Universidade de Aveiro

2021

Dimitri Alexandre da Silva

Plataforma de serviços para monitorização da cadeia de valor do pescado

Middleware for fish value chain treaceability

Universidade de Aveiro
2021

Dimitri Alexandre da Silva

Plataforma de serviços para monitorização da cadeia de valor do pescado

Middleware for fish value chain treaceability

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor Cláudio Jorge Vieira Teixeira, Diretor dos Serviços de Tecnologias da Informação e Comunicação da Universidade de Aveiro.

Tese escrita no âmbito do projeto mobilizador ValorMar - Valorização integral dos recursos marinhos: potencial, inovação tecnológica e novas aplicações”, especificamente no seu PPS 4 cofinanciado pelo Fundo Europeu de Desenvolvimento Regional (FEDER) no âmbito do Programa Portugal 2020, financiado pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional Competitividade e Internacionalização (COMPETE 2020)

o júri

presidente	Professor Doutor José Alberto dos Santos Rafael Professor Associado da Universidade de Aveiro
	Professor Doutor Fernando Joaquim Lopes Moreira Professor Catedrático da Universidade Portucalense Infante D. Henrique
	Doutor Cláudio Jorge Vieira Teixeira (Orientador) Diretor de Serviços da Universidade de Aveiro

agradecimentos

Ao Doutor Cláudio Teixeira, pelo acompanhamento durante o projeto e dissertação, mostrando sempre disponibilidade para ajudar e fornecer recursos quando possível.

Aos membros do consórcio Valormar por viabilizarem este projeto e terem criado esta oportunidade.

Aos colegas do projeto, pelos desafios criados no ambiente colaborativo, pelo apoio a resolver esses desafios e pelos momentos de convívio.

Aos membros dos STIC, que partilharam conhecimento e experiências e acolheram a equipa do projeto durante o desenvolvimento.

À família, pelas oportunidades criadas desde o início da minha educação, que acompanharam sempre no que podiam e incentivaram sempre a continuar o percurso académico.

Aos restantes professores, colegas, amigos e outros membros da comunidade académica de Aveiro, onde me integrei durante a licenciatura e mestrado por me acolherem e participarem nesta fase da minha vida.

palavras-chave Middleware, Rastreabilidade, Micro-serviços, Contentores

resumo

A rastreabilidade na cadeia de valor alimentar é um tema de interesse pelas vantagens que traz aos consumidores, produtores e autoridades reguladoras. Esta dissertação descreve as minhas contribuições durante a conceção e implementação de um middleware baseado em micro-serviços para a cadeia de valor do pescado portuguesa considerando as práticas atuais da indústria e os requisitos das partes interessadas envolvidas no projeto, com o objetivo de integrar toda a informação de rastreabilidade disponível de cada um dos operadores para fornecer aos clientes a história completa dos produtos que adquirem. Durante este projeto, assumi muitas funções, como desenvolvimento, operações e até mesmo alguma segurança, o que me permitiu melhorar as minhas capacidades em todos essas disciplinas e experimentar as mais recentes tecnologias nativas da nuvem, como contentores e práticas de DevOps.

keywords Middleware, Traceability, Micro-services, Containers

abstract

Traceability in the food value chain is a topic of interest due to the advantages it brings to both the consumers, producers and regulatory authorities. This thesis describes my contributions during the design and implementation of a microservice based middleware for the Portuguese fish value chain considering current practices in the industry and the requirements of the stakeholders involved in the project, with the goal of integrating all the traceability information available from each operator to provide customers with the full story of the products they purchase. During this project I assumed many roles such as development, operations and even some security allowing me to improve my skills in all these fields and experimenting with the latest cloud native technologies such as containers and with DevOps practices.

Index

Figure Index

Table Index

List of Acronyms

AM	WSO2 API Manager
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CD	Continuous Delivery
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
DNS	Domain Name System
EI	WSO2 Enterprise Integrator
EU	European Union
FaaS	Function-as-a-service
HA	High Availability
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IPVC	Politechnic Institute of Viana do Castelo
IaaS	Infrastructure-as-a-Service
IS	Identity Server

IT	Information Technology
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
PaaS	Platform-as-a-Service
MI	WSO2 Micro Integrator
MQTT	Message Queuing Telemetry Transport
MSOA	Microservice-oriented Architecture
NFC	Near Field Communications
NFS	Network File Server
RAM	Random-Access Memory
REST	Representational State Transfer
RFID	Radio-frequency Identification
ORM	Object-Relational Mapper
SOA	Service-oriented Architecture
SPA	Single-page Application
SSO	Single Sign On
STOMP	Simple Text Oriented Messaging Protocol
TFS	Microsoft’s Team Foundation Server
UA	Universidade de Aveiro
UI	User Interface
VM	Virtual Machine
VPN	Virtual Private Network

1 Introduction

In recent years there has been a lot of attention drawn towards the safety and quality standards of the food we consume due to the industrialization of the production processes that difficult transparency by having a high complexity on the structure of the value chains with a wide variety of stakeholders. Reported incidents and safety concerns made traceability systems a necessity to improve customer trust.

Portugal is the holder of one of the biggest Exclusive Economic Zones in the world coming in at 11th place with 1 721 751 km^2[1] which makes fishing a prominent activity in the country which enables consortiums like Valormar[2] to apply new technologies to add value and improve the efficiency to the sea food value chain.

The PPS4 subproject of the Valormar consortium described in this document has the goal of digitizing that value chain to gather as much traceability related information about the products along with static descriptions with the main goal of enhancing customer experience in the stores during the purchase of the final products displayed on the shelf with more transparency and helpful information.

1.1 Project Context

To digitize the value chain the proposal was to create a middleware with APIs to collect traceability information and enable the tracing of product lots over the value chain. This middleware must be versatile and prepared to integrate with a very broad range of operators, some with digital systems already in place that track their inventories and others with only physical records.

This project is being developed in collaboration with other universities and companies that participate in the Portuguese fish value chain giving us a case study, namely Docapesca[3] and Sonae[4]. Docapesca handles product registration at the start of the chain and interaction with the producers while Sonae participates closer to the consumer in the value chain being present both in distribution and consumer sales.

Both companies operate at a national level and the final goal of this project is to enable the collection of data at all stages of the value chain and to make this information available to any interested party that can be end users which will benefit from them during and after the purchase of the products as well as the companies that participate in the chain which will have the possibility to gain new insights from the collected data.

Other components in the PPS4 subproject that will interact with the middleware are being developed by other entities such as an in-store kiosk by IPVC[5] and operator integration components by FlowTech[6], formelly FoodInTech which works with transformation operators and other middlemen in the value chain.

1.2 Thesis Goals

For this thesis, my goals are to describe the design and implementation phases of the middleware with in-depth technical details of the components I developed or installed. Besides the development responsibilities of the middleware, I was also responsible for deploying and maintaining the staging environment for development use which allowed me to learn a bit regarding cloud computing and system administration. To complement my development tasks on the project with the knowledge gained while maintaining the system I included in my work automating the deployment process in cloud infrastructures and load tests to validate that deployment.

1.3 Working environment

The middleware was developed by a team of research fellows at the University of Aveiro working under the supervision of Doctor Cláudio Teixeira. The team was composed of two backend developers, a front-end developer, a tester and a biologist. I was one of the backend developers and my tasks started with designing a final segmented domain model for the middleware and then during the implementation phase I developed the information and user domains, cross cutting features and inter-domain communications with some sparse contribution to the development of the other domains.

Other responsibilities included integrating all the components together into a staging environment for all the developers and creating continuous integration pipelines. I joined the project in the design phases and have been contributing to the development phase which is the current stage of the project at the time of writing this document.

1.4 Document Structure

After the introductory section, this document contains a chapter with a literature and technology review of systems related to food traceability and industry standards currently in practice around the world and a brief overview of the technologies used by the operators in the Portuguese fish value chain.

Next comes a chapter covering the design phase of the middleware with an analysis of the previous prototype and the new proposed solution with the change of architecture which is then followed by a development chapter which contains in depth detail of the components I developed with a brief explanation of the remaining components and their integration with each other.

There is a dedicated chapter for the deployment and configuration of all the tools necessary for the system’s operation and the load tests used to validate that automated environment.

The chapter before the conclusion explains all the working methodology of the middleware team with agile techniques and the CI/CD pipelines developed for running tests and the staging environment.

2 State of the Art

During this chapter, a brief review of the existing research regarding food traceability systems will be made to establish a comparison with the current state of traceability in the value chains to be integrated in the middleware. Two main groups of studies were reviewed: studies related to the regulations currently enforced in the sector which relate to or force the implementation of traceability and the impact of this traceability on the businesses or consumers and studies focused on implementing traceability in specific cases or focused on the technologies used like blockchain or IoT related projects.

Furthermore, since the architecture type of the solution was already decided to be microservice-oriented and event-driven a review of the current technologies applied that could be interesting to use was done continuously during the development of the project.

2.1 Traceability studies – regulations, standards and social impact

On a global scale there clear differences in the regulations regarding traceability as reviewed[7] by Charlebois S. et al. Although Portugal was not included in that study but as it is a European Union (EU) member state its regulations are like the ones from the other EU countries included. The first big push regulation wise that forced traceability systems to be gradually implemented was EU regulation (EC) No. 178/2002 which laid down the general principles and requirements for food law, also establishing the European Food Safety Authority and procedures to follow to enable food safety. This law requires food related operators to: (1) be able to identify the source and destination of a product; (2) have systems and procedures in place to provide this information to the relevant authorities.

Aung and Chang' study[8] defines traceability and why it is becoming a major requirement to ensure high food quality and security standards due to the increase of consumer demand for this type of information after the first major incidents like the Bovine Spongiform Encephalopathy (BSE) outbreak, commonly known as the Mad Cow disease.

Even with the current laws in place and the improvements to the traceability infrastructure over the 2000s decade, issues are still common and new issues are starting to arise with the constant rise of demand for food commanded by the population growth. The 2013 horsemeat scandal was responsible for once again placing the customer’s focus on food traceability as this time the incident was related to fraudulent products which greatly damaged the consumer’s confidence in the supply chains, as reported by this article[9].

The current and future issues regarding food traceability are documented by King, T. et al in their study[10] that relates the global trends to the developments in food safety and conclude that issues like climate change, a growing and aging population, urbanization, and an increased affluence will difficult the current food safety challenges and create new demands on the industry participants. The rising issues singled out in the document are rising antibiotic resistance in bacteria, foodborne viruses, chemical contamination, economically motivated adulteration of food, allergens and intolerances, presence of nanoparticles and genetically modified food. These issues can be mitigated with advancements in science and the ones mentioned in the document most relevant to this dissertation are related to big data and traceability tools. The authors state that the establishment of a Big Data culture which would allow for the processing of the high volume of information captured by global supply chains to improve decision-making, insight discovery and optimization of processes. The advancements regarding traceability tools are related to the collection of information relevant to the informatics field are the employment of technologies like RFID[11] and NFC[12] and the improvements of the integration of information between businesses.

Regarding the technological advancement of the traceability in the fishing industry, Hardt. Marah J. et al reported in their study[13] the challenges of implementing chain wide traceability stating that, among other things, the lack of resources and the perception that traceability only provides social benefits and not economic ones are big constraints preventing more companies from pursuing such systems. The same study reports that only 40% of the producers inquired have traceability systems which is a big barrier to chain wide traceability as in many cases the first step of the product’s lifecycle is missing.

Last, but not least, he social impacts regarding traceability or lack thereof we have the example of the COVID-19 pandemic which was caused by a virus who’s likely zoonotic origin being a wet market in Wuhan, China, as reported[14] by Rothan, H. A., & Byrareddy, S. N. This incident has had a very wide impact on a global scale regarding the population’s health, the environment and the economy[15] as well as social impacts such as the negative effect it had on education[16], forcing hasty transitions to remote learning. This pandemic once again led to a raise in awareness of the customers who according to a study[17] in China have displayed and increase in food safety knowledge and practices.

2.2 Traceability studies – technological approaches

Regarding the more practical examples of traceability in the supply chains, several studies and projects have appeared in the recent years.

A. Kassahun et al. provide a reference architecture[18] for chain wide collaborative transparency systems using the EPCIS[19] standard based on a Service Oriented Architecture[20]. The authors that proposed the reference architecture placed heavy focus on the stakeholders that interact with the proposed system identifying consumers, chain operators and third-party service providers with an added focus to the chain operators which they divided in multiple tiers depending on the technology level or their internal traceability systems with the main challenge being the integration of all these different sources of information into the same system.

Regarding the EPCIS standard, it belongs to GS1 and its main goal is to allow the sharing of information across and within enterprises with the final goal being to create the full history of object as it travels through the business process steps. Most of the systems of the operators our test case is trying to integrate in the first versions of the middleware are not compliant with any standard so we are not implementing it for now, but in the future the platform can be made EPCIS compliant. In our case study the level of complexity and state of technology of these existing systems range from paper based records to retail industry standard software solutions by companies such as SAP[21] with some cases who have their own internally developed systems.

In the fish industry this example[22] implemented the traceability of live fish along a supply chain using RFID tags to collect information during the farming and transportation of the living organisms.

A practical example[23] of a chain wide system similar to the one described in this document was developed for the olive oil supply chain with the goal of using a web application to allow for customer access to the traceability information.

A lot of the more recent studies are focused on blockchain technology for supply chain collaboration with this paper[24] describing how integration could be achieved with this technology. While a blockchain does add many benefits to a collaborative traceability system regarding the integrity of the information collected and the decentralized aspect of the mode of operation, the stakeholders decided not to pursue such type of system due to the concerns over the higher complexity of implementation taking into consideration the existing systems that would be integrated in the middleware.

2.3 *Microservice based architectures*

As stated in the introduction section, the choice of architecture type for the middleware was a microservice oriented architecture. According to Sill, Allan's publication[25], microservices follow the old approach from service oriented architectures of splitting services into functions that can interact via programming interfaces but with some improvements, driven by the goals of rapid, interchangeable, adaptable and scalable components. This offers more flexibility during development over the more formal implementation of SOAs. The type of architecture also goes along well with the current container-based cloud environments and places a strong emphasis on the use of RESTful APIs for messaging.

According to Indrashiri and Siriwardena's book[26], SOAs appeared to combat the drawbacks of monolithic applications by segregating the functionalities into reusable and loosely coupled services usually segmented around the entities of the system with a Service Bus layer to centralize those features for the consuming applications that can also be responsible for composing services together to create more complex capabilities and the cross cutting features such as security. This centralized Service Bus remained a monolithic entity which all the developers would share to integrate their services. Due to the increase in complexity of the business capabilities required over time, the Service Bus layer became a hinder to development mostly due to the inter-service dependencies.

