope of different middleware components vary greatly. There may be a middleware that focuses on a single, specific type of communication. An example of this would be message brokers like RabbitMQ. Conversely, there may be a middleware such as web-servers that provide the full functionality needed to build an application. Abstracting another level, there is middleware, such as those on offer when using the HTTP protocol, that allow developers to build customised middleware functionality for their application. This concept received its name as the first middleware existed as a mediating layer between the application frontend and backend. Modern middleware has developed far beyond this scope though, with some middleware components encompassing aspects of either the frontend, the backend, or both.

Middleware, on the scale of entire applications, has become a notable approach in the IoT realm, where it is used to connect sensors and devices to processing platforms or users. Benayache *et al.* (2019) developed a microservice-based middleware for smart wireless sensor networks (WSN). Here, the microservice middleware (MsM) is proposed as a solution to the interoperability issues presented by WSN-based IoT projects. The MsM acts as an intermediate tool to allow for interactions between IoT devices without requiring large architectural changes. Ciavotta *et al.* (2019) describe two middleware components forming part of the MAYA project: the MAYA Support Infrastructure (MSI) and the MAYA

Communication Layer (MCL). The MSI is a large-scale, broadly-scoped microservices data processing middleware. It is responsible for the management of digital twins during their factory life cycle. The MCL, on the other hand, is a tighter-scoped communication middleware that hosts a runtime environment to enable aggregation, discovery, orchestration, and communication among CPSs. The MCL sits between the machines and the cloud, enabling data flows between the two; the MSI sits one layer higher than this and enables communication between the cloud and the user. In the integration of the MSI, the MCL, and the third component - the MAYA Simulation Framework (MSF) – finer-grained middleware, such as WebSockets and encryption middleware, are employed.

IBM (2021) and Bishop & Karne (2003) describe the most commonly-used types of middleware, with those relevant to this work being: message-oriented middleware (MOM), remote procedure call/procedure-oriented middleware, API middleware, and object request broker (ORB) middleware.

Message-oriented middleware acts as a translator to enable components using different messaging protocols to communicate with each other. In addition to translating messages between applications, MOM manages message routing to ensure that messages get to the correct components in the correct order. MOM is typically implemented as a proxy service.

Remote procedure call middleware usually manifests in the form of a framework. These frameworks allow an application on one machine to trigger a procedure on another machine, as if both processes were running on the same machine. This middleware takes care of all networking complexity on behalf of the developers by offering an intuitive interface for them to invoke these procedures.

API middleware can vary in scope but generally provide tools that developers can use to create and manage their APIs. This middleware is most notably used with the HTTP protocol, where it is typically employed to perform authorisation or to monetise an API call.

ORB middleware is again used for distributed networks, where it enables the fulfilment of requests between applications or components without these components needing to know where the other is hosted. This is a similar task to that of the MOM, but differs in its execution and application.

In addition to these middleware types, there is merit in defining security and monitoring middleware here. Al-Jaroodi *et al.* (2010) discusses approaches to security middleware. The authors discuss how middleware can be implemented such that it is responsible for authentication, authorisation and access control, and data security and integrity in a system. Finally, a more application-oriented middleware comes in the form of monitoring middleware. This middleware is responsible for collecting metrics describing the system during runtime. This is application-oriented middleware as it is typically just an implementation of

database middleware with the focus on structured metric collection. This middleware acts as an intermediate layer between the application recording metrics and the database that stores them.

It is evident that middleware is not governed by a strict definition. Many of the middleware discussed above describe concepts without a clear boundary to their scope. As such, there may exist a large overlap between middleware components, where certain categories encompass the functionality described by other categories. The following review of communication mechanisms and security provides a background on concepts that may be implemented through some form of middleware. Considering the potential overlaps in middleware, it is not impossible for security functionality to be considered in the middleware that is employed for specific communication mechanisms. In certain cases, it may in fact be beneficial for this to be the case.

## 2.4 Communication Mechanisms

The literature covered in the section on service-oriented architectures identified three main communication mechanisms that are leveraged when implementing SOAs and microservices. Communication mechanisms stipulate the procedure followed when transferring information between two or more software components. These allow for architectures to be further specified based on communication style and requirements. The three mechanisms identified are: REST APIs, event-driven architectures, and remote procedure calls.

### 2.4.1 REST API

REST, or REpresentational State Transfer, is an architectural style for distributed hypermedia systems. It was developed in 2000 by Roy Fielding (Fielding, 2000). It is predominantly used as an architecture for designing APIs based on HTTP calls, specifically for use in web-based systems. Supporting this, REST is designed to be efficient for hypermedia data transfer, which optimises it for the common use case of the web (Fielding, 2000).

Fielding (2000) presents the six guiding constraints of the REST architectural style as follows:

1. Strict client-server roles: This constraint is based on the separation of concerns principle. The user interface concerns (client) are decoupled from data storage concerns (server) to improve the portability of the user interface across platforms.

2. Statelessness: In the REST architecture, all communication must be stateless. This dictates that no session state is to be stored in the server. In order to achieve this, the request that is sent by the client must contain all the information required to perform the request. The state of a session is

thus stored in the client. This is said to provide visibility, reliability, and scalability.

3. Cacheable: Statelessness introduces inefficiency as repeated requests are re-processed instead of having their results reused where practical. In an attempt to improve network efficiency, clients can be given the right to reuse the response data. Caching improves efficiency and performance by potentially reducing interactions. The downside of this is that there is potential for clients to reuse stale data, reducing reliability.

4. Uniform interface: Employing a standardised interface simplifies the interactions performed in the system. Through this, implementations on the server-side are decoupled from the services that they provide. This further encourages independent evolvability. The trade-off with this is that the uniform interface reduces efficiency as information is transferred in a standardised form rather than one specified to an application's need.

5. Layered system: In a further attempt to improve scalability, a layered system constraint was added to the REST architecture. This specifies that components cannot see beyond the immediate layer that they are interfacing with. By restricting components to only require knowledge of a single layer, complexity is bounded.

6. Code on demand: The final constraint allows for client functionality to be extended by downloading and executing code as scripts. This allows functionality to be added after deployment, improving extensibility. This is an optional constraint in the REST specification.

The benefits and drawbacks of using REST APIs are provided by Richardson (2018) and Newman (2014), and can be observed in Table 1.

**Table 1: Benefits and drawbacks of REST APIs.**

| Benefits | Drawbacks |
|---|---|
| • Simplicity and familiarity<br>• Support for the request/response communication model<br>• Being firewall-friendly<br>• Requiring no intermediate broker<br>• The ability to test the API with a browser | • Being limited to only supporting request/response communication<br>• Being constrained by the fixed semantics; making it difficult to map multiple operations to HTTP verbs<br>• Reduced availability because no intermediary buffers is used<br>• Clients need to know the locations (URLS) of service instances |

### 2.4.2 Event-Driven Architectures

Tragatschnig *et al.* (2018) attributes the recent adoption of distributed event-driven architectures (EDA) to their ability to provide highly scalable, flexible, and concurrent solutions. Typically, an EDA consists of discrete components or agents that communicate with each other through sending and receiving events (Mühl *et al.*, 2006). Employing discrete components, EDAs map well to service-oriented architectures where the components or agents are implemented as services. Servers publish events to topics hosted on a message broker; interested parties would be subscribed to this topic and can thus consume information as the events are posted. Through this approach, clients and servers are fully decoupled with no knowledge of each other, or even whether the other exists in the system. This provides absolute isolation but at the same time provides no guarantee that requests (posted as events) are received by any servers.

Mühl *et al.* (2006) describe an 'event' as "any happening of interest that can be observed from within a computer". This could be a physical event where sensors are monitoring an environment, a timer event, or any state or information change in a system. This approach advocates for a push-based system where components react to changes to information instead of requesting it. Note that there is no specification forcing this, and it could be implemented as a pull system by adding request topics that servers would respond to. However, this is not how it was intended to operate as it is not a transactional communication mechanism.

Bellemare (2020) introduces event-driven microservices where services use consumable events to asynchronously and indirectly communicate with each other. An important distinction made in modern event-driven microservices architectures is that the information is not destroyed upon consumption, as it is in transactional message-passing systems. Instead, the information remains available for other consumers to read as is required. This provides message persistence and traceability as events can be tracked in hindsight. In event-driven architectures, services may be either stateful or stateless with no constraint placed on this (Bellemare, 2020).

Richardson (2018) strongly supports asynchronous messaging using event-driven architectures, with the stated benefits and drawbacks of using it shown in Table 2.

**Table 2: Benefits and drawbacks of event-driven communication.**

| Benefits | Drawbacks |
|---|---|
| • Loose coupling where clients can be unaware of service instances<br>• Messages are buffered until a time when they can be processed | • Risk of performance bottleneck<br>• Potential for a single point of failure<br>• Increased system complexity |

### 2.4.3 Remote Procedure Calls

Krishnamurthy & Maheswaran (2016) describe remote procedure calls (RPC) as an abstraction for performing procedural calls across languages, platforms, and protection mechanisms. In the context of IoT, this translates to supporting communication between distributed devices. RPC implements the request/response communication pattern, making it a transactional communication style.

Newman (2014) defines RPCs as the "technique of making a local call and having it execute on a remote service somewhere". A separate interface definition is said to make the generation of client and server stubs easier across different technology stacks. An example is given where a JAVA server exposes a SOAP interface, and a .NET (C#) client is generated from the Web Service Definition Language (WSDL) definition of the interface. Essentially, when employing RPC as a communication mechanism, one needs to select an interface definition language (IDL) and RPC framework; the former is often included in the framework. This IDL and framework allow a developer to define their interface, with the client and server stubs being generated automatically. These stubs perform parsing logic and consider the network complexity of the call on behalf of the developer. It is this functionality that allows one to invoke a remote call as if it were a local one.

Further mention is made of how one potential drawback of using RPC is its (potential) brittleness. Traditional RPC implementations such as SOAP, Java RMI, or Thrift have struggled to gain traction and widespread adoption due to issues in their implementations. Additional issues have been known to arise where the IDL allows some way of forcing language-specific objects into the message, as this creates havoc when the client/server on the other end of the connection cannot interpret it effectively. (Newman, 2014)

Considering microservice applications, Richardson (2018) describes remote procedure invocation as the process whereby a client sends a request to a service, with the service processing and returning the response. In his definition, no mention is given of the remote nature of the call; however, when considering the context of microservices, it is implicit.

An alternate approach to implementing remote procedure invocation than those highlighted by Newman (2014), and one that has gained immense popularity in recent times, is employing the gRPC framework. gRPC is the most recent offering of RPC frameworks, addressing shortfalls of previous forays into inter-process communication such as SOAP.

gRPC is a binary message-based protocol where one defines their API using Google's Protocol Buffers. This is a language-neutral mechanism for serialising structured data. The Protocol Buffer compiler then generates the client and server-side stubs in any of the (currently) 11 supported languages. gRPC can be used with alternate serialisation mechanisms, although it is optimised, and provides the most

support, for Protocol Buffers. Serialised messages are then transported between clients and servers using the HTTP/2 protocol – a performance-focussed revision of the HTTP protocol. HTTP/2 enables multiple message-streaming configurations on top of the standard request/response model when using RPCs. (Richardson, 2018)

Newman (2014) writes about RPCs with a focus on their shortcomings. The notable three drawbacks mentioned are the technology coupling, brittleness, and the fact that local calls are not remote calls (and making them appear that way is a difficult endeavour). The author does note that while his argument makes RPCs seem terrible, they are not – and their shortfalls are all related to the available implementations at that time of writing (2014). Richardson (2018) focuses on the gRPC framework in his more recent work. His discussion on this framework addresses the shortfalls identified by Newman (2014) and concludes with only two drawbacks to the approach. These are complimented with a list of benefits that gRPC, specifically, has over REST APIs – these can be seen in Table 3.

**Table 3: Benefits and drawbacks of (g)RPC**

| Benefits | Drawbacks |
|---|---|
| • API design is straightforward and boasts custom semantics<br>• The underlying serialisation is considerably more efficient<br>• Bi-directional streaming enables both request/response and messaging communication styles<br>• That their use enables interoperability between clients and servers written in a wide range of languages | • Implementing (g)RPC in JavaScript/the browser requires more work than REST APIs – especially considering that gRPC is predominantly based on the HTTP/2 protocol<br>• Older firewalls may not support HTTP/2 |

The issues with RPC have to do mostly with forward compatibility of legacy systems. However, since Richardson's (2018) work was published, Google has made great progress with their gRPC-Web project, dealing with the first drawback. Additionally, browsers have since provided support for HTTP/2. With this adoption, firewalls have almost all adopted support for the protocol in order to maintain relevance.

## 2.5 Security

According to Firdhous (2012), there are five key security considerations that one needs to take into account when working with distributed systems. These are: the protection of data in transit, user authentication, access control, explicit consideration for denial of service (DOS) attacks, and multi-level security considerations.

When considered, the protection of data in transit ensures that the data sent arrives at the intended destination untampered with and without its contents being viewed by any other party during transmission. This is achieved through encrypting messages using tested encryption mechanisms such as TLS encryption.

User authentication confirms that the user making requests is who they say they are and that this user is recognised by the system (Newman, 2014). Traditionally, this is implemented in computer systems through a user database that enables recognised users to log in before being given access to the system. The assumption is made that only the user knows of their username and password combination and thus, by verifying those details the system can trust that the user is in fact who they say they are.

Authorisation (or access control) is described as the mechanism that maps a principle (user) to an action (request) (Newman, 2014). More simply, this mechanism verifies that the user making the request is allowed to do so. This enables the protection of sensitive information and optionally allows a system to support users with different roles. Authorisation is often implemented in combination with authentication. When a user logs in, the authentication components will generate an access token. This token represents the user's session and is used to authorise the user for any further calls during that session.

The explicit consideration of DOS attacks is a means to ensure availability and quality of service (QoS), and stems from security requirements of computing clusters (Firdhous, 2012). DOS attacks overload systems with requests so that they are unable to process normal traffic effectively. Without being considered, the best case is that this attack results in a decrease in the QoS; the worst case is that the entire system buckles under the load and crashes.

Multi-level security considerations, also termed *defence in depth*, is an approach to minimise the possibility of a single-point-of-failure. This details the consideration and implementation of security measures at all points within a system. In complex systems with multiple potential points of access this is critically important, hence the advocation for it in distributed systems. This approach is an attempt to provide robust security where implementations may unknowingly expose vulnerabilities.

In addition to the above, isolation is considered a valuable security consideration in microservice architectures (Newman, 2014). This can be implemented as service

isolation and/or network segmentation. Service isolation ensures that malicious attacks such as the DOS attacks documented above do not affect that other than the component under attack. Network segmentation entails building subnetworks within your system of components that typically work together. This is akin to building multiple perimeters around a property to limit how far traffic can travel within it. Additionally, interfaces can be provided for each subnet which gives improved control over access to components in that subnet.

## 2.6 Conclusion

From the review presented above, it is evident that digitalisation in the maritime domain is a topic of great interest at present. The adoption of Maritime 4.0 seems to be focused on shipbuilding and manufacturing aspects rather than the operation of vessels themselves. The focus of this work will be on investigating Maritime 4.0 within the operational context.

As is mentioned in the Section 1.1, the maritime industry is considered to be one of the more traditional industries. Due to the immense associated costs, it tends not to be an early adopter of technologies, waiting for them to be proven before investment is considered viable. Review was thus performed on SOA use in the manufacturing and IoT realms as research in these fields are far more diverse and available at present. Multiple applications were found where SOAs were used as an architecture for digital factories. This domain seems to be the driving force behind digitalisation in the fourth industrial revolution. However, contributions from these domains are often offered for large-scale applications and are thus too complex and require too much connectivity for practical application on maritime vessels. Maritime vessels offer unique challenges for digitalisation, requiring a balance of complexity and connectivity in order to provide a viable solution. Across all domains, information siloes were identified as a significant challenge requiring thoughtful consideration in order to offer comprehensive digitalised assets and processes.

The review identified microservices as being a popular and prominent approach to digitalisation, with recent contributions all opting for this approach over the traditional service-oriented architectural style. When considering microservices, and referring specifically to the size of services, the general consensus encountered in literature is to decompose microservices into software components that can be managed by a single, specific domain team. This research will follow these recommendations by considering services as isolated units of code capturing the knowledge of domain experts. This will be a driving ideology throughout the remainder of this thesis, where it is used in the selection of an architecture in Chapter 4. The communication mechanisms and middleware presented above are considered in the context of this thesis to further specify this design in Chapter 5.

# 3 Problem Identification and Requirements

This chapter identifies the problem and solution spaces which are considered in this thesis. With these, a set of functional and non-functional requirements are generated based on the findings of the literature review and the objectives, set out in Section 1.2.

## 3.1 System Definition

Before considering what this system needs to achieve, it is important to define the scope of the system itself. The system described in this research encompasses only software aspects, decisions regarding data-collection and sensor interaction are not considered. This system exists between data collection components and the user. However, for the purpose of this study, an application layer/frontend has been included. SEBoK Authors (2020) state that "for a service system, and also when considering the service system context, the value is realised only through service transactions", elaborating that "the end-user co-creates value at the time of the request to use this service". Naturally then, when designing a service-oriented architecture, one needs to consider the service interaction in order to ensure that service value is fully realised.

Figure 3 shows the proposed architecture and the considered system within the broader context. The 'operational context' encompasses physical assets, their associated sensors, and their data acquisition systems, as they would exist on their own. This context accounts for subsystem digital twins that may exist within a vessel, too. Isolating the operational context guides the design towards being neutral regarding these implementation details. The 'environmental context' contains external data sources that may provide information about the operational environment relevant to vessel operation. The system in consideration communicates with the existing operational context, while leveraging information from external data sources, so as to allow for servitisation; the ability to offer the association of a physical asset with services, functionalities, processes, and data access (Minerva *et al.*, 2020). This servitisation allows the physical asset, or data, to move from being merely a good to a suite of services acting upon the good (Minerva *et al.*, 2020).

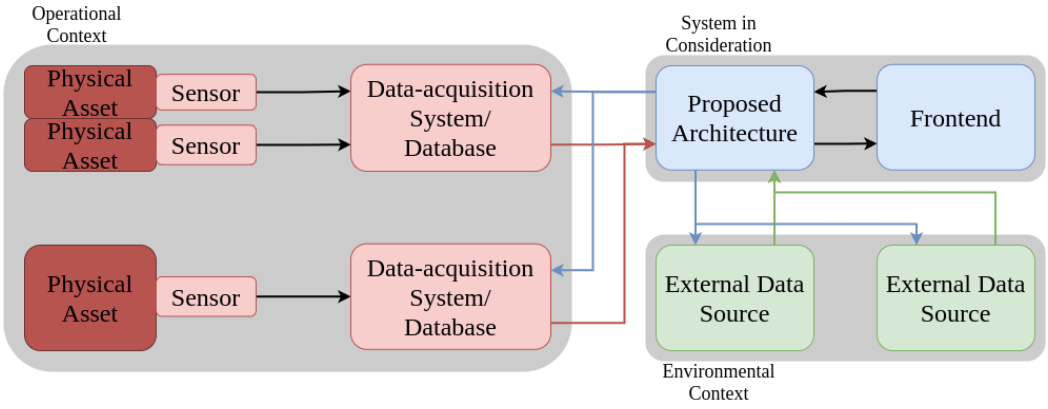



**Figure 3: System boundary diagram**

## 3.2 Problem Identification

"A system cannot be defined unless it is possible to clearly describe what it is supposed to accomplish" (SEBoK Authors, 2020). The proposed architecture serves to aid in decision-making on maritime vessels by communicating meaningful data to the stakeholders in (close to) real-time. The architecture should be capable of performing elementary data analysis such that the information it delivers is easily interpretable to users. The system should be able to communicate with services, whether they are standalone or offered by a digital twin, and aggregate the information received from them to deliver insight to operators. At the highest level, it should provide a service package to vessel operators, consisting of mission-relevant data derived from sensors and/or simulation – with this service package focussing on stakeholder needs, not the assets required to serve them. Finally, the architecture needs to be reconfigurable such that it can be customised for specific voyage requirements and can evolve with a vessel as it - and its stakeholder's needs - change throughout its life.

## 3.3 System Requirements

The systems engineering approach is well aligned with that of ISO/IEC FCD 25010 (Appendix A), which specifies how one firstly needs to understand what system must do, before considering how the system should work. The requirements for this system are split into two categories: functional requirements, and non-functional requirements. Functional requirements stipulate what the system should deliver in terms of the user, as driven by the problem described above. The non-functional requirements stipulate what needs to happen internally in order to successfully serve the functional requirements. For the most part, the non-functional requirements stem from the literature review.

In the presented requirements, broad needs are specified under “Need from source”. These are then broken down and focussed into more specific requirements under the “Need for architecture” column. Each need is subsequently given an ID for further reference during the design and evaluation chapters. The IDs are prefixed with an ‘F’ for functional requirements, and an ‘NF’ for non-functional requirements.

### 3.3.1 Functional Requirements

The functional requirements, found in Table 4, specify the functionality that the system should provide considering the context. Functional requirements can be thought of as higher-level requirements driving the design instead of constraining it. These detail the behaviour that the system strives to exhibit and assist in forming services catering to the needs of the user.

**Table 4: Functional requirements**

| Need from source | Need for architecture | Need ID |
|---|---|---|
| Aggregate information from multiple data sources | Facilitate data inputs from digital twins | F0.0 |
| | Facilitate data inputs from non-digital twin services | F0.1 |
| Provide stakeholders with insight into vessel operation, facilitating more informed decision-making | Perform (close to) real-time data processing on board for monitoring | F1.0 |
| | Encapsulate ‘simulation’ capabilities | F1.1 |
| | Facilitate machine-to-human communication | F1.2 |

There is merit in defining what real-time refers to in this thesis, as there is no explicit threshold under which a system should perform in order to be considered real-time. Instead, real-time can only be defined when considering the context in which it is being evaluated. In the context of this work, considering that simulation is a common use case, real-time does not place as strict of a latency requirement as would be on a control system, for example. To be considered real-time, the user should not notice additional latency resulting from supporting/platform functionality; the time required for these activities should be sufficiently short such that their effect is eclipsed by the processing times of the services. So long as the information flows of the system do not comprise a large portion of the overall response times, the system can be considered as real-time.

### 3.3.2 Non-Functional Requirements

The non-functional requirements, found in Table 5, define the criteria that are used to evaluate the whole system, but not for a specific behaviour. These requirements describe the functionality that the architecture should achieve internally to ensure proper operation, and will help form the services that serve only the architecture, rather than those that serve the user directly. These will help to structure the

architecture and set functional goals. The quality attributes derived from ISO/IEC FCD 25010 were consulted in the generation of these requirements. This standard is a quality model that can be used as a product quality evaluation system; it is discussed in Appendix A.

**Table 5: Non-functional requirements**

| Need from source | Need for architecture | Need ID |
|---|---|---|
| Support multiple services | Enable individual service development. | NF0.0 |
| | Enable individual service deployment and removal. | NF0.1 |
| Support multiple users | Keep track of users and their respective permissions. | NF1.0 |
| | Support multiple, concurrent requests from different users | NF1.1 |
| Service request routing | Interpret and route client requests to the relevant service(s). | NF2.0 |
| | Handle multiple, concurrent sessions. | NF2.1 |
| Security | Implement access control. | NF3.0 |
| | Ensure data integrity. | NF3.1 |
| Compatibility | Capable of interacting with legacy systems. | NF4.0 |
| | Make provision for the addition of future external systems. | NF4.1 |
| | Support distributed services, running on various machines across a vessel. | NF4.2 |
| Robustness and reliability | Recoverable in the case of failure. | NF5.0 |
| | Fault-tolerant to service and communication failures. | NF5.1 |
| | Available offline. | NF5.2 |
| Usability | Support access from multiple devices. | NF6.0 |
| | Hide back-end complexity from the users. | NF6.1 |
| Maintainability and supportability | Support logical fault-tracing and debugging. | NF7.0 |

# 4 Architecture Selection

This section considers certain non-functional requirements, identified in Table 5, to guide the selection of a suitable architectural style to follow in this design. Through literature, microservices and agent-based systems were identified as potential candidates for suitable architectural styles – attributed to their service-orientations. In the selection of architectural style, the most influential requirements will be those relating to supporting multiple services and maintainability (NF0.0, NF0.1) and request routing (NF2.0, NF6.1).

## 4.1 Microservices

Beginning with the need to support multiple services, a microservice architecture appears to be the most suitable candidate. Microservice architectures serve developers in that they strive to be more maintainable than traditional, monolithic architectures and design patterns. With a natural tendency towards loose service coupling, this architectural style fulfils the need for a modular (FN0.0) and modifiable (FN0.1) system.

While a robust and reliable system is predominantly attributed to effective implementation, a system boasting the low service coupling of a microservice architecture aids in this success. Having each service in the system existing independently isolates failures. With service failure and recovery happening independently of the greater system, full-system failure cannot originate from the failure of a single component. This isolated launching (and re-launching) of services serve the above-mentioned need for a modifiable system, too, as individual services can be updated and swapped out during deployment without affecting the operation of other services in the system.

In order to address the needs of usability and service-request routing, more detail regarding the specific microservice implementation is required. The API Gateway, a popular microservice pattern as is described in Chapter 2, provides a suitable solution to these needs. Implementing the backend-for-frontend (BFF) variation, multiple gateways are employed with each serving different frontend clients (be it mobile, web, or embedded desktop applications). This allows the architecture to serve multiple clients in a more optimised manner, giving greater control over the available information and allowing for better traffic management. This pattern serves the need for supporting access from multiple devices/systems (NF6.0).

Considering the need for service request routing, a suitable approach for this application would again be to implement the API Gateway pattern (or any of its variations) using some form of server-side discovery. This untethers the frontend from the inner workings of the architecture, limiting its concern to the effective delivery of information and providing a valuable level of abstraction for both

developers and users. Evidently, the use of this pattern serves needs NF2.0 and NF6.1 by abstracting the frontend from backend-complexity.

## 4.2 Multi-Agent Systems

With multi-agent systems holding their own in the Industry 4.0 and IoT movements, it is only fitting that they are considered in the context of digitisation. Considering the application at hand, the holonic nature of these systems provides great value through the aggregation of information. With an agent being somewhat akin to a microservice, holonic/multi-agent architectures are a suitable consideration for the service-oriented application at hand. The distributed nature of agents, where they are developed to act independently of other system components, embodies service-orientation in such a way so as to satisfy the need to support multiple services (NF0.0, NF0.1).

The social nature of agents is attractive for the distributed problem at hand, enabling components to readily share information to serve a goal. Considering the holonic nature demonstrated by agents, this provides the necessary support for request routing (NF2.0) where agents can serve a request through the employment of (routing of responsibilities to) other agents. The autonomous approach taken in doing so additionally serves the need to hide backend-complexity from users (NF6.1) as the aggregation is performed without requiring human intervention.

In an attempt to maintain component isolation and avoid rigid dependencies, agents provide a suitable solution. The reactive property of agents means that an agent is able to dynamically build an aggregation comprising the necessary agents to serve a specific request (or goal, to maintain MAS semantics). For the application at hand, this relates to ‘services’ maintaining their independence while still enabling value-adding collaboration. Additionally, the reactivity of agents means that they are able to respond to failures in the system such that they can still meet their goals - and thus, are still able to serve requests. This property serves the need for a fault-tolerant system (NF5.1).

## 4.3 Discussion

The discussion presented here considers the API Gateway and service-oriented multi-agent architectures mentioned above as potential approaches. While both these architectures boast aspects that make them attractive for this application, they each have shortfalls that require addressing.

The API Gateway, often considered an evolution of the standard service-oriented architecture (SOA), lends itself well as a digital service architecture. The focus placed on service-independence, and the separation of concerns this brings with it, could prove to be a valuable aspect in the proposed application. This separation of concerns allows teams of domain-specialists to develop their services

independently – without having to concern themselves with the development of other services/sub-domains (Harper *et al.*, 2019). This low coupling of services enhances the reliability of the system by isolating failures and enabling low-cost service recovery.

When considering a system that may consist of separate development teams, the low coupling of services makes it easier to understand the role and workings of individual services/components. Keeping the services largely independent and focussing them on performing specific tasks avoids intimate dependency between software components that developers would have to navigate when trying to understand and contribute to the system. This results in fewer integrations when adding to the system, and also means that dependencies need not be considered or adapted to facilitate the addition of new services.

Microservices, and the API Gateway specifically, are clearly a promising candidate. However, taking a traditional microservices approach presents a challenge for this application. The first issue relates to how services exist in the backend of a microservices architecture. The overwhelming majority of microservice applications, regardless of whether they follow the API Gateway pattern or not, describe their backend as a complex service-mesh. In general, when working with microservices, and especially the API Gateway pattern, aggregation is done at a high-level - taking place in the gateway itself. Because aggregation is done at this single point, coordination of services and information is the responsibility of components at the service level. While a seemingly minor detail, services are no longer focused solely on performing their task and now have to concern themselves with the tasks of other services in the system. This results in larger-grained services as they now have to contain more knowledge about the rest of the system. Adding to this, these communication links present new challenges such as failure-handling and traffic management, which need to be dealt with at a service, not system, level. There are design patterns such as the Circuit Breaker pattern (Richardson, 2018), and frameworks such as Istio that have been developed to deal with these issues – and they do so effectively.

The issue that this places on this application, however, is in how specialised domain teams interact with each other. Following this mesh approach would mean that these domain teams would have to collaborate in order for their services to work together. This is manageable when the software is being developed under one roof, but when the domain teams are developing services for their specific assets and the vessel is comprised of assets from various manufacturers, this approach is not practical. This problem reaches further by creating complexity barriers for developers wanting to add/update services, as they now have to navigate communication links between services. In this case, any non-backwards compatible changes to a service would require a re-design of all services that may be consuming that services' information.

A solution to the issue of communication links is provided in multi-agent systems where temporary dependencies are built dynamically, based on the state of the

system when an aggregate is required. Certain multi-agent architectures, such as ARTI (Valckenaers, 2019), provide explicit aggregation agents existing at lower levels of the system. These aggregating agents build holarchies that handle sub-system aggregation. This allows low-level agents to maintain their focus on achieving their goals, abstracting coordination activities to these aggregating agents. This creates a strong decoupling of low-level components, but still provides coherence among them.