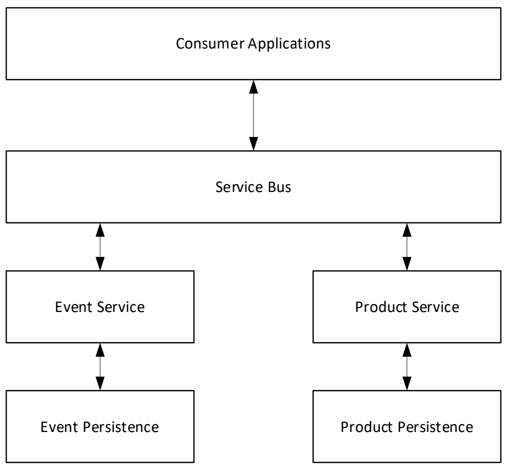


Figure 1 SOA Architecture example

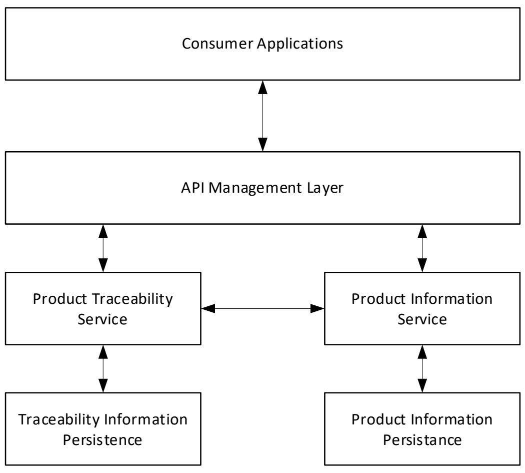

Consumer Applications
API Management Layer
Product Traceability Service
Product Information Service
Traceability Information Persistence
Product Information Persistance

Figure 2 MSOA Architecture example

To overcome this, in a microservice oriented architecture the responsibilities of the service bus are contained in each service so that they take care of the inter-service communications and composition with an API Management Layer taking over the remaining cross cutting features. The resulting microservices each offer their own well-defined business capability developed and deployed independently with corresponding interfaces to expose the information to the other microservices and to the outside consumers.

2.4 Technology review

Many tools are required for the development and deployment of a microservice oriented system, and a lot of the choices were already made when I joined but a technology review as still done to evaluate these choices.

For the API Management Layer, the API Manager from the WSO2[27] product family was chosen due to being open source, installable on premises and compatible with other products from the family such as the Identity Server. Most of the current solutions are offered only as a service like IBM's API Connect[28] or Azure's API Management[29] and could not be used in the context of this project due to the private cloud resources allocated. Other tools like Apigee[30] and 3Scale[31] were not considered due to their price as there was a preference for open-source tools. A valid alternative for WSO2 would be Kong[32] which presents similar features but is not as customizable as WSO2. The final reason for choosing WSO2 was its presence in other department projects which would enable the sharing of knowledge between teams.

Regarding the inter-service communication and some other messaging needs, a message broker or streaming platform was needed and, in this case, the most suitable was Apache Kafka[33], mostly due to its performance and the APIs it provides, like the Connect API which has many connectors available for the creation of data pipelines to and from a lot of different database engines. Alternatives such as ActiveMQ[34] or RabbitMQ[35] use higher level messaging protocols such as AMQP[36], STOMP[37] or MQTT[38] which makes it more interoperable than Kafka which uses plain TCP for communications but at the cost of having more overhead which makes them less efficient and performant.

For microservice development many frameworks are available. The one proposed for the development of the internal applications was .Net Core[39] which is the cross-platform successor to Microsoft's .Net Framework, with one of the main reasons being that the technological leader in the project is specialized in Microsoft tools. It is open source and supports the F# and C# languages and one of its main advantages over the others is the async programming instructions that allow for a better performing application. Alternatives such as JakartaEE[40] and Spring[41] based on Java are also very good offering more abstraction and with more community made libraries available. Another advantage of Java is that it is cross-platform due to the JVM based execution but nowadays C# can also be compiled for most environments. Other alternatives for microservice developments have also been popping up such as FaaS platforms like Azure Functions[42] or AWS

Lambda[43] which claim to substantially decrease development time with a serverless model with the drawback of becoming perhaps too dependent on the providers and sometimes limited to what other components can be used for storage and integration to what that specific provider has available.

For the persistence needs of each microservice a lot of different types of databases are available and, in each case, the most suitable one will be evaluated. A preference for Microsoft's SQL Server[44] was established in all the cases where a relational database engine is required unless it is more convenient to use another if any compatibility problems arise. Once again, this choice was imposed due to the familiarity with this software of our partner entities. In some cases MySQL[45] for compatibility purposes and in other cases non-relational engines were used. These non-relational engines were Neo4j[46], which a graph database and MongoDB[47], a document based database. The choice of these non-relational databases was made by the developers of the components that make use of them. In the case of MongoDB the main reason was the bigger flexibility regarding the data structure of relational engines. Neo4j was adopted as an experiment to store traceability information.

Regarding containers for the packaging and deployment of the applications, Docker[48] is currently the industry standard regarding container engines with OpenShift' rkt[49] and Apache Mesos[50] as very niche alternatives with less adoption and less supporting projects.

Container orchestration tools allow for the automation of deployment, management, scaling and networking of the containers, with the most popular solutions are Docker Swarm[51] and Kubernetes[52] with the latter one being way more advanced and feature rich with support for automated scaling and better auxiliary tools for deployment and monitoring of the components.

For situations when orchestration tools cannot be used or are not enough to achieve the level of automation desired, automation tools like Ansible[53], Chef[54] and Puppet[55] can be used with their main difference being in the approach they provide with Ansible being configuration-driven, Chef code-driven and Puppet model-driven. The choice made was Ansible as it was the most simple and easiest to learn and although it is not as complex, it had enough features for our needs and many community projects available online for automating the installation of popular software.

The hardware infrastructure used is managed using OpenNebula[56] which allows for the creation and management of virtual machines with volumes attached. Alternatives such as OpenStack[57] for cloud management although more advanced at the moment of writing were not used as the choice was out of the project's control. Services providers like AWS and Azure were not used as no budget was allocated since there was this private infrastructure available.

Completing the list of main technologies required is a Continuous Integration and Continuous Development platform and the one used was Microsoft’s Team Foundations Server 2017, which has recently been updated and renamed to Azure DevOps[58] and is maintained on premises by the university’s IT department. Alternatives like Jenkins[59] were also considered, primarily for the better flexibility but ended up not being necessary with the other platform already readily available.

3 Initial Project State and Design

When I joined the project there was already a substantial portion of the design phase done and documented such as a preliminary study of all the transformation processes in the value chain and a draft design of the middleware. The first steps of the development team were to study everything that had been researched and reported related to the project to make the final adjustments to the design before the start of the implementation.

3.1 Domain Analysis

The focus of the domain analysis was the information regarding the traceability of the products and mapping the operators that would participate in this early iteration of the middleware and what information they could provide. At this stage there were already plans to include descriptive information related to the products and value chain to complement the traceability information and monitorization features centered around the traceability information but no concrete ideas yet.

For the Portuguese fish value chain there was a study[60] conducted where the current businesses processes were modelled using BPMN[61]. The processes of several businesses were analysed and modelled and to fully understand the flow of the products over the value chain, abstract integrated processes were drawn. Taking the example of the fish and fishery value chain integrated process developed by the study present in Figure 3, we can understand the moments when information about the products is collected. In this process the products start their lifecycles either on a vessel operator or on an aquaculture producer operator with capture or production events, respectively. The product then goes through a series of retailers, transformers and logistics operators where quality assessment, storage, transformation, transportation and sale events may occur until it reaches the hands of the final consumer.

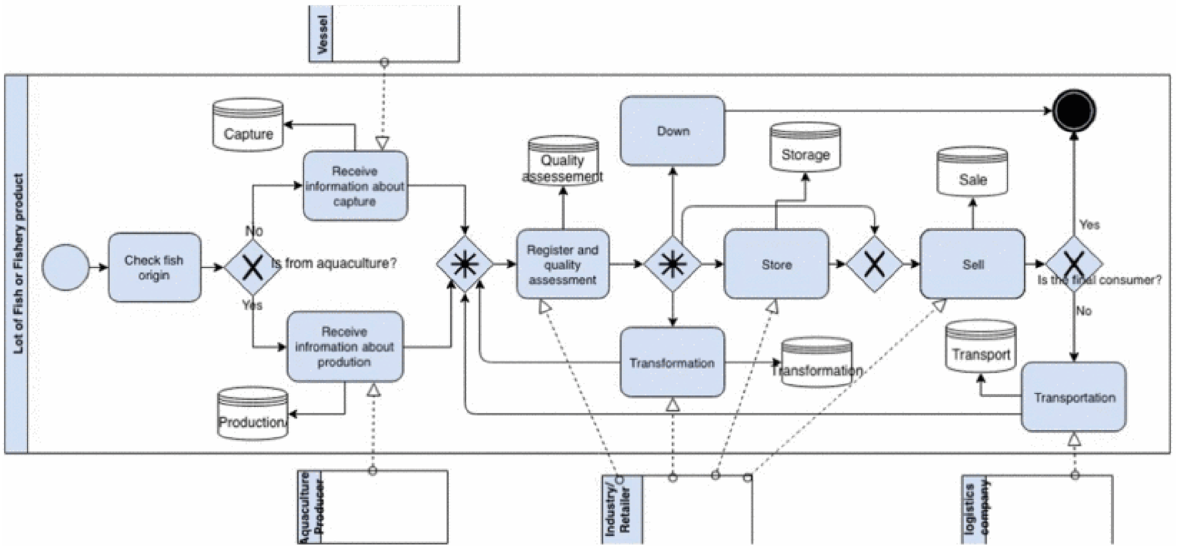


Figure 3 Fishery and Aquaculture integrated process (source: internal project documentation)

With the completion of this analysis the conclusion was that endpoints to register, query and update information about the products would be needed to achieve the purpose of the middleware of collecting and integrating all the information.

3.2 Middleware Requirements and Early Design

Following up the domain analysis an article[62] describing a possible design for this traceability middleware was published with the requirements. This first iteration was solely focused on the integration of the traceability information and split the requirements over 3 user roles and anonymous users:

1. Platform Admin which must be able to:
    a. Create, edit and deactivate Operators
    b. Create, edit and deactivate Operator Admins and Operator Users associated to an Operator
    c. Create, edit and delete information related to the global platform
2. Operator Admins which must be able to:
    a. Create, edit and deactivate Operator Users from their own Operator
    b. Create, edit and deactivate Event Records regarding lots related to their own Operator
3. Operator Workers which must be able to:
    a. Create Product Lots and Event Records for the Operator they work for
    b. Edit or delete the Event Records they created
4. Anonymous Users which must be able to:

a. Query the middleware for the Event Records related to a Product Lot using the Lot’s identifiers

Every product lot has an identification number associated. Uniqueness of these numbers is not guaranteed chain wide, not even inside the universe of identifiers belonging to that operator. To improve uniqueness of the identifier other attributes like the associated operator and the year of creation will be used.

Some other requirements regarding the structure of the traceability events were also present during this iteration but ended up being discarded in later versions to allow for configurable event structures.

On the non-functional requirements side of the list, it was established that the middleware should:

1. Be able to deal with the insertion of events asynchronously
2. Be able to scale horizontally

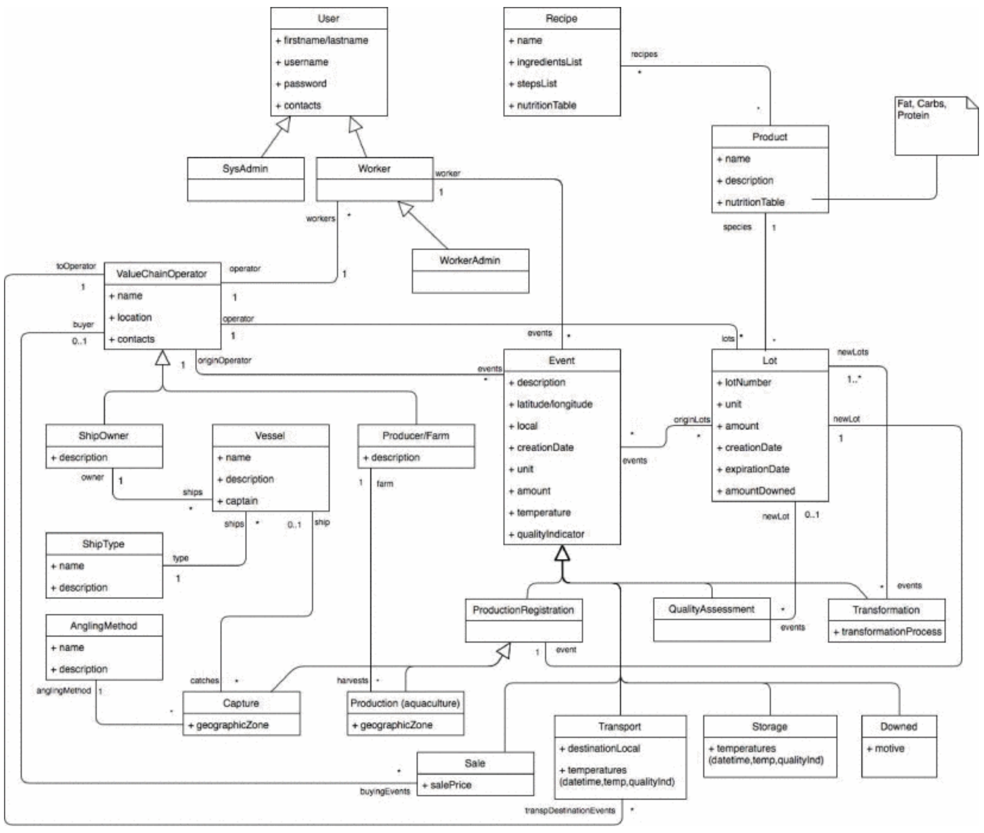


Figure 4 Initial Domain Model[62]

The domain model obtained from this draft design of the platform depicted in Figure 4 contains the main entities which will have information stored in the middleware: the users, events, products, lots and operators. Some of these entities like the operator and event types have many specializations to tailor the needs to each scenario which we viewed as an hinderance as with every new type of traceability event or operator added to the middleware, changes to the model would have to be made that would imply creating a lot of new code and changes to the database schema.

The architecture proposed in this phase consisted of a MongoDB[47] document database for the persistence needs connected to Kafka Consumers and Producers for the insertion of traceability events and a Node.js[63] Backend Application for the operations regarding the remaining entities. Everything was to be deployed using Docker containers. The Backend Application would expose REST APIs secured with JWT tokens using the Passport.js middleware for authentication.

The redesign of this version of the middleware was motivated by the desire to integrate some other systems like a wiki engine to store static information useful to the description of the value chains into the middleware and the changes to the domain to allow for more flexibility regarding the definition of new traceability event types, which in turn increased the complexity of the system, motivating the need to do some segmentation of the domain into smaller systems to completely embrace the microservice paradigm.

3.3 Improved Proposed Architecture

The type of architecture chosen for the development of the middleware was a microservice oriented architecture. This choice was made due to the diversity of data that will be saved and distributed by the platform and other advantages related to the scalability and extensibility of the system in relation to a monolithic architecture.

In a microservice oriented architecture, the complete system is segmented into small independent components with the least possible complexity. Each of these blocks can be developed independently, facilitating distributed development. In this architecture, priority should be given to the division of responsibilities of the components so that there are no repeated features. This type of architecture is indispensable in the development of applications for a native cloud model, taking advantage of the latest advances in technologies related to container virtualization and orchestration to deploy and maintain the system.

Figure 5 shows an example of the components contained in the architecture designed for the middleware with microservice examples and how they are related and the particularities used in this project like the Identity Provider component.