This aggregation ties in well to the dynamic nature of multi-agent systems, where agents exist independently and are contracted as needed. However, the dynamic nature of these systems may provide unnecessary functionality for the application at hand. Firstly, dynamic discovery and resource allocation is not a requirement for application on a ship. The components of a ship, and the services that may stem from them, are static and individual by nature (there is only one power-train on a ship, which is static and has one task/application; compared to robots in a manufacturing line which may serve multiple applications and need to adapt to the required task). By the same logic, the intelligence of agents is not required for application on a ship. The comparatively static nature of a service on the ship means that they should not need to 'bid' to serve; they are better suited as task-specific components that form a service-package, rather than goal-driven entities competing for a contract. The goals of agent-based systems are in the same region but do not fully align with those of this application. Implementing an agent-based system here may introduce unnecessary complexity, with potential developers avoiding adoption due to the effort required to understand and integrate their services within the system.

## 4.4 Selection

From the analysis above, it is evident that neither microservices nor agent-based systems perfectly suit the application at hand. Both offer attractive aspects, but have shortfalls that cannot be overlooked, namely in aggregation and the balance of complexity – which are arguably the two most important considerations in the proposed application. As a result, a hybrid architecture is proposed consisting of a combination of useful aspects from both architectures.

The architecture presented in Figure 4 is an adaptation of the BFF architecture proposed in Section 4.1. An (optional) aggregation layer has been added to the backend to create holarchies within the system. These holarchies appear as a single entity to clients while diluting the responsibilities of the gateways and providing finer-grained aggregation. The addition of this layer enables services to maintain their independence and focus on their tasks, leaving coordination activities to services in the aggregation layer. This division of lower-level services helps to reduce complexity by maintaining a strong separation of concerns. Aggregating services contain the necessary logic to negate the need for service-to-service communication, avoiding the webbed communication networks discussed above.

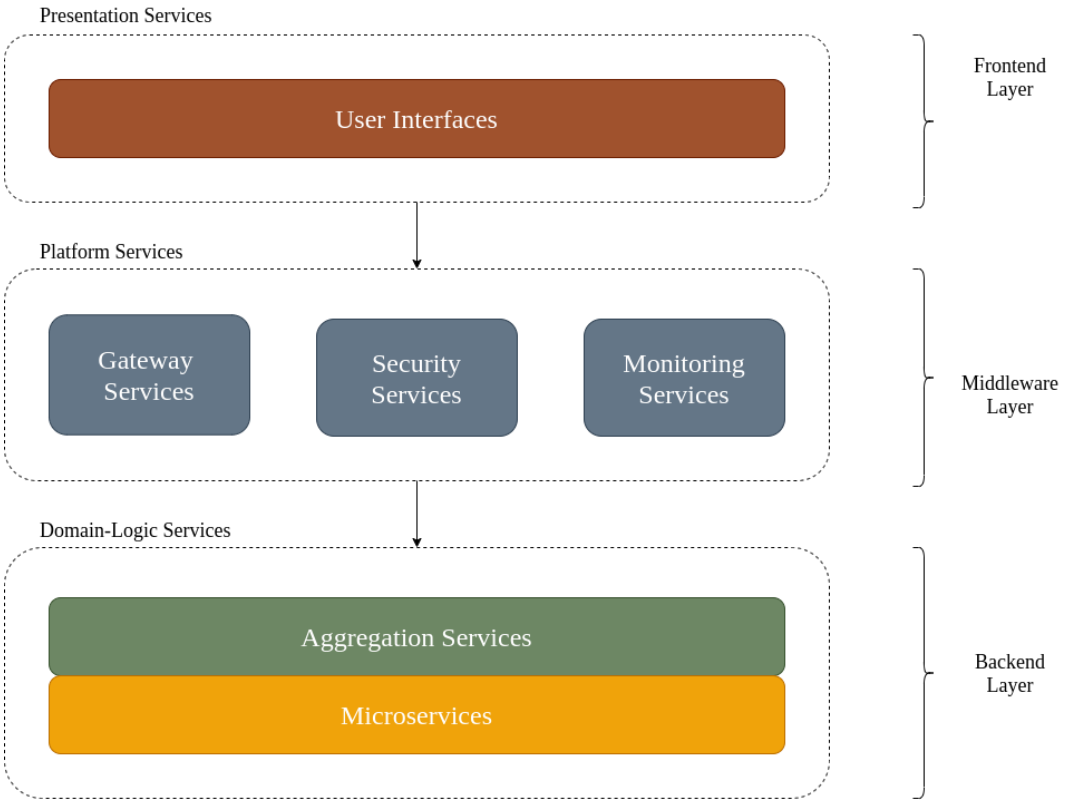



**Figure 4: Layered architecture diagram**

### 4.4.1 Frontend Layer

Services falling under the frontend layer encapsulate all user interfaces for the architecture. As the selected architecture is an adaptation of the BFF design pattern, different interfaces will exist for the different users of the architecture. No logic is placed in this layer - it simply receives the processed and aggregated data and displays it in an easy-to-interpret manner. As these services are user-facing, their source code is essentially made public. By keeping all program logic hidden in the backend, not only do users not have to navigate this complexity, but security is maintained as malicious users cannot reverse-engineer the system through analysing frontend code.

### 4.4.2 Middleware Layer

Services that live in the middleware layer are responsible for handling the majority of the non-functional requirements specified in Chapter 3. These services are all the internal components of a generic architecture, comprising all but the user interface and domain-specific components. Within this middleware layer, there are three

main service types: gateway services, security services, and monitoring services. Gateway services are the proxies associated with each frontend - these are the only entry point into the backend and handle request routing and system-wide rate-limiting and load-balancing (should the application require it). Security services are responsible for user account and authentication (UAA) – these services store user credentials and permissions, and handle authentication for the system. Monitoring services are responsible for collecting and displaying metadata about service performance and interaction for system analysis and introspection.

### 4.4.3 Backend Layer

Backend layer services relate to a specific implementation. These services are required in order for the architecture implementation to create value for the end-user. The backend contains two service types (divided vertically in Figure 4): aggregation services and microservices. For clarity, both types will contain microservices specific to the application, but the purpose of services in each layer differ. Aggregation services are driven by stakeholder needs and are responsible for coordinating calls between, and aggregating information from, services in the microservice layer. Services in the microservice layer focus on a single, specific task that provides information about whatever it is that they are focussed on. The microservice layer is therefore responsible for capturing domain knowledge, whereas the aggregation layer is instead responsible for performing business logic.

# 5 Architecture Design

The diagram presented above gives a high-level description of the structure of the architecture. This chapter delves deeper into the details of the architecture, considering lower-level requirements to guide towards a better-specified design. This chapter discusses various communication mechanisms that could be used to provide inter-service communication, as well as generic architecture components that will support services and developers. At the end of this chapter, a specific communication mechanism will be selected along with certain middleware components in order to specify a more detailed design.

## 5.1 Communication

Communication plays a fundamental role in all computer systems. Its importance is only amplified in distributed systems. The method of communication should, among other things, satisfy requirements concerning security, compatibility, and usability. Literature identified three prominent approaches to communication when implementing microservices: REST APIs, event-driven communication, and remote procedure calls (RPC). Each of these boasts a set of benefits and drawbacks, covered in the literature review. Neither is a silver bullet and as such, the decision about which to use needs to be made considering the intended application.

### 5.1.1 REST API

One of the most popular and prominent methods of communication used today is REST APIs. APIs expose information to the interested parties while acting as an interface for the services to external clients. This includes other services existing within the architecture itself.

REST, or REpresentational State Transfer, is an architectural style (for APIs) based on HTTP calls for use in distributed systems, specifically web-based systems. When a client invokes a REST API, the server hosting that API will provide a representation of the state of the resource (server). These are universally recognised as the de-facto standard for making web calls and as such, are well understood by most developers.

REST, at its core, is a set of constraints that need to be followed when *transferring* or *representing* information. Consequently, communication can be done either synchronously or asynchronously - where REST itself is neither. Synchronicity depends rather on the provisions made by the language used in the server, as REST is exclusively implemented on the server-side.

REST is a universally recognised standard. This means that it is easy for external clients to use the interface because of the accessibility to the large developer community. The learning curve for implementing REST is relatively short because

of the comprehensive resources available. Contributing to this ease-of-adoption is the call simplicity that results from following the CRUD semantics. By limiting the client to four predefined calls/actions, the communication complexity is greatly reduced.

The drawbacks of REST APIs result from one of their advantages, which is the simplicity of their semantics. REST APIs are somewhat outdated in the modern era - owing their popularity to the fact that they are already well-understood and well-adopted, and not because they are necessarily the best option. The CRUD semantics they follow can become limiting in certain customised applications. This is especially the case when one moves beyond the frontend of an application. However, the most limiting factor is the restriction to the HTTP protocol. This restriction is disadvantageous as any application wanting to make use of REST APIs is constrained to using the HTTP protocol, which is gradually being replaced by new revisions.

Another limitation one could encounter when using REST is the fact that a service can be either a server or a client, but cannot be both simultaneously. This restriction makes service-to-service invocation impossible, which is severely limiting to its application in the backend of a service-oriented architecture.

### 5.1.2 Event-Driven Architecture

An event-driven communication system is based on the publish-subscribe communication model, necessitating some form of message broker. This broker acts as a single intermediary for messages. Clients and servers subscribe to the topics that they are interested in and then post and/or consume events to/from the relevant topics. With a messaging broker, a service is able to act as both a client and a server, needing only to post requests/responses to suitable topics. This lack of role constraint makes service to service invocation a simple task.

The publish-subscribe communication model is not a transactional communication style, making it asynchronous by nature. Asynchronous messaging decouple the request from the response where each can be queued in the messaging broker. This enables more efficient communication over unreliable connections as there is less of a ‘tie’ between clients and servers. Consequently, a network failure does not necessarily cause complete transaction failure.

The most immediate advantage of using an event-driven communication model is the low service coupling it enforces. Services need not know of each other or even whether others exist; they simply need to know which topics they are interested in and process requests. Services will remain dormant until a triggering event is posted to a topic that it is subscribed to. This will hold true even if there are no other services to post events to that topic. This “ignorance” to the existence of other services provides a high level of abstraction within the architecture as services are truly independent, existing with no coupling other than the message broker itself.

This forced service independence compliments the decoupled nature of a microservice architecture.

The broker, being a relatively mature piece of software, manages message queuing and distribution. This adds a level of robustness towards service failure as the messages persist in the queue, to be picked up again once the relevant services have recovered. The queuing feature of the broker additionally unlocks the potential for streaming data; this means receiving information from the servers as it is made available and pushing that through to the clients instead of waiting for the server's process to complete before receiving any information.

The level of decoupling provided by an event-driven architecture makes service maintenance a clean process. Services can be updated or replaced with ease as they should not be tied into communication with other services. The new service simply needs to subscribe or post to the topics it is interested in and everything else will carry on working as it did before the change. This also means that it is easy to add new services and functionality to the system as there is no need to interface with a highly interdependent system. This allows domain teams to focus on their specific service without concerning themselves with the work of other teams. Should one service require another, it simply needs to post its request to the relevant topic, instead of having to know the location and capabilities of the other service. This style of service invocation helps to avoid communication chains and dependencies that have the potential to introduce latency and cascading failures. Subsequently, the freedom provided by this decoupling makes it easy to add or update services on the fly without having to worry about its effect on the rest of the architecture. This is useful once the system has been deployed and rapid service updates need to be implemented without pulling the system down.

The publish-subscribe model that event-driven communication is built on allows for one-to-many, many-to-one, and many-to-many communication without additional communication overhead. This makes it well-suited for the task of aggregation. As the communication is indirect, one-to-many communication can be achieved in a very lightweight manner requiring a single 'message' to be sent out for all recipients (as opposed to transactional communication where a single message would be sent out for each recipient). Unsurprisingly, this means that event-driven communication is naturally asynchronous – allowing for multiple requests to be sent out simultaneously before aggregating the responses.

Conversely, the low service coupling provided by an event-driven architecture comes at the cost of system complexity. While the services themselves and their interfacing procedure are kept simple and isolated – maintaining the communication channels becomes complex even at small scale. Essentially, the communication complexity is shifted from the services to the system when taking this approach, requiring a detailed model of the communication to fully describe it.

An event-driven architecture provides a highly robust system in terms of service robustness. Ultimately though, the reliability of the system depends entirely on the reliability of the broker and the machine hosting it. This single point of failure necessitates a highly robust piece of software for the message broker and contends the robustness provided by a distributed system in itself. In addition to being a single point of failure, the messaging broker has the potential to act as a bottleneck as it is a central communication channel that all components need to consult when they require information.

Event-driven architectures work well when the tasks of some services depend on the status of other services, but may not be as suitable for cases where information transfer is the goal. When task status is important, services simply post their status updates and the interested parties can consume and respond to these events. However, when information transfer is the priority, the process is not as simple. A service can post an update with available information, which may not be in the form of a structured message and, consequently, no message structure can be enforced by either the client or the server. This places more responsibility on the service developers as they now have to ensure that the messages are correctly and robustly parsed so that the information is properly extracted. This increases service complexity and introduces a potential point for human error to occur. Additionally, aggregation is achieved through the implicit design of information flows. With complete isolation of components, any aggregation of information has to be considered when designing the execution order that services follow in their consumption and publishing of information. This requires some forward thought towards service configuration so that services can effectively be re-used.

### 5.1.3 Remote Procedure Calls

Remote Procedure Calls (RPCs) is a communication mechanism that is well suited to distributed systems. This is a result of its ability to hide networking complexity and as such is one of the most suitable communication styles for microservice architectures (Murthy, 2017). An RPC can invoke a procedure to execute in a different address space, or a different machine, while programmed as if it were a normal function call. When using RPCs, the servers and clients are stubbed such that the RPCs mimic local procedure calls – shielding them from networking details. This provides a useful level of abstraction to service developers as they do not have to concern themselves with details pertaining to the remote interaction. A client requiring a service from a server on another device simply makes a call to invoke this service in the same way that it would invoke a local method. In this interaction, the stubs take care of the network details between the server and client. Since no specialised message-broker is required for RPC communication, the related potential bottlenecks and dependence on highly-mature software are eliminated. Even without a broker, RPCs can still be made asynchronously, depending on the provisions made by the implementation language of the server.

RPCs additionally give the freedom to create customised semantics, which provides valuable flexibility for application in backend services. Additional freedom is granted due to RPCs protocol-agnostic nature, giving designers the ability to select whichever protocol suits their application best. This allows one to employ as many protocols as wanted within a given application, should that be desirable. Additionally, RPC implementations are not limited by a set of constraints like REST APIs are. Consequently, server and client roles are more relaxed, which allows a server to exist as both if necessary.

To be suitable in the application at hand, RPCs would need to allow for information to be aggregated. Fortunately, RPCs allow for one-to-many and many-to-one communication to take place – with aggregation being a common activity when using this communication style.

While avoiding the need for a specialised broker when using RPCs, it is required to stub all clients and servers. This concept is a potential barrier that new developers will have to grasp and overcome when contributing to the system. With the stubbed message, communication is also more intimate, following a transactional style. This is not necessarily an issue, but it does increase the coupling between services and reduces the speed at which messages can be exchanged when compared with a brokered implementation. However, RPCs are more committed to maintaining the core principles of distributed systems when compared with the event-driven style. Additionally, in order to generate the server and client stubs for RPC, a standard interface needs to be defined. This interface acts as a contract between servers and clients, specifying a standard message structure communicators will use. With this approach, parsing is taken care of on behalf of the developers, removing the potential for errors relating to message interpretation.

Unlike event-driven architectures, data streaming is generally not a feature that is considered in traditional RPC implementations. Following the transactional communication style rather than using queued messages, information is only returned to the client once the invocation process has completed, ending the transaction. However, while not an explicit feature of RPC, certain RPC frameworks do offer streaming functionality depending on the underlying messaging protocol. HTTP/2, for example, supports streaming in multiple configurations - so any RPC framework leveraging this protocol should support message streaming.

## 5.2 Middleware

Recall that Section 2.3 described middleware as a component that simplifies the connectivity between application components (IBM, 2021). Through leveraging middleware, one can provide standard functionality across components of a system. In the context of this work, this provides a valuable abstraction for developers wanting to contribute as middleware can perform the majority of the system-

integration functionality on their behalf. Considering the middleware layer proposed in Chapter 4, and the communication protocols discussed in Section 5.1, there are four relevant types of middleware to consider in this application. These are security middleware, monitoring middleware, message-oriented middleware, and communication middleware.

### 5.2.1 Security Middleware

Security middleware refers to all software components that contribute to guaranteeing secure communication within the system. Considering the application at hand, this encompasses everything that deals with authorisation, authentication, and information integrity. This includes services that handle user profiles, their permissions, and access control. These services can - and most likely will - vary in their scope, with some acting at a global level (looking at the greater system) and others at a more granular level (looking at individual services). In this implementation, these components may materialise as a service that users can log in with or as a service that other services can use to verify a user's request.

### 5.2.2 Monitoring Middleware

Monitoring middleware encompasses the components that track or monitor transactions within the system. These do not add inherent value to the user, instead, they assist developers to analyse the system and its usage. These components track all service interactions and enable insight into how the services communicate within the system. In distributed systems, monitoring middleware plays a vital role as it provides developers with a look into how services are performing and aids in fault finding within the context of the greater system. These components usually manifest in the form of logging frameworks and performance trackers, which record service interaction states and metrics that describe individual service performance.

### 5.2.3 Message-Oriented Middleware

Message-oriented middleware enables components using different messaging protocols to exchange messages. In addition to providing a 'translation' service, these components manage routing so that messages are delivered to the correct services in the correct order. For the presented application this type of middleware encompasses the gateway component, which handles message routing and traffic management. Typical message-oriented middleware components are implemented as proxies or proxy-based frameworks. These implement load-balancers, rate limiters and message routing based on application-specific criteria.

### 5.2.4 Communication Middleware

Communication middleware is a type of middleware that assists in the implementation of the selected communication mechanism. There will be specific middleware components for REST APIs, RPCs, and brokered-communication within this category. These middleware components act at a higher level than those

discussed above; potentially including other middleware components within themselves. For example, certain communication middleware may perform authorisation itself, negating the need for such middleware components.

RPC middleware is the software that enables the distributed nature of the call while allowing it to be used as if it were a local call. This software handles the networking complexity of RPC calls, abstracting it from the programs employing the RPC. RPC middleware performs the stub generation of clients and servers and, as such, generally dictates a message format. In doing so, the RPC middleware handles the parsing of messages as the protocol and structure are somewhat embedded in the stubs. Broker middleware, also known as Object Request Broker middleware (ORBM), refers to the aforementioned 'mature' piece of software. ORBM executes the message queuing functionality in a publish-subscribe model. This software manages the topics implemented in event-driven architectures and queues the messages as they are posted and consumed. It is worth noting that ORBM does not specify message structure and, as such, message parsing is the responsibility of services. As REST is a set of constraints and not an explicit communication procedure, no middleware components are required to implement it. Middleware is commonly used to refer to software that acts on an API call before it is processed by the server when working with HTTP. This is applicable when using REST APIs as they often use the HTTP protocol. In this case, the communication middleware is not used as a means to implement the REST API. It rather supports the implementation by enabling other middleware to be implemented.

## 5.3 Security

As is mentioned in Chapter 2, Firdhous (2012) defines five key security considerations that one needs to make when working with distributed systems. In addition to this, Newman (2014) added a sixth security consideration when developing microservices specifically.

Successful consideration of the protection of data in transit ensures that the data sent arrives at the intended destination untampered with, and without being viewed by any other party during transmission. This is achieved through encrypting messages and is advocated for in microservice applications by Richardson (2018) where TLS encryption is suggested as an encryption mechanism.

User authentication and access control tend to be considered together. This combination enables the protection of sensitive information and allows a system to support users with different roles. When implementing the API Gateway pattern, Richardson (2018) advocates for authentication to be done in the gateway and authorisation to be done in the service(s). He explains this as the access token pattern, where the gateway provides access tokens to the services when routing requests to them. Through this approach, the gateway's responsibility is to ensure that the user has access to the system. This necessitates that microservices ensure

that the user has permission to access the specific information that they are querying. This divide enables authentication to be a system-wide consideration, with authorisation being more granular. This is a logical approach as services are given full control over who has access to each of their offerings, but do not have to concern themselves with the security requirements of the greater system.

The explicit consideration of DOS attacks is a means to ensure the availability of services and stems from security requirements of computing clusters (Firdhous, 2012). By implementing a rate limiter, one can slow down the attacks enough to minimise the effect that they have on service delivery. Rate limiting can be implemented globally at the gateway, and more granularly at a service level. This multifaceted approach to security allows for tailored rate-limiting of services based on their individual performance. This would be beneficial for computationally-heavy services, such as those that run simulations. Additionally, considering rate limiting both globally and locally ties into the final requirement described by Firdhous (2012) as multi-level security considerations.

Multi-level security considerations was initially identified for distributed systems in Firdhous (2012) but was found to be explicitly advocated for in microservices applications by Newman (2014). This details the implementation of security measures at all points within a system. In complex systems with multiple potential points of access this is critically important, hence the relevance in distributed systems. This does not detail specific security measures. Instead, it specifies that security measures span all levels of a system where possible. Accounting for the other considerations discussed above, this philosophy can be applied in three instances. The first would be to configure the gateway to only offer services that the user has access to. As mentioned above, rate limiting is considered at both a global and service level – this contributes to defence in depth. Finally, messages should be encrypted within the system in addition to at external interfaces.

Isolation is inherent when designing microservices, and as such is more of an implementation consideration. While being isolated in nature, running services on a common machine presents potential faults to this isolation. In the case where services share an operating system, any malicious access to the host OS could result in all services deployed on that OS being compromised. It is not always feasible to run each service on a separate machine as this could result in thousands of greatly-underutilised machines. An approach to dealing with this that has become commonplace in microservices, especially with the advent of cloud-computing services, is that of containerisation. Containers are lightweight operating system instances that can run on the same machine concurrently. Containers are far more lightweight than virtual machines as they operate on a single kernel. This approach provides protection against this kind of infiltration as invalid access is constrained to a single service/OS.

## 5.4 Architecture Specification

### 5.4.1 Communication

REST, constrained by its semantics and ties to the HTTP protocol, is unsuitable to use anywhere other than the frontend for the application at hand. Considering RPC, allowing for tailored semantics and boasting protocol-agnosticism, is a far more suitable communication style for a backend implementation. Considering that the frontend in this application is not dominantly web-based, RPC is equally suitable for use in the frontend.

The decoupling of services, indirect messaging style, and ease of messaging multiple services makes event-driven communication an attractive option for use in microservices. Additionally, the ability to stream data improves real-time communication in a system boasting simulation capabilities – as is the case here. With streaming, processing throughput is increased as large, timely simulations can be broken down into a series of smaller simulations; where the series of smaller simulations yield more timely results. However, the potential issues involved with unenforced information parsing is a concern. This is an especially important consideration where different teams will be responsible for defining their interfaces. Considering that the focus of this system is on information and not task coordination, an event-driven approach introduces unnecessary complexity and risk. Conversely, the stub generation process of RPC requires that a message structure and standardised protocol be defined in the interface. In doing so, the stubs perform message parsing on behalf of the service. This removes the potential for message misinterpretation.

An option that is not unheard of, and is described in an example by Murthy (2017), is to design an architecture combining the three discussed communication styles. This approach suggests using REST APIs in the frontend, where user interfaces need to communicate with the architecture. RPCs and message brokers are used for backend communication, where information transfer is handled using RPCs and task coordination is handled by a message broker/event-driven communication. Considering the size of the application being considered, this fusion approach is unnecessarily complicated.

The considerations presented in Section 5.1 indicate that RPC is likely the most suitable communication mechanism for use in the proposed architecture. In the discussion to follow, it is specified that the selected RPC framework should support message encryption, relating to the need for data integrity (NF3.1), and some form of interceptor middleware. Additionally, to cater to further security requirements, implementations should utilise containerisation technology to ensure true isolation and portability of services. In the application considered here, where internet connection is not guaranteed, having pre-configured containers means that services do not need to be built or have dependencies fetched from online repositories during

a voyage. Instead, their containers, which contain these dependencies already, can be launched and migrated as required.

With a communication style selected, the layered architecture presented in Figure 4 can be expanded upon. A generic, lower-level architecture is depicted in Figure 5. The discussion to follow specifies how the presented components are to be implemented considering the discussion on middleware presented in Section 2.3.

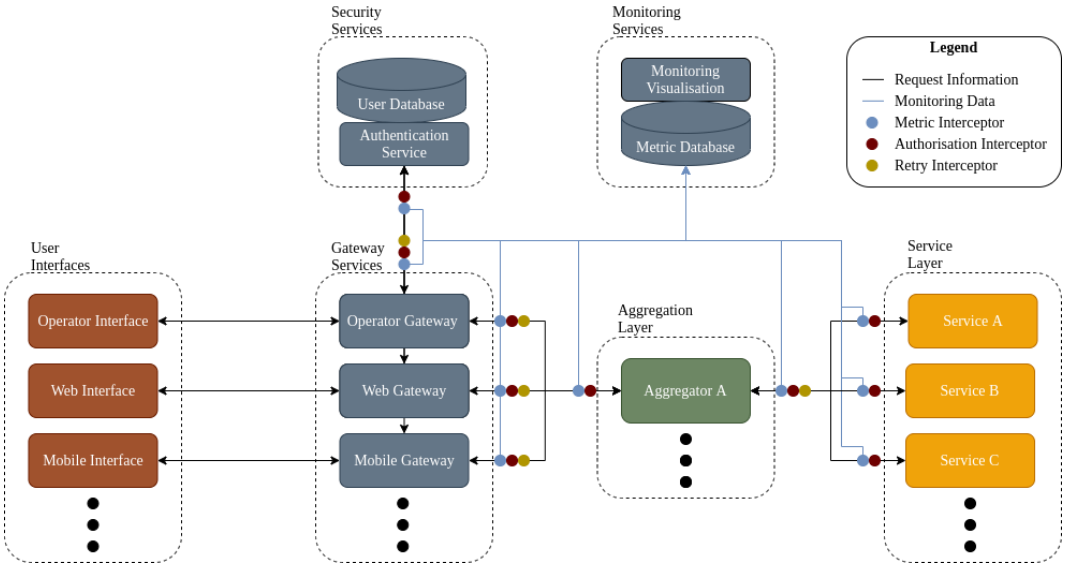



**Figure 5: Architecture diagram**

### 5.4.2 User Interfaces

Multiple user interfaces can exist, with each being tailored to a specific client. Operator interfaces, for example, will be optimised to require minimal navigation, allowing operators to focus on piloting the vessel instead of navigating their information services. Mobile interfaces will likely be more involved and display less information per page due to the nature of the device.

### 5.4.3 Gateway Services

The gateway services are the only access point to the system, acting as a receptionist. In order to confirm access to the system and its services, the gateway communicates with security services to authenticate users who want to use the system. Richardson (2018) shows how providing each individual frontend with a gateway allows for better traffic control and finer-grained access to services.

An off-the-shelf message-oriented middleware component can be used to implement the gateway. This allows for easy reconfiguration of the backend system as protocol translation and routing is handled through a configuration file rather than a custom codebase. A suitable message-oriented middleware component for

this task would be a proxy that can essentially represent the backend system to the frontend/user. Most proxies exhibit the described translation and routing functionality out of the box, with the additional option of rate-limiting and load balancing being a standard feature. By configuring the proxy to implement rate-limiting at the gateway, system-wide rate limiting can be implemented in addition to service-level rate limiting. This can be used to provide preference to priority users at the system level, which could be useful in situations where networks are constrained.

Considering defence-in-depth, the gateway that is selected should at least provide facilities for message encryption and rate-limiting. Employing encryption at the gateway, as well as in the selected RPC framework, allows for one to encrypt both internal and external messages. Additionally, through leveraging two points and approaches to encryption, separate certificates can be used for internal and external communication. This provides an additional layer of security - where one set of certificates become compromised, the extent of the damage is constrained to that communication channel only. Rate limiting in the gateway allows for the system to be configured independently of the services, which provides high-level control over inbound traffic and acts as the first line of defence against DOS attacks. Here, the system can be configured to only allow the maximum expected traffic on board the specific vessel (145 pax in the case of the SAAII).

### 5.4.4 Security Services

Considering the points discussed in Section 5.3, it is evident that security needs to be considered at both a global level and a service level. With this in mind, multiple security middleware components will be included in the architecture specification. To deal with authentication needs, and following the suggestions of Richardson (2018), an authentication service (with an associated database of users) should be included to only allow valid users to log into the system. This service facilitates security at a global level by generating access tokens for valid users to use with their queries. To successfully implement the access token pattern (Richardson, 2018), each service should enforce authorisation for itself. This can be achieved through a generic middleware component that is configured and employed by each service. For RPCs, this can be achieved through the use of an authorisation interceptor. By doing so, developers do not have to deal with implementing the system logic of authorisation, having only to add the generic interceptor to their server. This interceptor can be added to all components, not just low-level microservices, to provide the system with defence-in-depth. Considering authentication and authorisation in this manner provides the required support for different users (NF1.0, NF1.1, and NF3.1).

To round out security middleware considerations, rate-limiting at a service level can be implemented through the use of a generic interceptor, too. This interceptor can be configured to rate limit based on user or IP, providing tailored DOS

protection to the services. Again, by using a generic interceptor, developers can focus on their service without the need to deal with complicated system integration.

The final security consideration is that of message encryption. Encryption has become an industry standard and as such, is generally provided for in messaging frameworks. This specification is not assigned to a specific component but will be used as a requirement for selecting an RPC framework and proxy technology during implementation.

### 5.4.5 Monitoring Services

Monitoring services collect performance and usage data from the services, including information about service usage, failure, and response times. These can be used for debugging and introspection and can help to identify sub-standard services and target backend optimisations. These services consist of the metric database, recording quantitative metrics about service interactions, and a monitoring console that displays this information and trends to system administrators.