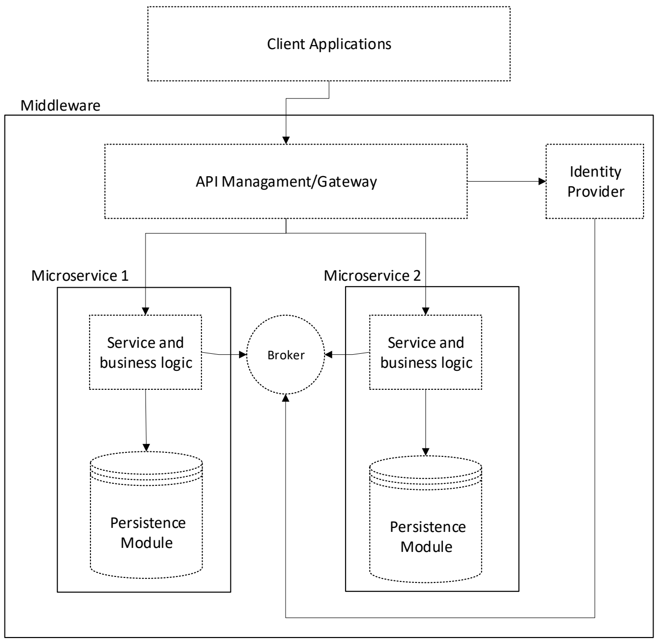


Figure 5 Middleware reference architecture[64]

The internal communication between the developed components will be done through a message broker, as is common in this type of system[65]. A message broker is a software component responsible for receiving messages and forwarding them to their destination. They are generally used in systems for asynchronous communication, the publish and subscribe standard being the most used.

Another component present in this type of architecture is an API Manager[66] or API Management Layer. A disadvantage of this type of architecture is the complexity added to the system due to the existence of several APIs with different versions that must be managed and maintained, being that the layer of management facilitates this work and implements the functionalities of the system transversal to all microservices that will be explained in more detail in the implementation chapter. Also displayed the architecture diagram example is an Identity Provider that serves as a single piece to authenticate users in the system during the routing phase and removing the need to implement security in microservices as long as the deployment isolates the components properly from outside connections.

3.4 Domain Segmentation

By sticking to the patterns and practices used in a Microservice Oriented Architecture, Domain Driven Design[67] was used which, as implied by the name itself, focuses primarily on the data domain and the decomposition of this domain in sub-domains to enable the right level of granularity for each microservice.

During the design step, four domains were identified that should be developed as separate modules. This segmentation was done focusing primarily on the volume of data and the frequency of change of the entities present in the data model of the full middleware as well as the relations between the entities. Following is the list of domains:

1. Information: Mostly static data regarding the businesses, processes, products and people related to the supply chain. This data can be of any type and unstructured such as long text descriptions, videos or images.

2. Tracking: Data related to the events and product lots registered by the operators in the value chain, which is used to create the history of the final product lot and for logging and tracing purposes.

3. Governance: Comprises the management of the runtime configurations for the traceability event data structures, related monitoring triggers and internal access control. This approach allows the operators to adapt the middleware to their needs while maintaining a fixed structure providing advantages for data warehousing and data mining operations.

4. Users and Access control: Contains information about users, roles and permission and the user data related to the security mechanisms of the API Gateway.

4 Middleware Design and Development

This chapter describes the development phase of the middleware focusing on the final version of the components as of the moment of writing of this document, starting with a simplified broad overview of the entire system and the moving to a more detailed description of each domain. An earlier prototype of this middleware was documented in a published article[64] for the CISTI 2020 conference.

4.1 Full Overview

The domains obtained in section 3.4 were developed as separate systems, each with their own domain models and entities. The complete class model with all the entities and their cross-domain relations can be viewed in Figure 6.

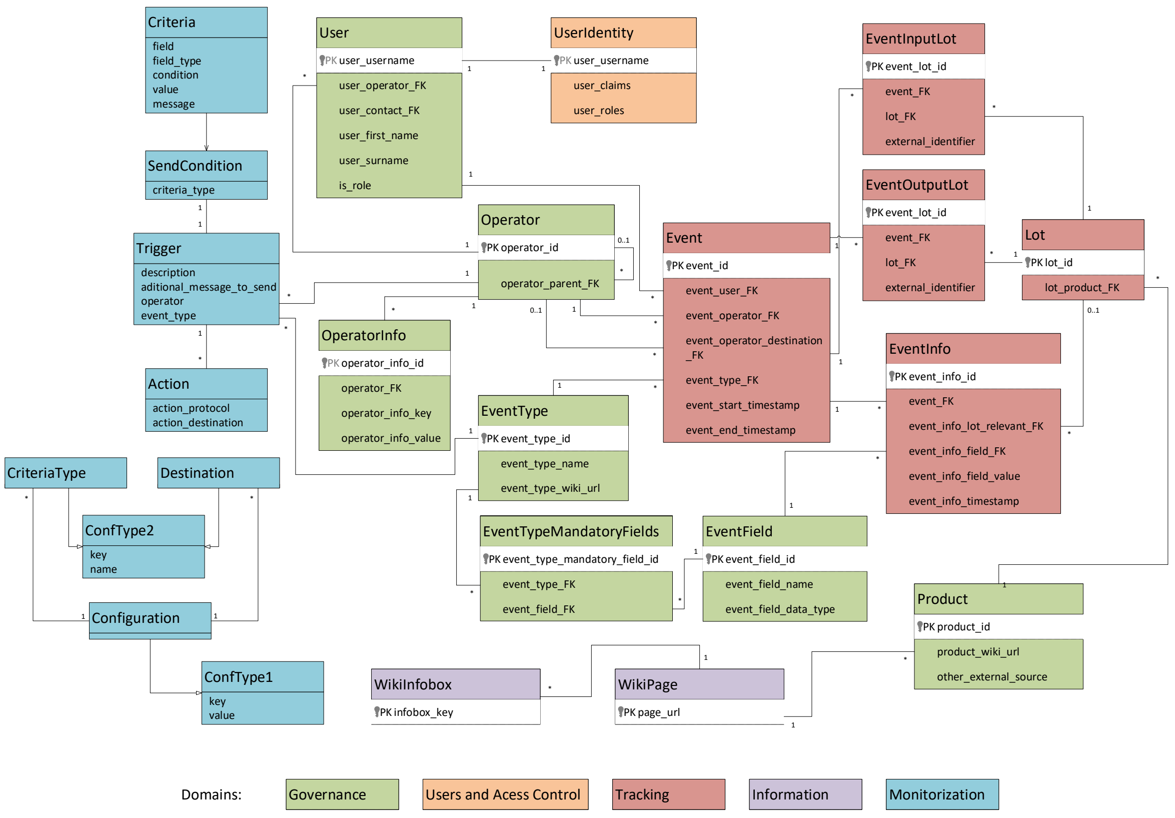


Figure 6 Middleware Class Diagram, adapted from the journal[68]

Each domain is colour coded and in the cases of the Information Users and Access Control only simplified views of the classes inside that specific software were drawn, as they are very complex tools and the goal here was to provide a view of what information is relevant in the context of the Middleware.

The final visualization of all the components and their communications is present in Figure 7. The focus of this diagram is on the flow of information, placing an emphasis on the Kafka broker, with each rectangle inside representing topics used to create data pipelines.

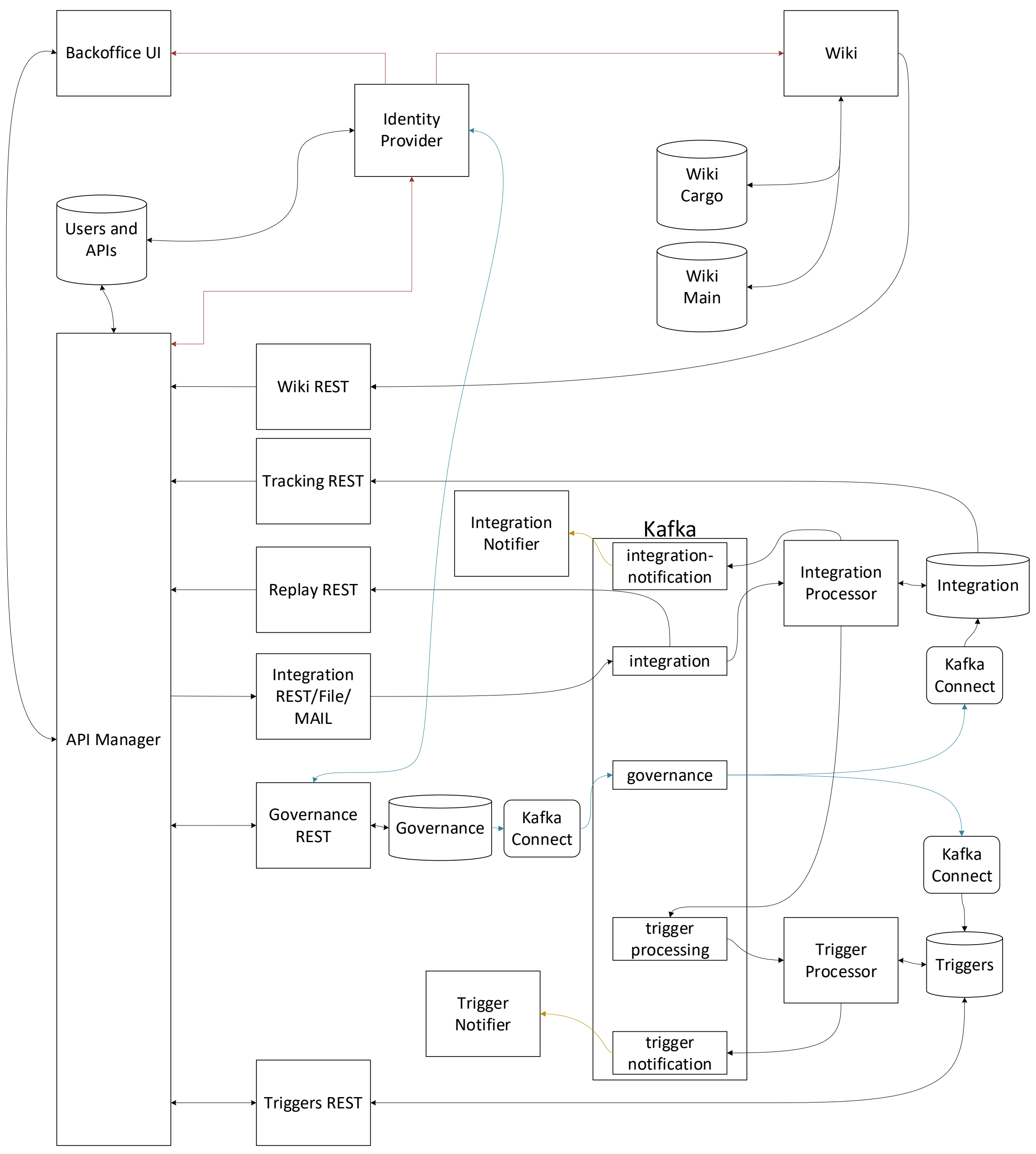


Figure 7 Middleware components and information flow

The black connections represent the flow of information from the API Management Layer to each component that implements an API and then to each data source or sink components. The connections coloured in blue represent the cross-domain communication of all the information required in multiple contexts. Red connections are related to the flow of information regarding user credentials and authentication. Finally, the yellow connections represent the flow of outbound asynchronous messages sent by the Middleware to external systems.

4.2 *API Management and Gateway*

For the management and gateway layer, the WSO2 product family was chosen. These products are designed to support service and microservice-oriented systems. They are open source, highly versatile and modular, allowing developers to change or add features when they are not yet provided by the base product.

All components of the WSO2 ecosystem are developed in Java and composed of the WSO2 Carbon platform and the modules of the necessary functionalities for that component. Figure 8 illustrates all the tools available with the WSO2 API Manager and the integrations it can do with the backend services.

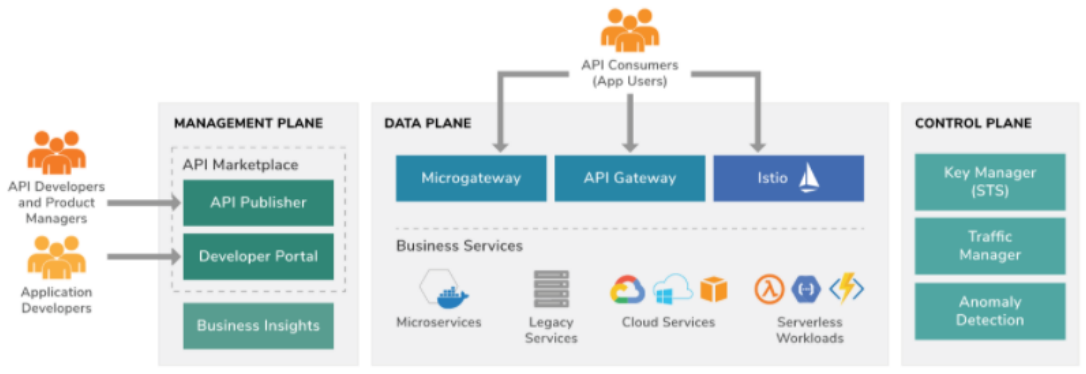


Figure 8 WSO2 Core Features (source: wso2.com)

The WSO2 API Manager comes segmented into separate components that are configured to work together as the gateway and management layer of the middleware. These components can be deployed in an all-in-one instance or deployed as separate entities to optimize the resource allocation.

4.2.1 Developer and Publisher Portal

The developer portal (API Developer Portal) allows developers of applications that will communicate with middleware to consult the specification of the APIs that are available. This portal makes it possible to register client applications to generate the necessary credentials for the consumption of services. This specification follows the OpenAPI 3.0 standard[69] implemented by Swagger[70]. Other information available on this portal is documentation written in markup language, access points to services and self-generated libraries for the consumption of APIs in various languages. The current list of APIs available from the middleware is shown in the Developer Portal, as in Figure 9.

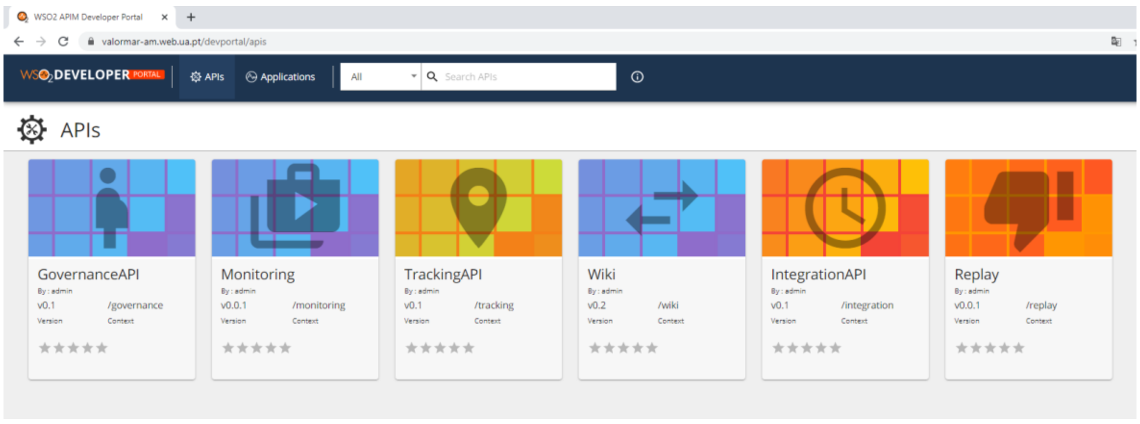


Figure 9 WSO2 AM Developer Portal homepage

The management portal (API Publisher) is used by platform administrators to control the entire operation of each API, such as transport settings, security, permissions, change the specification or limit the number of requests allowed in a given time interval.

4.2.2 API Gateway and Traffic Manager

The centralization of the various APIs in a single access point is done in the API Gateway. This Gateway puts into practice the settings selected in API Publisher for each API, mediating between applications and the backend. During this mediation, the tokens are validated with the Key Manager.

Traffic Manager monitors the API Gateway and communicates events of system state changes to the other components, such as blocking APIs above the usage limit, revoking tokens and changing settings.