Monitoring middleware can be achieved in one of two ways: either through a service registry component or a monitoring interceptor. Adding a service registry requires that each service registers itself on start-up and posts interaction information either before or after each interaction. This approach requires service developers to interface with other components in the system which conflicts with the approach taken in this system design. A better approach would be to specify a generic monitoring interceptor that records the service interaction of the service it is added to, and have this interceptor post information on behalf of the service. This allows the developer to maintain focus on their service rather than the system, which is better aligned with the objectives of this work. Additionally, with interceptors already being considered for security middleware, this decision does not add any additional component types to the system specification.

### 5.4.6 Aggregation Layer

The aggregation layer is the first layer of the backend, consisting of custom software. Services that fall under this layer are inspired by the staff holons of the PROSA aggregation (Valckenaers, 2019), differing from standard microservices in that they do not have an explicit task to do. These services are responsible for invoking services required to provide certain information, and aggregating the responses. To the gateways and users, aggregators can be thought of as providing a ‘service package’. These services send out concurrent, non-blocking requests to all the necessary services, aggregating the responses and returning them to the user (through the gateway).

While a seemingly simple adaptation from having a pure service mesh in the backend, having this aggregation layer provides great value for implementation

with specialised domain teams. The aggregators coordinate information among services, so that in the case where a Service A requires information provided by a Service B, it will not have to know of Service B or its location. The aggregator has this logic programmed into it, first invoking Service B to get the necessary information, then relaying that to Service A as a message argument when making the invocation. This approach provides a valuable level of abstraction, especially when working with services developed by domain experts. These experts can focus on developing services that encapsulate their knowledge, without having to concern themselves with how their service will interact with others. This abstraction further helps to avoid backend complexity, avoiding communication webs between services that could cause lock-ins and cascading failures.

### 5.4.7 Service Layer

The service layer embodies the lowest level of this architecture and is where all the microservices reside. Because of the inclusion of the aggregation layer, these services truly live up to the vision of the microservice, dedicating themselves to performing a single task and nothing else. These services are not concerned with other services or locating the required information, they simply receive a request and serve it. The simplicity and independence that is afforded by the aggregation layer makes updating and optimising services incredibly easy, and allows them to remain as lightweight as possible. Because the code of these services only serves to achieve the task it exists to do, and not any system-integration activities, it is also easier for new developers to understand and contribute to the service.

The customised semantics enabled by RPCs make it easy to tailor the calls that can be made to these service. In this architecture, digital twins simply expose available services through an RPC server. These look the same as any other service would to clients in the architecture, and are interacted with in the same manner, too. In the case that a digital twin is unable to expose itself through RPCs due to its siloed development, an RPC client can be set up to act as a ‘translator’ between the architecture and the service exposed by the digital twin. This is enabled by the relaxed server/client roles of RPC.

# 6 Case Study Implementation

This chapter details the implementation of the specified architecture in a case study. The case study manifests as an implementation of the architecture proposed in Figure 5, applied to the SAAII. This implementation will serve as a basis for the evaluation of the architecture design, with the evaluation described in Chapter 7. This chapter outlines the objectives of this case study, the methodology taken in implementing it, and the specific components comprising the case study itself.

## 6.1 Objectives

The implementation of the proposed architecture is designed such that it aggregates information from various sources in a way that creates value beyond that which each source could provide independently. Additionally, the implementation should servitise existing engineering models and algorithms that have been developed by domain experts in previous studies. This showcases the design's considerations towards service development by independent teams.

Beyond meeting the design requirements, the case study is designed such that it showcases specific characteristics. These are the characteristics that are omitted from the evaluation due to their implementation-specific nature, this is discussed in Section 7.1. Recalling that service isolation enables varying technology stacks to be employed, the implementation includes different but suitable technologies based on the service requirements. The services are decomposed such that they are aligned with specialised domain teams and can be reused to provide different information to the user. This displays the composability characteristic of microservices. Finally, this case study deliberately includes services existing as both clients and servers, since this functionality was a key consideration in the architecture design and communication mechanism selection.

This case study, therefore, aims to verify the suitability of the proposed architecture for the aggregation of information on maritime vessels. The case study comprises a minimal implementation meeting the requirements set out in Chapter 3. This implementation will serve as a basis for testing and evaluation.

## 6.2 Methodology

The development of services that encapsulate advanced domain knowledge is beyond the scope of this thesis. As such, the majority of the services used in this case study implementation were curated from a repository of past studies carried out by the Sound and Vibration Research Group (SVRG). Each of these studies imparts their own contributions to the research community, but are used here merely as examples of service instances that may exist within the maritime context.

These services were originally developed as stand-alone solutions, they all followed a siloed development style to serve the studies' initial goal. These were developed in complete isolation, with no intention for future collaboration or servitisation. Many of these studies were conducted over different time periods too, serving research needs that evolved with the vessel. The individuals who developed these models and algorithms are considered domain experts, with their studies focussing on specific and intricate details of the SAAII. This siloed development by domain experts is a recurring theme when working with maritime systems and, by leveraging this, verifies the design choices made when designing the architecture regarding individual service development by specialised domain teams (NF0.0).

## 6.3 Implementation

This section describes the implementation of the proposed architecture. It details the process followed to develop the case study, as well as the technologies selected to implement it.

### 6.3.1 Implementation Strategy

This case study involved the development of both generic and study-specific components. The generic components are those that hold no relation to a specific implementation and can thus be reused across multiple implementations with reconfiguration. These were developed without considering domain logic to maintain their generic nature. The study-specific components relate to this case study, specifically, and consist of models and algorithms made available through the SVRG. These are included to effectively demonstrate the aggregation and coordination of information in a maritime environment through this architecture.

The implementation will consist of various services which would traditionally exist as a monolithic system (or multiple, independent monolithic systems). In this hypothetical monolithic system, these services would likely be deeply intertwined with repeated functionality and replicated data. By instead following the proposed microservices approach, each service exists independently within the system such that the services, and the information that they provide, are reused where practical. Additionally, by designing each operation with clearly defined interfaces, their data can be used with or by other operations to enhance the value that it can provide.

During testing, the gateway, authentication service, and relevant interceptors (rate-limit, retry, authorisation) were reconfigured as needed to change how services would react to specific requests. As is mentioned in Chapter 3, the frontend needed to be considered to demonstrate certain architectural requirements but doesn't fall into the scope of this project. As such, a suitably simple frontend was developed to serve the needs of this case study. It allows for the required inputs to be provided when invoking services, displays the outputs, and only displays offered services based on the user's role.

### 6.3.2 Implementation Platform and Technology Selection

Considering the specification made in Chapter 5 that the selected RPC framework should support message encryption and some form of interceptor middleware, gRPC has been selected for this implementation. gRPC is covered in Section 2.4.3 and is a highly-suitable framework for use in microservices, providing interceptor support and encrypted messaging out of the box.

Following the recommendations of Berger *et al.* (2017), each service will be deployed in their own Docker container with all containers running within their own network. This is done to support migration between machines during development and deployment as containers improve the portability of services – requiring no further installations or builds between machines as dependencies are implicit to the container. Containerisation is a popular deployment approach for microservices, ensuring fully de-coupled, lightweight services that fail in complete isolation while being easily recoverable. Additionally, building Docker containers beforehand, known as *baking*, yields Docker images containing all the required dependencies. Through the *baking* of containers, offline operation is guaranteed as everything required by the service is available locally. This is likely how this system would be deployed on board a vessel, so it will be tested in this manner to best mimic the real-world deployment environment.

#### 6.3.2.1 Generic Components

The gateway component was said to most likely be implemented as a proxy. Google advocate for using Envoy as a proxy for gRPC applications, as it supports the translation of HTTP/1 to and from HTTP/2. This is done to enable gRPC to run in the browser, as modern browsers do not yet support the full range of HTTP/2 functionality used by gRPC. Based on this support, and its successful adoption in industry, Envoy will be used to implement the gateway component. This proxy will route user requests to the relevant backend services while translating any requests where necessary. In addition to this, Envoy will be configured to rate-limit requests at the system level. The gateway is documented in Appendix C.9, and enables the web gateway to be written in JavaScript using gRPC Web. In addition to the gateway, a generic authentication service has been developed in Golang, as is documented in Appendix C.11.

A custom retry interceptor was developed in order to offer clients with fault-handling functionality. This interceptor adds exponential back-off retry logic to connections. In the case that a server is inaccessible the very moment a client tries to connect to it, or in the case that a server goes offline while processing a request, the client-side interceptor will catch the error and retry the request, with the back-off logic gradually increasing the time between retries instead of throwing an immediate error. This provides relief by giving the server enough time to process internal issues, or to restart itself before the next call is received. The source code for this interceptor can be found in Listing 1.

A custom metric interceptor was also developed, allowing for service metrics to be recorded. This interceptor records all client and server interactions with metrics for latency, traffic, and packet size. These metrics are sent to a Prometheus server, which is running in its own container, ensuring that the data describing service interactions does not reside with that service. This centralises the system monitoring and ensures that metrics about service performance persist the services themselves. All source code relevant to the logic performed by this interceptor can be found in Listing 2, Listing 3, and Listing 4.

Additionally, a rate limit interceptor was developed for servers. This interceptor boasts logic that tracks the number of active calls being processed by the server as well as the limit to concurrent calls imposed by that server. Once this interceptor has recorded that the server is currently processing its maximum number of concurrent requests, any further requests are rejected so as to protect against DOS attacks. Source code for the logic performed by this interceptor is presented in Listing 5.

The final middleware component developed for generic use is a custom authorisation interceptor. This was developed to enable role-based authorisation, complimented by the authentication service mentioned above. This interceptor can be added to any server in order to achieve service-level authorisation. This interceptor loads in permissions for every service that it is added to, catching requests before they reach the server. The interceptor extracts an access token from the request metadata and verifies that the user requesting a service call is permitted to do so before forwarding the request to the server. The relevant source code for this interceptor can be found in Listing 6, Listing 7, and Listing 8.

At this point, it is worth noting that interceptors are language-specific components. As such, any additional language that is employed requires that the interceptor logic be replicated for that language.

#### 6.3.2.2 Domain-Specific Components

Figure 6 shows the case study's information flows and the relationships between them. Note that in this diagram, the data flows into the metric database (Prometheus server) have been omitted. These links were purposely left out in an attempt to avoid a cluttered diagram. For the same reason, interceptors have not been shown on this diagram. However, in both cases, the same metric database connections and interceptors have been employed as are shown in Figure 5, previously. Each of the services, and their information flows, are individually discussed in detail in Appendix C.

As is mentioned above, microservices were selected from a repository of past projects of the SVRG. Certain projects were documented such they could be servitised by adding server code and making minor adjustments to the original source code, while others required complete development from scratch. The

microservices used in this case study include: the ocean weather service (responsible for collecting environmental data), the power-train service and vibration estimate service (showcasing different data-driven modelling approaches), the comfort service (running remotely), and the propeller monitor service (running remotely to showcase numerical modelling and services offered by digital twins). Here, majority of services have been written in Python 3, with the vibration estimate service being implemented in C#.

Aggregator services were designed to demonstrate service coordination, service reuse, and information aggregation. Three aggregators have been included in this case study, documented in detail in Appendix C.6 to C.8. The route analysis aggregator coordinates and aggregates information between multiple microservices to provide a multifaceted, high-level summary of the proposed route. The associated user interface allows users to click on metrics of interest, where they are provided with more detailed insights through automated requests to the power-train aggregator or vessel vibration aggregator. The power-train aggregator provides a high-resolution time series description of the power consumption along a proposed route. This power consumption is overlayed with the additional cost incurred at each point along the route – providing stakeholders with insight into the power consumption/cost trade-off and allowing them to alter routes to minimise power and cost requirements. The vessel vibration aggregator orchestrates service calls to provide an estimation of the whole body vibration that may be experienced on the bridge along the given route. The aggregators additionally contain specific logic to convert the information as is required for various services. For example, the ocean weather service should not have to contain logic for the ship's heading to provide relative wind and wave information. Instead, the ocean weather service provides absolute values for these, with the aggregator converting them to relative values where subsequent service invocations may require them. All aggregators have been written in Golang, as is documented in Appendix C.

The middleware layer encompasses all generic and reconfigurable services, and have been configured specifically for this case study. Additionally, the middleware layer encompasses the interceptors described in Section 5.4, although it is not displayed in the figure below for reasons already mentioned.

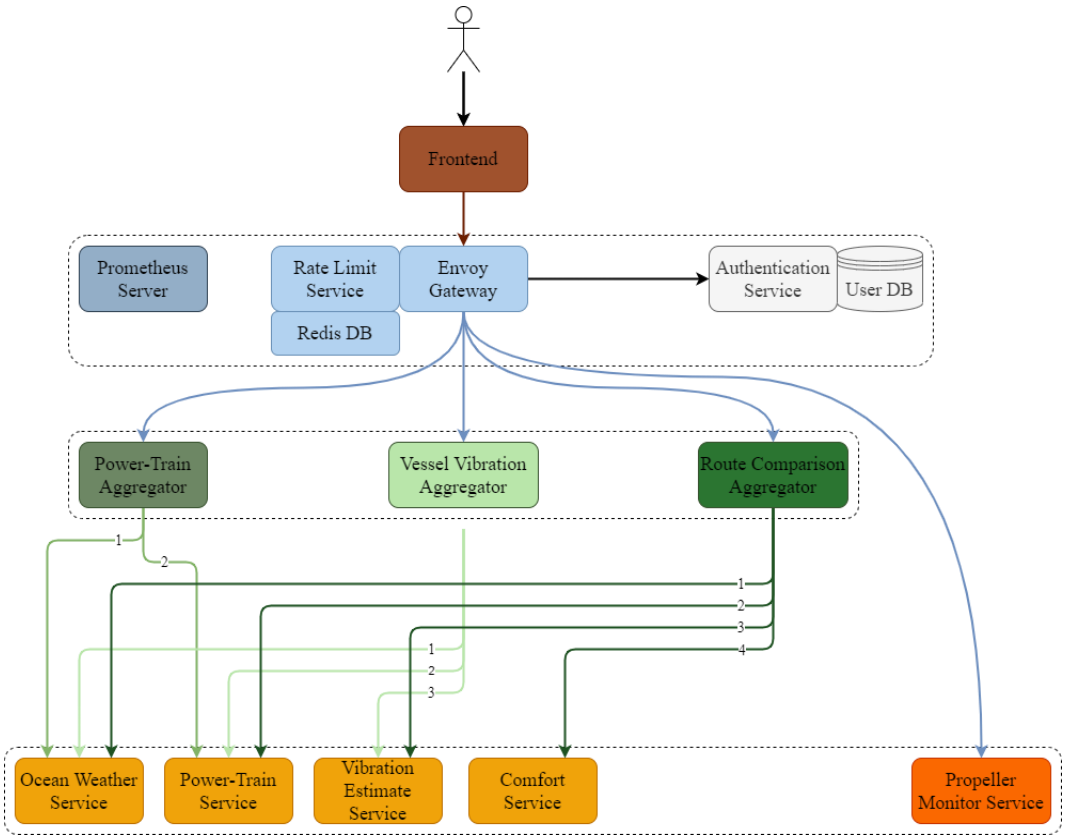



**Figure 6: Case study diagram**

# 7 Case Study Evaluation

This chapter serves to evaluate the architecture implemented in the case study described in Chapter 6, considering the requirements formulated in Chapter 3. This evaluation utilises the ISO 25010 standard, presented in Appendix A, as a guideline for evaluating the criteria. This chapter details the generation of the evaluation criteria below. Thereafter, it presents the results before discussing them considering the formulated criteria.

## 7.1 Evaluation Criteria

This section details the evaluation criteria. Table 6 presents the metrics appropriate to the evaluation of the design. For criteria where metrics could not be derived, a discussion is provided in Section 7.4 along with a discussion of the metrics – these are included in the table but have no metrics assigned to them. The proposed architecture design builds on the core characteristics of microservices. The case study showcases the design decisions made, with this evaluation serving to validate these characteristics and decisions. Beyond the design decisions, this evaluation should benchmark the behaviour of the system for those requirements that can only be considered during implementation. The criteria specified here dictate the experiments required to perform an evaluation, as discussed in Section 7.2.

Note that in Table 6, not all microservice characteristics and decisions have been included. Characteristics, such as enabling technological heterogeneity, allowing for organisational alignment, and supporting composability cannot be evaluated for a generic architecture. Instead, these were considered in the case study design. In validating the decisions to use RPC as a communication mechanism, motivation based on flexible server and client roles is again considered in the case study design, where aggregators and gateways act as both servers and clients in the system.

Considering that the proposed architecture follows the microservice architectural style, the initial evaluation criteria are based on the characteristics of microservices (presented in Chapter 2). Accounting for service independence, the evaluation should verify that faults originating in a single service do not affect any other component of the system. Microservices are often favoured as each service can be individually deployed and updated. This evaluation should verify that this architecture supports individual service deployment and removal during operation.

Beyond the criteria for a microservices architecture, certain application-specific criteria require evaluation. These refer to the aforementioned design decisions and benchmarks. A major decision made in the architecture selection was to employ the API gateway pattern with an additional aggregation component. The gateway was selected to handle request routing, protocol translation, and provide an easily reconfigurable system; while the additional aggregation component was included to enable greater service independence and simplicity. The experiments should thus

be designed such that they can reasonably validate the system's behaviour regarding request routing, protocol translation, and reconfigurability, without requiring complex service navigation. The experiments should additionally verify that through the inclusion of the aggregator component, services are allowed to exist independently of others in the system.

**Table 6: Evaluation criteria**

| | Source of Criterion | Microservice Characteristics | | | Architecture Selection Decisions | | | | | Architecture Design Decisions | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Criterion** | Fault isolation | Enable independent service deployment | Enhanced maintainability | API gateway for request routing | API gateway for protocol translation | Reconfigurability | Ensure service independence | Aggregation layer for hiding complexity | RPC for real-time response | Gateway for authentication | Services for authorisation | Message encryption | Gateway and service rate limiting | Metric collection in service providing structure | Monitoring service for persistent metrics |
| **Metrics** | | | | | | | | | | | | | | | | |
| **Quantitative** | Response time | | | | X | X | | | | X | X | X | | X | | |
| | Number of concurrent requests | | | | X | | | | | | | X | | | | |
| | Number of concurrent sessions | | | | X | | | | | | X | | | | | |
| | Launch time | | X | X | | | X | | | | | | | | | |
| | Lines of code | | X | X | | | X | | | | | | | | | |

Following the selection of the architecture, important decisions were made in the Architecture Design. Most notably was the selection of RPC as the communication mechanism in the architecture. Relevant to this evaluation, the selection was motivated by RPC's support for distributed services. The experiments should thus evaluate the support for running, and communicating with, services remotely. In the architecture design, decisions were made regarding middleware components, too. Security middleware was specified to perform authorisation at each service interface, message encryption both internally and externally, and rate-limiting of both the system and services. Along with authorisation at each service interface, it was stated that authentication would be performed at the gateway level. The experiments should thus support the evaluation of these security decisions. Beyond security, middleware was specified to provide system monitoring functionality. The experiments should verify that metrics are collected in a structured manner and that they persist the services that they describe.

## 7.2 Experiments

Considering the evaluation criteria, four experiments are identified to evaluate this system: a standard operations experiment, a forced failure experiment, a security experiment, and a reconfigurability experiment. Metrics regarding system performance will be collected by the monitoring middleware throughout the experiments, with the raw results documented in Appendix E. Through the use of these metrics for evaluation, the monitoring middleware requirement for structured data storage is verified. Here, each service produces labelled data that can be efficiently searched and compiled based on its assignment.

### 7.2.1 Standard Operation Experiment

The standard operations experiment aids in evaluating whether or not standard functionality is achieved. This includes aggregation activities, request routing, protocol translation, and metric collection, as well as how the system operates in a simulated deployment environment (unreliable internet connection). With the help of the data collected by the metric interceptors during this experiment, benchmarks for F1.0, F1.1, F2.1, and F6.0 can be provided. The procedure for the standard operations experiment is provided in Appendix D.1, with the benchmark data presented in Appendix E and discussed in Section 7.3.

This experiment involves running the system as it would be in a deployment. The system will be run for a prolonged period (for this experiment, 24 hours was deemed sufficient to ensure stability). During this time, sporadic but regular requests will be made to the system from various machines on the local network with the developer verifying that the system is behaving as expected. Calls will be made from various devices and by different users to ensure that the system's requirements for multiple device and user support is stable.

From the metrics recorded for this experiment, the service with the most consistent response times will be identified. This service will be used to benchmark all interceptor-based functionality. Initially, the service will receive a control set of requests without any interceptors added to it, to provide a performance baseline. Subsequently, each interceptor will be added and tested individually using the same control set of requests. It is expected that longer response times will result from incorporating interceptor functionality. The results can be used to evaluate if the associated design decisions allow for the system to maintain a real-time response.

To fully consider edge use cases, where every passenger on board the SAAII may want to make use of the system at the same time, the final test in this experiment makes 145 requests to the system at a single point in time. This tests if the system can handle the maximum expected traffic on the SAAII.

### 7.2.2 Forced Failure Experiment

The forced failure experiment describes a controlled environment in which potential failures in the system are forcibly invoked. These failures should be handled by the system and, as such, the system's observed response is compared to the expected response. This experiment serves to verify that the system is robust against internal failures and that services are truly isolated – insomuch that their failures do not induce failure elsewhere within the system. The test procedures for this experiment are presented in Appendix D.2, with the related benchmark data documented in Appendix E and discussed in Section 7.3.

This experiment invokes failures at multiple points in the system. The first point that is tested is in the system's response to failure in a singular service. For this test, a service, running locally, is pulled offline during a call and the system's reaction to this is observed. The same test is done for a remote service. This failure should not propagate beyond the failing service to ensure service independence and isolation. The second test point is in the autonomy of system recovery, this test entails modifying a service to force a failure only once a call has been made to the service. By forcing a failure of this type, where a fatal error will be thrown, the autonomous recovery of services can be evaluated. Again, this test will be performed on both local and remote services. Through testing the recovery of services, the requirement for independent service deployment is tested too. Finally, handling and recovery from network failure are tested. In order to provide a robust distributed system, the system should handle network failures and should recover once the network is re-established. To ensure the persistence of metrics to aid in diagnosis, the performance metrics should persist service failures without being overwritten when the services are re-launched. This ensures that a complete history of system behaviour is maintained even when failures occur.

### 7.2.3 Security Experiment

The security experiment serves to validate all security considerations in the architecture, verifying that the security requirements identified in Section 3.3 are satisfied in this implementation. The procedure for this experiment is documented in Appendix D.3, with results discussed in Section 7.4.

For the first test in this experiment, the system is reconfigured to provide a specified user with the option of requesting information from a service that they do not have access to. Thereby the first layer of security, authentication at the gateway, is maintained, but represents a misconfiguration of the system. This tests whether authorising calls in the service is an effective design decision, providing multilayer security. An additional test on the suitability of authorisation in the service is to attempt service access without the request being routed through the gateway. This approach fails to present an access token and it is again expected that the request will be rejected. The explicit considerations against a DOS attack necessitate launching a DOS attack while ensuring that other users are granted access to the system without degrading service delivery. This is the final test of this experiment and is performed by overloading a service with requests from a single user.

### 7.2.4 Reconfigurability Experiment

The reconfigurability experiment serves to evaluate the process required to add a new user-facing service to a deployed system. The design proposed in this research strives to minimise the system knowledge required for contributors. It is fitting then that contributing to the system should not be a difficult endeavour. This experiment serves to test the modularity of the architecture, by evaluating the modularity requirements (NF0.0, NF0.1). The test procedures are presented in Appendix D.4, with Listing 9 to Listing 14 containing the code snippets where changes are required.

This experiment requires that the system be running as it would be in a deployment environment. From this stable state, the process required to add an existing service to the system is followed, documenting the required changes to source code. Once the service is ready to be run within this system, the service needs to be integrated with the deployed system. To do so, the gateway needs to be updated (through its configuration file). This update necessitates that the gateway be pulled offline momentarily with the updated instance relaunching. A successful outcome would result if the system performed consistently during this downtime as it would in standard operation. This test verifies that individual service deployment is successfully considered and that the architecture has been designed to support maintainability.

## 7.3 Results

This section presents the results of the evaluation. All raw data has been included in Appendix E, with the data relevant to the evaluation presented here. Quantitative results are presented in Section 7.3.1, with the qualitative results presented in Section 7.3.2.

### 7.3.1 Quantitative Metrics

#### 7.3.1.1 System Stability Test

Using the measured service request latencies presented in Table 10 to Table 14, the average communication times can be calculated. Knowing the call list of each aggregator, one can subtract the request latency of each service invoked by that aggregator to determine the *net communication time* (time for messages to be sent and received, without considering server processing time) for a call chain. This can then be divided by the number of services invoked by the aggregator to obtain the average *communication time* between a client and a server. The *communication time* is the time required for a request to be sent to, and for a response to be sent from, a server, excluding the time required for any service logic to be performed. This has been performed on the metrics recorded in Appendix E by matching request ID, with the results are presented in Figure 7, below:

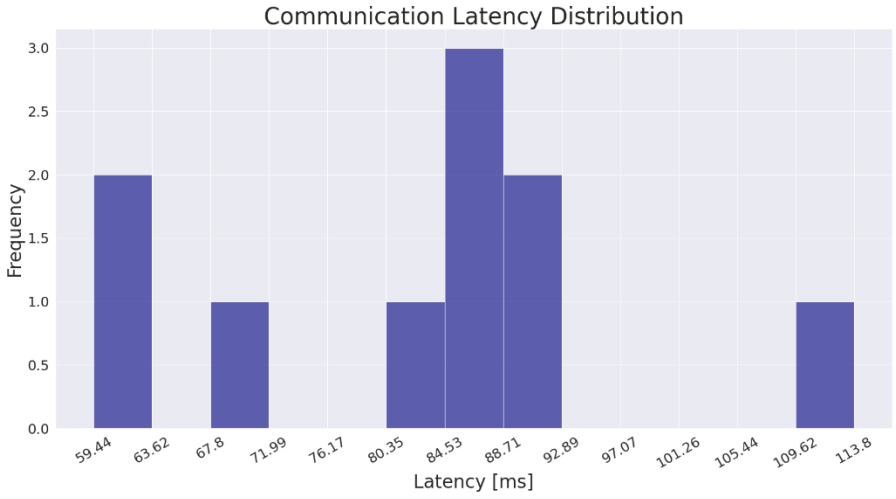



**Figure 7: Communication latency histogram**

From the data presented in Figure 7, the average communication time is calculated to be approximately 80 ms, with the slowest communication times nearing 114 ms and the fastest falling within 60 ms.

The gateway, being a standard component containing no domain logic, was benchmarked during the standard operations stability test. The results are presented in Table 18 and Figure 8.

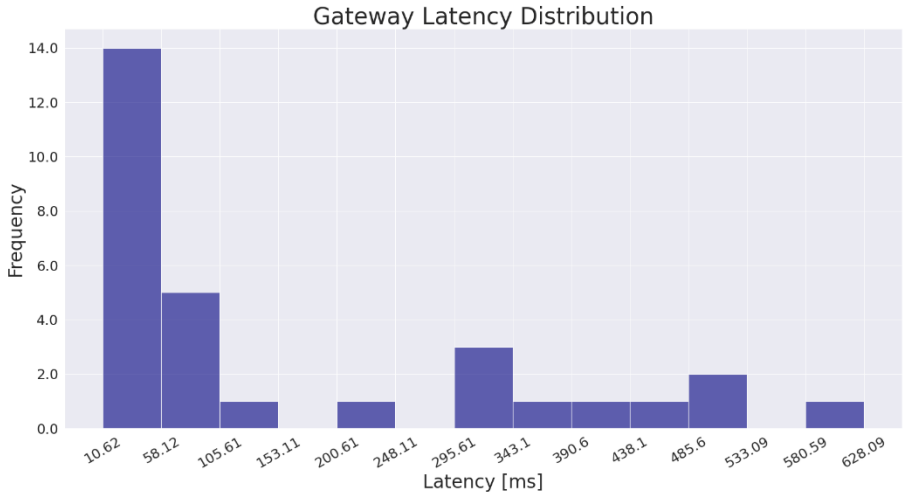



**Figure 8: Gateway latency histogram**

The average response time of the gateway, while performing protocol translation, request routing, and system-wide rate-limiting, was calculated to be approximately 160 ms, with the majority of the recorded interactions taking place within 100 ms. The slowest response received from the gateway took 628 ms, with the best performance providing a response in close to 10 ms.

Figure 10 and Figure 9 show the hardware requirements of each service used in this case study. These graphs present the information provided in Table 21. In both figures, the requirement for a dormant service is presented with the requirement for an active service – note that the active requirements are stacked on top of the dormant requirements in these graphs, not alongside them. In Figure 10, it can be seen that the two services offering simulation functionality, the propeller monitor service and the power-train service, place notably greater requirements on the processer. Here it can be seen that both of these services require approximately 50% of the CPU when processing requests. Note that for all services, the ‘dormant’ CPU requirements are nearly negligible, as can be expected.

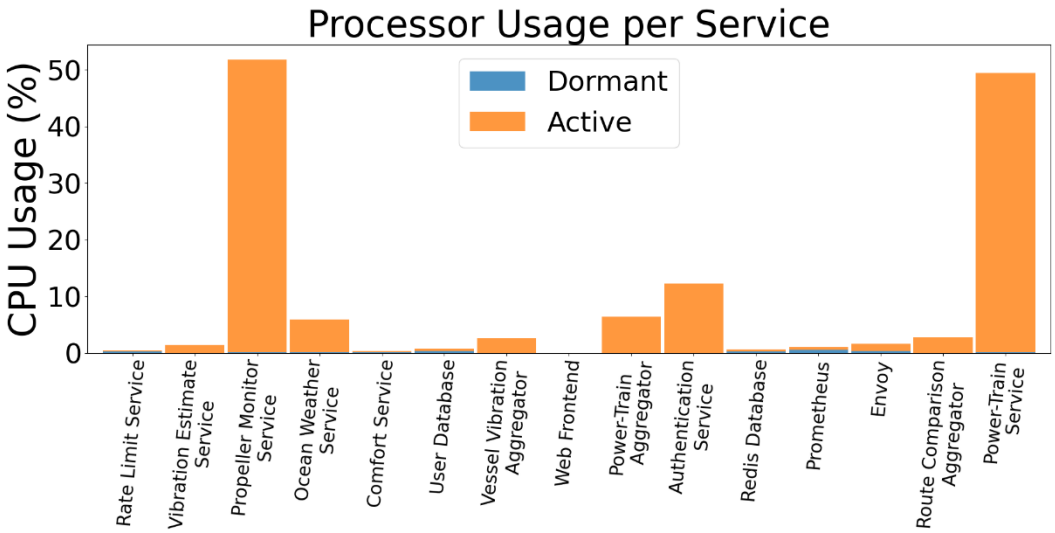



**Figure 10: CPU usage of case study services**

It can be observed in Figure 9 that the user database and power-train service require the most memory, with the user database requiring nearly 500MB of memory when both dormant and active.