4.2.3 Token Generation and Validation

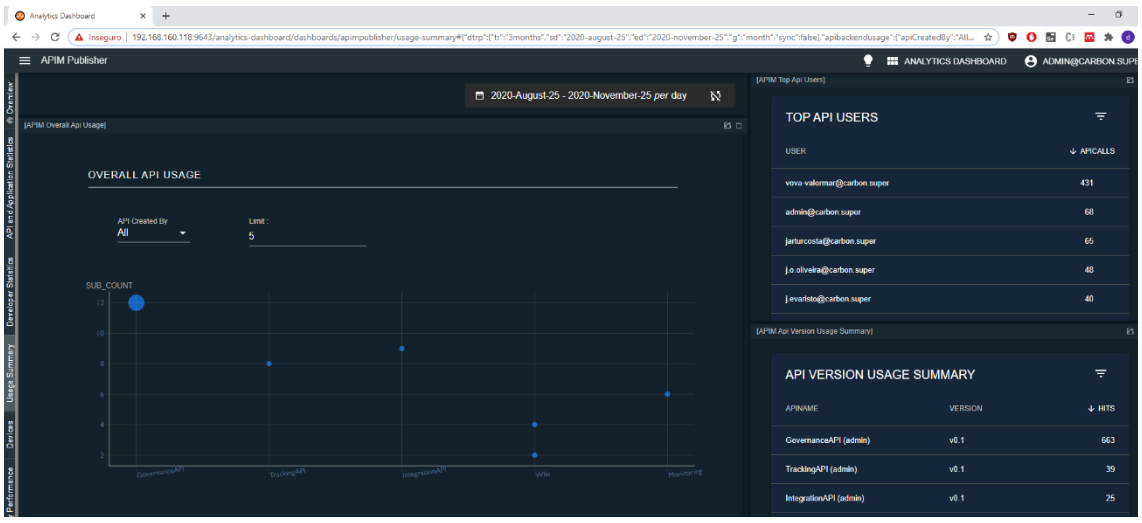
For user authentication in the APIs, the WSO2 API Manager has a key management component (Key Manager) that can be replaced by the WSO2 Identity Server with the installed Key Manager package that expands the features available as predefined integrations with other Identity Providers.

For authorization purposes, oauth2 tokens and JWT tokens are available and applications can choose the type that is most convenient for them.

When a request reaches the API Gateway, it queries the Key Manager to validate the token. After the default validation, the tokens are removed but if the backend needs contextualization of the application or the user, this is done during the mediation phase by issuing a JWT token specific for the backend or adding certain metadata to the request, such as the user or application's information.

4.2.4 Monitorization

Many of the products in the WSO2 family have monitoring components such as API Manager and Identity Server. There are Analytics Worker components that receive information from the components to be monitored and store them in a database and the Analytics Dashboards that queries that database to create graphical visualizations. This information concerns the use of APIs and applications, as exemplified in Figure 10.



```
192.168.160.118:9643/analytics-dashboard/dashboards/apimpublisher/usage-summary#{"dtrp":{"tr":"3months","sd":"2020-august-25","ed":"2020-november-25","g":"month","sync":false},"apibackendusage":{"apiCreatedBy":"All...
```

[APIM Top Api Users]

TOP API USERS

USER	APICALLS
vova-valormar@carbon.super	431
admin@carbon.super	68
jarturcosta@carbon.super	65
j.o.oliveira@carbon.super	48
j.evaristo@carbon.super	40

[APIM Api Version Usage Summary]

API VERSION USAGE SUMMARY

APINAME	VERSION	HITS
GovernanceAPI (admin)	v0.1	663
TrackingAPI (admin)	v0.1	39
IntegrationAPI (admin)	v0.1	25

Figure 10 WSO2 AM Analytics Dashboard

4.3 *Inter-Domain Communication*

Regarding the communication and replication of information shared between domains, the Kafka distributed system was selected. This system is an event streaming platform and has a high adoption in microservices oriented architectures. The main advantages are its performance, scalability and fault tolerance, with the Connect API being a nice extra that eases some development tasks related to the creation of data pipelines.

4.3.1 Kafka in event driven scenarios

Kafka has 3 main capabilities: publish and subscribe to event streams to import or export information, the storage of event streams and the processing of those events. The communication of events is done through the exchange of messages using TCP connections.

To use the platform, libraries are available in several languages that use the APIs exposed by Kafka from which we use the Consumer API, Producer API, and the Kafka Connect API to create data pipelines, as exemplified in Figure 11.

Messages are organized into topics that can have multiple partitions to share the load between nodes and replicas for better availability. The messages are stored in the partitions in order of arrival with the offset being the identifier used. In addition, the message may also have an associated key that is unique to each topic.

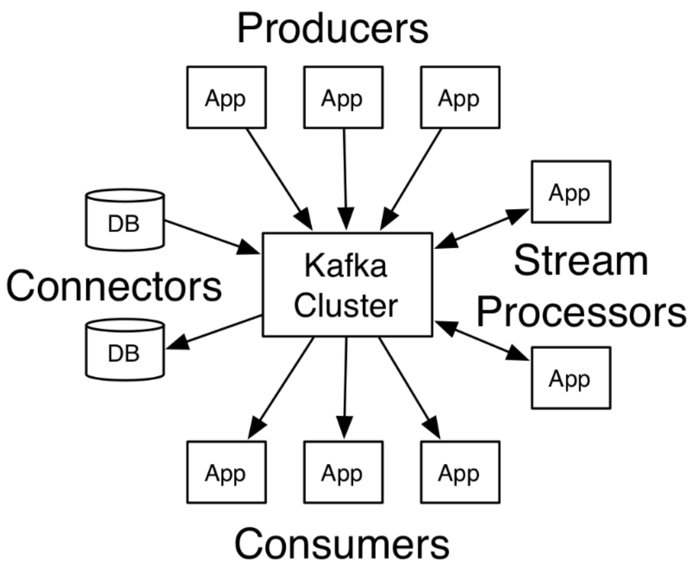


Figure 11 Kafka APIs (source: dzone.com)

4.3.2 Auxiliary components

Within the Kafka ecosystem there are several tools necessary for the cluster to function and others that facilitate integration between microservices.

Zookeeper is used as the cluster controller; it is there that all settings and metadata of existing topics are saved. To view this information, dashboards such as Kafka Manager, Lenses IO and Confluent Control Centre were used.

The Schema Registry uses Kafka topics to store additional metadata about the topics and exposes a query API for that information. This query API can be used by applications to force a certain structure in the messages for each topic.

By placing a schema in the topics, it is possible to use the Kafka Connect API with modules already made to establish data pipelines between the system components and Kafka. The main connector used was the Debezium[71] for SQL Server uses the events generated by the database engine's monitoring agent and the database's transaction log to create events as changes are made. Another advantage of Debezium is that it automatically detects the database schema and creates a corresponding Avro[72] schema in the Schema Registry to facilitate the insertion of that data in other domains.

The remaining connectors used were JDBC Sink, which allows insertion of data into a relational database and sinks for ElasticSearch[73] and MongoDB.

4.4 Information Microservice

The goal for this microservice was to collect and provide static information about the businesses, processes and entities that make up the value chain. This information can be used to enrich the traceability story presented to the customers during their product selection process.

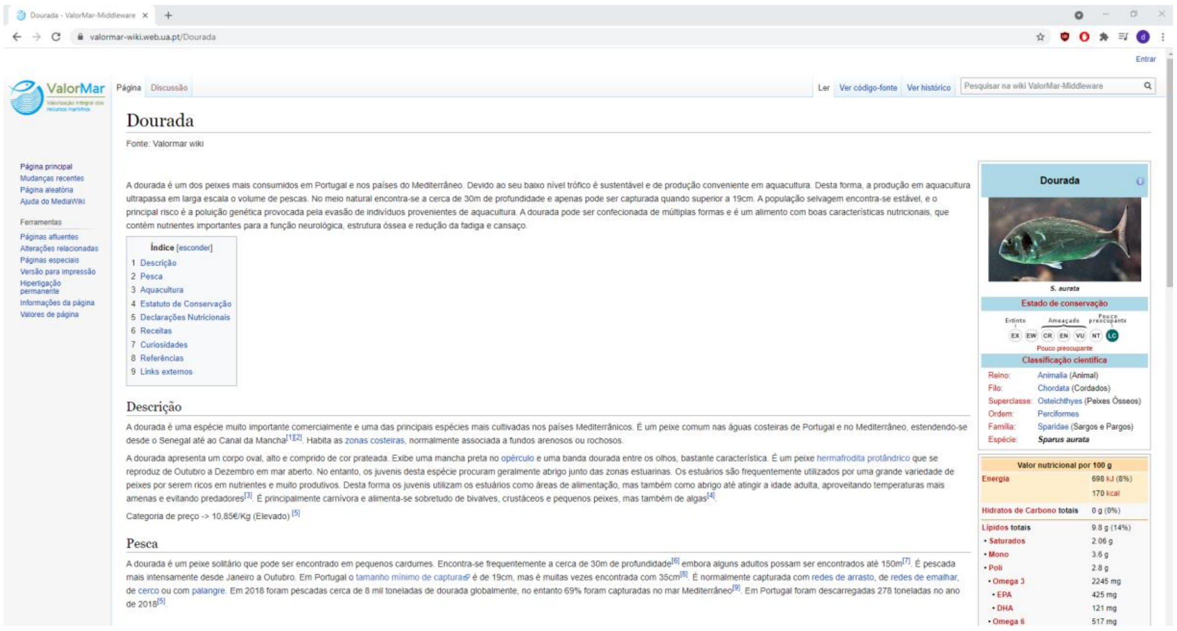
For this purpose, we decided to configure and deploy a wiki for the middleware. A wiki is a website that makes use of a wiki engine which works like a content management system. The content is usually stored in pages usually written in a markup language which can then be grouped and linked with lists and categories. One of the main advantages of using a wiki is the collaborative editing capability that can be configured to allow for very open settings so that anyone can produce content for the wiki. This supresses the need to go through a very extensive data collection process to properly document the supply chain and shifts that responsibility to the businesses and entities themselves which can insert any information they see fit and link it to their products or events.

4.4.1 Mediawiki Engine

Several wiki engines exist with the most popular being Mediawiki[74]. Mediawiki is the engine behind the Wikipedia Project which is a collection of wikis written in multiple languages that stores information about everything and anything. This engine was written in php and uses a relational database to store the created pages. Some other engines may only support static content and use a git repository to store the wiki pages such as Wiki.js[75]. The database engine used during the development process was MySQL, deployed using Docker. All the extensions necessary are available on the Mediawiki project website.

To install all the necessary components and prepare the Mediawiki environment a Dockerfile was used to extend the base Mediawiki docker image fetching and installing plugins from the Mediawiki repositories to the local extension folder. Most of the wiki related configurations are made in the file LocalSettings.php such as the database connection string, wiki hostname and security settings.

A complete example of a wiki page created in the middleware is shown Figure 12 with the Sparatus Aurata page containing descriptive information, some images and some infoboxes for structured information.



Dourada

Fonte: Valormar wiki

A dourada é um dos peixes mais consumidos em Portugal e nos países do Mediterrâneo. Devido ao seu baixo nível trófico é sustentável e de produção conveniente em aquacultura. Desta forma, a produção em aquacultura ultrapassa em larga escala o volume de pescas. No meio natural encontra-se a cerca de 30m de profundidade e apenas pode ser capturada quando superior a 19cm. A população selvagem encontra-se estável, e o principal risco é a poluição genética provocada pela evasão de indivíduos provenientes de aquacultura. A dourada pode ser confecionada de múltiplas formas e é um alimento com boas características nutricionais, que contêm nutrientes importantes para a função neurológica, estrutura óssea e redução da fadiga e cansaço.

Índice [esconder]
1 Descrição
2 Pesca
3 Aquacultura
4 Estatuto de Conservação
5 Declarações Nutricionais
6 Receitas
7 Curiosidades
8 Referências
9 Links externos

Descrição

A dourada é uma espécie muito importante comercialmente e uma das principais espécies mais cultivadas nos países Mediterrânicos. É um peixe comum nas águas costeiras de Portugal e no Mediterrâneo, estendendo-se desde o Senegal até ao Canal da Mancha[1][2]. Habita as zonas costeiras, normalmente associada a fundos arenosos ou rochosos.

A dourada apresenta um corpo oval, alto e comprido de cor prateada. Exibe uma mancha preta no opérculo e uma banda dourada entre os olhos, bastante característica. É um peixe hermafrodita protândrico que se reproduz de Outubro a Dezembro em mar aberto. No entanto, os juvenis desta espécie procuram geralmente abrigo junto das zonas estuarinas. Os estuários são frequentemente utilizados por uma grande variedade de peixes por serem ricos em nutrientes e muito produtivos. Desta forma os juvenis utilizam os estuários como áreas de alimentação, mas também como abrigo até atingir a idade adulta, aproveitando temperaturas mais amenas e evitando predadores[3]. É principalmente carnívora e alimenta-se sobretudo de bivalves, crustáceos e pequenos peixes, mas também de algas[4].

Categoria de preço -> 10,85€/Kg (Elevado) [5]

Pesca

A dourada é um peixe solitário que pode ser encontrado em pequenos cardumes. Encontra-se frequentemente a cerca de 30m de profundidade[6] embora alguns adultos possam ser encontrados até 150m[7]. É pescada mais intensamente desde Janeiro a Outubro. Em Portugal o tamanho mínimo de captura é de 19cm, mas é muitas vezes encontrada com 35cm[8]. É normalmente capturada com redes de arrasto, de redes de emalhar, de cerco ou com palangre. Em 2018 foram pescadas cerca de 8 mil toneladas de dourada globalmente, no entanto 69% foram capturadas no mar Mediterrâneo[9]. Em Portugal foram descarregadas 278 toneladas no ano de 2018[5].

Dourada

S. aurata

Estado de conservação

Pouco preocupante

Classificação científica	
Reino:	Animalia (Animal)
Filo:	Chordata (Cordados)
Superclasse:	Osteichthyes (Peixes Ósseos)
Ordem:	Perciformes
Família:	Sparidae (Sargos e Pargos)
Espécie:	***Sparus aurata***

Valor nutricional por 100 g	
Energia	698 kJ (8%)
	170 kcal
Hidratos de Carbono totais	0 g (0%)
Lípidos totais	9.8 g (14%)
• Saturados	2.06 g
• Mono	3.6 g
• Poli	2.8 g
• Omega 3	2245 mg
• EPA	425 mg
• DHA	121 mg
• Omega 6	517 mg

Figure 12 Wiki page (Sparatus aurata)

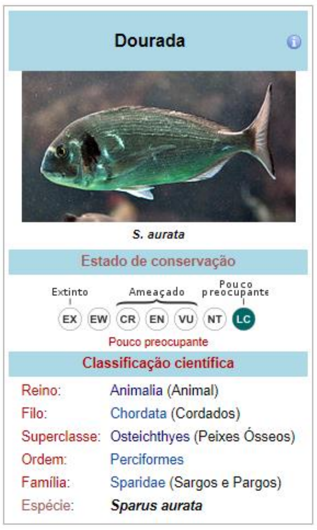
4.4.2 Infoboxes and Text Extraction

To improve the information insertion and retrieval side of the wiki we used a combination of the following extensions: Scribunto[76] and Cargo[77] to allow the usage of templated infoboxes with a fixed structure. One of the many challenges of working with a wiki is the wikitext structure which is not very friendly to extract data from, as stated in this article[78] about Semantic Mediawiki which is another extension for Mediawiki that improves the functionality in some key areas: Consistency of content, Accessing knowledge and Reusing knowledge. Cargo is an extension that provides similar but simpler functionality and with better reported performance in relation to Semantic Mediawiki and it allows us to use relational database tables to store and read information from the wiki. The infobox templates created with the Scribunto parser are modified to make use of these Cargo tables so any infobox created in a new wiki page gets stored in the corresponding Cargo Table. Cargo also extends the REST API endpoints available in Mediawiki with services that allow the querying of the infobox tables to allow other applications to use this data. The tables created are stored in a separate database using the same MySQL database engine as the wiki.