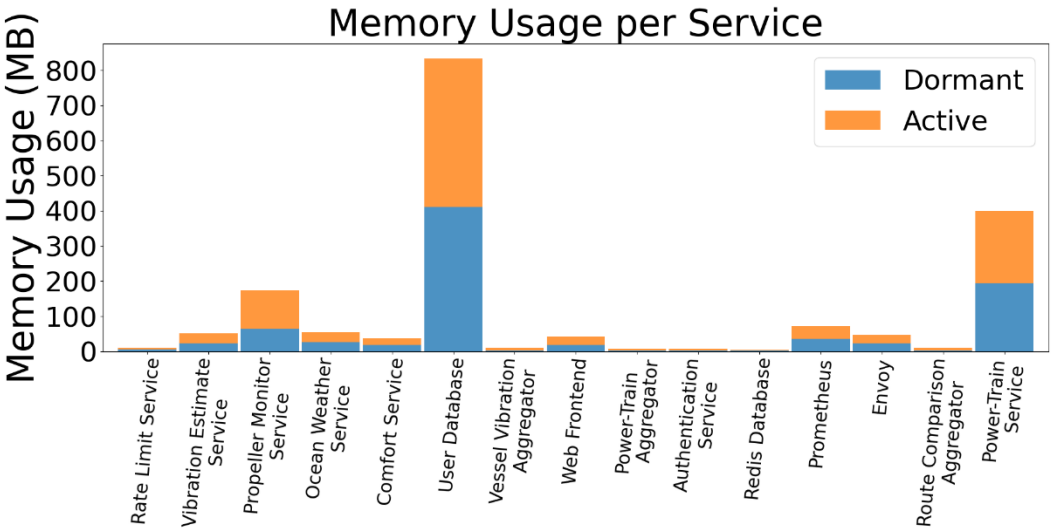



**Figure 9: Memory usage of case study services**

In Figure 11 the times required to build the container for the case study services are presented – this is based on the data documented in Table 22. In this figure, the initial build time for each service container is provided alongside the time required to build the service container once all necessary dependencies have been cached. These dependencies includes container images and packages/libraries required by

the service. In this figure, it is evident that the power-train service and web frontend boast the lengthiest build times, with the power-train service needing almost 7 minutes to complete the initial build. One can also observe that, with all necessary dependencies cached, container build times dropped to approximately 6 seconds across all services except for the Ocean Weather Service, which took 11 seconds in this case.

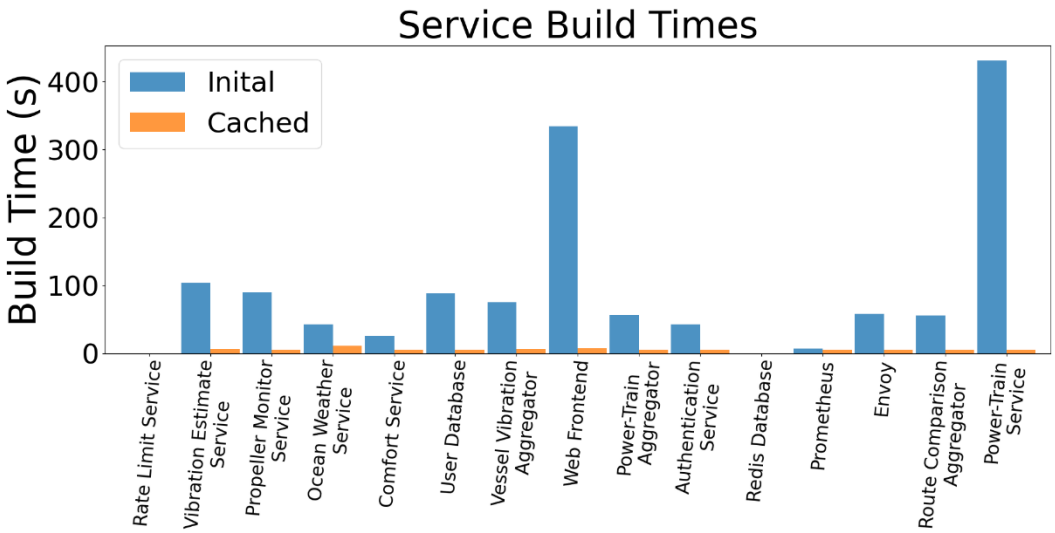



**Figure 11: Container build times for case study services**

#### 7.3.1.2 Interceptor Benchmark Test

The interceptor benchmarks, recorded in Table 19, were averaged as each recorded transaction was unique, and were subsequently compared. The average response times with various interceptor configurations were recorded and can be observed in Figure 12.

From the response times presented in Figure 12, the additional latency added to a request by each interceptor is calculated and presented in Table 7.

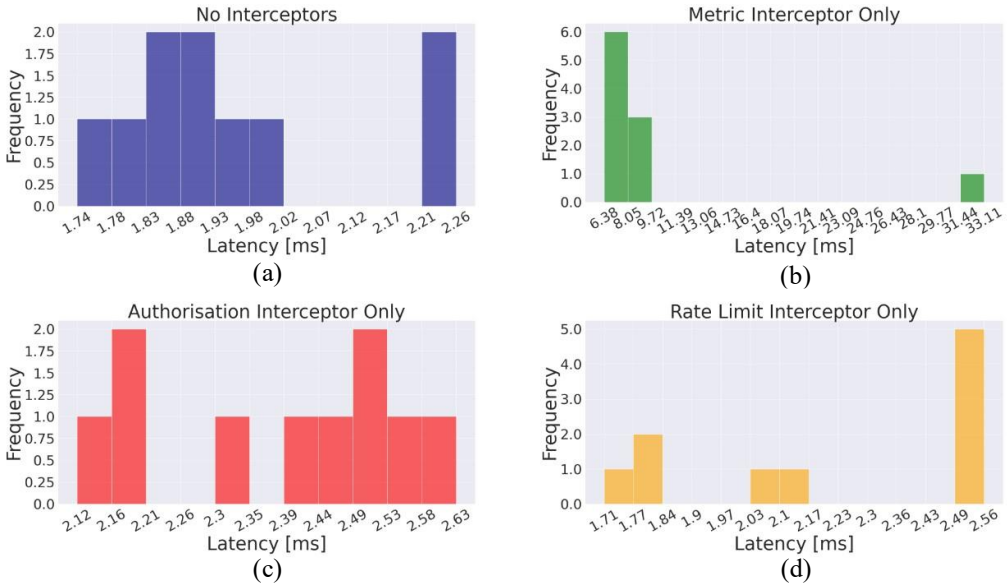



**Figure 12: Service latency with (a) no interceptors, (b) metric interceptor only, (c) authorisation interceptor only, and (d) rate limit interceptor only**

**Table 7: Interceptor latencies**

| Interceptor | Average request latency [ms] | Additional request latency [ms] | Percentage increase on baseline [%] |
|---|---|---|---|
| None | 1.96 | 0 | 0 |
| Metric interceptor | 9.83 | 7.87 | 402.53 |
| Authorisation interceptor | 2.39 | 0.43 | 21.94 |
| Rate limit interceptor | 2.22 | 0.26 | 13.26 |

### 7.3.1.3 Rate Limit Test

The rate limit rejection results, recorded in Table 20, show successful results when implementing service-level rate-limiting in Golang and system-wide rate-limiting in Envoy. Here, the service and the gateway rejected all calls once the limits had been reached. Service-level rate-limiting in Python, however, provided varying results depending on the service configuration. Consistent results were obtained where the server processing requests was constrained to using 20 or fewer workers (threads) to handle requests, and the server set its rate limit to enable fewer than 16 concurrent requests. It can be observed that inconsistent results are provided for any configuration where there are: more than 20 threads serving requests, the rate limit is set to more than 16, or the rate limit is set to be greater than the number of threads.

### 7.3.2 Qualitative

#### 7.3.2.1 System Stability Test

For the duration of this test, the system remained online and accessible from all devices on the network. During this time, all authorised requests were successfully processed except for those that were sent when the host machine had its internet connection intentionally revoked. In this case, the requests were still sent and processed, but the ocean weather service recorded and returned an error when trying to access the Stormglass API as the service was unable to connect to the Stormglass servers.

#### 7.3.2.2 Request Limit Test

The request limit test saw the system being sent 145 concurrent requests. With the gateway rate limit and the service rate limits set to allow for this many requests, all requests were processed. During this test, user functionality was maintained, with service access being provided only to those users who were authorised. It is worth noting, however, that Python servers did not process all requests concurrently. gRPC servers that are implemented in Python are configured using workers (threads), with each worker being assigned a single request. It was observed that Python was able to process a maximum of 20 requests concurrently, with subsequent requests being queued. This resulted in all requests being served, but with them being processed in batches of 20 requests at a time.

#### 7.3.2.3 Fault Isolation Test

For the fault isolation test, none of the invoked failures originating from a single service were observed to invoke faults anywhere else in the system. When a fault occurred, all other components in the system were still capable of processing requests as intended. This held true for both locally-hosted and remote services.

#### 7.3.2.4 Service Recovery Test

In the service recovery test, all services were observed to relaunch themselves without intervention. This held true for both local and remote connections. When a Golang client was used to invoke a failing service, its retry interceptor successfully picked up failed calls where the service ran on the same machine. However, it failed to re-establish a connection to a recovered remote service, returning “code = Unavailable desc = transport is closing”. This is a known bug in Golang when programs are presented with dropped connections. As such, the test was repeated with a Python client. In this case, the same error was logged by the client, with the retry interceptor successfully re-establishing the connection and completing a successful call chain for both local and remote connections.

#### 7.3.2.5 Network Failure Test

Observing the server logs, the network was unplugged once the remote server received a call and was only re-connected once the logs showed that the server was attempting to return a response. The server retried returning the response such that the response was successfully returned once the remote machine's network connection was re-established. In this interaction, both the server and client maintained their connection through the network failure, allowing for the same connection to be re-used once the physical connection was regained.

#### 7.3.2.6 Unauthorised Access Test

Unauthorised requests that were validly sent to servers were rejected as expected. Validly sending requests dictates that the requests are routed through the gateway. In all cases, the server logs documented a "Failed to authorise: the user does not have permission to access the requested service" internally, returning an "rpc error: code=PermissionDenied desc=user does not have permission to access this RPC" message.

#### 7.3.2.7 Gateway Bypass Test

With requests being sent to servers in an invalid fashion, the client fails to provide the server with an access token in the request. In all cases, these invalid requests were rejected as anticipated. Server logs documented a "Failed to authorise: JWT has not been provided" internally, returning an "rpc error: code=PermissionDenied desc=authentication token has not been provided" message in all cases.

#### 7.3.2.8 Service Development Test

The process documented in Appendix D.4 describes changes to at least eighteen lines of code to convert service logic into a service capable of being run in this system. In addition to these changes, two build commands need to be called. These eighteen lines of code span two separate files, with the majority of changes attributed to customised service call semantics. Due to the nature of these semantics, there is no possible way to further automate the creation or reconfiguration of services. These semantic changes necessitate changes in the service file. Generic configuration variables, however, have been implemented in a manner such that they can all be changed through the configuration file.

It is worth noting here that this thesis does not account for any line changes to the frontend to display the information of new services. This was considered out of scope as the line changes required for this depend heavily on the implementation of the frontend (web vs. embedded) and the design of the frontend itself. Additionally, the architecture proposed in this thesis does not specify how to implement the frontend. Keep in mind, though, that the addition of any new service would require that the frontend be updated to display the new information on offer. Here, the

frontend should not need updating to offer the new service, as this is done automatically when the user logs in and their access permissions are provided through the gateway.

#### 7.3.2.9 Service Integration Test

The process of updating the system to include a newly-developed service is described in Appendix D.4. This procedure requires five-line changes in a single file, as well as one build command. These line changes are purely configuration and are required for the gateway to know where to route new requests. The process of updating the gateway during deployment requires a single command, which pulls the old container offline and replaces it with the new one. This process executes in the order of milliseconds. Where there is an existing connection to the gateway when it is pulled offline, the client receives an "ERR: CONNECTION REFUSED" error as the connection to the gateway is immediately killed. The server request continues to be processed, yielding an error only when the response is returned. In order to handle this dropped request, the client's retry interceptor re-establishes a connection to the new gateway instance and retries the call through that connection. This behaviour mirrors that which was observed during the fault isolation test.

## 7.4 Discussion

This section presents a discussion of the results, recorded in Section 7.3 and Appendix E, in the context of the case study and thesis objectives. The results are evaluated in terms of the satisfaction of the requirements stipulated in Section 3.3. As is mentioned above, the ISO 25010 standard is consulted in structuring this evaluation; because this standard is intended for evaluating software products, not designs specifically, categories that relate to implementation such as supportability, usability, and portability, have been omitted here to maintain relevance.

### 7.4.1 Functional Suitability

Functional suitability considers the extent to which the system is able to provide the functionality described by the objectives and requirements. The emphasis here is on qualitative achievement and as such, the requirements associated with the qualitative results presented in Section 7.3.2 are mostly discussed in this section.

#### 7.4.1.1 Functional Completeness

The system implemented in the case study successfully provides a facility in which to run engineering models and information collectors. The case study sufficiently demonstrates this through running both data-driven (power-train service and vibration estimate service) and mathematical models (propeller monitor service), as well as the inclusion of an environmental information collector (ocean weather service). From the service types used in this case study, it can be seen that the proposed architecture is capable of running modern engineering models (regardless

of whether they are mathematically derived or built off repositories of data). This allows models of standard components, such as motors or generators, to be deployed as services in this system, as well as models that would be specific to each vessel, such as hull or propeller models. The case study additionally showcased how the system can coordinate information between information sources and these models for enhanced insights, too. Here, a standard set of information services could be offered, such as those for open ocean weather, with custom information services being developed based on specialised use cases. An example of this would be a custom information service to provide ice information, which could use satellite imagery to provide information about ice fields, or it could have a more specialised implementation extracting this information from a drone or camera setup installed on the vessel instead. The architecture proposed here does not place any constraints on this, offering a system that is flexible and customisable for each vessel and voyage it is used on.

The addition of an aggregation layer considers the objective to coordinate information between models and information collectors. The three aggregators were effective in displaying this, enabling each microservice to focus solely on capturing domain knowledge, with the aggregators providing additional value through the fusion of information.

The final objective specified that the architecture be designed such that it supports contributions from various vendors, creating a modular and adaptable system. The selection of a microservices-inspired architecture was the first step towards achieving this. The discrete nature of microservices inherently promotes a modular system. Additionally, the abstraction provided by the aggregation layer allows for microservices to be oriented around domain knowledge – as is advocated for in the domain-driven design ideology (Evans, 2003). This makes contribution easier as developers need only focus on providing domain-specific information, not on how the information will be used in the system. To consider information use, the aggregators are oriented around business logic, with their development focussing on using domain-specific services to derive value for end-users. This combination produces a system that can be easily reconfigured for changing stakeholder needs and vessel expansions. The case study demonstrates this through the use of services that were developed by domain experts prior to this research, which are deployed within the system to provide enhanced value to users. This allows the system to grow along with the vessel and her needs, with additional services and information being made available as new requirements are formed and new insights are obtained.

The architecture has also been designed such that the value that a single service can provide is not limited to that use case alone. It was demonstrated how the information provided by services could be used to provide value greater than just what is offered by that service. In the case study this was seen where the power-train service was capable of providing power and cost estimates, but could provide even greater value when its information was used to derive information from the

vessel vibration service too – providing insights beyond what one could with the power-train service alone.

Through the case study, the selection of well-supported and well-documented open-source technologies, such as gRPC, encourages contribution as interfacing complexity is minimised. This was demonstrated in the Reconfigurability Experiment, where it was shown how the interfacing procedure does not necessitate a depth of knowledge into the system or the technologies employed by it. Apart from the service logic, the only changes required from the boilerplate code provided is in configuration variables. These variables relate to standard computing concepts that any developer capable of writing code should understand. Additionally, the semantics required to create a service definition are human-readable and logical, with comprehensive support provided through the greater software community.

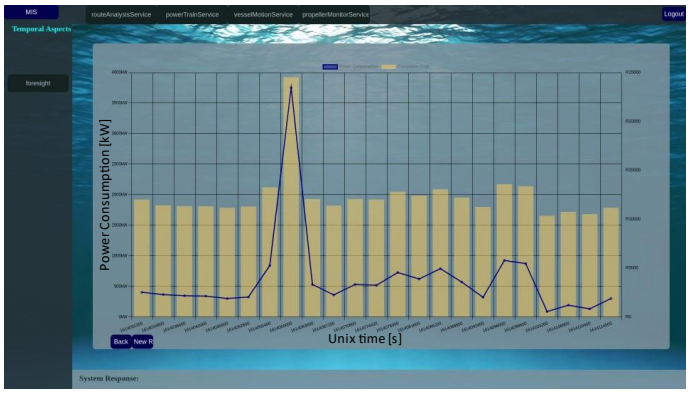
#### 7.4.1.2 Functional Correctness

Functional correctness is the degree to which a system provides the correct results, considering the precision required by the application. This refers to mathematical precision when transferring data. Considering this work, it can be thought of as ensuring that the essence of the data is not lost through aggregation.

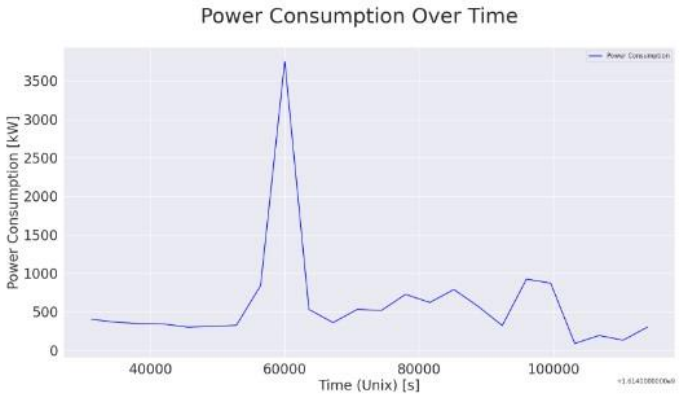
Figure 13 and Figure 14 show the data displayed to the user after aggregation and transfer through the system, alongside the data plotted by the service where it originated. The timestamps and positions provided by the user for these graphs were those of the SAAII on the 23rd of January 2021, taken at twelve hour-long intervals. This day fell on the return leg of the vessel from her annual Antarctic relief voyage; it was a notably stormy day, challenging the vessel with large waves, low visibility, and high winds. These conditions forced the vessel to drop her speed to approximately 5 knots, hence the low power requirements.

Figure 13 shows the frontend plot against the service-generated plot for power consumption. Similarly, Figure 14 shows the frontend plot against the service-generated plot for bridge accelerations. From these plots, it is clear that the essence of the data is not lost in transit through the system. In both cases, the plot offered to users perfectly represents that which is generated by the root service. This sufficiently proves that the aggregation of data does not affect the information delivered to users. It is worth noting that the service interfaces were designed to use appropriate data types to avoid potential aggregation issues. Figure 13 and Figure

14 verify that the facilities required to support the high-precision data generated by engineering models are available.



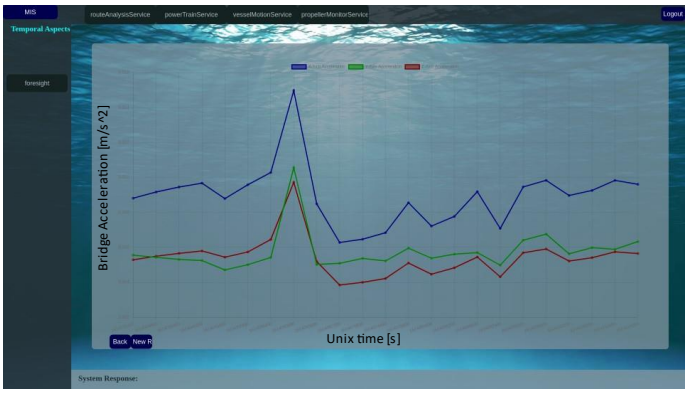

(a)



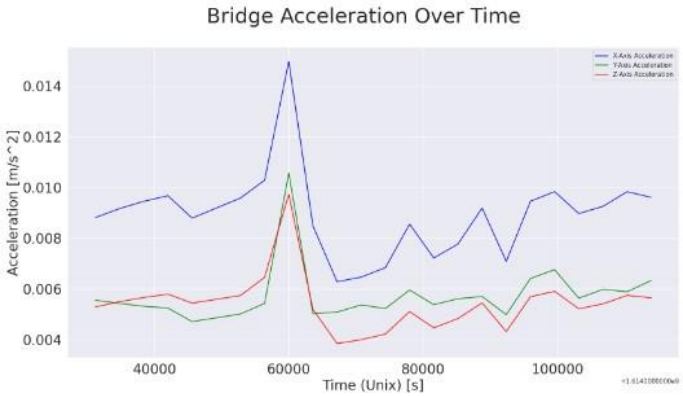

(b)

**Figure 13: Plot comparison for power consumption with (a) showing the graph presented after transport and (b) showing the graph generated in the power-train service**

In Figure 13, the power plot is indicated by the blue line, with the left hand side axis indicating the values. The yellow bars show the incremental cost for that portion of the route, with the right hand side axis indicating the cost. This overlay of data gives insight into which portions of a route will be the most costly and support operators in reducing costs by altering specific parts of the route. In Figure 14, the human-weighted RMS vibration on the SAAII bridge are plotted over time. The blue lines represent the x-axis acceleration, the green represents the y-axis, and the red represents the z-axis. The y-axis of the graph indicates the RMS acceleration, in $m/s^2$, with the x-axis showing the unix timestamps [s].




(a)




(b)

**Figure 14: Plot comparison for bridge acceleration with (a) showing the graph presented after transport and (b) showing the graph generated in the vibration estimate service**

### 7.4.1.3 Functional Appropriateness

The need to enable individual service development (NF0.0) is ensured by following a microservices approach to digital services. Through the inclusion of the aggregation layer, services can be developed in complete isolation of the greater system as coordination with other services within the system is not required. This was tested through the service development test where a new service was developed from existing domain-logic and configured to run in the system while it was online.

The requirement to enable individual service deployment and removal (NF0.1) shares similar merit and was tested in the service integration and fault isolation tests, respectively. During these tests, it was found that services could indeed be deployed and removed independently of the greater system. Launching a service (such that it exposes itself on the host machine and within the local network) could be done without having any effect on other services in the system. The addition of new services, however, required the gateway to be relaunched which, while not

affecting any of the existing services, resulted in current requests being cancelled. In order to maintain autonomous recovery, this necessitates that the user-facing client boast retry logic to retry the failed calls.

Note that the system was not required to enable dynamic service addition as it is highly unlikely for the needs of the vessel to change during a given voyage. The system was instead designed to be configured for each voyage before the time (in part due to unreliable internet required for additional development to take place on board).

The requirement for individual service deployment and removal is further guaranteed by specifying service deployment in containers. These containers can be pulled offline and relaunched in a matter of microseconds without directly affecting any other services in the system. The failure isolation test verified this whereby services were individually pulled offline and removed from the system. Each of the remaining services were tested, ensuring normal operation after the removal. Services were subsequently re-launched through the service recovery test, displaying successful behaviour for individual service deployment.

This capability to independently deploy and remove services enables the system to be updated while online. While this was not a requirement, there are cases where it may be beneficial. If, for whatever reason, a service was observed to have a bug while the vessel was at sea, this service can be pulled offline and replaced with a patched version without affecting the rest of the system. It was also observed how this could be done in approximately 6 seconds. Additionally, should an opportunity for new information be identified during a voyage, a service capable of offering this information could be developed and deployed on that voyage, with a worst-case downtime of 12 seconds, and without affecting any of the other services deployed within the system. Additionally, taking the aforementioned approach to deployment with containers, these services (or more suitably, their containers) can be built in the absence of internet - as is likely the case on a maritime vessel.

The need to keep track of users and permissions (NF1.0) was considered through the specification of an authentication service and user database. This was satisfied in the case study by adding four separate users, with different permissions, and dynamically tailoring the frontend to them depending on their respective permissions. The login functionality offered by the authentication service further provided support for access control (NF3.0) – discussed further in Section 7.4.5.1.

The request interpretation and routing requirement (NF2.0) is satisfied at two levels in the architecture. Firstly, requests are routed by the gateway component. This routing ensures that users do not require knowledge of backend address spaces in order to derive value from the system. This abstraction additionally aids in hiding backend complexity from users (NF6.1). During all tests, request routing through the gateway proved to be a robust solution. Additionally, the gateway negated the need for advanced service discovery, all the while without requiring the users to

know of service locations. Reconfigurability of the gateway proved a simple task, requiring minimal adjustments (in the order of five-line changes of code) to reflect changes to backend configurations.

The second point of consideration is in the aggregator component. Requests for information (implicitly implemented as an information aggregate) are routed to the relevant aggregator by the gateway, with the aggregator routing and coordinating subsequent requests to the microservices required to provide the information for this aggregate. The aggregation layer proved to simplify reconfiguration as adjustments to the gateway were limited to a single service (or aggregator). In the absence of the aggregator, custom logic would need to be placed in the gateway, requiring a more complex piece of software than the proxy used in this case study. Additionally, aggregation logic requires more complex changes, which would increase the chance of a relaunched gateway instance containing errors and increasing downtime. Finally, if aggregation was performed in the gateway, users would be interacting directly with aggregation logic; with this logic sitting so close to the user it is easier to unintentionally expose program/aggregation logic to them.

Access control and authentication needs (NF3.0) are considered through the specification of the authorisation interceptor on each service. This, coupled with the access token logic described for the authentication service, ensures that users are authorised before requests are processed at all points within the system. The security experiment proved this approach to be successful in providing defence-in-depth, ensuring that information remains protected even in the cases where a user may bypass system-level security mechanisms. For authorisation to be performed at the service, the use of interceptor middleware proved to be a valid design decision. In the tests, all invalid requests were rejected with the services exhibiting the anticipated behaviour perfectly. The performance implications of adding authorisation functionality to each service are discussed in Section 7.4.2.1.

Data integrity (NF3.1) was not tested in this thesis but is considered by specifying an RPC framework that supports message encryption for internal integrity, and a gateway that provides message encryption externally. In the case study, TLS encryption was supported by all components used in the system, with gRPC encrypting messages internally (for service-service invocation) and Envoy, being the only external interface, encrypting communication with the browser. Enabling the use of message encryption both internally and externally satisfies the need to guarantee data integrity.

The capability of this system to integrate with legacy systems and consider future systems (NF4.0 and NF4.1, respectively) could not be explicitly tested in this case study. However, the modularity of this system ensures that it can adapt to changing vessel configurations and stakeholder needs where required. Additionally, the reconfigurability experiment showed that, while an unlikely and unintended use case, the system can be updated during deployment to include new or updated services. Where services are updated and not added, other requests being processed

by the system would remain unaffected as the gateway would not require reconfiguration in this case. Considering digital twins, specifically, it is reasonable to assume that a digital twin would offer multiple interfaces to ensure their compatibility with common systems. With RPC being a prominent communication mechanism in the field, one of these interfaces would likely be RPC. Alternatively, the relaxed roles of an RPC implementation allow one to configure a translation client or proxy that converts requests and responses between proprietary interfaces and RPC where required. The nature of RPCs in this context is that they support services running on remote machines. This is verified in the case study by running the propeller monitor and comfort services on a remote machine, thus satisfying the requirement for distributed services (NF4.2). For a maritime vessel, this allows one to deploy services as close to their data as possible. A service that encapsulates a model may be deployed on the same machine responsible for collecting and storing the data that the model acts on. This allows for the service to remain synchronised with the data, and allows the service to act on the data where required even when there are network issues present. As microservices are stateless by nature, any ‘state’ is maintained by the data that belongs to that services. By deploying the service on the same hardware as its data, service ‘state’ is maintained while keeping the service itself stateless; keeping a service stateless in this manner greatly reduces complexity making the services more manageable for contributors.

The service and network recovery tests displayed that the system was easily and autonomously recoverable. Recovery times could not be reliably recorded; however, all services relaunched themselves within the first retry bracket (i.e. within 100ms). Launching the system entails the parallel launching of each service and, similarly, system recovery entails the parallel recovery of individual services. This satisfies the need for recoverability (NF5.0), which is further discussed in Section 7.4.4.4.

In the case study, the system enabled access from any device on the local network that had access to a browser. This includes laptops and mobile phones. The BFF pattern that was employed describes a customised gateway for different device/user types. This was deemed unnecessary to test through the case study and as such all web traffic was handled by a single gateway. There is no constraint on this, however, and a mobile application could just as easily route its requests to a gateway optimised for mobile traffic. The specification of a gateway component providing customised translation and routing satisfies the need for access from multiple devices (NF6.0).

The requirement for logical fault-tracing and debugging support (NF7.0) is satisfied through the addition of the metric interceptors and service logging. During the forced-failure test, the root cause of the failure could be easily identified by the messages provided through the user interface, as well as through any service involved in the failing call chain.

### 7.4.2 Performance Efficiency

Performance efficiency is the category concerned with resource usage and system performance when the software is used under specified conditions. Considering the requirements presented in Chapter 3, this mostly refers to real-time requirements where it is specified that the system is to achieve soft real-time performance (i.e. platform functionality should not comprise a significant portion of the overall response times).