After the installation, the extensions must be configured with the connection properties and credentials to access the database engine in the LocalSettings.php.

To create an Infobox template there are 2 main components: the visual component and the text extraction component. The visual component is written in wikitext and includes some html code. Instead of developing my own visual infobox template I exported the base one from the Portuguese Wikipedia website and imported it and its dependencies to the middleware’s wiki with some minor visual adjustments. This base template allows for the usage of a limited set of key value pairs to create more specific infobox templates and the name of the page in the wiki is Predefinição:Info. An example of a template needed was a Taxonomy box to insert and display information regarding fish species. The visual portion of the template is also base on a similar one in the Portuguese wiki which extends the base Info template and is called Predefinição:Info/Taxonomia. This template was then again extended with another final layer to add the cargo text storage and extraction instructions which is located in the page Predefinição:Info/Taxonomia/Cargo. After saving the cargo template the corresponding database table must be created manually by using the recreate table button on the template page or the php scripts included by the extension. The final visual result from this template can be seen in Figure 13.

With another extension called TemplateData we can create metadata for the infoboxes with the list of fields supported, data type and a brief description. This metadata is used by the Visual Editor extension to generate forms, as in Figure 14, to allow for a more user-friendly insertion of each field instead of using wikitext.


Dourada

S. aurata

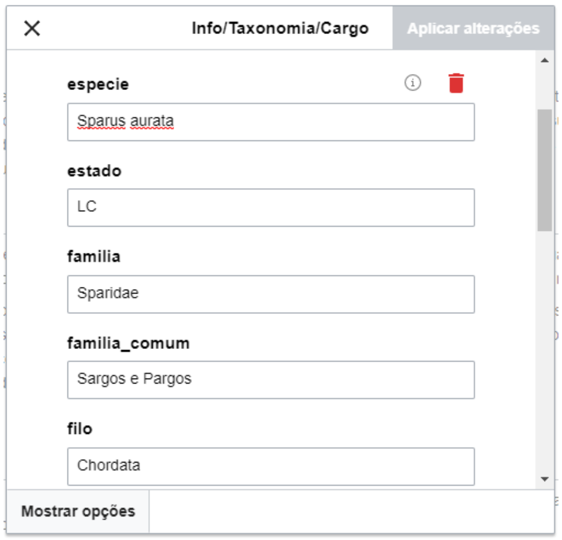
Estado de conservação

Pouco preocupante

Classificação científica

Reino:	Animalia (Animal)
Filo:	Chordata (Cordados)
Superclasse:	Osteichthyes (Peixes Ósseos)
Ordem:	Perciformes
Família:	Sparidae (Sargos e Pargos)
Espécie:	*Sparus aurata*

Figure 13 Taxonomy Infobox display (Sparus Aurata)


Info/Taxonomia/Cargo

Aplicar alterações

especie

Sparus aurata

estado

LC

familia

Sparidae

familia_comum

Sargos e Pargos

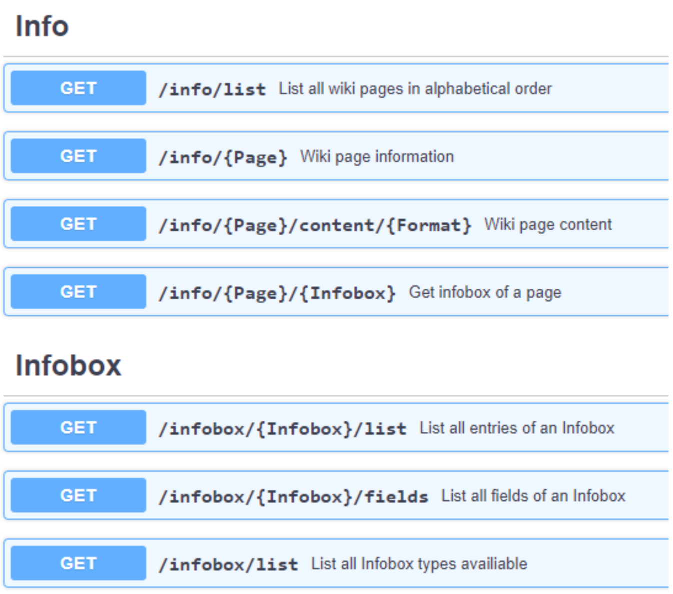
filo

Chordata

Mostrar opções

Figure 14 Taxonomy Infobox insertion (Sparus aurata)

The REST API provided by Mediawiki and Cargo for information is very extensive and a bit complex to use as for the extraction of an infobox the developer must know the complete structure of the infobox which might not always be possible. To simplify this API, we used an Enterprise Integrator to remove some of the metadata gathering steps and clean some of the internal structure of the tables that does not need to be extracted from the wiki. The integrator used was WSO2 Micro Integrator as it provides multiple ways to develop the solution with a graphical drag and drop IDE or xml configuration files. Other advantages are that it is highly compatible with the other WSO2 products used in the project and it allows for the addition of a Swagger documentation which is then added to developer portal when the API is deployed.


Info

GET /info/list List all wiki pages in alphabetical order

GET /info/{Page} Wiki page information

GET /info/{Page}/content/{Format} Wiki page content

GET /info/{Page}/{Infobox} Get infobox of a page

Infobox

GET /infobox/{Infobox}/list List all entries of an Infobox

GET /infobox/{Infobox}/fields List all fields of an Infobox

GET /infobox/list List all Infobox types availiable

Figure 15 Infobox API Swagger definition

The API only has GET methods and is divided in 2 main groups: Info and Infobox, with the corresponding Swagger console in Figure 15. The info group has methods to extract content based on the page it is at. The methods allow for the entire page to be extracted in multiple formats like html or wikitext or only some metadata such as inboxes that exist on the page which can then also be extracted. The infobox group is related to the infoboxes themselves such as their fields, list of entries and other metadata. The information is always exported in JSON as exemplified in Figure 16.

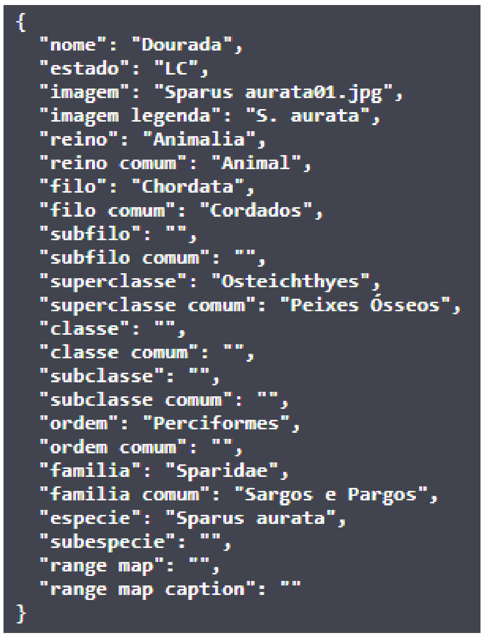

```
{
  "nome": "Dourada",
  "estado": "LC",
  "imagem": "Sparus aurata01.jpg",
  "imagem legenda": "S. aurata",
  "reino": "Animalia",
  "reino comum": "Animal",
  "filo": "Chordata",
  "filo comum": "Cordados",
  "subfilo": "",
  "subfilo comum": "",
  "superclasse": "Osteichthyes",
  "superclasse comum": "Peixes Ósseos",
  "classe": "",
  "classe comum": "",
  "subclasse": "",
  "subclasse comum": "",
  "ordem": "Perciformes",
  "ordem comum": "",
  "familia": "Sparidae",
  "familia comum": "Sargos e Pargos",
  "especie": "Sparus aurata",
  "subespecie": "",
  "range map": "",
  "range map caption": ""
}
```

Figure 16 Taxonomy Infobox API extraction (Sparus Aurata)

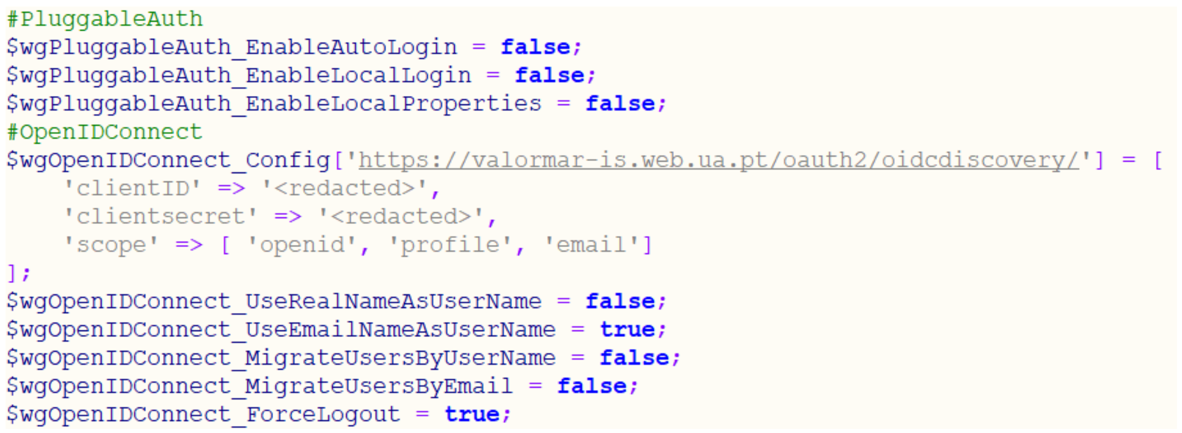
4.4.3 Categorization

One of the main features of the Mediawiki engine for page indexation is Categorization. Categorization allows for the grouping of pages into categories by using category tags placed in the wikitext. In our Taxonomy infobox a lot of the fields stored and displayed can also be interpreted as categories and by adding category tags to the applicable fields in the infobox during the wikitext generation process we can create Category pages which will automatically list all the species belonging to that category and since this list is still a page some generic information can also be added.

4.4.4 Authentication and Permissions

Regarding authentication, the extensions adopted were OpenIDConnect[79] and Plugabble Auth[80] to implement Single Sign On using our WSO2 Identity Server and the OpenID Connect standard.

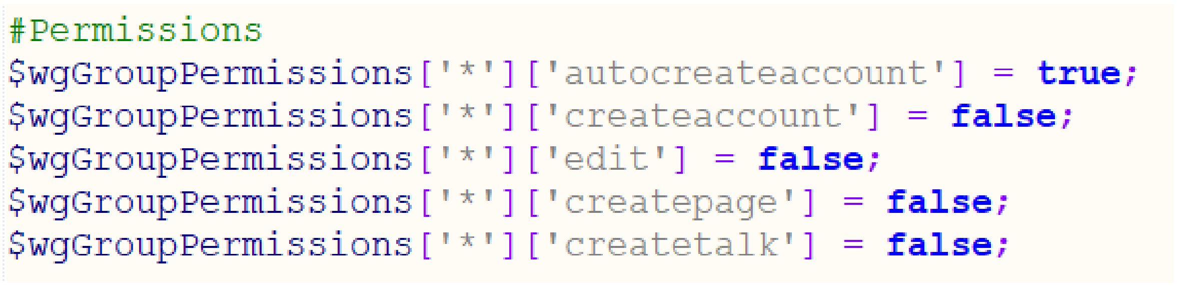
The extensions are configured in the LocalSettings.php files with the OpendID Connect client ID, client secret and scope, as shown in Figure 17. This php extension uses the authorization code authentication flow.



```php
#PluggableAuth
$wgPluggableAuth_EnableAutoLogin = false;
$wgPluggableAuth_EnableLocalLogin = false;
$wgPluggableAuth_EnableLocalProperties = false;
#OpenIDConnect
$wgOpenIDConnect_Config['https://valormar-is.web.ua.pt/oauth2/oidcdiscovery/'] = [
    'clientID' => '<redacted>',
    'clientsecret' => '<redacted>',
    'scope' => [ 'openid', 'profile', 'email']
];
$wgOpenIDConnect_UseRealNameAsUserName = false;
$wgOpenIDConnect_UseEmailNameAsUserName = true;
$wgOpenIDConnect_MigrateUsersByUserName = false;
$wgOpenIDConnect_MigrateUsersByEmail = false;
$wgOpenIDConnect_ForceLogout = true;
```

Figure 17 Mediawiki SSO configurations

For the wiki permissions we disabled local account creation to use only SSO and enabled account autocreation to automatically confirm the accounts generated by the extension during a user’s first login. As of the moment of writing, the feature of the OpenID Connect extension that was used that would enable the wiki to receive roles for the user from the Identity Server is not fully tested and not part of the major release so currently, the local wiki groups are being used for that purpose, as configured in Figure 18. By default, users are not allowed to edit or create pages and must be promoted by an admin to start contributing to the wiki. In the future this can be changed to allow for public users to send articles that are reviewed by wiki admins and published if appropriate.



```
#Permissions
$wgGroupPermissions['*']['autocreateaccount'] = true;
$wgGroupPermissions['*']['createaccount'] = false;
$wgGroupPermissions['*']['edit'] = false;
$wgGroupPermissions['*']['createpage'] = false;
$wgGroupPermissions['*']['createtalk'] = false;
```

Figure 18 Mediawiki permissions configurations

4.4.5 Static Content and Visual Tweaks

The Mediawiki engine allows for the upload of static content such as videos and images, which are stored in the filesystem mounted to the container while also storing metadata about the content in the database. For these kinds of files to be uploaded, some request size limits had to be raised. This had to be done for the wiki engine, php interpreter and for the NGINX reverse proxy allowing fronting the wiki for public access.

Some visual themes were tested like Tweeki[81] to improve the appearance of the wiki and the usability, as this theme used bootstrap to make the web application responsive but due to some incompatibility issues with the infoboxes at the time of writing, the default skin chosen was Vector which is the current Mediawiki default.

The remaining visual tweaks done were some adjustments to some of the infoboxes and the usage of the project logos for the footer and icons.

4.5 Governance Microservice

This microservice was not part of the original plans but after the analysis of the draft database model, concerns were raised that with a limited pre-defined set of event types a lot of incompatibility issues could appear when integrating operators and they would have to be solved at the database level and on a case-by-case situation.

To mitigate that problem, we opted instead for this governance component which would allow for each operator to manage the structure of the information they intend to push to the system. These new features were grouped with all the management features that were planned from the start during the domain segmentation phase.

4.5.1 Data Model

The ER Diagram in Figure 19 presents the final database model conceived for this data domain. The Operator entity can represent any entity belonging to the value chain from an entire company to automated systems. In this domain it is mostly used to define the User's scope inside the middleware by only allowing this user to manage configurations inside its own Operator context. Private information regarding the operators can also be inserted using the OperatorInfo entity as a key-value store and more descriptive information about the Operator can also be added in the Information Domain and linked to the corresponding entry in this domain.

The User entity contains replicated information from the User Domain and is present in this domain for the relation with the Operator entity.