#### 7.4.2.1 Time Behaviour

All time behaviour results obtained refer to the need for soft real-time data processing (F1.0). The first aspect to be addressed here was the decision to use an aggregation component that sends out a request and waits for the necessary response before sending requests to subsequent services. This was an adaption to traditional microservice approaches where each microservice would be responsible for obtaining the information it requires. The decision to use the aggregated approach does not notably affect response times as, by the end of the call invocation, only one additional request and response pair will have occurred – that of the aggregator itself. With the same messages being sent to and from microservices, the same request latencies can be expected. Considering the communication latency reported in Section 7.3, with an average communication time (request + response latency) of 80ms, an additional call is not detrimental to achieving soft real-time considering the service latencies recorded in Table 10 to Table 17: Power train service latency.

On top of the specification of an (optional) aggregation layer, the inclusion of a gateway layer could potentially affect real-time operation as requests need to be processed by an additional component before reaching the intended service in the backend. The gateway recorded an average response of 160ms, twice that of the communication time for services. This time was required to perform system-wide rate limiting, protocol translation, and request routing. Again considering the context of this work, the addition of 160 ms is a reasonable trade-off to provide the additional security and functionality on offer through this gateway. The worst-case for the gateway of 628 ms is a notable outlier. This outlier could have a more significant effect on the system where the gateway took longer to process than the cumulative communication times of all down-the-line services. Considering that this was an outlier, the average response times (service processing and communication times), and that there is a fairly relaxed definition for real-time in this context, stakeholders should not notice a dip in performance. Thus the additional latency incurred through the inclusion of both the aggregation and gateway components is justified when considering the value of the functionality and abstraction provided by each.

Considering the service latencies recorded during the standard operations test, a discussion can be provided on system and service use. The service response times

showed a worst-case response of 2.5 s for the aggregators and 2 s for the ocean weather service. This suggests that the system is suitable for route planning, allowing for multiple route simulations to be made without limiting, or placing time-constraints on, the decision-making of stakeholders. Looking at the latencies of platform components (such as the gateway and interceptors), it is unlikely that this system be recommended for use in high-performance control applications and would be better kept as a decision-support tool. For a service such as the propeller monitor service, where real-time monitoring is offered, the system places minor constraints on deployment. In the original work of Nickerson (2021), real-time response is recorded for data bins as small as 2 seconds – where the algorithm is capable of processing all data before a new data bin is recorded. Running this algorithm as a service in this system reduces the real-time capabilities as, on average, an additional 160 ms needs to be accounted for the gateway processing, as well as an additional 80 ms for the service communication time. This does not have a detrimental impact on a ship, considering the rate at which decisions are made, but is worth noting. It is also worth noting that the results recorded both in this work and in the work of Nickerson (2021) do not consider data pre-processing times, reporting instead purely on service and algorithm processing times, respectively. The effect of pre-processing/converting data on real-time application needs to be considered before one can say that this system is truly capable of running real-time monitoring for such a service.

A decision was made in the architecture design to use interceptors for service-level rate-limiting, authorisation, and metric collection. Each of these interceptors were benchmarked in the

interceptor benchmark test to analyse their effect on the real-time performance of this system. The additional latency incurred by each is documented in Table 7. Of the three interceptors, the metric interceptor was observed to consume the most time. This is to be expected as this interceptor communicates with an external server in order to store the metrics. The authorisation interceptor was an order of 10 faster than the metric interceptor, with the rate limit interceptor executing in half of that time. On average, the cumulative time incurred by these interceptors was less than 9 ms. Considering that this comprises approximately 11% of the average communication time in a service, the interceptors are not considered to have a notable impact on the real-time operation of the system. Additionally, considering the fine-grained control each provide to services in the system, and the additional functionality, the approximately 9 ms of additional latency is seen as a worthwhile trade-off.

The build times presented in Figure 11 show that the initial build times can vary greatly between services. Here, it is clear that the power-train service and web frontend require the most time for their initial builds. These two services require the most, and largest, packages of all services used in the case study. The power-train service, for example, requires the TensorFlow package and all of its dependencies, which exceed 500MB in size. It can also be seen that all aggregators

require similar initial build times, far less than that observed for the power-train service. An important service to consider is the Envoy proxy, as for any new service addition, this will require an update and re-build. In this case, we expect an initial build to take 1 minute and any subsequent builds requiring 6 seconds. From the data, one can expect an initial build of up to 7 minutes for services requiring larger packages, and a minimum of ~30s for simpler services with fewer dependencies, such as the comfort service. With all necessary packages cached, though, we see all builds taking place in approximately 6 seconds, bar the Ocean Weather Service. This means that, should a bug in a service be identified while the system is deployed and the vessel is at sea, or a new use case/service need be identified, the patched/new service can be built and relaunched with anywhere between 6 and 12 seconds of downtime. Here, 6 seconds would be the case of updating an existing service, and 12 seconds would be the case of a new service addition where the additional 6 seconds accounts for the time required to relaunch an updated gateway (Envoy) instance to include the new service. Here, it is worthwhile noting that building a service with packages cached still uses the internet to perform checks – this could introduce issues considering the state of internet connectivity on an ocean-going vessel. To work around this, a Docker image can be build before the voyage to contain all packages required by services. This way, instead of having to check for packages online, the service containers can instead inherit the services from this prebuilt container. This container can then be cached so that no online checks are required.

Considering how long a vessel spends in port prior to a voyage (at least 2 weeks in the case of the SAAII), the worst case service build time of 431s is considered completely reasonable. Additionally, considering the processing times of services, the cached build time of 6s is considered acceptable, and the worst case service downtime of 12s is thought not to have a notable effect on how the system is used.

#### 7.4.2.2 Capacity

The request limit test resulted in the system processing all 145 requests without error. However, not all 145 requests were processed concurrently. As was highlighted in Section 7.3, Python servers only had the capacity to process 20 requests at any given time. The Python implementation of gRPC uses the concurrent/futures module to process concurrent requests in the server using a thread pool. In the documentation of this module, it is noted that a limit of 32 workers (threads) is imposed by the module itself. Theoretically, the maximum number of threads that Python will be able to run on the host machine is described by the microchip architecture. Each core of the processor on the host machine used in the case study enables two threads, and with 8 cores present the machine should be able to run 16 concurrent Python processes. This means that 16 threads can be processed at a single instance. When there are more than 16 threads requiring processing, the machine multi-threads to rapidly change between which 16 threads are being processed at a given instance. It was observed during testing that with this multi-threading, the Python servers were capable of concurrently handling 20

requests with the full system deployed. Services written in other languages, such as Golang which is renowned for its concurrent performance, were capable of processing all requests concurrently. Additionally, the gateway was successful in concurrently routing all requests while maintaining its core functionality. Thus, it is shown that the system itself is capable of processing the maximum number of requests that can be expected on board the SAAII, satisfying the need for concurrent request handling (NF1.1). However, the importance of choosing an appropriate implementation language is highlighted. If the service being developed is expected to handle high loads of concurrent traffic, the service should be implemented in a language that can handle the expected concurrent traffic.

The requirement for supporting multiple user sessions (NF2.1) stimulates an interesting discussion. The decision to employ 'transparent' access tokens, implemented with JWTs in this case study, allows for a theoretically infinite number of user sessions. Transparent tokens - and the number of tokens issued - are not tracked by the backend system, and once issued are the sole responsibility of the user that they are issued to. As a result, no additional load is placed on the system by generating further access tokens or sessions. Considering this, an infinite number of tokens could be generated without impacting system performance as the mere existence of tokens, representing sessions, places no load on the system at all. The only performance degradation that may result from issuing this infinite number of tokens surfaces when these hypothetical users all request information from the system at the same time. In this case, the system would be heavily overloaded, mimicking a DOS attack. In order to enable multiple sessions then, the system should be able to handle concurrent requests as there is no actual limit on the sessions. Rather, the limit is on processing requests from multiple sessions. Therefore, this implementation was shown to successfully support 145 user sessions as that was the extent to which the capacity of the architecture was tested. Thus, through successfully supporting concurrent requests (NF1.1), the requirement for supporting concurrent sessions (NF2.1) is satisfied.

Considering the hardware requirements presented in Figure 10 and Figure 9, observations can be made on the resource requirements of different service types. It is clear that considering both processer and memory usage, services offering simulation functionality such as the power-train and propeller monitor service require more resources that average. This makes sense as the logic performed by such services perform computationally-expensive math operations and store more intermediate data than other services. These two services are shown to use up to 50% of the CPU capacity on the test machine when performing their logic. It is likely then, that if simulation services are deployed on a machine with similar CPU specs, that they would be deployed in isolation (alongside no further services) and would rate limit themselves to process at most 2 concurrent requests.

It can also be seen that aggregator services do not place particularly strict requirements on the host machine. Considering this, it may be appropriate to deploy all aggregators on a single machine and with platform services (gateway,

authentication, rate limit, etc.) being deployed on their own machine. Services that are less computationally expensive, such as the information collectors, could be deployed on a single machine, with simulation services deployed on dedicated machines or on the machines responsible for collecting their data where that is appropriate. This approach provides platform services and aggregators with the resources required to handle the scale of requests expected on a maritime vessel without unnecessarily under-utilising machines. Additionally, it provides a guarantee that where a service is offered based on specific, recorded data, the service maintains its state during network interruptions by existing on the same machine as the data providing its state (as the service itself is stateless, but the data it is associated with is not).

### 7.4.3 Compatibility

Compatibility considers the ability of the system to exist and interoperate with other systems, both on the same machine and remotely. Co-existence is considered an implementation detail and has been omitted from the evaluation. However, it is noted that while running the suite of tests, other software such as Google Chrome and Microsoft Word were used with no noticeable performance degradation in either. In this work, compatibility refers to the system's ability to communicate information to and from external systems.

#### 7.4.3.1 Interoperability

This architecture was designed to be interoperable with digital twins where they offer services, and by nature, any other service offering that caters to RPC clients. Where RPC is not offered, an RPC server can be created to act as a translation client for whatever communication mechanism is offered. Although this may not result in the most elegant solution, the relaxed client/server roles offered by RPC allow for it. Essentially, this system can then retrieve information from any source offering it. This capability enables this system to interoperate with any existing system as a client, granted that the system makes provision for interoperability itself.

The case study demonstrated the language-agnostic nature of RPC, where four mainstream programming languages were used in the implementation. This demonstrated the interoperability between services written in different languages, enabling experts to use whichever language best suits their service in its implementation. This was considered sufficient proof of interoperability considering that microservices (and gRPC itself) have already been accepted and proved by industry as offering this language interoperability. gRPC offers a further 7 languages at present, encompassing all major languages, to provide further choice to those wanting to interface.

### 7.4.4 Reliability

Reliability encompasses the ability of the system to maintain performance under specified, sub-optimal conditions. In the context of application on a maritime

vessel, the anticipated sub-optimal conditions include unreliable and occasionally absent internet connections. Additionally, due to the nature of the system, reliability also considers the system robustness to dropped local networks and individual component failures.

#### 7.4.4.1 Maturity

The system maintained a stable state throughout the standard operations test. During this time the system did not exhibit any unexpected behaviour and produced no errors other than those related to network connectivity, as is described in the section on availability.

For the remainder of the tests, the system maintained stability in terms of exhibiting the expected behaviour, apart from in two of the tests. During the rate limit test and service recovery tests, inconsistent behaviour was observed in certain interceptors. In the rate limit test, it was identified that the Python rate limit interceptor provided inconsistent results depending on the service configuration. The results are briefly discussed in Section 7.3.1.3. Based on the manner in which the results vary with configuration, it is hypothesised that the inconsistency can be attributed to how concurrent requests are dealt with in Python, discussed in the section on capacity, above. Consistent results were observed when the server limited requests to fewer than the machine's theoretical maximum of 16 concurrent requests, with imperfect but bounded deviations observed where the server was limited to between 16 requests and the observed maximum of 20 requests. Completely unreliable results were observed when the server limited connections to anything greater than the observed maximum number of concurrent requests. In the case where the limit was set to be greater than 20, the rate limit interceptor had no effect at all, allowing all traffic through. Conversely, the Golang implementation provided consistent results for all limits as the underlying implementation differs from that in Python. Python's gRPC implementation is built on top of the C core, whereas Golang's implementation is built on a Golang core. Using a native core, as is done in the C, Golang, and Java implementations of gRPC, provides finer-grained control over, and access to, the underlying connection. The unreliable results observed in the Python rate limiter are attributed to a lack of control over the underlying connection because of how the Python implementation of gRPC is only a wrapper on the C core.

In contrast, the opposite was observed during the service recovery test. When servers were pulled offline while handling calls, the server sends a termination message to the server to kill the underlying TCP connection. The system was designed for this, with the clients catching this error and attempting to re-establish the connection in order to complete the request. During the test, it was observed that the specific error raised by servers was not handled by the Golang interceptor as expected, with the client not retrying and immediately logging and returning an error up the call chain. After some research, it was discovered that this specific error is known to produce unexpected behaviour in certain versions of Golang and at the

time of testing it had not been patched. A Python client was written with a Python implementation of the retry interceptor in order to test the logic and handling of failed services. This approach produced satisfactory results with the Python client successfully re-establishing the connection with the server once the server had restarted itself. The root cause of the Golang bug remains unknown at the time of writing. However, the fact that both shortfalls were associated with interceptors working on the underlying connection may be indicative that processing connections through interceptors is not a stable approach.

The Open Systems Interconnection (OSI) Model of networking is a model used to provide a conceptual framework to describe the communication between a computer system using seven layers of abstraction. These layers serve to divide the flow of data in a communication system to reduce, or rather abstract, complexity. In this model, a higher layer (represented by a higher layer number) is served by, and relies on, the layer below it. In the model, layer 7, being the highest layer, describes the application layer; this is the interface responsible for communicating with host-based and user-facing applications. Maintaining relevance to this work, Layers 3 and 4 represent the network and transport layers, respectively. The former of these is responsible for forwarding (routing) packets of information from the data source to its destination in a network, whereas the latter is responsible for delivering the packets to the appropriate process on the host computer once it is there. When considering the software used in this work, layer 7 describes the RPC interface that developers can use to communicate between services – here, it is dealt with in code at the application layer. Layer 3 describes the functionality performed by the proxy to route a request from one machine to another over the network, and layer 4 is the functionality that sends this request to the correct service (process) once it has arrived at the correct machine. It is evident, then, that gRPC is a high-level piece of software, offering less control over the underlying connection than something like the proxy. This provides an explanation as to why rate limiting was shown to be effective when performed by Envoy, a proxy, but ineffective when performed by gRPC, an RPC middleware; similarly, this may explain why unreliable results were obtained when working with connections in the rate limit interceptors.

#### 7.4.4.2 Availability

Again referencing the standard operation test, the system remained available from all devices on the network as long as the local network remained online. Accessing the system through various devices on the network satisfies the requirement for multi-device access (NF6.0). During the test period, where internet connection was lost, the system was still available to all users on the network. As discussed in Section 7.3.2, services requiring internet access returned errors during this time, but the system itself was not reliant on internet and was thus still accessible. This satisfies the requirement that the system should be available offline (NF5.2).

#### 7.4.4.3 Fault Tolerance

Fault tolerance was tested through the fault-isolation and network-failure tests, with results discussed in Section 7.3.2. In both the tests, the system exhibited expected behaviour with the invoked fault having no effect on other components of the system. All components were capable of receiving and processing requests, with their automated tests passing following the failure of other services. In addition to service failure, network failures displayed the same results in the system. In this test, all local and remote services operated as expected. With network failure, the remote invocations failed, but this is to be expected as the communication medium was no longer present. The displayed robustness towards failures in the system is sufficient to satisfy the need for fault-tolerance (NF5.1).

This behaviour towards fault tolerance was to be expected, given that the architecture follows a microservices approach. A poor implementation could result in dependencies between components that would result in shared failure, however, it could be argued that an implementation of this nature no longer follows a microservices approach. Consequently, this hypothetical implementation can no longer be considered an implementation of the proposed architecture either.

#### 7.4.4.4 Recoverability

The recoverability of the system was evaluated through the service recovery and network failure tests. The service recovery test showcased the successful autonomous recovery of failed services and the re-establishment of any failed calls resulting from the failed service. In both the case of local and remote service failures, the services were successful in autonomously relaunching themselves. Additionally, the services relaunched themselves fast enough such that the client retry logic was able to re-establish the connections before timing out. For the re-establishment of calls, however, not all cases were successful, as is discussed in Section 7.4.4.1.

For network recovery, the system was successful in re-establishing connections after a network failure, re-using the existing connections where the connection had not yet timed out. In the cases of connection timeout, the client was successful in retrying the call by re-establishing a new connection every time. This is an expected outcome and successfully provides recovery and robustness to network failures.

These results prove that the decision to place retry logic in all clients is practical, but again suggests that implementing this through interceptors may not be a suitable approach as retry logic requires functionality acting at a connection level.

This automatic system recovery offers valuable functionality for application on a vessel when the system is deployed in a distributed manner. Considering the above discussions, where it is shown how computationally expensive services will likely be deployed on their own machines, and where services are recommended to be

deployed on the same machine responsible for collecting the data that they operate on, having services recover themselves and re-establish failed connections greatly improves availability. This allows for computationally-heavier services to be run on suitable machines, and in isolation, while providing a mechanism to handle sporadic network failures. Additionally, in the case where a service is deployed on the same hardware as its data storage, the service protects the data from having to expose itself on the network. This has the added benefit of abstracting responsibilities for access control to the service. In the case of a network failure, and where a client might cache failed requests to re-attempt once the connection has been re-established, there is a risk of the client essentially launching a DOS attack on the server by re-attempting all failed calls simultaneously. If the service were on a separate machine to the database, the service would act as the client in this hypothetical and would launch this attack on the database itself – risking the data collection and storage in doing so. By instead having the service act as the server, by running on the same machine as the data, the service would take the brunt of the attack and, if it fails, database integrity is maintained. As the data is where all state is maintained, this abstraction provides a lot of value to protecting measurements on the vessel. And with measurements generally being used for control systems on maritime vessels, ensuring its integrity is of utmost importance.

Considering this, partial satisfaction of the need for a recoverable system (NF5.0) is provided. In this partial satisfaction, failed services were all successful in autonomously relaunching themselves. However, the system was not successful in re-establishing connections or completing failed calls for all test cases.

### 7.4.5 Security

Security is evaluated through the considerations advocated for distributed systems. In the design of the proposed system, explicit decisions were made around security considerations. The suitability of these decisions is evaluated below.

#### 7.4.5.1 Confidentiality

In order to ensure that data is only accessible to those authorised to it, call authorisation was specified as a responsibility of each individual service. This was based on a recommendation by (Richardson, 2018). In this implementation, authorisation was specified through an authorisation interceptor. During the unauthorised and gateway bypass tests, all requests were sent with the knowledge that they were unauthorised requests. In all cases, these unauthorised requests were rejected on logical grounds. These results suggest that authorisation through interceptors is sufficient to provide fine-grained access control and confidentiality in the system. Considering this, the need for access control (NF3.0) is satisfied.

#### 7.4.5.2 Non-repudiation

A truthful understanding of events within the system can be obtained through the analysis of log files. The inclusion of logging in each service enhances

maintainability as is required by NF7.0. Program logs record interactions in services that describe the services' lifetime. This allows for interested parties to track events throughout the system. In the current design, events need to be tracked based on the time series and can only be tracked by following the call chain from its root. For example, one would need to start at the first service to receive a request and move to subsequent services based on the calls made by that service. The subsequent calls can only be identified based on the times that they were received. Additionally, log files live with the service and in the case of failure, die along with the service. A better approach to service logging, and thus non-repudiation would be to include a logging interceptor in the design, responsible for exporting service logs for each interaction to a central location. This would enable log files to persist the services that they are describing, as is done with the metric collection. Additionally, assigning an ID to call chains would enhance traceability, as system events could be grouped by ID instead of by following their time series. This would be especially beneficial where concurrent requests are handled and there may be multiple call chains being logged at the same time. IDs could be generated in the gateways such that all inbound traffic is seen as unique when analysing the logs.

#### 7.4.5.3 Accountability

The current logging system includes the user ultimately responsible for invoking a service in its logs. This provides accountability by allowing one to associate specific requests, or events resulting from requests, to specific users. This accountability additionally allows one to monitor the usage of the system based on user. Further, in the case of system misuse, and when combined with system authentication, this allows one to identify the responsible parties. This knowledge can enable stricter rate-limiting policies to be placed on that specific user for future requests.

#### 7.4.5.4 Authenticity

The specification of a user database and/or authentication service and the manner in which it is included in this system ensures that all users are known to the system. In order to gain access to any information through the system, an access token is required. This access token can only be obtained through consulting the authentication service. Additionally, this authentication service will only provide an access token to valid users of the system. By enforcing this, the system can guarantee the identity of any user capable of querying information from it. The gateway bypass test proved that should a user gain access to services without navigating this non-optional information flow, they would be unable to access information through that service. This is because the system, or more specifically the components within the system, are unable to verify that the client has been successfully authenticated by the greater system. Considering that the gateway was capable of routing all requests during the request limit test, and that the authentication service was implemented in Golang, specifying this single point of authentication does not act as a bottleneck for the system at the scale required. This

satisfies the needs to implement access control, support multiple requests, and route client requests (NF3.0, NF1.1 and NF2.0, respectively).

### 7.4.6 Maintainability

Maintainability is predominantly related to the implementation of a system, depending on the considerations made during its development. However, certain sub-categories are relevant and have been included considering that the proposed system should be maintainable in order to enable further expansion and reconfiguration. The following discussion considers the design decisions related to service independence and system modularity.

#### 7.4.6.1 Modularity

In terms of modularity, microservices are arguably the most suitable architecture style to follow. Specifying that each component should be discrete to provide a low-coupling of services results in services that can be added, removed, or updated without any effect to others. With interfaces defined, the logic serving a call is abstracted from the call interface itself. This makes updating logic easy as, unless new functionality is added, the interface remains the same. Where new functionality is added, the interface can be expanded such that the change remains backwards compatible. The reconfigurability experiment displayed how services can be developed in isolation of the system and can be near seamlessly added to it. Additionally, it shows how existing services could be “hot-swapped” and updated without any apparent downtime. The fault isolation test additionally displayed the discrete nature of these services by showcasing how services are indifferent to the operation of others within the system.

#### 7.4.6.2 Reusability

The reusability of services is again attributed to the discrete nature of microservices, with emphasis placed on their specialised focus on domain logic. The information flows implemented in the case study showcase the re-use of low-level services in serving different user needs. Additionally, these low-level services are reused among different aggregators in a manner such that they are used to compose additional value-adding services.

#### 7.4.6.3 Modifiability

In employing a common interface definition language (IDL) through a standardised RPC framework, interfaces (or APIs) are decoupled from the logic serving them. Through this, the logic of a service can be updated freely without requiring any changes elsewhere in the system. This means that clients can remain unchanged even if server logic is updated. This modifiability is what enables services to be independently updated within the system. Considering the implementation, every service includes a set of unit and integration tests. These enable one to quickly

verify whether an update to a service is backwards-compatible or if changes break core program logic.

# 8 Conclusion and Recommendations

This research aimed to develop a service-oriented software architecture capable of delivering digital services on maritime vessels. Based on the analysis of its implementation in a case study on the S.A. Agulhas II, it was shown how the proposed design is successful in achieving the specified objectives. The system was able to provide information to users, giving them greater insight into vessel operation for enhanced decision-making. This information comprised discrete contributions from domain-experts, where these contributions originated from engineering models and environmental information services. The case study conducted showcases the initial value that this approach to digitalisation could have in the maritime domain, where the services leveraged were capable of providing insight into how proposed routes relate to operational cost. This allows stakeholders to make more informed decisions regarding viable routes by balancing sailing time and incurred costs.

The design was successful in coordinating information between its services, where the aggregating services were shown to provide a valuable level of abstraction to the system. The specification of aggregation services enables a separation of concerns for developers; allowing the microservices to focus solely on capturing domain knowledge, with aggregation services focussing on serving stakeholder needs by employing these microservices.

Reconfigurability was tested, where the proposed design was shown to be easily reconfigurable. Service contributions were demonstrated to require little effort from developers, with the majority of service setup and code being generic enough to enable autonomous generation. Additionally, integrating services into the system was shown to require minimal adjustment, necessitating changes to configuration files only. The design was shown to successfully enable dynamic service updates, as well as service addition – although the latter was not a requirement for this application.

The discussion provided on user session support highlights the importance that system-wide rate-limiting has in this design. Considering that the effect of multiple sessions can only be controlled through rate-limiting, the gateway component is shown to be of utmost importance – and any additional latency incurred through its inclusion is seen as worthwhile.

It can be concluded that the proposed architecture is capable of providing a flexible digital service system for use on maritime vessels, however, the success of its application requires careful consideration of the hardware on which the system and its components are deployed. Contribution was shown to require little understanding of the greater system, with platform integration and support functionality being added to services by default. Thus, this system can be reconfigured on a per-voyage basis, allowing it to adapt to changes in the vessel, available services, and/or its specific voyage goals.

During testing, a concern was identified in the implementation of service-level rate-limiting and client-side retry logic. While it is concluded from the evaluation that the specification of interceptor middleware did not have a significant impact on performance, the experiments suggest that interceptors are not a suitable approach for providing functionality acting on underlying connections. Considering the OSI model of networking, RPC middleware, and the interceptors that they provide, operate at the application layer, layer 7. This makes it high-level software, and with the underlying TCP connections existing at the transport layer, layer 4, RPC frameworks may not have the control over the transport details required to provide reliable functionality regarding the connection. It is recommended that service-level rate limiting and client-side retry logic be performed by a proxy middleware instead of through interceptors. Proxies traditionally operate at layers 3 and 4, giving them greater control over the connection. A generic proxy configuration could be added to each service, performing the required rate-limiting or retry logic, while interceptors are maintained to perform authorisation, logging, and metric collection. This maintains the abstraction for developers, but provides better control over connections.

Additionally, a use case that was not considered, and may be worthwhile investigating, is that where a digital twin acts as a client to the system. Currently, the system makes provisions for interaction with digital twins following the digital-twin-as-a-service model. Here, digital twins offer a service and the system queries information from this service. In this interaction, the system acts as a client only. No provisions have been made for the situation where a digital twin might want information from this system, in which case the system would offer services to a digital twin client. Note that the proposed architecture places no constraints preventing this behaviour, it just has not been evaluated in this case study.

Thus, four opportunities for further research have been identified:

- To evaluate reliability enhancements through the use of proxy middleware for certain platform functionality. The Istio project has been identified as a possible solution for this research. This project provides a solution to offering generic platform functionality to microservices, wrapping each service with a custom implementation of the Envoy Proxy. The Envoy proxy proves to be a reliable rate limiter in this case study, where it is employed at a system level. It is assumed that it would maintain this reliability when implemented at a service level with an Istio configuration.

- Evaluate the system where data is not available in a usable format. This work is based on the assumption that data is available in a format that is readily-usable. This work used simulated data instead of a simulated data pipeline where the effect of obtaining the data stored in a proprietary format is not considered. Before the system can be considered fully capable of offering maritime services in real-time, an investigation is required on the effect that converting/pre-processing data may have on real-time insight

delivery. This investigation should place a focus on shaft-line services, such as the propeller monitor service, which could be tested on a simulated measurement system employing the actual hardware used to record data on board the SAAII.

- Investigating the inclusion of a digital twin as a client in this architecture, where mature digital twin implementations could query environmental or service-based information from this system. A possible approach to this would be to host a gateway catering to digital twins as a user. This gateway would be configured to provide translation functionality for common digital twin communication protocols. Additionally, the encryption performed by this gateway when communicating outside of the system would use a separate set of certificates than those used internally and at the web interface, for enhanced security.

- With the architecture and generic platform functionality available, explore more advanced service implementations within the architecture. The case study implemented here employed existing models and algorithms as a proof of concept for the design, with a rudimentary digital twin representation. Future work that considers fatigue damage and/or asset lifecycle would be of great value to the maritime domain. These services would likely be offered by a sub-system digital twin as they describe specific assets and their lifetimes. The implementation of such services could be evaluated in the same context of route planning, allowing for potential routes to be described by the estimated wear on sub-systems, or the vessel itself.

# 9 References

Abdelhedi, K. & Bouassidar, N. 2020. An SOA Design Patterns Recommendation System Based on Ontology. *Advances in Intelligent Systems and Computing*. Springer. 940:1020–1030.

Al-Jaroodi, J., Jawhar, I., Al-Dhaheri, A., Al-Abdouli, F. & Mohamed, N. 2010. Security Middleware Approaches and Issues for Ubiquitous Applications. *Computers & Mathematics with Applications*. 60:187–197.

Alexander, C., Silverstein, M. & Sara Ishikawa. 1977. *A Pattern Language*. Oxford University Press.

Basu, S. 2018. *Microservices Aggregator Design Pattern Using AWS Lambda*. [Online], Available: https://dzone.com/articles/microservices-aggregator-design-pattern-using-aws [2020, May 26].

Bekker, A. 2017. From (Big) Data to Insight – A Roadmap for the " SA Agulhas II ". *High Performance Marine Vessels*. (September, 11).

Bellemare, A. 2020. *Building Event-Driven Microservices*. O'Reilly.

Benayache, A., Bilami, A., Barkat, S., Lorenz, P. & Taleb, H. 2019. MsM: A Microservice Middleware for Smart WSN-Based IoT Application. *Journal of Network and Computer Applications*. 144:138–154.

Berger, C., Nguyen, B. & Benderius, O. 2017. Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices. In *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*. 7–12. (April, 5).

Bishop, T. & Karne, R. 2003. A Survey of Middleware. In *Proceedings of the ISCA 18th International Conference Computers and Their Application*. 254–258. (March, 26-28).

Borodulin, K., Sokolinsky, L., Radchenko, G., Tchernykh, A., Shestakov, A. & Prodan, R. 2017. Towards Digital Twins Cloud Platform: Microservices and Computational Workflows to Rule a Smart Factory. In *UCC 2017 - Proceedings of the10th International Conference on Utility and Cloud Computing*. 205–206. (December, 5-8).

Botti, V. & Giret, A. 2008. Holons and Agents. *Journal of Intelligent Manufacturing*. 15:645–659.

Ciavotta, M., Dal Maso, G., Rovere, D., Tsvetanov, R. & Menato, S. 2019. Towards the Digital Factory: A Microservices-Based Middleware for Real-to-Digital Synchronization. In *Microservices, Science and Engineering*.