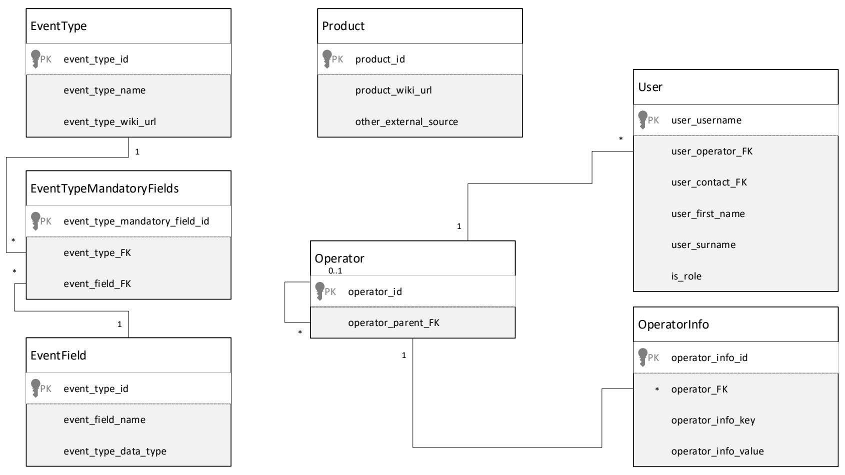


Figure 19 Governance Database Model

To establish the connection with the information stored in the wiki the Product entity allows for the management of the wiki page where the description of each product is accessible.

Regarding the event related governance of the system, three Entities were used: EventType, EventTypeMandatoryFields and EventField. These allow for the definition of Event structures with a set of mandatory fields which can be of any primitive data type or string.

Other permission related entities were planned but not fully implemented and tested as of the moment of writing. These were to be used to provide even finer granularity to the permissions of each user like for example allowing only the insertion of a specific type of event. Currently the access control can be first implemented based only on the Operator the User belongs to and the user profile they currently have by adding Operator contextualization to the EventType and Product entities. These users' profiles are defined and managed in the User Domain and can be: Operator User, Operator Admin, Platform Admin.

4.5.2 Operator Trees

The Operator entity possesses a foreign key field that points to the same entity that can be used to specify a parent Operator. The main reason for this implementation was the potential lack of collaboration, due to lack of means from some of the Operators in the value chain that could prevent the collection of information from the entire supply chain, allowing for a proxy operator to manage their information for them. In other cases, like the bigger companies, it also makes sense to allow for segmentation into multiple operators as a single entity may be responsible for dozens of stores or warehouses. By allowing Operators to be associated with a parent Operator Trees can be formed with increasing levels of granularity as the tree is expanded. This allows for the supply chain to be integrated gradually with the event insertion responsibilities being delegated further down the tree as it expands.

4.5.3 Architecture and Implementation

For the implementation of all the CRUD operations to the entities of this data domain we opted to use Microsoft's .Net Core framework (version 2.2) to implement RESTfull API Controllers for each entity. These entities were created using the Entity Framework Object-Relation Mapper which allows for the usage of Object-Oriented Programming's classes to generate all the database code required. Thus, the development of this component was done using the code-first methodology allowing EF Migrations to evolve the database during the development process. For the deployment process we opted instead to use DACPAC[82] packages with the goal of maintaining the information in the staging environment and this allowed us more control over that migration operations in the cases that this process could not be done autonomously.

Figure 20 describes the flow of the information through the components used to implement this domain. The inbound requests made to the API Gateway are routed to the .Net Core applications which communicate with the persistence layer to perform the desired operation and generate the output required for the request's reply. The remaining components displayed are related to the replication of information across all the domains using Kafka as central component to collect and distribute this information.

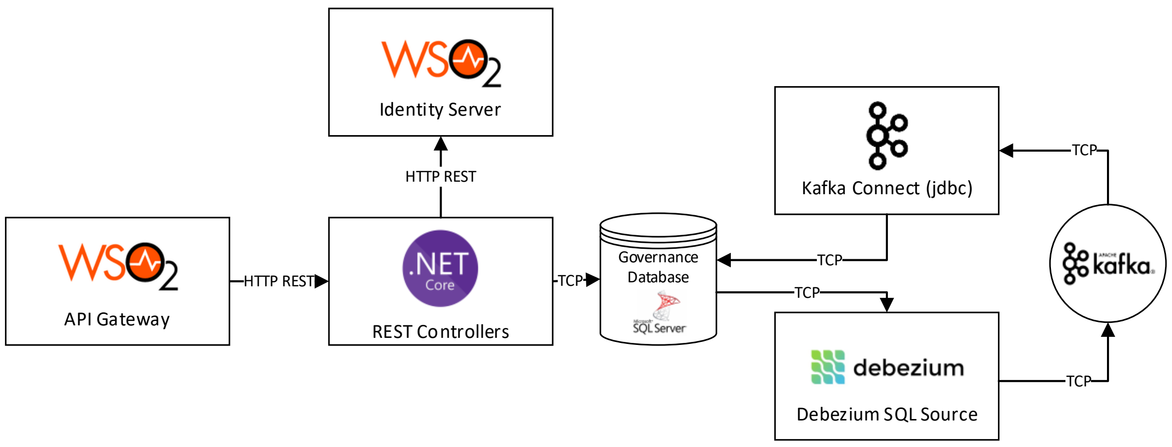


Figure 20 Governance Microservice Implementation Diagram

4.5.4 Information Exporting

For the information managed in this domain that is also required in other domains we had 2 main solutions for this problem. Initially we planned to implement the push of all this information manually using the Kafka Producer API during the CRUD operations of the REST API. After further research we opted instead to use the database's transaction logs as this would be easier to implement with already existing software solutions. Another advantage is that it does not place an extra load on the engine as it does not constantly query it to retrieve the information.

With Microsoft' SQL Server this is done by the SQL Agent component which is responsible for capturing all of the events that occur in the database engine. This feature is only available in the Developer and Enterprise editions of the software. With the agent enabled the only other configuration step on the database side is to enable Change Data Capture on the desired databases and tables with SQL queries.

To push the information from the logs to the Kafka Topics, Debezium was used which is a distributed platform for change data capture which uses the Kafka connect API to implement connectors to transfer information from Kafka to a database and vice-versa. In this case we used the Debezium MSSQL Server Source which observes the transaction log and inserts all the changes as events to the Kafka platform. These events are organized in topics with each entity having its own dedicated topic and this allows for any interested party with access to the broker to receive the events of the entities they require information on.

4.5.5 Profile Replication

Regarding the information required in this domain that is managed by other domains, JDBC Sink Kafka Connectors were used to update those entities in this domain. In this case the user related information is replicated from the User and Access Control domain which will be described later.

One small difference regarding this information replication in relation to the rest of the project is that the component that was responsible by the information was not developed internally, thus the level of control and customization was much lower.

To complete the centralization of the middleware's operator management related features we were required to provide a way to manage the user access profile from the Governance API and for that purpose this feature was implemented in this domain with a proxy to the API that allows this in the User and Access Control that uses implements the SCIM2[83] standard. We opted for this method of information replication instead of having this feature handled by this domain and having connectors manipulating the users' database. This provides better interoperability in case there was a need to change the components implementing the User and Access Control domains for other similar Identity Providers.

4.6 Tracking Microservice

This data domain is responsible for the aggregation of the traceability events that occur in the supply chain.

4.6.1 Data Model

The data model depicted in Figure 21 shows the tables currently implemented in this domain. The Product, User, Operator, EventType, EventTypeMandatoryFields and EventField are all replicated from the Governance domain and are represented in a darker colour. The Event, EvenInfo, Lot, EvenInputLot, EventOutputLot and EventInfo are the tables used to store all the traceability information.

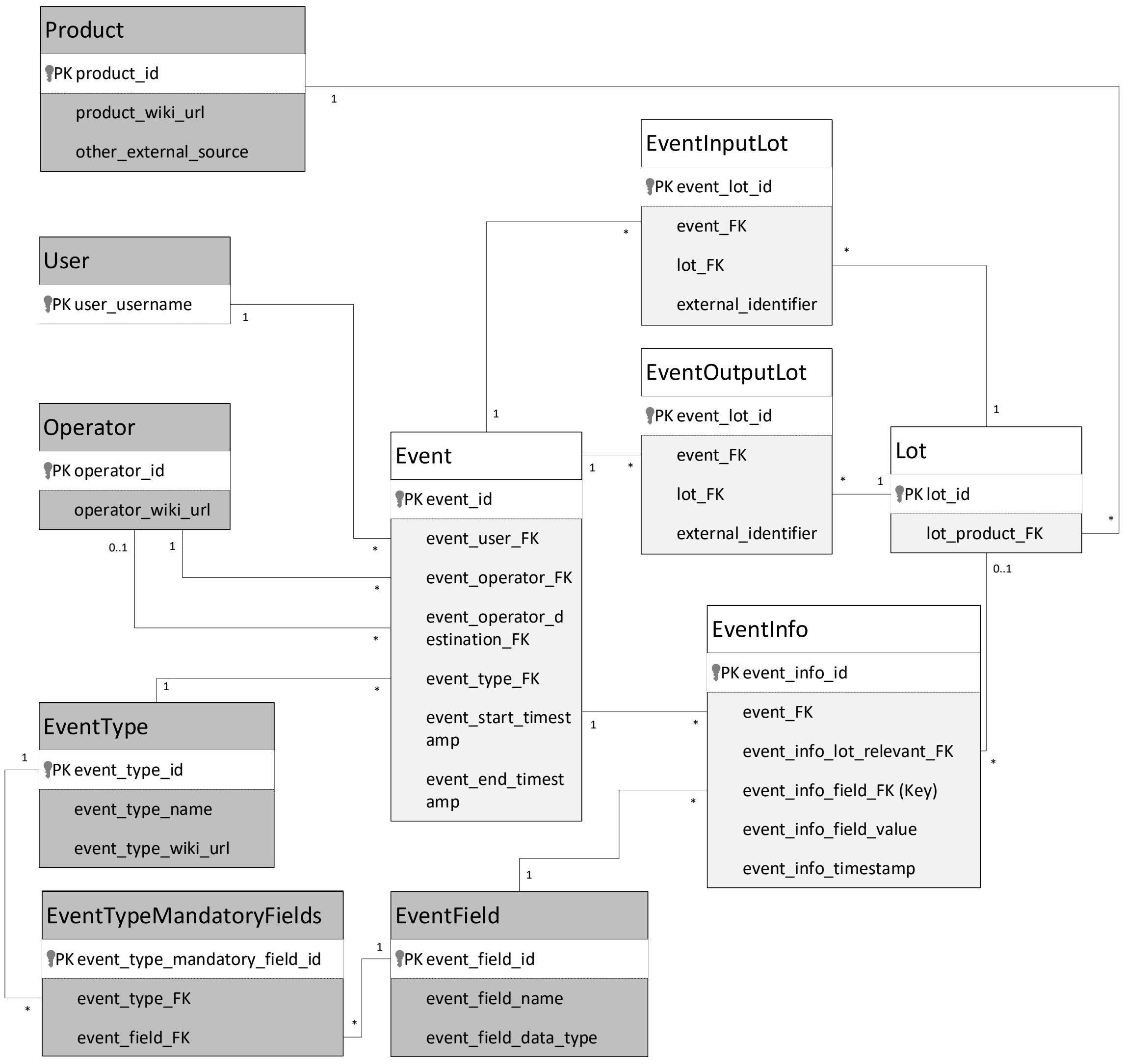


Figure 21 Tracking Data Model

4.6.2 Architecture and Implementation

The focus during the implementation of this microservice was enabling asynchronous insertion of traceability Events through a REST API. The requests from the integration applications are routed through the API Gateway to a WSO2 Micro-Integrator instance that registers the requests in Kafka to be processed and persisted in a database. This process is done by a .Net Core application that consumes requests from Kafka and inserts them to the Microsoft SQL Server engine used with the Entity Framework ORM (Object Relational Mapper). In later instances of development other interfaces were created to push traceability information to the middleware using table files. These files can be sent attached to emails or through a REST API and are then processed into the same format as the requests from the original API to maintain the remaining event insertion pipeline. During the insertion of the events to the database the rules configured for that event type are enforced. These configurations are present in this database by using Kafka Connect to replicate the information managed by the Governance domain.

The complete flow of information requests can be seen in Figure 22, with the client applications on the left using the API Gateway to send their requests.

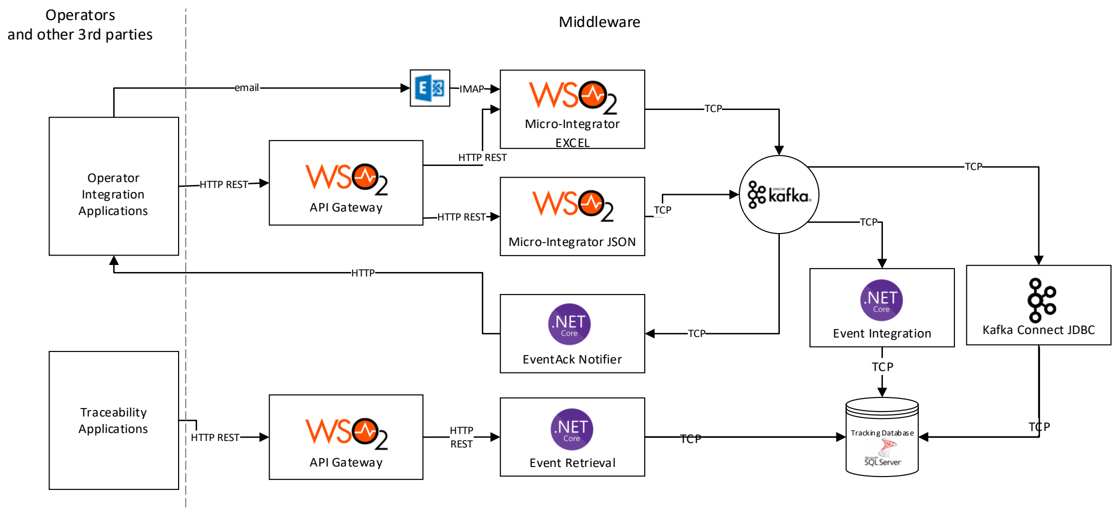


Figure 22 Tracking Microservice Implementation Diagram[68]

The reply to the HTTP requests for event insertion only acknowledges that the middleware as queued the request for processing with the proper identification. The acknowledgement for the correct persistence of the information or the error report in the non-complying cases is sent via HTTP to a REST API. This requires the integration applications to implement this functionality with a REST API to enable this feature.

The event retrieval and traceability reports are exposed through the API Gateway which routes to REST API implemented with .Net Core and Entity Framework to query the database and return the result.

4.6.3 Excel/Table File Insertion

This type of insertion was done as an exploratory project and in two different attempts. The attempt I focused on was done in a push architecture where excel templates were generated for specific event type definitions in the governance domain. These templates could be requested using that REST API and would be filled out by an operator to be inserted into the system. For this insertion, two methods were tested:

1. Email attachments,

2. REST API.

The REST API approach was easier to integrate with the other components of the middleware as the same security features as the other APIs from the API Management layer could be applied. Other communication protocols were suggested such as FTP, but it would be much harder to implement a collaborative environment where multiple entities can push files into the same system. This approach overall I do not think is good as it forces the operators to adopt our Excel templates while the earlier goal was to provide methods to integrate existing operators' systems that operate using this format and perhaps the effort of implementing such solution might be even harder than just using the REST APIs with JSON payloads. Could still be useful for operators that will manually insert the information as an alternative to the REST API using forms.

Another approach was attempted with a pull architecture by another developer with a daemon specifically designed for an operator do pull their daily generated CSV files which was also abandoned due to lack of cooperation and could have set a precedent where the middleware would have to take on this effort of integration for each different operator traceability system.