Springer. 273–297.

Damyanov, I. 2019. Data Aggregation in Microservice Architecture. *International Journal of Online and Biomedical Engineering*. 15:81–87.

de la Peña Zarzuelo, I., Freire Soeane, M.J. & López Bermúdez, B. 2020. Industry 4.0 in the Port and Maritime Industry: A Literature Review. *Journal of Industrial Information Integration*. 20:100-173.

Derigent, W., Cardin, O. & Trentesaux, D. 2021. Industry 4.0: Contributions of Holonic Manufacturing Control Architectures and Future Challenges. *Journal of Intelligent Manufacturing*. 32:1797–1818.

Durandt, P.G. 2020. *Data-Driven Regression Models for Voyage Cost Optimization Based on the Operating Conditions of the SA Agulhas II*. Masters Thesis. Stellenbosch University.

Egert, R., Grube, T., Volk, F. & Mühlhäuser, M. 2021. Holonic System Model for Resilient Energy Grid Operation. *Energies*. 14:4120.

Ellingsen, O. & Aasland, K.E. 2019. Digitalizing the Maritime Industry: A Case Study of Technology Acquisition and Enabling Advanced Manufacturing Technology. *Journal of Engineering and Technology Management - JET-M*. 54:12–27.

Erikstad, S.O. 2019. Designing Ship Digital Services. In *Computer Applications and Information Technology (Compit)*. 458–469. (March, 25-27).

Evans, E. 2003. *Domain-Driven Desgin*. 1st ed. Addison-Wesley Professional.

Fielding, R. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral Thesis. University of California.

Firdhous, M. 2012. Implementation of Security in Distributed Systems - A Comparative Study. *International Journal of Computer and Information Sytems*. 2:1–6.

Fonseca, Í. & Gaspar, H. 2020. Challenges when Creating a Cohesive Digital Twin Ship: a Data Modelling Perspective. *Ship Technology Research*. 68:1–14.

Fowler, M. & Lewis, J. 2014. *Microservices*. [Online], Available: https://martinfowler.com/articles/microservices.html [2020, May 26].

Gamboa, F., Cardin, O., L'anton, A. & Castagna, P. 2015. Process Specification Framework in a Service Oriented Holonic Manufacturing Systems. *Studies in Computational Intelligence*. 594:81–89.

Harper, K.E., Ganz, C. & Malakuti, S. 2019. Digital Twin Architecture and Standards. *Industrial Internet Consortium Journal of Innovation*. 12:72-83.

He, S., Zhao, L. & Pan, M. 2019. The Design of Inland River Ship Microservice Information System Based on Spring Cloud. In *Proceedings - 2018 5th International Conference on Information Science and Control Engineering, ICISCE*. 548–551. (July, 20-22).

Henning, K., Wahlster, W. & Helbig, J. 2013. *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0*. Forschungsunion.

IBM. 2021. *What is Middleware?* [Online], Available: https://www.ibm.com/cloud/learn/middleware [2021, October 18].

International Organization for Standardization, 1997. *ISO 2631-1:1997. Mechanical Vibration and Shock - Evaluation of Human Exposure to Whole Body Vibration*. Organization for Standardization.

International Organization for Standardization, 2011. *ISO/IEC 25010:2011. Information Technology - Systems and Software Quality Requirements and Evaluation - System and Software Quality Models*. Organization for Standardization.

Jansen, K., Whiting, C. & Hulbert, G. 2000. Generalized-Alpha Method for Integrating the Filtered Navier-Stokes Equations with a Stabilized Finite Element Method. *Computer Methods in Applied Mechanics and Engineering*. 190:305-319.

Jones, S. 2014. *Addressing Cyber Security Risks at Ports and Terminals*. London. [Online], Available: www.seasecurity.org [2021, October 19].

Kappagantula, S. 2019. *Top Microservices Design Patterns To Build Your Applications*. [Online], Available: https://medium.com/edureka/microservices-design-patterns-50640c7bf4a9 [2020, May 26].

Kavallieratos, G., Katsikas, S. & Gkioulos, V. 2020. Modelling Shipping 4.0: A Reference Architecture for the Cyber-Enabled Ship. *Intelligent Information and Database Systems*. 202–217. (February, 16).

Kelley, M. & Ó Gráda, C. 2018. *Speed under sail during the early Industrial Revolution*. [Online], Available: https://voxeu.org/article/speed-under-sail-during-early-industrial-revolution [2020, May 26].

Krishnamurthy, J. & Maheswaran, M. 2016. Programming frameworks for Internet of Things. *Internet of Things: Principles and Paradigms*. Morgan Kaufmann. 79-102.

Kruger, K., Human, C. & Basson, A.H. 2021. Towards the Integration of Digital Twins and Service- Oriented Architectures. In *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*. (October 2021).

Malik, S., Ahmad, S. & Kim, D.H. 2019. A Novel Approach of IoT Services Orchestration Based on Multiple Sensor and Actuator Platforms using Virtual Objects in Online IoT App-Store. *Sustainability (Switzerland)*. 11:204.

Maoudj, A., Bouzouia, B., Hentout, A., Kouider, A. & Redouane, T. 2019. Distributed multi-agent scheduling and control system for robotic flexible assembly cells. *Journal of Intelligent Manufacturing*. 30:1629–1644.

McDuffie, J. 2017. *Why the Military Released GPS to the Public*. [Online], Available: https://www.popularmechanics.com/technology/gadgets/a26980/why-the-military-released-gps-to-the-public/ [2020, May 26].

Meyer, J. 2007. Service Oriented Architecture (SOA) Migration Strategy for U.S. Operational Naval Meteorology and Oceanography (METOC). In *Oceans Conference*. (June, 18-21).

Minerva, R., Lee, G.M. & Crespi, N. 2020. Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models. *Proceedings of the IEEE*. 108:1–40.

Mühl, G., Fiege, L. & Pietzuch, P. 2006. *Distributed Event-Based Systems*. Springer International Publishing.

Murthy, N. 2017. *REST, RPC, and Brokered Messaging*. [Online], Available: https://medium.com/@natemurthy/rest-rpc-and-brokered-messaging-b775aeb0db3 [2020, October 20].

Nair, S. n.d. *History of Sea Navigation Before the GPS*. [Online], Available: https://www.teletracnavman.com/gps-fleet-tracking-education/history-of-sea-navigation-before-the-gps [2020, May 26].

Newman, S. 2014. *Building Microservices: Designing Fine-Grained Sytems*. 1st ed. O'Reilly.

Nickerson, B. 2021. *Inverse Models for Ice-Induced Propeller Moments on a Polar Vessel*. Doctoral Thesis. Stellenbosch University.

North, D. 1968. Sources of Productivity Change in Ocean Shipping, 1600-1850. *Journal of Political Economy*. 76:953–970.

Penobscop Marine Museum. 2012. *Navigation: the 20th Century to the Present*.

[Online], Available: https://penobscotmarinemuseum.org/pbho-1/history-of-navigation/navigation-20th-century-present [2020, May 26].

Richardson, C. 2018. *Microservice Patterns*. 1st ed. Manning.

Rodriguez, S., Hilaire, V. & Koukam, A. 2005. Formal Specification of Holonic Multi-Agent Systems Framework. *International Conference on Cyber Security and Privacy*. 3516:719–726.

Rodriguez, S., Hilaire, V., Gaud, N., Galland, S. & Koukam, A. 2011. Holonic Multi-Agent Systems. *Natural Computing Series*. 37:238–263.

Russell, D., Looker, N., Liu, L. & Xu, J. 2008. Service-Oriented Integration of Systems for Military Capability. In *Object Oriented Real-Time Distributed Computing*. 33–41. (May, 5-7).

SEBoK Editorial Board. 2021. *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*. v.2.5, Cloutier, R.J (Ed.). Hoboken, NJ: The Trustees of the Stevens Institute of Technology. Accessed [2020, July 13]. www.sebokwiki.org.

Singh Gill, N. 2020. *A Quick Guide to Service-Oriented Architecture (SOA)*. [Online], Available: https://www.xenonstack.com/insights/service-oriented-architecture [2021, October 18].

Soal, K.I. 2014. *Vibration Response of the Polar Supply and Research Vessel the S.A. Agulhas II in Antarctica and the Southern Ocean*. Masters Thesis. Stellenbosch University.

Tragatschnig, S., Stevanetic, S. & Zdun, U. 2018. Supporting the Evolution of Event-Driven Service-Oriented Architectures using Change Patterns. *Information and Software Technology*. 100:133–146.

Tyson, M. 2020. *What is service-oriented architecture?* [Online], Available: https://www.javaworld.com/article/2071889/what-is-service-oriented-architecture.html [2020, May 24].

UNCTAD. 2018. *Digitalization set to revolutionize shipping – new United Nations report*. [Online], Available: https://unctad.org/press-material/digitalization-set-revolutionize-shipping-new-united-nations-report [2020, May 12].

Valckenaers, P. 2019. ARTI Reference Architecture – PROSA Revisited: Proceedings of SOHOMA 2018. In *Service Orientation in Holonic Manufacturing and Multi-Agent Manufacturing*. 1–19.

Weyer, S., Schmitt, M., Ohmer, M. & Gorecky, D. 2015. Towards Industry 4.0 - Standardization as the crucial challenge for highly modular, multi-vendor

production systems. *IFAC-PapersOnLine*. 48:579–584.

Zoughbi, G., Kattnig, G., Parkinson, S., Lindqvist, N., al Nuaimi, M. & Balooshi, H. 2011. Considerations for Service-Oriented Architecture (SOA) in military environments. *IEEE GCC Conference & Exhibition*. (February, 19-22).

# Appendix A ISO/IEC 25010: Product Quality Evaluation System

The ISO/IEC 25010 standard is a quality model describing the characteristics to be accounted for when evaluating a software product. The standard defines the quality of a system as the degree to which said system satisfies the needs of its stakeholders. The quality model defined in this standard comprises of eight characteristics for quality, as is shown in Figure 15. (ISO/IEC & JTC1/SC7/WG6, 2011)

## A.1 Functional Suitability

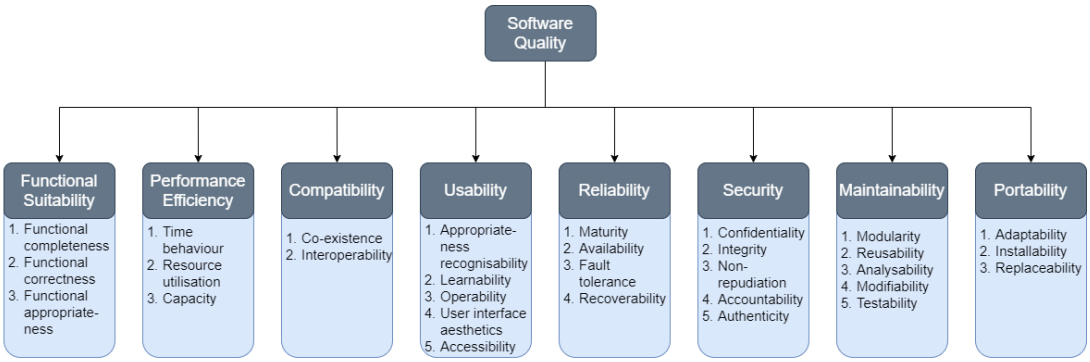



**Figure 15: ISO 25010 (adapted from ISO/IEC & JTC1/SC7/WG6 (2011))**

Functional suitability is described as the degree to which a system boasts functionality that meets the needs when used under specific conditions. This can be seen as the coverage a system provides to the user requirements. The sub-categories of functional suitability are as follows:

- *Functional Completeness*: This is the degree to which the set of functions offered by the system cover the specified objectives. Essentially, it is a means towards acceptance testing to ensure that all required functionality is offered by the system.

- *Functional Correctness*: This is the degree to which a system provides the correct results, considering the precision required by the application. Generally, this refers to mathematical precision available when transferring data.

- *Functional Appropriateness*: This is the degree to which the functions provided for by the system facilitate the accomplishment of objectives. While similar to functional completeness, this considers the suitability of choices made towards implementing the functions, not just the presence of the functionality.

## A.2 Performance Efficiency

Performance efficiency is the category that represents the systems performance, considering resource usage under specific conditions. Oftentimes, this is dependent on the implementation and the machine(s) running the software product. The evaluation should thus be carried out considering the application. The three sub-categories of performance efficiency are:

- *Time behaviour*: This considers the degree to which the response and processing times, and throughput rates of the system meet their requirements during standard operation.

- *Resource Utilisation*: This sub-category considers the amount and type of resources used by the system when it is performing its functions. This includes CPU, RAM, and memory.

- *Capacity*: This assesses how close the system is to its limits during normal operation.

## A.3 Compatibility

This characteristic serves to evaluate the degree to which a system can interoperate with other components while sharing a common hardware or software environment. Compatibility sub-categories include:

- *Co-Existence*: This evaluates how efficiently the system can perform its functionality while sharing a common environment and resources with other products. Effectively, the system, and any other system sharing resources, should not have notable reductions in performance when operating together.

- *Interoperability*: This is the degree to which the system can exchange information with and use information that has been exchanged with other systems.

## A.4 Usability

This is how effectively, efficiently, and satisfactorily the system can be used to achieve its goals. Principally, this category deals with ergonomics and user-experience. The sub-categories of usability are as follows:

- *Appropriateness and Recognisability*: This evaluates how easily users can recognise whether a product or system is appropriate for their needs.

- *Learnability*: This evaluates how easily a user can learn to use the system. It considers how intuitive the system is for new users.

- *Operability*: This deals with the user interface and evaluates the attributes included to make operation and control of the system intuitive.

- *User Error Protection*: This evaluates the degree to which the system protects users against making errors. This predominantly deals with error handling and input-validation where user are asked to provide information.

- *User Interface Aesthetics*: This is a more subjective characteristic, considering how satisfying the user interface, and interactions with it, are.

- *Accessibility*: This evaluates how accessible the system is to users with a range of capabilities. It evaluates the extent to which specific expertise/capabilities are required to use the system.

## A.5 Reliability

Reliability considers the extent to which the system can perform its functions under specified conditions. Reliability encompasses the following four sub-categories:

- *Maturity*: This is the degree of reliability to which the system performs under normal operation.

- *Availability*: This is the degree to which the system is operational and accessible when required by users.

- *Fault Tolerance*: This is the degree to which the system can perform its functions given the presence of specific hardware or software failures.

- *Recoverability*: This is the degree to which the system is able to recover itself and its data, as well as to re-establish its state in the event of a failure.

## A.6 Security

The security characteristic is used to evaluate how well the system protects information and data. Additionally, this encompasses ensuring that data access is only provided to those who should have access to it. Security encompasses the following aspects:

- *Confidentiality*: This is the degree to which the system can ensure that data is only accessible to those who are authorised to it.

- *Integrity*: This is how well a system can prevent unauthorised access or modification to programs and data. It evaluates the extent to which the system can guarantee data to its consumers.

- *Non-repudiation*: This is the degree to which events within the system can be traced back to a unique identity or source.

- *Accountability*: This evaluates the extent to which the actions of an entity can be traced back to that unique entity within the system.

- *Authenticity*: This evaluates the extent to which the identity of a subject/resource can be verified to be that which is claimed.

## A.7 Maintainability

This represents how effectively and efficiently the system can be modified for improvements, corrections, or adaptions to its environment and requirements. Maintainability includes:

- *Modularity*: This evaluates the degree to which the system is composed of discrete components. Through this, it evaluates the scope of impact that a change to a single component has to others within the system.

- *Reusability*: This evaluates the extent to which an asset or component can be used for more than one purpose or in more than one system, as well as in composing additional assets or components.

- *Modifiability*: This evaluates how the system can be modified without introducing defects to the system functionality, or degrading the quality of the system to its users.

- *Testability*: Testability evaluates how effectively the system can be tested to verify whether pre-defined criteria have been met.

## A.8 Portability

Portability encompasses aspects that can be used to evaluate whether the system can be transferred across hardware or software environments while maintaining its core functionality. Portability describes the following three aspects:

- *Adaptability*: The is the degree to which the system can be adapted for evolving hardware or software environments.

- *Installability*: This evaluates how effectively the system can be installed or uninstalled in a specific environment.

- *Replaceability*: This evaluates how well the system can replace a specified software product for the same purpose in the same environment.

# Appendix B Interceptor Source Code

This appendix contains the source code for the interceptors used in this case study. These are generic and are thus reusable across any and all implementations. For the sake of readability, the interceptors documented here are those written in Python. However, the same logic is applied in the interceptors written in other languages.

## B.1 Retry Interceptor

The retry interceptor contains the logic required to retry failed calls. To perform this logic, only the intercept method (Listing 1) is required. If the response contains an error, the intercept method will wait a specific amount of time (using exponential backoff logic) before retrying the call. The wait times and retry limit are set when adding the interceptor to a server.

```
def intercept(
    self,
    method,
    request_or_iterator,
    call_details: ClientCallDetails,
    ):
    '''
    This is the function that runs when the call is received.
    '''

    while self.failureCount < (self.maxRetries - 1):
        logger.info(f"Trying to make a call, call number: {self.failureCount}")

        # Attempt to make the call
        response =  method(request_or_iterator, call_details)

        # Evaluate whether the response has an error
        if(hasattr(response, "_state") and (response._state.code)):
            print(f"Call attempt failed")
            time.sleep((self.waitTime/100)*(2**self.failureCount)) ''' Wait function
            that implements exponential backoff based on the number of failed calls
            '''
            self.failureCount += 1 # Increment the failed call counter
        else:
            return response
    return method(request_or_iterator, call_details)
```

**Listing 1: Retry interceptor, ‘intercept’ method**

## B.2 Metric Interceptor

The metric interceptor is implemented using a Python decorator. Two helper functions are used to separate operations. The pushToPrometheus function (Listing 2) is responsible for labelling metrics and posting them to the Prometheus push gateway. The sendMetrics decorator function (Listing 4) is responsible for generating the metric data, invoking the server call, and calling the pushToPrometheus function. This decorator function wraps the intercept method (Listing 3) allowing for additional logic to be added to the interceptor both before and after the call has been made.

```
def pushToPrometheus(
    c,
    g,
    h,
    executionTime,
    serviceName,
    address,
    method,
    job,
    registry
    ):
    '''This function sets the labels for a Prometheus entry and pushes metrics to the
    push-gateway.
    '''
    c.labels(
        Role="Server",
        grpc_type = 'unary',
        grpc_service = serviceName,
        grpc_method = method
        ).inc() # Increment the call counter
    g.labels(
        Role="Server",
        grpc_type = 'unary',
        grpc_service = serviceName,
        grpc_method = method
        ).set_to_current_time() # Set last call time to the current time
    h.labels(
        Role="Server",
        grpc_type = 'unary',
        grpc_service = serviceName,
        grpc_method = method
        ).observe(executionTime) ''' Set the response latency to the execution time of
        the service '''

    # Post the metrics to the gateway
    prometheus.push_to_gateway(address, job=job, registry=registry)
    logger.info("Succesfully pushed metrics")
```

**Listing 2: Metric interceptor, 'pushToPrometheus' function**

```python
def sendMetrics(func):
    '''This decorator wraps the intercept method, setting the metrics before invoking
    the pushToPrometheus function.
    '''
    from functools import wraps

    @wraps(func)
    def wrapper(*args, **kw):
        logger.debug("Starting Interceptor decorator")

        # Extract the service name and method from the incoming request
        if isinstance(args[3], grpc._server._Context):
            servicerContext = args[3]
            # This gives us <service>/<method name>
            serviceMethod = servicerContext._rpc_event.call_details.method
            serviceName, methodName = str(serviceMethod).rsplit('/')[1::]
        else:
            logger.warning('Cannot derive the service name and method')
            raise Exception("Could not derive service name or method.")

        try:
            # Set the start time of the call
            startTime = time.time()
            # Invoke the service call
            result = func(*args, **kw, )
            resultStatus = "Success"
            logger.debug("Function call: {}".format(resultStatus))
        except Exception:
            resultStatus = "Error"
            logger.warning("Function call: {}".format(resultStatus))
            raise
        finally:
            # Calculate the time since the start of the call
            responseTime = time.time() - startTime
            # Push metrics to the Prometheus server
            pushToPrometheus(
                args[0].c,
                args[0].g,
                args[0].h,
                responseTime,
                serviceName.rsplit(".")[2],
                args[0].address,
                methodName,
                args[0].microserviceName,
                args[0].registry)
        return result
    return wrapper
```

**Listing 4: Metric interceptor, ‘sendMetrics’ decorator function**

```python
@sendMetrics
def intercept(self, method, request, context, methodName):
    '''This is the function that runs when the call is received. In this interceptor
    the logic is performed by a decorater, wrapping this function with the required
    functionality.
    '''

    logger.info("Starting server-side metric interceptor method")

    return method(request, context)
```

**Listing 3: Metric interceptor, ‘intercept’ method**

## B.3 Rate Limit Interceptor

The rate limit interceptor performs all necessary logic in its intercept method (Listing 5). Whenever a new request is received, it increments its counter, decreasing it when a response is returned.

```
def intercept(self, method, request, context, methodName):
    '''This is the function that runs when the call is received.
    '''

    self.callCounter += 1 # Increment the number of active calls

    # Check if call counter exceeds the specified call limit
    if (self.callCounter > self.callLimit):
        logger.error("Concurrent request limit reached.")
        self.callCounter -= 1 ''' Decrement the number of active calls as the
        current call is not going to be made '''
        context.set_code(grpc.StatusCode.RESOURCE_EXHAUSTED)
        context.set_details('Concurrent request limit reached.')
        return None
    else:
        # Invoke the requested method
        response =  method(request, context)
        ''' Decrement the numebr of active calls as the current call has been
        processed '''
        self.callCounter -= 1

        # Return the response
        return response
```

**Listing 5: Rate limit interceptor, 'intercept' method**

## B.4 Authorisation Interceptor

The authorisation interceptor employs three methods to perform authorisation logic. The authorise method (Listing 6) firstly evaluates whether the requested RPC requires authentication. If it does, the interceptor extracts the metadata from the request to obtain the authentication token. The interceptor subsequently evaluates whether the token is valid, before verifying that the user has permission to access the RPC that they are requesting. If none of these checks return, the interceptor defaults to rejecting the request based on the premise that their claims could not be verified.

```
def authorise(self, methodName, context):
    '''This function goes through a series of checks to verify that the user
    making a request is properly authenticated for that request.
    '''

    # Check if the requested method requires authentication
    accessibleRoles = self.authenticatedMethods.get(methodName)
    if (accessibleRoles == None):
        logger.info(f"Authentication is not required for {methodName}")
        return

    # Extract the metadata from the request
    try:
        metadata = dict(context.invocation_metadata())
    except:
        logger.debug("Failed to authenticate: metadata is not provided")
        raise grpc.StatusCode.PERMISSION_DENIED

    # Check if a JWT has been included in the metadata
    try:
        encodedToken = metadata["authorisation"]
    except:
        logger.debug("Failed to authenticate: JWT has not been provided")
        raise grpc.StatusCode.PERMISSION_DENIED

    # Check that the provided JWT is valid
    claims, err = self.verifyJWT(encodedToken) ''' Extract the user claims from the
    token '''
    if (err != None):
        logger.debug("Failed to authenticate: Provided JWT is invalid")
        raise err

    ''' Check that the role of the user making the service call authenticates them
    for the service being called
    '''
    for role in accessibleRoles:
        if (role == claims["role"]):
            logger.debug(f"Successfully authenticated request for {methodName}")
            return

    logger.debug('''Failed to authenticate: the user does not have permission to
    access the requested service''')
    raise grpc.StatusCode.PERMISSION_DENIED
```

**Listing 6: Authorisation interceptor, ‘authorise’ method**

The verifyJWT method (Listing 8) decodes the token it is provided with, returning a structured data object (key-value pair) containing token and user claim info. The HS256 signing algorithm was used for token encryption in this case study.

```python
def verifyJWT(self, accessToken):
    '''This function takes a JWT token as an input and decodes it to ensure that it
    is valid (according to the HS256 algorithm).
    '''

    try:
        token = jwt.decode(accessToken, self.secretKey, algorithms=["HS256"])
    except Exception as e:
        logger.debug(f"Invalid token: {e}")
        raise grpc.StatusCode.PERMISSION_DENIED

    return token, None
```

**Listing 8: Authorisation interceptor, 'verifyJWT' method**

Upon receiving a request, the intercept method (Listing 7) invokes the authorise function to verify the user making the request. If an error is returned, the RPC is not made to the server, with the interceptor instead returning an error.

```python
def intercept(self, method, request, context, methodName):
    '''This is the function that runs when the call is received.
    '''

    logger.info("Starting server-side authentication interceptor")

    # Attempt to authorise the user for the request
    err = self.authorise(methodName, context)
    if err:
        context.set_code(err)
        return None

    # Return the result of the service/method call
    return method(request, context)
```

**Listing 7: Authorisation interceptor, 'intercept' method**

# Appendix C Case Study Components

Where not specified, the rate limit interceptor has been configured to allow 150 concurrent calls to be made to the service at any given time. All client connections have been fitted with a retry interceptor. The retry interceptors are configured to retry a call when a gRPC error is received, and to retry a maximum of 5 times. The first retry waits 100ms, with each subsequent retry waiting twice the previous waiting period. The maximum time spent retrying is thus 3.1s.

All aggregator services in this case study have been written in Golang due to its suitability for system development and high-performance approach to concurrency. Additionally, Golang, being a product of Google, has extensive support for gRPC, with both server and client interceptors being well documented for the language.

Where applicable, service interfaces are provided, detailing what calls are on offer and the relevant message contents. The fields of the request and response messages have been colour coded in these figures to show the origin of the data. Additionally, fields encapsulated by square brackets indicate an array of values – and in most cases represent a time-series.

All source code can be found in the project repository on Github.

## C.1 Ocean Weather Service

The ocean weather service provides information about the weather that the vessel encounters during open-water passage. This service queries the APIs of select, reputable marine weather services, fetching a tailored set of parameters that are required to describe the vessel's environment along a route. The outputs of this service are intended to serve other services that may require environmental inputs. The interface for this service is shown in Figure 16.

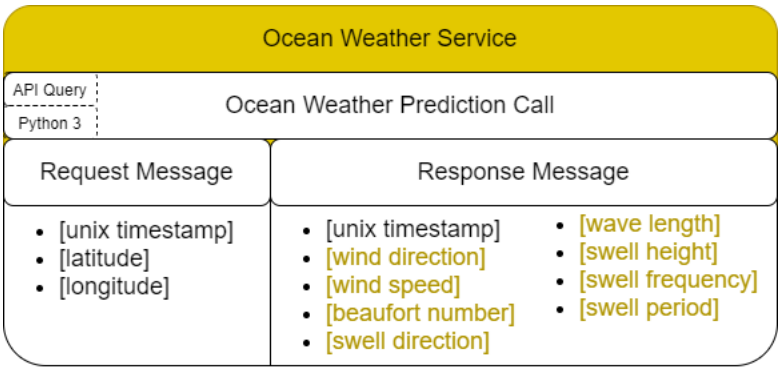



**Figure 16: Ocean weather service interface**

This service receives a set of associated latitudes, longitudes, and (Unix) timestamps as input, which it uses to query the pre-defined parameters. For the models present in this implementation, these parameters are: wind speed, wind direction, swell direction, swell height, and swell period. From these, the service further calculates the wave length, swell frequency, and derives the Beaufort number. For the case study, a single call is offered. The *Ocean Weather Prediction* call offers a marine weather prediction, providing foresight for tactical decision-making. This call uses the Stormglass API to fetch future predictions as it offers an intuitive interface for retrieving marine weather predictions, intended for navigation and route planning of maritime vessels.

The Stormglass API does not provide swell frequency or wave length, but does provide the information required to calculate such parameters. The swell frequency can be determined by taking the reciprocal of the swell period. The wave length is calculated using the relationship shown in Equation 1, where $\boldsymbol{\lambda}$ represents the wave length and T represents the swell period.

$$\boldsymbol{\lambda = 1.56 \cdot T^2} \tag{1}$$

It was observed that majority of marine weather services offer ready-to-use API libraries and well-documented interfaces for Python and, as such, this service was written in Python 3. The rate limit interceptor has been configured to allow for 50 concurrent requests to be made at any given time – this was selected to limit the Stormglass API usage during testing.

## C.2 Power-Train Service

The power-train service offers information on the power-train of the SAAII. It is a servitisation of the work done by Durandt (2020), focussing on providing power and cost estimates for the operation of the vessel through a data-driven model. The outputs of this service are intended to be used for route-planning and analysis.

This service requires a combination of the proposed sailing configuration (motor speeds, propeller pitches, and speed over ground) and predicted environmental conditions (relative wind direction, wind speed, Beaufort number, wave direction, and wavelength). Two calls are offered by this service. The *Power Estimate* call provides foresight for tactical decision-making by estimating the power required for a proposed route-plan - this is served by a data-driven model of the SAAII's power requirements. The *Cost Estimate* call serves the same temporal aspect and value space, building on this by analysing the cost as a result of the required power and elapsed time of a proposed route. The additional cost ($C_{add}$) for a time period is calculated using Equation 2:

$$\boldsymbol{C_{add} = C_{hrly} \cdot \left(\frac{\delta_{time}}{3600}\right) + C_{kWh} \cdot P_{Est} \cdot \left(\frac{\delta_{time}}{3600}\right)} \tag{2}$$

Here, the $C_{hrly}$ is the cost of running the vessel per hour. This includes crew salaries and equates to R10000. $C_{kWh}$ is the cost incurred by each kWh used by the SAAII – this is dependent on the price of diesel for each voyage and the efficiency of the

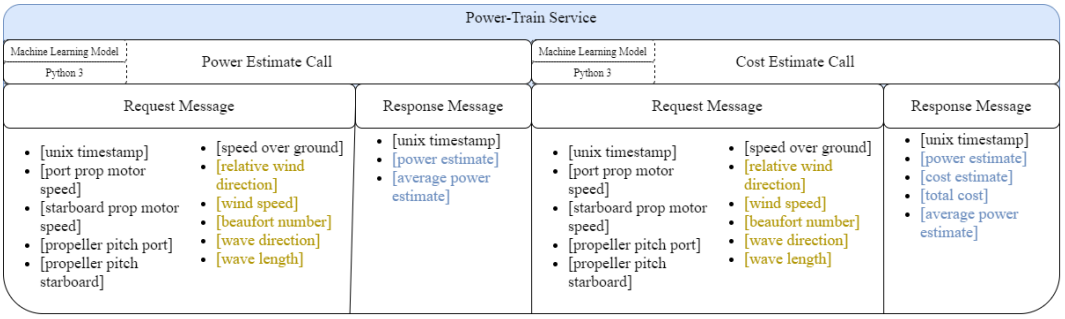



**Figure 17: Power-train service interface**

SAAII, which is documented as being 179 $g_{diesel}$/kWh (Durandt, 2020). $P_{est}$ is the power estimate, produced by the data-driven model used in the power estimate call.

The original model, as developed by Durandt (2020), was done in Python using the Keras and Tensorflow libraries. This choice was made as Python offers outstanding support for machine learning and data manipulation. This service was thus written in Python 3 to demonstrate the ease with which one can servitise work done by domain experts. The interfaces for this service are shown in Figure 17, with source code available in the project's Github repository.

## C.3 Vibration Estimate Service

This service offers information about whole body vibration on the SAAII. Whole body vibration refers to vibrations in the range of 0.5Hz to 80Hz (Soal, 2014). It is a servitisation of the work done by Soal & Bekker (2014) with a focus on estimating vessel response to wave conditions. The service interface is presented in Figure 18.

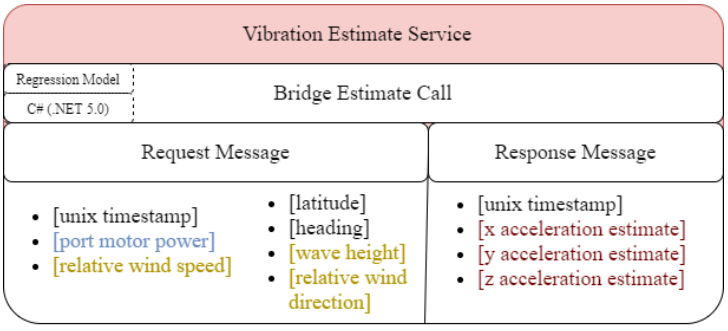



**Figure 18: Vibration estimate service interface**

Again, this service requires a combination of wave conditions (relative wind speed, wave height, relative wind direction) and sailing configuration (port prop motor

power, latitude, and heading) as inputs to the call. For this case study, only a single call is offered. The *Bridge Estimate* call provides foresight for tactical decision-making by providing human-weighted whole body RMS vibration estimates for the three primary axes (x, y, and z) on the bridge; this is done by implementing the regression model developed by Soal & Bekker (2014) for human-weighted vibration estimation on the bridge of the SAAII. The vibration response, $Y_n$, can be calculated using the regression model described by Equation 3 (Soal, 2014), with the coefficients provided in Table 8. This provides the whole body vibration response in m/s$^2$.

$$\boldsymbol{Y_n = C_n + \alpha \cdot C_n + \beta \cdot C_n + \gamma \cdot C_n + \delta \cdot C_n + \zeta \cdot C_n + \eta \cdot C_n} \quad \textbf{(3)}$$

**Table 8: Regression coefficients for open-water bridge estimates (Soal, 2014)**

| Coefficient | X-axis | Y-axis | Z-axis |
|---|---|---|---|
| Intercept C | 2.7298 | 2.5711 | 1.7605 |
| Port Prop Motor Power $\alpha$ | 0.0013 | 0.0016 | 0.0010 |
| Latitude $\beta$ | -0.0004 | -0.0004 | -0.0004 |
| Relative Wind Speed $\gamma$ | 0 | 0 | 0.2050 |
| Relative Wind Direction $\delta$ | -0.0010 | -0.0010 | -0.0008 |
| Heading $\zeta$ | 0.0037 | 0.0011 | 0.0017 |
| Wave height η | 0 | 0.8668 | 0 |

The implementation of this service is relatively simple as it required only basic math operations. To showcase a more diverse technology stack, this service was implemented in C# using the .NET 5 framework – source code can be found in the project repository.

## C.4 Comfort Service

The comfort service offers a single call that serves all three temporal aspects, depending on the context in which it is used. The *Comfort Rating* service call takes either a single value for, or a series of, human-weighted RMS vibration(s) as input. Where a series is given, it calculates an equivalent vibration of the time-series using Equation 4 (Equation C1, Standardization, 1997). This provides an aggregate of the vibration that passengers are exposed to for multiple periods of exposure to vibration of different magnitudes. Where the equivalent vibration is used, high-resolution measurements are diluted in order to provide a "summary" of the vibrations imposed on passengers.

$$\boldsymbol{a_{w,e} = \sqrt{\frac{\sum a_{w_i}{}^2 \cdot T_i}{\sum T_i}}} \quad \textbf{(4)}$$

This (equivalent) vibration is then used to classify the comfort of passengers on board, based on the thresholds outlined in ISO 2631-1, which are documented in Table 9. This service was again implemented in Python 3, with its interface recorded in Figure 19 – source code can be found in the project repository.

**Table 9: Comfort ratings (adapted from Appendix C, Standardization, 1997)**

| Threshold | Classification |
|---|---|
| $a < 0.314$ [m/s$^2$] | Not uncomfortable |
| 0.315 [m/s$^2$] < a <0.63 [m/s$^2$] | A little uncomfortable |
| 0.5 [m/s$^2$] < a < 1 [m/s$^2$] | Fairly uncomfortable |
| 0.8 [m/s$^2$] < a < 1.6 [m/s$^2$] | Uncomfortable |
| 1.25 [m/s$^2$] < a < 2.5 [m/s$^2$] | Very uncomfortable |
| a > 2 [m/s$^2$] | Extremely uncomfortable |

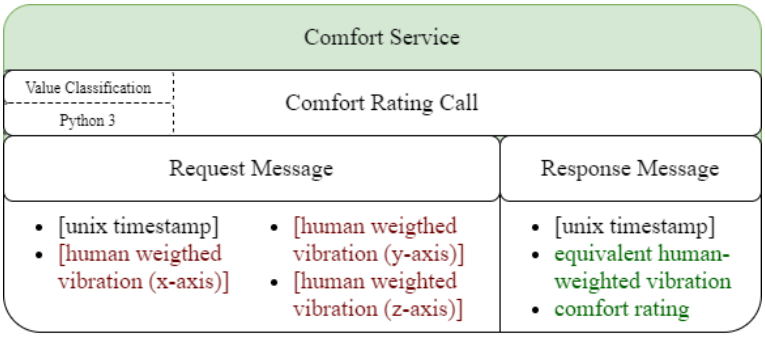



**Figure 19: Comfort service interface**

## C.5 Propeller Monitor Service

The propeller monitor service offers a single call providing real-time insights. This service servitises the model developed by Nickerson (2021), calculating the inverse problem to estimate propeller and ice induced torques based on internal shaft measurements. This service can be used to determine if the calculated ice load exceeds the threshold at the propeller. The information required by this model (shaft RPM and internal shaft torque) is information that would likely be recorded by, and belong to, a shaft line digital twin. Additionally, shaft information such as material characteristics and dimensions would be documented by this digital twin too. As such, this service is run remotely to represent a digital twin's service interface, with the interface available in Figure 20. Note that in this interface, the request message is empty. This is because all information required by the model belongs to the digital twin offering the service.

The solution presented by Nickerson (2021) performs matrix manipulations to solve the inverse problem of a continuous model of the SAAII's shaft. The solution derives a set of matrices, which are solved using the modified generalised-alpha

method (Jansen, Whiting & Hulbert, 2000). This is offered as a service through the *Estimate Propeller Load* call, providing soft real-time insight into the state of the shaft line. The equation derived by Nickerson (2021) and is solved by this service can be seen in Equation 5(5, with relevant matrices shown in Equations 6 to 9.

$$\boldsymbol{J\ddot{q} + C\dot{q} + Kq = Q} \tag{5}$$

$$\boldsymbol{J} = \begin{bmatrix} (\rho JL + J_p + J_{motor}) & (J_p - J_{motor}) & \cdots & (J_p - J_{motor}) & 0 & 0 \\ (J_p - J_{motor}) & (\rho JL + J_p + J_{motor}) & \cdots & (J_p + J_{motor}) & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ (J_p - J_{motor}) & (J_p + J_{motor}) & \cdots & (\rho J\frac{L}{2} + J_p + J_{motor}) & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

$$\boldsymbol{C} = \begin{bmatrix} C_p & C_p & C_p & \cdots & C_p & 0 & 0 \\ C_p & C_p & C_p & \cdots & C_p & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ C_p & C_p & C_p & \cdots & C_p & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \varphi_0(x_a) & \varphi_1(x_a) & \varphi_3(x_a) & \cdots & \varphi_{N-1}(x_a) & 0 & 0 \end{bmatrix} \tag{7}$$

$$\boldsymbol{K} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 1 & -1 \\ 0 & GJ\frac{\pi^2}{2L} & 0 & \cdots & 0 & 1 & 1 \\ 0 & 0 & GJ\frac{(3\pi)^2}{2L} & \cdots & 0 & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & GJ\frac{((N-1)\pi)^2}{2L} & 1 & 1 \\ 0 & GJ\varphi_1'(x_a) & GJ\varphi_3'(x_a) & \cdots & GJ\varphi_{N-1}'(x_a) & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \tag{8}$$

$$\boldsymbol{Q} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ Q(x_0, t) \\ \dot{\theta}(x_0, t) \end{bmatrix} \tag{9}$$

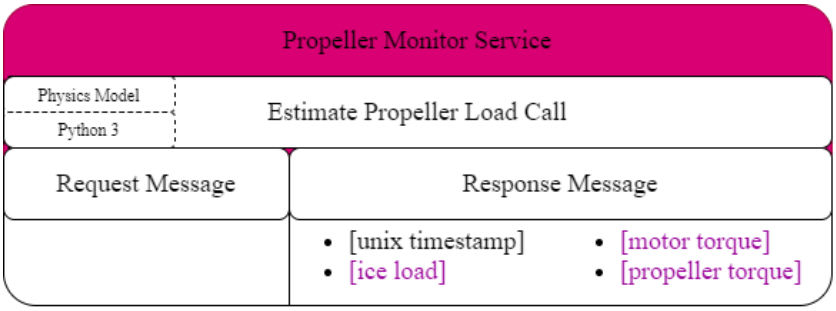



**Figure 20: Propeller monitor service interface**

The original study tested the algorithm in Matlab which is not natively supported by gRPC. The model was thus translated into a Python 3 implementation employing NumPy libraries so that it could be run as a service – source code can be found in the project repository. This implementation leverages gRPC's server-side streaming functionality to provide a constant feed of data to users. The service monitors the "digital twin's" data stores for updated data, running the algorithm and streaming the results when new data is made available. This breaks the traditional request/response mould where a single request receives a single response. Instead, a single request for monitoring is made, with multiple responses provided, so long as new data is observed. It is worth noting that the implementation of this service does not require aggregation of information outside of the service's scope. As such, this service is communicated with directly from the gateway, without an aggregation service, as can be seen in Figure 6.

## C.6 Route Analysis Aggregator

The route analysis aggregator takes a route as an input and provides a summary of it using high-level statistics (e.g. using ensemble estimates instead of full time-series descriptions). This summary aggregates information describing the route to provide multiple, low-resolution descriptions. The beginning of this flow invokes the ocean weather service to fetch weather estimates along a route, which are fed through to the power-train service with the sailing configuration to obtain power and cost estimates. These estimates are added to a new request message used to invoke the vibration estimate service and, subsequently, the comfort service. The response contains summary information from almost every service in the invocation chain such that the user can get a low-resolution picture of the proposed route. An approach without this aggregation component would not be able to provide information from multiple services like this without adding logic into each service that enables it. The addition of this logic, however, would create a tight, static coupling between services that makes microservice management incredibly difficult. The service interface is shown in Figure 21.

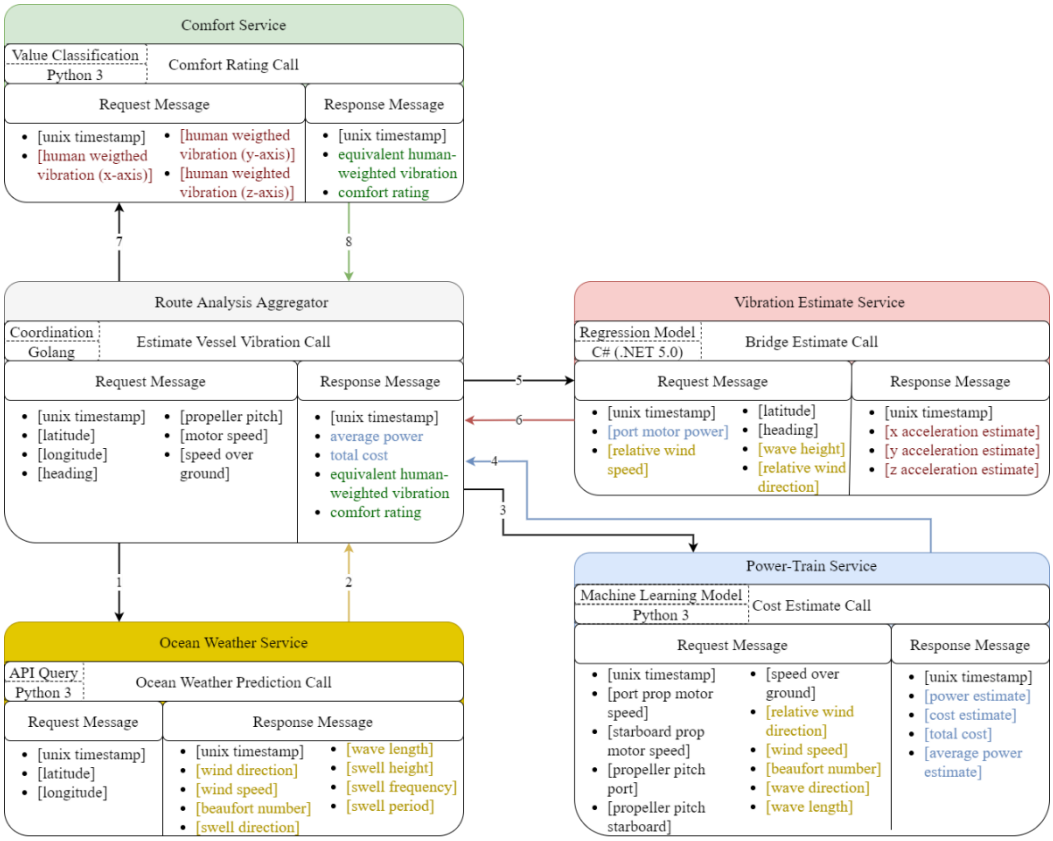



**Figure 21: Route analysis aggregator interface and information flow**

## C.7 Power-Train Aggregator

The power-train aggregator aggregates information to provide the power estimates for the requested routes. It firstly invokes the ocean-weather service to get the predicted weather for the provided route. Relative wind and wave speeds are then derived and combined with the provided sailing configuration. This information is sent with the request to the power-train service, which returns the power estimate and cost estimate along, and total cost for, the requested route. This aggregator provides more granular information about the power-train for a specified route than the route analysis aggregator. The service interface is provided in Figure 23. This information is presented by overlaying the incremental power requirements with the costs incurred by each, as is seen in Figure 13 (a). This provides stakeholders with insight into the cost/power trade-offs at points along the route, giving them the information required to reduce costs if preferable.

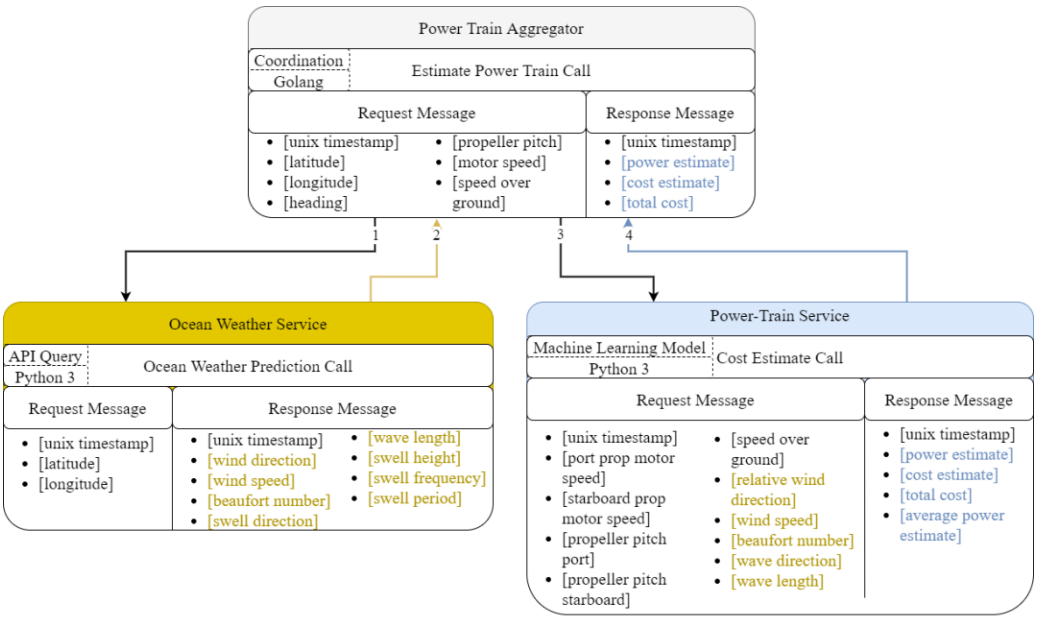



**Figure 23: Power train aggregator interface and information flow**

# C.8 Vessel Vibration Aggregator

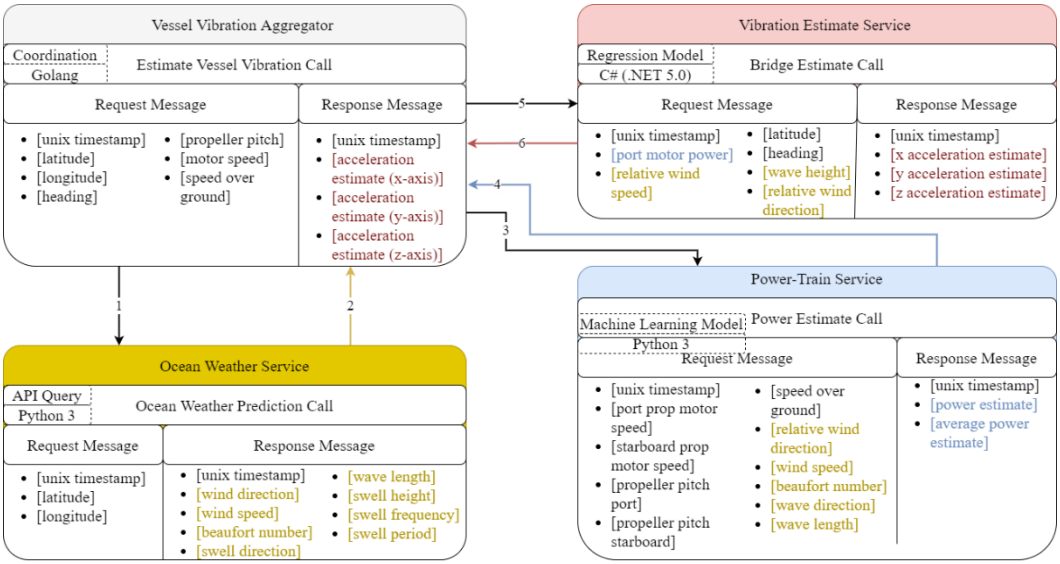



**Figure 22: Vessel vibration aggregator interface**

The vessel vibration aggregator coordinates information among services to provide high-resolution vibration estimates for a specified route. The initial call is, again, to the ocean weather service, followed by a call to the power-train service – the same as for the power-train aggregator; except that in the vessel vibration aggregator, the

power estimate call is invoked instead of the cost estimate call as cost is not necessary for this use case. A combination of information from both services is sent through to the vibration estimate service to produce whole body vibration estimates for the bridge, which is returned for display to the user. The service interface is shown in Figure 22.

## C.9 Web Gateway

The web gateway has been implemented using the Envoy Proxy. The configuration offered by Google for gRPC has been adapted to this case study to route requests to the relevant aggregators and microservices. In addition to this, TLS encryption has been added to the gateway, with a rate limiting filter to limit the number of calls allowed to the greater system. This filter is employed alongside Lyft's rate limit service and a Redis database cache – used to keep track of current connections and perform the rate limit logic. For routes where filters are added, an RPC call is firstly made to the rate limit service to verify that the call can be made according to a pre-defined set of rules, before the request is routed to the relevant backend service.

## C.10 Rate Limit Service

The rate limit service is deployed alongside the web gateway, with a Redis database cache associated to it. This service is an implementation of Lyft's rate limit service, which has been open-sourced for rate limiting using the Envoy proxy. This is a gRPC service that the proxy communicates with before routing requests. Rate limit rules are passed in the query message automatically and the service evaluates whether the request can be sent to down-the-line services based on the rule set and call history provided.

## C.11 Authentication Service

The authentication service offers login functionality, checking the provided user details with a user database. The user database has been implemented using mySQL, with the authentication service implemented in Golang and built on top of the database. Once a user's details have been verified, the authentication service generates a JSON Web Token (JWT) for the user to use in subsequent requests to the system. JWTs are used as the access token in this architecture, taking a transparent token approach. This approach dictates that access tokens are not managed or recorded in the backend, being the sole responsibility of the user that they are issued to once generated. This approach reduces the number of backend calls, as users can be authorised by each service's authorisation interceptor instead of having the tokens verified with a central authority for every service for every call. In this implementation, the tokens are signed using the HS256 algorithm. Golang has been adopted by the financial technology industry as a result of its cryptographic support and is suitable for this service because of this. Golang's database driver has been used such that SQL queries are sanitised when built. This

sanitisation ensures that user inputs cannot be provided such that they comment out the remainder of the SQL statement that the user inputs are being added to, providing protection against SQL injection.

## C.12 Prometheus Server

Prometheus has been selected as the technology for performance monitoring, being an open-source server and database intended specifically for metric collection. The metric interceptors (Listing 2 to Listing 4)were developed to post to the Prometheus server, with Prometheus providing supporting libraries for most popular languages. This server has been configured to use a push gateway, where services push their metrics to the server endpoint. This differs from standard approaches where the services expose their metrics over an endpoint and have the server scrape that endpoint to collect metrics. Administrators are provided access to the metrics on the machine running the server. This server is independent of any services in the system allowing for the data it records to persist the services providing it.

## C.13 Web Frontend

The web gateway leverages gRPC web to make gRPC requests from the browser. This allows the system to be accessed from any device that has access to the internet through a browser. The frontend has been developed using a combination of JavaScript, HTML, and CSS. Users are met with a login page where they will have to log in through before they are provided access to the actual system. When successfully logged in, the user is provided with an access token that is automatically added to subsequent requests – ensuring that authorisation is performed without the user needing to perform further actions. Based on the user's privileges, HTML elements representing the services available to them are loaded dynamically. This ensures that the code cannot be edited in the browser to provide access to services that the user should not have access to. The user can provide inputs when requesting a service, with the errors or responses displayed using either standard HTML elements or GraphJS, where relevant.

# Appendix D Test Procedures

This appendix documents the test procedures followed during the evaluation of the case study. These are mentioned in Section 7.2, with the exact procedures documented here.

For all tests documented below, the implementation was run across two machines. The core services were all run on a Dell XPS 13 boasting 16GB of RAM and a 10$^{th}$ gen Intel I7-1051U (1.8GHz) processor running 64 bit Ubuntu 20.04.3. The propeller monitor and comfort services were run on an 8GB Raspberry Pi 4 running 64 bit Kali Linux.

## D.1 Standard Operations Experiment

### D.1.1 System Stability Test

This test is used to collect the standard operation metrics for this system. This includes latency, packet size, and request volume. This test requires that the system be run for a sufficient time period such that stability is guaranteed. The tester will query the system regularly during this period to simulate vessel passengers looking for information. Additionally, the system should be run in a similar manner as it would be during a deployment, i.e. with occasional drops in internet connection. In the case study that has been designed, the ocean weather service requires an internet connection in order to fetch environmental data. Considering these points, the system will be run with an internet connection. However, the host machine's internet connection will be interrupted at random points during the test. At least one of these interruptions should overlap with a user querying information from the ocean weather service. The raw data from this test can be found in Appendix E.

The test procedure is as follows:

1. Deploy the system with all services as documented in Chapter 6.
2. Inspect all ports on the machines involved in the test to ensure that all expected services were running.
3. Load the user interface and log in as an administrator. Once logged in, all service offerings should be tested to ensure that they are working as intended. This can be done with by sending requests and observing the responses
4. Following the service tests, log into the Prometheus server to ensure that the metrics for all services are being collected.

5. The tester should log back in and query all services on offer at least once every hour, varying the request density each session. Note that the tester should log in with different profiles during the different sessions to verify that the user support is functioning correctly.

6. At the end of the 24 hour test period, all metrics should be downloaded from the Prometheus server.

### D.1.2 Interceptor Benchmark Test

The service with the most consistent (smallest standard deviation in) response times, identified in Test 1, will be used for this test. The interceptor benchmark test provides metrics describing the additional latency added by including each type of interceptor. The service will first be benchmarked without any interceptors added to it. This provides a baseline performance for the service. This baseline will be performed with a control set of requests. The actual content of this control set does not matter as the message structure required by the server's interface dictates a standard format. As long as the content of the messages is kept constant throughout the test consistent results can be produced. Interceptors will be added to the service individually, running the same control set of requests against the service to record the response latency. The raw data from this test is presented in Appendix E.

The steps followed for this test are as follows:

1. Reconfigure the selected service such that it employs no interceptors.

2. Deploy the Prometheus monitoring service.

3. Deploy the "naked" service and run the control set of requests against it. The omission of a metric interceptor necessitates that the client requesting the service records response times. The only additional latency added through this approach is the network/message travel time. Running the client and server on the same machine, however, minimises this contribution as well as providing a standard communication medium such this variation is negligible.

4. With the baseline metrics recorded, pull the server offline and reconfigure it to don only the metric interceptor. Subsequently, relaunch the service and run the control set of requests against it, as was done in Step 2.

5. Repeat step 4 for the authorisation interceptor, retry interceptor, and rate limit interceptor.

6. Repeat steps 1-5 ten times to obtain a stable, averaged result.

7. Download the metrics from the Prometheus server.

### D.1.3 Request Limit Test

Still in the realm of standard operation is the request limit test. This test evaluates whether the system can handle the maximum expected load. This would occur where every pax onboard the S.A. Agulhas II attempts to access the system at once. The SAAII was designed to accommodate 100 passengers and 45 crew members. Considering this, 145 concurrent requests will be made to the system at a single time. The requests will be made from users with varying privileges such that the access control functionality can be verified to remain effective. The results of the test were recorded in Section 7.3.2.

The procedure followed for this test is as follows:

1. Set up three clients to iterate over the request process, creating and sending each request in its own thread (concurrently).

2. Each client should be manually issued an access token with different user claims (privileges).

3. The requests are all sent to the same service in this test, however they could just as well be sent to different services as it is the system being analysed and not a specific service. The service receiving the requests should have its rate limit interceptor modified to allow for this many concurrent requests to be made.

4. Each client should have the anticipated, successful response preloaded such that it can autonomously verify the actual responses. Along with this, logic should be added to notify of a response match or mismatch. This enables the tester to easily confirm whether the requests have been correctly handled.

5. Observe the notifications from the client, looking out for response mismatches.

6. Record results.

7. Repeat steps 2-6 ten times to obtain a stable, averaged result.

## D.2 Forced Failure Experiment

### D.2.1 Failure Isolation Test

This tests the considerations towards robustness to failure in the system, specifically fault isolation. To do this, a service is pulled offline with the system's response documented. The results of this test were recorded in Section 7.3.2.

The steps followed for this test are as followed:

1. Deploy the system with all services as documented in Chapter 6

2. Make a request to a service known to be running on the host machine, monitoring said service's log files. The tester should monitor the logs to confirm once the service begins processing the request.

3. Once the service has begun processing the request, the tester should forcibly shut down the service before it can complete the request.

4. The tester should observe that the client to the removed service is still running. This ensures that the client does not fail if the call does.

5. The tester should observe the log files of the client to ensure that the error is effectively caught.

6. The tester should additionally confirm that all other services in the system are still operational by sending requests to them using a client program (Bloom RPC was used in this thesis).

7. Record results.

8. Repeat steps 1-7 ten times to obtain a stable, averaged result.

9. Repeat steps 1-8 for a service running on a remote machine.

### D.2.2 Service Recovery Test

This test evaluates the autonomy of the system regarding service recovery. For this test, a service is modified such that it starts up correctly but raises a fatal error when receiving a call. By forcing a failure of this type the autonomous recovery of the service can be tested. It is expected that the service restarts itself and the client retries the request. This behaviour will be monitored and confirmed by the tester. This test additionally confirms that the requirement for independent service deployment is met. The results to this test are documented in Section 7.3.2.

The procedure followed for this test is as follows:

1. A selected service running on the host machine should be modified to raise a fatal error when invoked. This allows the service to host itself as expected, failing only when a request is made to it.

2. Make a request to the modified service, with the client donning a retry interceptor. This interceptor employs exponential backoff logic, retrying the call after 100 ms, and waiting an double that time for subsequent failures. The retry limit is set to 5 by default, providing 3.1 seconds of relief.

3. The tester should observer that the service restarts itself autonomously. With the service restarted, the tester should inspect the log files of the client to determine which retry bracket was successful. This allows the tester to determine in which timeframe the service recovered itself.