4.7 Monitoring Microservice

This domain is responsible for a post commit analysis of every traceability event inserted into the middleware. It is composed of three independent applications developed in Node.js that communicate using Kafka and use MongoDB for the persistence needs

4.7.1 Configuration and Event Definitions

The monitoring of the events is done through triggers that can be configured in the Backoffice dashboard which uses the API implemented by a monitoring specific governance component. This component combines the event definitions from the governance domain with the trigger definitions to make the decisions regarding which events generate alerts and where those alerts are pushed to.

4.7.2 Event Reception and Analysis

The traceability events are received in a Kafka topic from the tracking domain after being properly validated and inserted. In the Event Analysis component, the rules specified in the configuration component are applied and if triggered, a notification is generated and pushed to a Kafka topic for later processing.

Other planned features regarding the analysis of the events were the collection of aggregate data such as totals of traceability events and amounts of product traced. This would be done by processing the events and storing them in ElasticSearch which is an analytics engine that could compute those types of aggregate queries if the information is present.

4.7.3 Notifications

A monitoring notification component is responsible for consuming the notifications generated from the Kafka topic and pushing it to the correct recipient. This can be done through multiple communication channels, currently supporting Web Push notification or Emails, with the option of adding more consumers for new types of channels to expand the system.

4.8 Playback Microservice

The purpose of this microservice would be to provide historical data of the middleware's usage, mainly using Kafka's message retention and persistency features. This microservice was not developed very much as the entity within the project responsible for the data science withdrew very early on. Nonetheless, a fully working REST API to access this information was still developed and deployed.

To achieve the purpose of exposing the Kafka messages contained in the topics, Confluent's Kafka REST Proxy[84] was considered but with some concerns were raised regarding the customization of the solution as the initial goals could require custom permissions or filters. Another caveat of this technology is that it requires a paid license which went against the project priority of using open-source tools.

The final implementation ended up being done using .Net Core to provide a REST API that allows for the querying the Kafka cluster using a consumer. This was done in a very generic way as the arguments for the API are the topic, partition and offset and allow for any messages stored in the cluster to be exported. This serves as a basis for later development if more specific methods are required or filters and permissions with also the possibility of merging information for multiple topics.

4.9 Users and Authentication

In this domain the main goal was to store information about the users registered in the middleware, their permissions and to provide authentication and authorization methods for the other domains' services.

4.9.1 Authentication and Authorization Standards

Regarding REST APIs over HTTP, there are a few options for authentication and authorization. At the lowest level, the Authorization header is generally used to transmit some credentials for either purpose. These headers are usually of the Basic or Bearer type with the first one using encoded credentials and the latter using cryptography-based tokens generated by the server during login requests. Both should be used over HTTPS to protect the credentials from a MITM[85] attack.

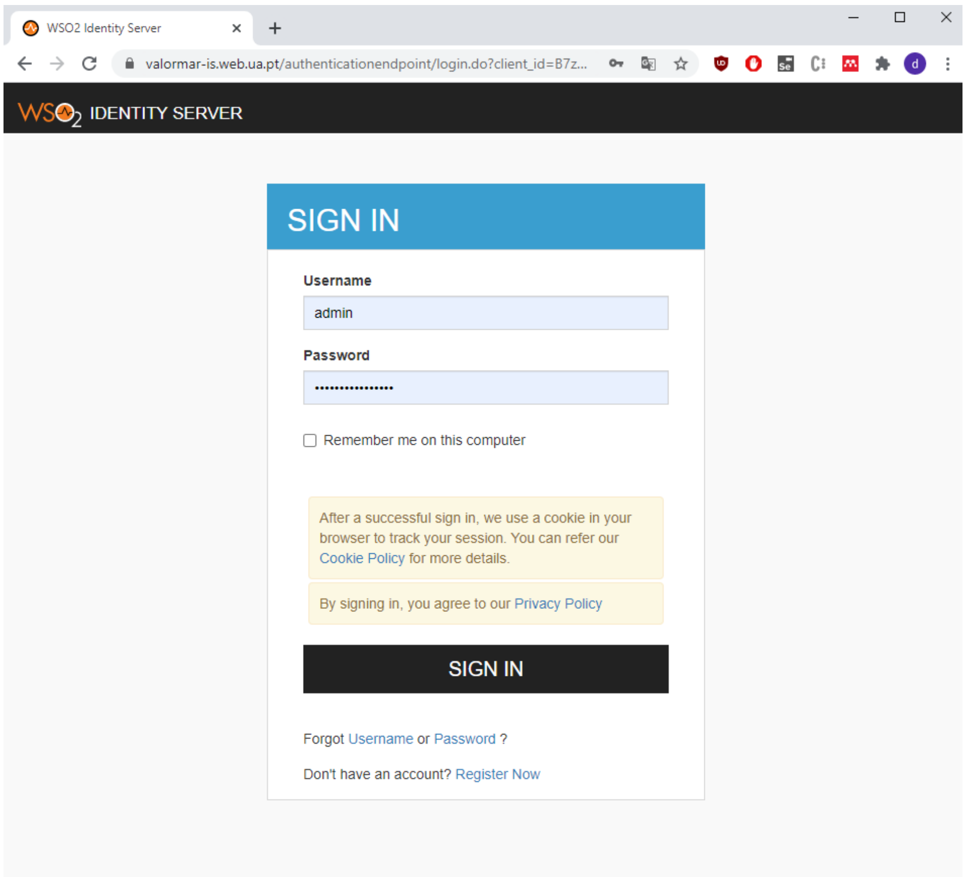
The advantage of the previous methods is that they can be used to contextualize the user in the API. Other approaches are also used such as API Keys, an early approach to REST API security that works well in cases where the APIs are read-only, and no user specific permissions are required and application authentication is enough.

4.9.2 Single Sign On with OpenIDConnect

Authentication is done through the OpenID Connect[86] protocol. OpenID Connect adds an authentication layer to the Oauth2 authorization protocol. This authentication is done through an id_token that contains the user's identifier and some of its attributes.

Client applications are registered with the Identity Server and the attributes to be exported are defined at the end of the authentication process. There are several flows available for different types of clients.

In the case of authentication of a human user in web applications, there are the authorization code and implicit flows that work through redirects to the WSO2 Identity Server authentication pages and back to the application.


WSO2 Identity Server

valormar-is.web.ua.pt/authenticationendpoint/login.do?client_id=B7z...

WSO2 IDENTITY SERVER

SIGN IN

Username: admin

Password: ••••••••••••••••

☐ Remember me on this computer

After a successful sign in, we use a cookie in your browser to track your session. You can refer our Cookie Policy for more details.

By signing in, you agree to our Privacy Policy

SIGN IN

Forgot Username or Password ?

Don't have an account? Register Now

Figure 23 Login Page WSO2 Identity Server

Currently the discovery endpoint of information about the Identity Server is https://valormar-is.web.ua.pt/oauth2/oidcdiscovery/ and is used by some applications to discover the Identity Provider methods.

For application authentication there is flow client credentials where only the application's credentials are used.

These flows are documented in WSO2's documentation website[87] and there are also libraries that can facilitate the integration of OpenID Connect in client applications.

4.9.3 User Profiles and Claims

The three user profiles planned for the platform were implemented using the groups/roles available on the WSO2 Identity Server. The great advantage of this is that we can block access to the API methods at the Gateway through this information.

The platform administrator and operator profiles have access to all methods, while operator users can only access the methods of reading information and entering events.

The integration of this functionality with the rest of the system was done through the SCIM2 API that allows to consult and modify users. When a user's profile is changed by BackOffice, this change is made in the management domain and then replicated in WSO2 IS with the API. To ensure the consistency of the information between the 2 systems, data replication in the management domain is also made when there are changes in the WSO2 version of the information. This is done with an OSGI bundle developed in java to communicate user changes to a Kafka topic that is consumed by a connector connected to the Governance domain database.

User attributes are stored in OpenID Connect claims. The predefined list has about 30 attributes that can be customized. Applications registered through the Developer Portal only receive the user's identifier in the ID token but any of the claims can be exported if necessary, with different rules for each application.

4.9.4 Identity Federation and Shibboleth Example

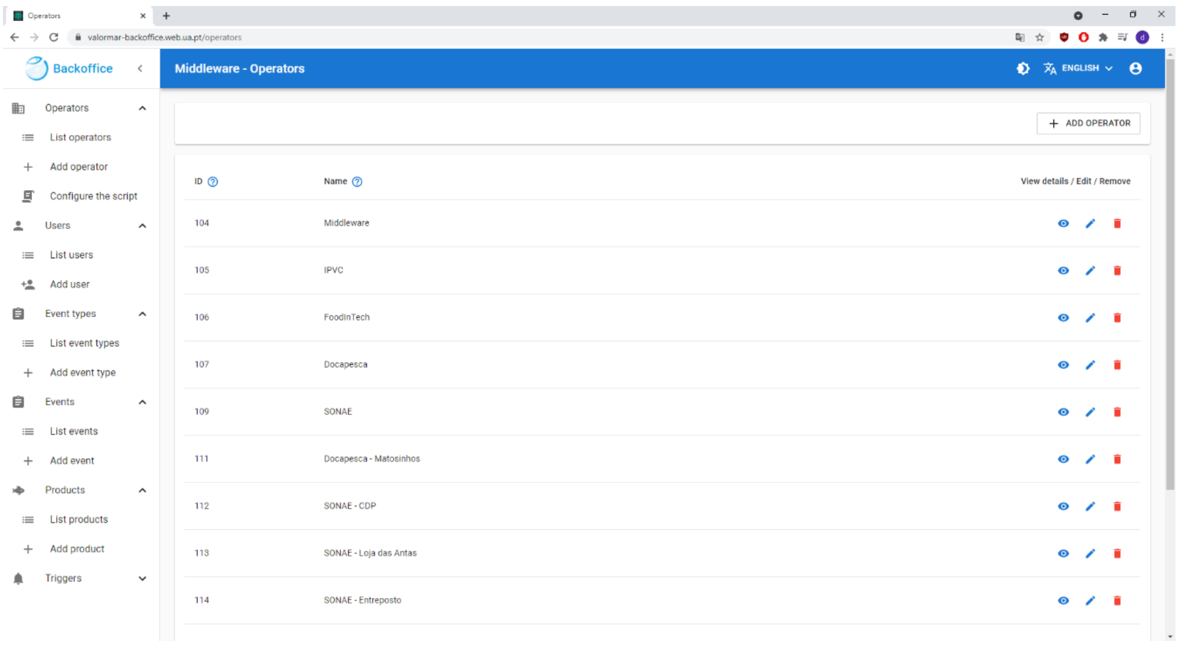
Another concept that we find of interest to middleware related to authentication is federated authentication. The WSO2 Identity Server allows the delegation of user authentication to another Identity Provider. In the case of operators with existing Single Sign On systems, this can facilitate the management of users and the sharing of attributes. The WSO2 Identity Server allows integration with the protocols SAML, SAML2, Oauth1.0, Oauth2.0, OpenID, OpenID Connect and includes integrations with some social logins like Google and Facebook that also use the same protocols.

To test the capabilities of the system, a test integration was made with an Identity Provider SAML2[88], more specifically the Shibboleth Identity Provider[89] used for SSO at the University of Aveiro.

4.10 Backoffice Web Application

The backoffice web application allows for the management of the governance related entities of the middleware. Non-anonymous users may login and the application features are restricted based on their role. Regular Operator Users only have access to a read-only view of some elements and can insert traceability events while the Admin roles can create and edit the remaining information.

Figure 24 depicts the backoffice menu where all the operators can be viewed from a Platform Admin perspective with dummy information from a real-world integration scenario.



ID	Name	View details / Edit / Remove
104	Middleware	
105	IPVC	
106	FoodInTech	
107	Docapesca	
109	SONAE	
111	Docapesca - Matosinhos	
112	SONAE - CDP	
113	SONAE - Loja das Antas	
114	SONAE - Entreposto	

Figure 24 Backoffice Web Application - Operator menu

This web application is responsive and was developed using ReactJS[90] with the MaterialUI[91] theme. For the authentication with the Identity Provider the OpenID Connect implicit flow was used, redirecting the user to the provider's login page with the application credentials and after the authentication is successful, the user gets redirected back to the Backoffice application with the access_token and id_token. This access_token is used in the authentication header for all the requests made to the APIs via the gateway and the id_token is where the application extracts the user claims like the username and the roles which limit the user's permissions in the application.

4.11 External Client Applications

Other entities working in this PPS4 subproject also developed components to interact with the middleware. FlowTech developed operator integration application that using the .Net framework for desktop application development.

IPVC developed a kiosk and a mobile application to query and display the traceability information. The architecture of the kiosk solution is described a published article[92]. This kiosk was designed to support multiple devices in a shop or supermarket, using a Zero Configuration network model to keep all the nodes synchronized. Another article was later published[93] with further refinements to the application and usability tests conducted using the System Usability Scale (SUS) method. Regarding the mobile application, a journal[68] has been accepted and published by Elsevier to the Journal of Agriculture and Food Research resulting of a collaborative effort from the middleware and client application teams that provides an overview of the middleware's traceability functionalities and its usage with the mobile application along with usability test feedback and load tests of those traceability APIs which are also present in this document.

5 Middleware Operational Management

After the early phases of prototype development, challenges appeared regarding the deployment of the middleware. The high complexity of the solution developed and the number of distinct components to manage paired with the requirements of testing the system and maintaining a staging environment for the application developers to use forced us to adopt technologies and methodologies that simplify the deployment process. During the early stages, Docker was used as a container engine. We standardized all the components into containers to ease the manual deployment process, which was later replaced with container orchestration using Kubernetes provisioned using automation tools such as Ansible.

5.1 Docker and image creation

Docker containers are lightweight, standalone, ready-to-run software components that package everything that is strictly necessary to run an application: code, libraries, dependencies and configurations. One of the first steps taken was to standardize all the software components of the middleware into Docker images. A lot of the software used was already available in the public Docker image registry[94]. Some of these images are provided and maintained by the original developers of the software while others are developed by the community so some caution must be taken to pick the best image available as some might be missing features or broken.

For the developed applications, some customization that was required with some of the software we created our own images based on the ones provided by the vendors such as Microsoft's .Net Core runtime images and the Mediawiki image.

To distribute these custom images a local Docker Registry was deployed and configured. This image repository is available for free and maintained by the Docker developers and has the basic image storage and version control functionalities.

One challenge to install this Docker Registry was the security related configuration since to enable SSL a properly signed certificate with the corresponding and reachable hostname was mandatory as there were issues when attempting the usage of self-signed certificates. To get around this problem a valid certificate was requested from the University's IT department along with the registration of the hostname in the local DNS server.

For the staging environment, these docker images were deployed across some Ubuntu 18.04 LTS VMs that acted as docker engines. Initially these VMs were supposed to act as Docker swarm and the deployment was to be done with horizontal scalability and other availability related features but due to the lack of knowledge on the clustered deployment of some of the components at the time and later the decision to adopt Kubernetes for the final production deployment only 1 instance of most of the components was deployed using either docker-compose or just the standalone container deployment commands.

5.2 Kubernetes and Container Orchestration

Container orchestration is the automatic management of runtime containers, normally used in microservice based applications. A container orchestration tool is a centralized software component that handles the distribution of the applications across the deployment environment's machines allowing for the efficient management of resources. As stated at the end of the previous section the first choice of software for this purpose was Docker Swarm. This choice was made due to Docker Swarm being more lightweight and easier to learn than Kubernetes. After the first attempts its limitations started to show, namely the lack of features such as a UI or auto scaling of components. Although a UI could be installed using Portainer[95] and the auto scaling could be achieved using a monitoring system and triggers to change the system when under certain conditions, this would seem to require more effort to implement than to adopt Kubernetes. Other advantages gained from the change were better volume definition and management functionalities and more advanced load balancing features.