4. Record results.

5. Repeat steps 1-4 ten times to obtain a stable, averaged result.

6. Repeat steps 1-5 for a service running on a remote machine.

D.2.3 Network Recovery Test

This test evaluates the recovery of the system from a network failure. This is a crucial consideration to ensure a robust distributed system. This test requires a request to a remote server, with the network connection between the client and server dropping during communication. The fault isolation is evaluated by confirming the operation of all other components of the system as is done in test 1. The network connection is re-established, and the behaviour is observed. It is not expected for the failed call to resume, except in cases where the network is reconnected before the client's retry interceptor times out. The successful outcome of this test specifies that the system (specifically, the failed service) can be queried without any manual intervention once the network connection is re-established. This ensures system autonomy where networks may fail or be updated during a voyage. The results of this test are documented in Section 7.3.2.

The test procedure followed was as follows:

1. Deploy the system with all services as documented in Chapter 6.

2. Make a request to the propeller monitor service, with the tester monitoring the service's log files.

3. Once the tester observes that the propeller monitor service has received the request, they should unplug the network cable connecting the two machines.

4. The network cable should be immediately plugged back in, with the response observed. This is done to observe the behaviour of the system where the client's retry interceptor is still relevant.

5. Repeat steps 1-4 ten times to obtain a stable, averaged result.

6. Record results.

7. Repeat steps 1-3.

8. The network cable should only be plugged back in after 10 seconds. This allows sufficient time such that the retry interceptor is no longer relevant.

9. Record results.

10. Repeat steps 7-9 ten times to obtain a stable, averaged result.

## D.3 Security Experiment

### D.3.1 Unauthorised Access Test

This test evaluates the design decision to authorise calls in the service. To perform this test, the system is reconfigured to provide a specific user (guest) the option of requesting information from a service that they do not have access to. This represents a misconfiguration of the gateway and frontend components, or a malicious user forcing access of services through the gateway. It is anticipated that the request be received by the service as no authorisation is performed through the gateway. However, the request should be rejected based on the user not having access permissions to said service. The results of this test are documented in Section 7.3.2.

The procedure followed was:

1. Reconfigure the frontend to offer a guest user access to a service offering that they are not authorised to access.

2. Deploy the system with the new (mis)configuration.

3. Log in as a guest and make a request to the service that you are unauthorised to access to but are offered.

4. Observe an error on the frontend and inspect the log files of the service. The logs should document that the request was rejected based on the user's access token.

5. Record results.

6. Repeat steps 1-5 ten times to obtain a stable, averaged result.

### D.3.2 Gateway Bypass Test

This tests that services cannot be accessed without the client presenting an access token. This would be the case where a user somehow managed to enter the closed network that the system runs on, where they would be able bypass the gateway to query individual services. The expected result is that the request is rejected as was done in Test 1. The results are recorded in Section 7.3.2.

The procedure taken was as follows:

1. Expose a service outside of the closed network. This service should don all interceptors as it would in a full deployment.

2. Using an RPC client (Bloom RPC was used for this test), make a request to the exposed service.

3. Observe the response from the server, it should return an unauthenticated error.

4. Observe the log files of the service to ensure that the request was not processed by the service.

5. Record results.

6. Repeat steps 1-5 ten times to obtain a stable, averaged result.

### D.3.3 Rate Limit Test

This test evaluates the explicit considerations against DOS attacks. For this test, the client used in the request limit test is reused to launch multiple, concurrent requests. The system is reconfigured to limit the number of requests to less than the number sent by the client, with the client recording the number of rejected requests. The server will still be recording latency metrics so that performance reduction can be quantified. The test is performed both at a system and service level. Results can be found in Section 7.3.2 and Appendix E.

The test procedure followed is as follows:

1. Deploy the system with all services as documented in Chapter 6.

2. Modify the client developed in the Request Limit Test to launch twenty requests concurrently. Additionally, change the response code of the client to count the number of failed requests as well as the number of successful requests.

3. Reconfigure the gateway to rate limit to ten calls total.

4. Run the client, observe the response counter. The client should print out that there were ten successful requests and ten unsuccessful requests.

5. Record the results. Additionally, download Prometheus metrics for the system.

6. Repeat steps 1-5 ten times to obtain a stable, averaged result.

7. Reconfigure the gateway to allow for more than twenty requests. Reconfigure the authentication interceptor of the service being targeted to limit to ten calls.

8. Repeat steps 4-6 for the new configuration, downloading only the Prometheus metrics for the involved service.

## D.4 Reconfigurability Experiment

The reconfigurability experiment serves to evaluate the process required to add a new, user-facing, service to a deployed system. The design propose in this research strives to minimise the system knowledge required for contributors. It is fitting, then that contributing to the system should not be a difficult endeavour. This experiment serves to test the modularity of the design, evaluating the requirement for individual service development, deployment and removal (NF0.0, and NF0.1, respectively).

This experiment requires that the system be running as it would be in a deployment environment. From this stable state, the process required to add an existing service to the system is documented placing emphasis on the required code complexity and integration. Once the service is ready to be run within this system, the service needs to be integrated with the deployed system. To do so, the gateway needs to be updated (through its configuration file). This update necessitates that the gateway be pulled offline momentarily with the updated instance relaunching. This process will be done multiple times to evaluate the behaviour of the system during this downtime given different states (gateway receiving requests, processing requests, returning requests). A successful outcome would result if the system performs the same during this downtime as it would without the downtime. This test verifies that individual service deployment is successfully considered, and that the system has been designed in an easily maintainable manner.

### D.4.1 Service Development Test

This test involves converting a program into a service that is ready to be run in this system. The ‘program’ encapsulates the service logic, and the process of running that logic as a service is focussed on here. This is the typical procedure a domain expert would need to follow when contributing to this system. For this test, the number of lines of code requiring changes is recorded, as well as how many different files these changes occur in.

The procedure required to servitise a program is as follows:

1. First, generate the service file structure and boilerplate code using an automated build command

2. Open the 'configuration.yaml’ file and change at least three lines of code

2.1. Update the port that the new service will expose itself on (line 3 in Listing 11).

2.2. Add 1 line per call for the access level ‘name’ field. This is used to match requests in the authorisation interceptor (line 10 in Listing 11).

2.3. Add 1 line per call for the access level ‘role’ field. This is used to evaluate whether a user has access to the matched request (line 12 in Listing 11).

3. Create the .proto file in the “proto/v1” directory.

    3.1. Add 1 line for the package name (3 in Listing 9).

    3.2. Add 1 line for request message definition. Add an extra line for each subsequent message field required (line 5 in Listing 9).

    3.3. Add 1 line for the response message definition. Add an extra line for each subsequent message field required (line 13 in Listing 9).

    3.4. Add at least 1 more line for each additional message required (lines 9 and 17 in Listing 9).

    3.5. Add 1 line defining the service name (line 21 in Listing 9).

    3.6. Add 1 line defining the service call (line 22 in Listing 9). Add an extra line for each subsequent call on offer (line 23 in Listing 9).

4. Open the service file/script.

    4.1. Update 2 lines for protofile imports. This only requires updating the path name (omitted in the figures below as it does not fall into a function).

    4.2. Update 1 line in the metric interceptor integration (line 9 in Listing 12). This change assigns a label to the service when recording metrics.

    4.3. Add 1 line per call in the authorisation interceptor (line 13 in Listing 12). This provides the authorisation interceptor with a key-value pair of request names and request permissions.

    4.4. Update 1 line for the rate limit interceptor specifying how many calls to limit the service to (line 16 in Listing 12).

4.5. Update 1 line for service registration (line 28 in Listing 12). This tells the server which class definition to override, with these classes automatically generated by the proto compiler.

4.6. Copy the service definition from the proto files. This definition is automatically generated for you (Listing 10).

4.7. Add 1 line in the service handler function to create a response (line 5 in Listing 10).

4.8. Add 1 line in the service handler function to return the response (line 5 in Listing 10). Note that step 4.8 and 4.9 are displayed on the same line in Listing 10 for the sake of brevity; in reality, the response message would be created prior to returning it so that the message fields can be populated.

```
1  syntax = "proto3";
2
3  package serviceNameAPI.v1;
4
5  message RequestMessageOneName {
6      // Message fields are specified here
7  }
8
9  message RequestMessageTwoName {
10     // Message fields are specified here
11 }
12
13 message ResponseMessageOneName {
14     // Message fields are specified here
15 }
16
17 message ResponseMessageTwoName {
18     // Message fields are specified here
19 }
20
21 service ServiceName {
22     rpc ServiceCallOneName(RequestMessageOneName) returns (ResponseMessageOneName);
23     rpc ServiceCallTwoName(RequestMessageTwoName) returns (ResponseMessageTwoName);
24 }
```

**Listing 9: Generic proto file**

```yaml
server:
  port:
    myself: "portNumber"
  authentication:
    jwt:
      secretKey: "encryptionKey"
      tokenDuration: 15 # Duration (in minutes) that the token is valid for
    accessLevel:
      name:
        ServiceCallOneName: "/serviceNameAPI.v1.ServiceName/ServiceCallOneName"
      role:
        ServiceCallOneName: ["user1", "user2", "user3"]
```

**Listing 11: Generic service, configuration file**

```python
class ServiceNameServicer(object):

    def ServiceCallOneName(self, request, context):
        # Add service logic here, returning a response message object
        return serverStub.ResponseMessageOneName()

    def ServiceCallTwoName(self, request, context):
        # Add service logic here, returning a response message object
        return serverStub.ResponseMessageTwoName()
```

**Listing 10: Generic service, service class**

```python
def serve():
    ''' This function creates a server with specified interceptors, registers the
    service calls offered by that server, and exposes the server over a specified port.
    '''

    # ________CREATE A SERVER OBJECT________
    # Create interceptor chain
    activeInterceptors = [
        metricInterceptor.MetricInterceptor("ServiceName"),
        authenticationInterceptor.AuthenticationInterceptor(
            config["authentication"]["jwt"]["secretKey"],
            config["authentication"]["jwt"]["tokenDuration"],
            {config["authentication"]["accessLevel"]["name"]["ServiceCallName"]:
            config["authentication"]["accessLevel"]["role"]["ServiceCallName"]}
        ),
        rateLimitInterceptor.RateLimitInterceptor(maximumConnections)
    ] # List containing the interceptors to be chained

    # Create a server to serve calls using 16 workers
    server = grpc.server(
        futures.ThreadPoolExecutor(max_workers=16),
        interceptors = activeInterceptors
    )
    logging.debug("Successfully created server.")

    # ________REGISTER SERVICES TO SERVER________
    # Register an ocean weather service on the server
    service_name_pb2_grpc.add_ServiceNameServicer_to_server(
        ServiceNameServicer(),
        server
        )
    logging.debug("Successfully registered service to server.")

    # ________CREATE A CONNECTION FOR THE SERVER TO HOST ITSELF ON________
    # Load TLS credentials
    creds = loadTLSCredentials("certification")

    # Create a secure connection on port
    serverHost = os.getenv(key = "SERVERHOST", default = "[::]") ''' Receives the
    hostname from the environmental variables (for Docker network), or defaults to
    localhost for local testing
    '''
    try:
        server.add_secure_port(f'{serverHOST}:{config["port"]["myself"]}', creds)
        logging.debug("Succesfully added (secure) port to server.")
    except Exception as e:
        logging.debug(f"Failed to add (secure) port to server: \n{e}")

    # ________LAUNCH THE SERVER________
    try:
        # Start server and listen for calls on the specified port
        server.start()
        logging.info(f'Server started on port {config["port"]["myself"]}')

        # Defer termination for a 'persistent' service
        server.wait_for_termination()
    except Exception as e:
        logging.debug(f'Failed to start server on port {config["port"]["myself:"]}:
        \n{e}')
```

**Listing 12: Generic service, 'serve' function**

### D.4.2 Service Integration Test

This test involves using the service developed in the Service Development Test and integrating it with the deployed system. Note that the system was not initially designed with the goal of dynamic service addition in mind. The original goal was to enable hot-swappable service updates without any system downtime, i.e to reconfigure existing services during a voyage where required. The performance of the system, however, presented interesting opportunities to evaluate dynamic service addition, observing downtime and the system's response to it. In order to add a new service to the system is a simple case of adding the service to the closed network which does not affect any other components of the system as the services are all completely independent. In order to present this new, independent, service offering to the user, however, the gateway needs to be reconfigured. This necessitates that the current instance of the gateway be pulled down with the updated instance being relaunched instead. In reality, this downtime is so small that a user would not notice it. However, a user would notice if their request failed. Thus, the behaviour of existing calls will be evaluated during this downtime. The results of this are documented in Section 7.3.2.

The process of integrating a service with a running system is documented below:

1. Open the "envoy.yaml" configuration file.
2. Add a new route option. This is where how Envoy know how to route requests.
    2.1. Update 1 line for the message prefix of this route (line 13 in Listing 14). This is how envoy matches or identifies an incoming request.
    2.2. Update 1 line for the cluster for this route (line 17 in Listing 14). Once Envoy has matched or identified said incoming request, this is where it routes that request.
3. Add a new cluster. This cluster holds the information about where envoy is routing the request.
    3.1. Update the name of the cluster to match that set in step 2.2 (line 2 in Listing 13).
    3.2. Update the address of the cluster to represent the address space that the service is hosting itself on. This can be the local address, internal Docker address, or the network hostname of a remote service (line 14 in Listing 13).
    3.3. Update the port that the service is exposing itself on (line 15 in Listing 13).
4. Re-build the Envoy Docker image.

5. Pull the current Envoy container offline and re-launch the new one.

```
virtual_hosts:
                    - name: routing_service
                      domains: ["*"]
                      rate_limits:
                        - stage: 0
                          actions:
                            - generic_key:
                                descriptor_value: "global"
                      routes:
                        - {
                            match:
                              {
                                prefix: "/serviceNameAPI.v1.ServiceName",
                              },
                            route:
                              {
                                cluster: generic_service_cluster,
                                timeout: 30s,
                                # Add the global rate limit filter to this call
                                include_vh_rate_limits: true,
                              },
                          }
```

**Listing 14: Envoy configuration, virtual_hosts**

```
clusters:
  - name: generic_service_cluster
    connect_timeout: 0.25s
    type: logical_dns
    http2_protocol_options: {}
    lb_policy: round_robin
    load_assignment:
      cluster_name: cluster_0
      endpoints:
        - lb_endpoints:
            - endpoint:
                address:
                  socket_address:
                    address: genericServiceHostAddress
                    port_value: genericServicePortAddress
```

**Listing 13: Envoy configuration, clusters**

# Appendix E Results

This appendix documents the raw results generated by the experiments of Appendix D. Majority of the results below were extracted from the data recorded by the metric interceptors, which was downloaded from the Prometheus server.

**Table 10: Route analysis aggregator latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] | Client-side request latency [s] |
|---|---|---|---|
| 1 | 2.361533295 | 2.361533295 | 2.38 |
| 2 | 4.524789331 | 2.163256036 | 2.28 |
| 3 | 7.034973034 | 2.510183703 | 2.53 |
| 4 | 9.372878725 | 2.337905691 | 2.35 |
| 5 | 12.196905886 | 2.824027161 | 2.84 |
| 6 | 14.073475482 | 1.876569596 | 2.4 |
| 7 | 16.132097777 | 2.058622295 | 2.37 |
| 8 | 18.571937616 | 2.439839839 | 2.46 |
| 9 | 21.098379991 | 2.526442375 | 2.55 |
| 10 | 22.77029045 | 1.671910459 | 2.3 |

**Table 11: Power train aggregator latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] | Client-side request latency [s] |
|---|---|---|---|
| 11 | 1.826012551 | 1.826012551 | 1.9 |
| 12 | 3.565061844 | 1.739049293 | 1.75 |
| 13 | 4.948039508 | 1.382977664 | 1.7 |
| 14 | 6.526804434 | 1.578764926 | 1.65 |
| 15 | 7.908688122 | 1.381883688 | 1.602 |
| 16 | 9.215126543 | 1.306438421 | 1.62 |
| 17 | 10.723540284 | 1.508413741 | 1.54 |
| 18 | 12.350896179 | 1.627355895 | 1.64 |
| 19 | 13.926617598 | 1.575721419 | 1.59 |
| 20 | 15.581704508 | 1.65508691 | 1.66 |

**Table 12: Vessel vibration aggregator latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] | Client-side request latency [s] |
|---|---|---|---|
| 21 | 2.402917585 | 2.402917585 | 2.48 |
| 22 | 4.026829691 | 1.623912106 | 2.04 |
| 23 | 5.453413511 | 1.42658382 | 1.9 |
| 24 | 7.522794904 | 2.069381393 | 2.08 |
| 25 | 8.768254816 | 1.245459912 | 1.77 |
| 26 | 10.124946209 | 1.356691393 | 1.72 |
| 27 | 11.953640754 | 1.828694545 | 1.85 |
| 28 | 13.558839418 | 1.605198664 | 1.698 |
| 29 | 15.580809639 | 2.021970221 | 2.04 |
| 30 | 17.601399086 | 2.020589447 | 2.04 |

**Table 13: Comfort service latency (remote)**

| Request ID | Cumulative request latency [s] | Request latency per call [s] |
|---|---|---|
| 1 | 0.00195956230163574 | 0.001959562301636 |
| 2 | 0.00318455696105957 | 0.001224994659424 |
| 3 | 0.00572299957275391 | 0.002538442611694 |
| 4 | 0.0074770450592041 | 0.00175404548645 |
| 5 | 0.00891613960266113 | 0.001439094543457 |
| 6 | 0.0105624198913574 | 0.001646280288696 |
| 7 | 0.0117754936218262 | 0.001213073730469 |
| 8 | 0.0140478610992432 | 0.002272367477417 |
| 9 | 0.0152480602264404 | 0.001200199127197 |
| 10 | 0.0164375305175781 | 0.001189470291138 |

**Table 14: Comfort service latency (local)**

| Request ID | Cumulative request latency [s] | Request latency per call [s] |
|---|---|---|
| 1 | 0.000347852706909 | 0.000347852706909 |
| 2 | 0.0006945133209228516 | 0.000346660614014 |
| 3 | 0.00101470947265625 | 0.000320196151733 |
| 4 | 0.0013153553009033203 | 0.000300645828247 |
| 5 | 0.00160980224609375 | 0.00029444694519 |
| 6 | 0.0019791126251220703 | 0.000369310379028 |
| 7 | 0.002270936965942383 | 0.00029182434082 |
| 8 | 0.002644777297973633 | 0.000373840332031 |
| 9 | 0.0030362606048583984 | 0.000391483306885 |
| 10 | 0.003438472478027344 | 0.000402212141 |

**Table 15: Authentication service latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] |
|---|---|---|
| 1 | 0.210221521 | 0.210221521 |
| 2 | 0.389366484 | 0.179144963 |
| 3 | 0.582184472 | 0.192817988 |
| 4 | 0.798584816 | 0.216400344 |
| 5 | 1.01954035 | 0.220955534 |
| 6 | 1.124142274 | 0.104601924 |
| 7 | 1.346817525 | 0.222675251 |
| 8 | 1.580945182 | 0.234127657 |
| 9 | 1.802512729 | 0.221567547 |
| 10 | 2.017874745 | 0.215362016 |
| 11 | 2.194310619 | 0.176435874 |
| 12 | 2.389455358 | 0.195144739 |
| 13 | 2.683609029 | 0.294153671 |
| 14 | 2.897722194 | 0.214113165 |
| 15 | 3.108086534 | 0.21036434 |
| 16 | 3.301726905 | 0.193640371 |
| 17 | 3.523337678 | 0.221610773 |
| 18 | 3.724208388 | 0.20087071 |
| 19 | 3.972768943 | 0.248560555 |
| 20 | 4.21270021 | 0.239931267 |
| 21 | 4.42402428 | 0.21132407 |
| 22 | 4.667611809 | 0.243587529 |
| 23 | 4.893179804 | 0.225567995 |
| 24 | 5.126438674 | 0.23325887 |
| 25 | 5.35874871 | 0.232310036 |
| 26 | 5.575922523 | 0.217173813 |
| 27 | 5.812214529 | 0.236292006 |
| 28 | 5.92660355 | 0.114389021 |
| 29 | 6.159280489 | 0.232676939 |
| 30 | 6.390500895 | 0.231220406 |

**Table 16: Ocean weather service latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] |
|---|---|---|
| 1 | 1.29532504081726 | 1.29532504081726 |
| 2 | 2.87037992477417 | 1.57505488395691 |
| 3 | 4.65417695045471 | 1.78379702568054 |
| 4 | 6.41064429283142 | 1.75646734237671 |
| 5 | 8.65156841278076 | 2.24092411994934 |
| 6 | 9.78731989860535 | 1.13575148582459 |
| 7 | 11.2693922519684 | 1.48207235336305 |
| 8 | 13.1938104629517 | 1.48207235336305 |
| 9 | 15.1854875087738 | 1.9916770458221 |
| 10 | 16.2334952354431 | 1.0480077266693 |
| 11 | 17.5952196121216 | 1.3617243766785 |
| 12 | 18.995772600174 | 1.4005529880524 |
| 13 | 19.936910867691 | 0.941138267517001 |
| 14 | 21.0939555168152 | 1.1570446491242 |
| 15 | 22.0187573432922 | 0.924801826477001 |
| 16 | 23.0000140666962 | 0.981256723403998 |
| 17 | 24.0042028427124 | 1.0041887760162 |
| 18 | 25.2457418441772 | 1.2415390014648 |
| 19 | 26.3478558063507 | 1.1021139621735 |
| 20 | 27.4698433876038 | 1.1219875812531 |
| 21 | 29.4647974967957 | 1.9949541091919 |
| 22 | 30.6727859973907 | 1.207988500595 |
| 23 | 31.6648359298706 | 0.992049932479898 |
| 24 | 33.2741410732269 | 1.6093051433563 |
| 25 | 34.2037889957428 | 0.929647922515898 |
| 26 | 35.1744561195374 | 0.970667123794605 |
| 27 | 36.5849206447601 | 1.4104645252227 |
| 28 | 37.6711971759796 | 1.0862765312195 |
| 29 | 39.2684338092804 | 1.5972366333008 |
| 30 | 40.6126866817474 | 1.344252872467 |

**Table 17: Power train service latency**

| Request ID | Cumulative request latency [s] | Request latency per call [s] |
|---|---|---|
| 1 | 0.378655195236206 | 0.378655195236206 |
| 2 | 0.619446277618408 | 0.240791082382202 |
| 3 | 0.839204072952271 | 0.219757795333863 |
| 4 | 1.16547274589539 | 0.326268672943119 |
| 5 | 1.39550733566284 | 0.23003458976745 |
| 6 | 1.59398651123047 | 0.19847917556763 |
| 7 | 1.90351963043213 | 0.30953311920166 |
| 8 | 2.11553430557251 | 0.21201467514038 |
| 9 | 2.30516934394836 | 0.18963503837585 |
| 10 | 2.55764961242676 | 0.2524802684784 |
| 11 | 2.79433917999268 | 0.23668956756592 |
| 12 | 2.99378323554993 | 0.19944405555725 |
| 13 | 3.25463080406189 | 0.26084756851196 |
| 14 | 3.50545740127563 | 0.25082659721374 |
| 15 | 3.78313684463501 | 0.27767944335938 |
| 16 | 3.98943448066711 | 0.2062976360321 |
| 17 | 4.31889295578003 | 0.32945847511292 |
| 18 | 4.53442811965942 | 0.21553516387939 |
| 19 | 4.84532999992371 | 0.310901880264289 |
| 20 | 5.05721092224121 | 0.2118809223175 |
| 21 | 5.26983880996704 | 0.21262788772583 |
| 22 | 5.51028990745544 | 0.2404510974884 |
| 23 | 5.74655318260193 | 0.236263275146491 |
| 24 | 6.02551031112671 | 0.278957128524779 |
| 25 | 6.20798873901367 | 0.18247842788696 |
| 26 | 6.45428419113159 | 0.24629545211792 |
| 27 | 6.67841720581055 | 0.22413301467896 |
| 28 | 7.01321482658386 | 0.334797620773309 |
| 29 | 7.25205588340759 | 0.23884105682373 |
| 30 | 7.52893612861633 | 0.27688024520874 |

## Propeller monitor service

Results are not able to contribute to evaluation (the response time that was recorded indicated the time required for the server to acknowledge the response, not the time used to process the request. As such they can't be used to extrapolate data).

## Gateway

Data describing the gateway latency was extrapolated by subtracting request latency of aggregators from the associated client-side call time (as recorded by the browser network monitor).

**Table 18: Gateway latency**

| Request ID | Request latency per call [s] |
|---|---|
| 1 | 0.018466705 |
| 2 | 0.116743964 |
| 3 | 0.019816297 |
| 4 | 0.012094309 |
| 5 | 0.015972839 |
| 6 | 0.523430404 |
| 7 | 0.311377705 |
| 8 | 0.020160161 |
| 9 | 0.023557625 |
| 10 | 0.628089541 |
| 11 | 0.073987449 |
| 12 | 0.010950707 |
| 13 | 0.317022336 |
| 14 | 0.071235074 |
| 15 | 0.220116312 |
| 16 | 0.313561579 |
| 17 | 0.031586259 |
| 18 | 0.012644105 |
| 19 | 0.014278581 |
| 20 | 0.10491309 |
| 21 | 0.077082415 |
| 22 | 0.416087894 |
| 23 | 0.47341618 |
| 24 | 0.010618607 |
| 25 | 0.524540088 |
| 26 | 0.363308607 |
| 27 | 0.021305455 |
| 28 | 0.092801336 |
| 29 | 0.018029779 |
| 30 | 0.019410553 |

Table 19: Interceptor benchmark test latencies

| Request latency for naked service [s] | Request latency for metric interceptor only [s] | Request latency for authentication interceptor only [s] | Request latency for rate limit interceptor only [s] |
|---|---|---|---|
| 0.002262036 | 0.03311139 | 0.002197871 | 0.002558613 |
| 0.002065373 | 0.008264827 | 0.002492698 | 0.002006417 |
| 0.001879877 | 0.00724314 | 0.002428657 | 0.001831754 |
| 0.00185192 | 0.007141475 | 0.00257878 | 0.001666799 |
| 0.002021791 | 0.006376783 | 0.002626695 | 0.001875669 |
| 0.001925043 | 0.008294034 | 0.002531447 | 0.001782087 |
| 0.001736588 | 0.006581529 | 0.002302119 | 0.00196225 |
| 0.001939543 | 0.00638695 | 0.002165371 | 0.00205579 |
| 0.001903918 | 0.008425824 | 0.002115789 | 0.002115955 |
| 0. 00181692 | 0. 006501906 | 0. 002467573 | 0. 001707131 |

**Table 20: Rate limit test results**

| Rate limit setpoint | Number of requests sent | Number of requests processed | Implementation/configuration |
|---|---|---|---|
| 25 | 100 | 100 | Python ('None' workers) |
| 25 | 100 | 100 | Python (10 workers) |
| 17 | 100 | 100 | Python (10 workers) |
| 17 | 100 | 100 | Python (12 workers) |
| 17 | 100 | 17 | Python (20 workers) |
| 17 | 100 | 20 | Python (20 workers) |
| 17 | 100 | 18 | Python (20 workers) |
| 14 | 100 | 14 | Python (20 workers) |
| 14 | 100 | 14 | Python (20 workers) |
| 14 | 100 | 14 | Python (20 workers) |
| 14 | 100 | 14 | Python (16 workers) |
| 14 | 100 | 14 | Python (16 workers) |
| 14 | 100 | 14 | Python (16 workers) |
| 25 | 100 | 25 | Golang |
| 25 | 100 | 25 | Golang |
| 17 | 100 | 17 | Golang |
| 17 | 100 | 17 | Golang |
| 14 | 100 | 14 | Golang |
| 25 | 100 | 25 | Envoy (system-wide) |
| 25 | 100 | 25 | Envoy (system-wide) |
| 25 | 100 | 25 | Envoy (system-wide) |
| 17 | 100 | 17 | Envoy (system-wide) |
| 17 | 100 | 17 | Envoy (system-wide) |
| 14 | 100 | 14 | Envoy (system-wide) |
| 14 | 100 | 14 | Envoy (system-wide) |

**Table 21: Resource requirements of case study services**

| Service name | Dormant service | | Active service | |
|---|---|---|---|---|
| | Maximum CPU usage (%) | Memory usage (MB) | Maximum CPU usage (%) | Memory usage (MB) |
| Rate limit service | 0.22 | 5.109-5.121 | 0.28 | 5.648 |
| Vibration estimate service | 0.01 | 22.86 | 1.42 | 27.84 |
| Propeller monitor service | 0.14 | 63.23 | 51.7 | 110.7 |
| Ocean weather service | 0.16 | 26.65 | 5.73 | 28.61 |
| Comfort service | 0.16 | 17.68 | 0.2 | 18.84 |
| User database | 0.35 | 411.1 | 0.42 | 422.9 |
| Vessel vibration aggregator | 0.01 | 3.289 | 2.56 | 6.555 |
| Web frontend | 0.01 | 17.67 | 0.01 | 23.69 |
| Power-train aggregator | 0.01 | 3.254 | 6.41 | 4.129 |
| Authentication service | 0.01 | 2.852 | 12.28 | 4.023 |
| Redis database | 0.28 | 2.094 | 0.33 | 2.7293-2.797 |
| Prometheus | 0.513 | 35.31 | 0.513 | 36.45 |
| Envoy | 0.41 | 23.18 | 1.24 | 23.79 |
| Route comparison aggregator | 0.01 | 2.988 | 2.78 | 7.102 |
| Power-train aggregator | 0.13 | 194.8 | 49.4 | 205.4 |

**Table 22: Container build times for case study services**

| Service name | Initial build time (s) | Cached build time (s) |
|---|---|---|
| Rate limit service | N/A | N/A |
| Vibration estimate service | 104.037 | 5.905 |
| Propeller monitor service | 90.017 | 5.655 |
| Ocean weather service | 43.247 | 11.171 |
| Comfort service | 25.691 | 5.845 |
| User database | 88.147 | 5.757 |
| Vessel vibration aggregator | 75.242 | 6.048 |
| Web frontend | 334.166 | 7.701 |
| Power-train aggregator | 56.887 | 5.638 |
| Authentication service | 42.513 | 5.682 |
| Redis database | N/A | N/A |
| Prometheus | 6.979 | 5.466 |
| Envoy | 58.629 | 5.73 |
| Route comparison aggregator | 55.571 | 5.205 |
| Power-train service | 431.508 | 5.428 |