Just like Docker Swarm, Kubernetes also uses yaml files for the configuration of the components to be deployed but there seems that a lot more community projects and examples are available for Kubernetes than for Docker Swarm. Helm[96] can be used for version control of these types of configurations acting also as a repository. There are many Helm repositories with deployment configurations for the most popular software being widely available and sometimes even provided by the vendor themselves.

The development process for this environment consisted of searching for adequate Helm charts for all the software required as well as developing our own charts for the applications that were created. Some of the public charts come well prepared and only some configuration tweaks are required to deploy a cluster of that tool. In other cases, substantial changes were made to the charts to achieve the required purpose and some even were outdated or incompatible with the remaining setup.

Each Helm chart contains at the root a Chart.yaml file which contains a description of the chart and some other repository related metadata. The values.yaml file is usually used for all the configurations related to the software itself along with the deployment specifications. Any entry in this file can be accessed from the deployment files and this enables the development of dynamic deployments.

The Helm charts developed required multiple types of Kubernetes resources. For example, a stateless REST API application requires a Deployment file where the hardware allocation for the pods is defined as well as the image to be deployed, update strategies and replication constraints. This Deployment then requires a Service resource for the routing of the requests to all the pods of that type in the cluster. In the case of stateful applications a StatefulSet is used instead of the Deployment which creates a set of unique pods with persistence and unique hostnames. To allow external access to the cluster, an Ingress resource is used which can create HTTPS Nginx reverse proxies with auto generated certificates and hostnames. These certificates and hostnames can also be added manually for the final production deployment with a reachable hostname and a valid certificate. Software related configurations can be added with multiple approaches. For environment variables and other similar arguments that would be added during the container deployment these can be passed to the container from the values.yaml file in the Deployment file. However, for file related configurations which are also common in the software used, configuration volumes are required which are created using ConfigMap resources. These resources allow the loading of files from the chart directory for configuration purposes. These files are then mounted into the containers in the Deployment resources, overriding the defaults present in the image used. Software that requires persistence of files also required further attention, PersistentVolumeClaim resources can be used to create volumes access control rules which tailor the type of volume to the specific use case. This allows for the creation of single shared volume accessible by multiple pods or assign separate storage blocks to every node in a deployment, commonly used when the software cluster presents some form of sharding of the persistent information like Kafka or MongoDB.

5.3 Cluster Installation and Automation

The main goal of employing automation tools in this project was to provision a dynamic Kubernetes environment for the deployment of the complete middleware with the orchestration detailed in the previous section.

Ansible is the automation tool chosen for this purpose as it provided all the features needed and had a configuration driven approach. There are also many community driven ansible repositories available with templates to deploy commonly used software such as container engines or databases.

As stated before, the computing resources allocated to this project are in the form of an OpenNebula private cloud hosted on campus so these automation scripts are specialized to manage virtual machines on that cloud, but they can easily be adapted to work with another Cloud Management Platforms. In Ansible the resources are referred to as inventory which can be fixed or dynamic. This private cloud contains four physical hosts, two with Intel Xeon E5-2670 v3 and the other two with Intel Xeon X5675 CPUs. Each physical host contains two CPUs, with the total cloud resources 72 cores, 144 threads and 785.9 GB of RAM.

The inventory is usually described in the inventory file and when the inventory is dynamic there is a script that interacts with the desired cloud management platform and generates the inventory file. Templates for inventory generation are available for OpenStack, OpenNebula, AWS, Azure and Google Cloud so it is possible to interchange or even mix cloud providers while maintaining the same deployment scripts for the middleware although some of the platforms mentioned provide not only IaaS but also PaaS which may have other proprietary tools that provide Kubernetes clusters or even more software specific like database engine clusters.

For OpenNebula the ansible module available is the one_vm_module[97] which allows for the creation and deletion of virtual machines when provided with proper credentials and the specification of the virtual machines to be created. After the VM creation the playbook developed that invokes the one_vm_module generates an inventory script with the hostnames of the generated VMs.

The VMs were generated using an Ubuntu 18.04 template imported from the OpenNebula repository although custom templates were also created from scratch by installing the operating system on an empty VM during development.

After the inventory file is generated the Kubernetes installation playbook may be executed. This playbook is from the Kubespray[98] project which provides ansible scripts to install Kubernetes clusters with all the HA features and supports a wide array of Linux distributions.

The inventory file also specifies the roles of each VM in the cluster. HA Kubernetes clusters have control plane nodes which contain the controller and scheduler components responsible for the container orchestration as well has the api server that allows the Kubectl client to manage the deployments. In production environments these nodes might have the deployment of pods disabled to isolate the control plane features of the cluster from the applications.

Another component required in a Kubernetes HA cluster is an etcd[99] cluster. Etcd is a distributed key value store which Kubernetes uses to store all the information about the configurations and deployment instructions for the applications. This cluster can be deployed independently from Kubernetes or stacked with the master nodes. Kubespray's default settings deploy an uneven number of etcd nodes stacked with the other components on some VMs. The number of nodes must ben uneven to avoid deadlocks during the etcd master node election process. The stacked topology can be viewed in Figure 25.

The worker nodes are the final component responsible for deploying the Kubernetes Pods and are connected to the master nodes through a load balancer.

The downside of using stacked etcd instead of the standalone deployment is that the cluster loses redundancy since if one of those nodes fail both an etcd and a control plane node are lost, so the official recommendation is to have at least 3 stacked etcd and control plane nodes to mitigate this problem. On the other hand, the standalone deployment would require double the nodes for control plane and etcd and due to some networking constraints, the cluster would end up not having as many resources for pod deployment.

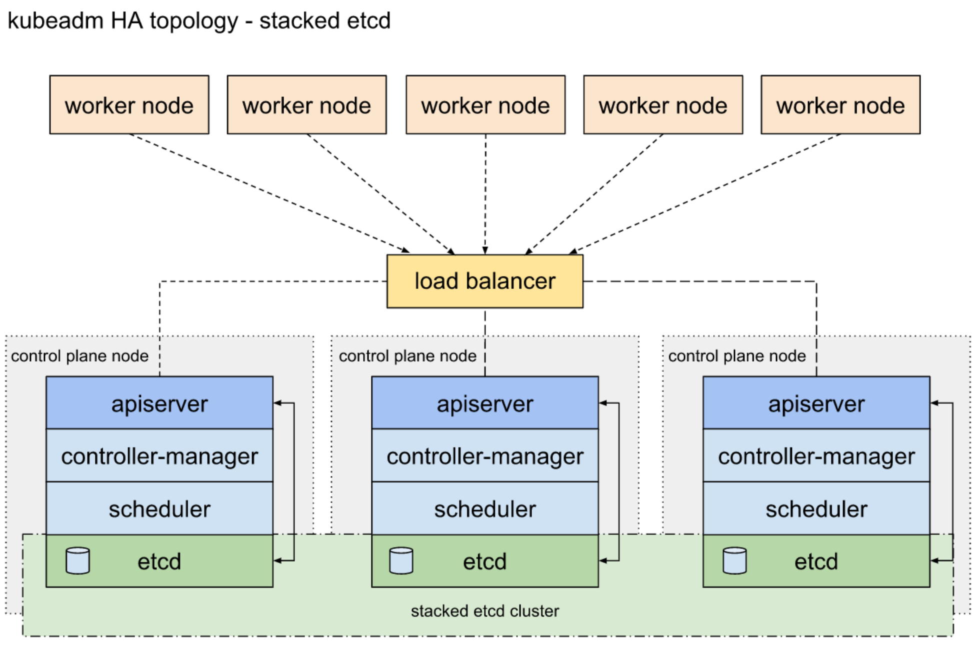


Figure 25 Kubernetes HA Topology - stacked etcd (source: kubernetes.io)

Regarding the configurations of the cluster most of the defaults provided by kubespray are enough to achieve this HA deployment but there were some issues to resolve.

One of the issues was ip collisions with the university’s VPN. The VPN uses the ip ranges that Docker uses by default to create the internal networks so docker configurations were added to the docker.yml file of the inventory’s group variables.

Some optional features were also enabled such as the dashboard, Helm, metrics server and nginx ingress to provide layer 7 load balancing to the applications across all the nodes.

Storage wise there are many options of connecting the storage hardware to the Kubernetes cluster with some plugins specialized for many cloud providers’ storage options like OpenStack Cinder or AWS Elastic Block Store. In our project there were no storage resources allocated besides the space allocated for virtual machines so the only way to provide storage for the storage would be creating a virtual machine dedicated for this purpose and exposing it with an NFS server. To simplify things as this was just for testing purposes a local volume provisioner was used that would allow volumes to be created from a node’s storage. These volumes would not be shareable between pods which was required in some cases so the NFS Server chart was used to use this storage class to create a new one which would create virtual NFS volumes as needed.

The result was a 5-node cluster with 2 control plane nodes and 3 etcd nodes with all the nodes being allowed to deploy pods. Each node was allocated 32 GB of RAM and 16 physical CPU units exposed to the VM as 16 virtual CPU units with a 100 GB storage share of the cloud’s file storage.

Table 1 Kubernetes cluster node roles

Node/Role	Control Plane	Etcd	Worker
node1	x	x	x
node2	x	x	x
node3		x	x
node4			x
node5			x

When testing the cluster some issues appeared with the performance of the cluster. The deployment of the applications was extremely slow and the etcd nodes would fail making the cluster unavailable a lot of the time. A lot of troubleshooting was done to find the problem, this was one of the most challenging roadblocks in the entire project.

After touching a lot of the configurations in kubespray and installing the cluster several times in different operating systems and increasing VM resources the problem continued, sometimes even forbidding me from connecting to the VMs with ssh. At this point I even wondered if it could be a physical hardware issue as no other projects were using this cloud for the same purpose, so I focused my troubleshooting on the virtual machine's resource usage and spotted that there was always a ksoftirqd thread with high load. Ksoftirqd is a per CPU kernel thread responsible for queuing interrupt requests when these arrive in high volume and the OS cannot handle them fast enough. Heavy network traffic can cause a flood of interrupt requests and in this case the Etcd and Kubernetes' components communications were responsible for the cluster's instability.

To resolve this problem some newer versions of Kubernetes and Etcd were tested to no success moving on to looking at the VM's network card drivers. After learning that the virtual network card installed was based on a Realtek driver and that those drivers were not recommended on heavy load environments some further research into VM optimization for OpenNebula were done leading to this blog post[100] which contained instructions on how to switch the default Realtek card to another more high performance reliant on Red Hat's virtual networking drivers by changing the NIC model of the VMs to "virtio" in the configurations.

5.4 Deployment

With the Kubernetes cluster working the next step was to deploy all the software with the same configuration as the Docker staging environment. For the developed apps custom deployments had to be created while in the other cases the vendor provided examples were used as a base and merged with the configurations specific with each software achieved when configuring the staging environment. These deployments were defined using Helm charts with a chart being used to deploy each software independently.

5.4.1 WSO2 APIM and IS

The WSO2 Developers provide sample Helm charts for the deployment of clusters of their software. The charts used in this project are available in the kubernetes-apim[101] and kubernetes-is[102] repositories in WSO2's Github account.

Both charts were required as we intended to use WSO2 Identity Server as the Key Manager in the APIM cluster. Instead of ending up with 2 separate charts highly dependent on each other, the default Key Manager configurations from the APIM chart were merged with the files from the IS chart creating a new chart with some configuration changes to tailor the cluster to our requirements. One component added to the pods was Secrets containing a KeyStore with the certificates to be trusted and the keys to be used. This was done to ensure that the SSL handshake between the component’s communication would succeed and to allow for the generation of valid JWT tokens for any entity trusting that self-signed certificate. The default keys included do not allow this as the corresponding certificate belongs to the localhost hostname.

The remainder of the WSO2 cluster components were deployed with mostly default configurations with some tweaks to the resource allocation and image versions to use the latest software.

To take full advantage of WSO2 APIM’s feature segmentation, deployment pattern 3 (Figure 26) was used where each role of the cluster is deployed in a separate pod group. The APIM Docker image by default uses a standalone deployment where every node can fulfil every role in the cluster. In pattern 3 the images are specialized for each role with role specific configurations which are inserted using ConfigMaps.

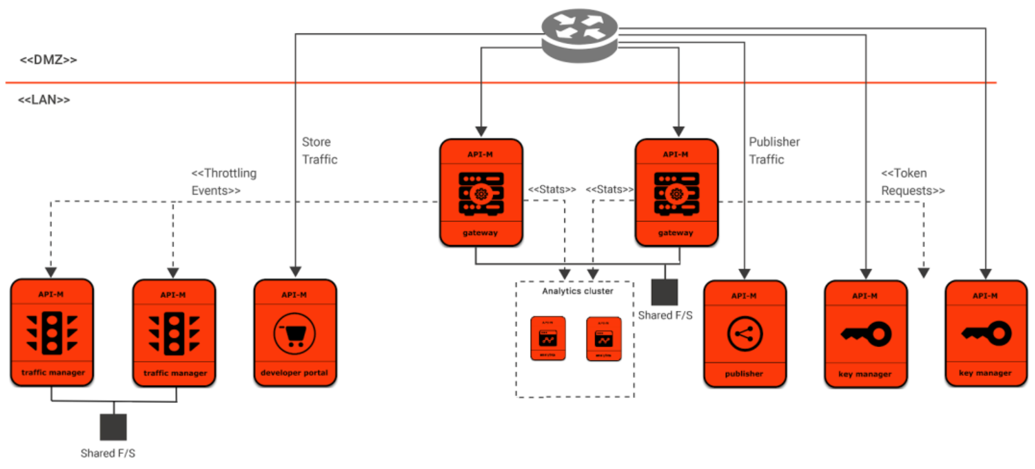


Figure 26 WSO2 APIM Deployment Pattern 3 (source: github.com/wso2/kubernetes-apim)

With Identity Server acting as the Key Manager, a separate Analytics Server cluster can be added to monitor this component which was not present in Figure 26.

For each component at least two replicas were deployed to have redundancy with all the components and allowing for system to operate normally in the case of failure of any node.

In theory, the components that will be put under higher load are the Gateway components, responsible for the routing of all the middleware’s inbound requests and the Key Manager components responsible for token generation and validation so these components will be the ones where we focus our performance evaluation during the load tests. The Traffic Manager components will not be placed under much load during the validation since we did not adopt rate limiting policies to any of the APIs.

5.4.2 WSO2 Micro Integrator

For the deployment of the WSO2 Micro Integrator instances required the official tutorial[103] was used as a base for the development of the Kubernetes resources needed which were then added to each data domain's respective Helm chart.

Since all the nodes in a WSO2 Micro Integrator cluster is stateless, Deployment, Service and Ingress roles were created. Software wise no configurations were required due to the CI pipeline used for the creation of images. These images come with the correct configuration which is created during the development process in the Integration Studio IDE. The integration projects are compiled into CAR files which contain the service definitions and registry entries. These projects can then be further packaged into docker image with the IDE built in tool that generates Dockerfiles and the WSO2 Micro Integrator configurations in the deployment.toml file.

There were some issues with the IDE's execution of the Dockerfile build process which uses maven due to the local Docker host's configurations which was the main reason for the development of the pipeline to create and push the Docker images. This pipeline requires a bash script in the project's root to run all the Maven commands that are required to package the solution.

For these images to be accessible by the Kubernetes cluster the Helm chart also contains configurations in the values.yaml file to define the repository and image name to be used as well as a secret with valid credentials for the project's private Docker Registry.

5.4.3 .Net Core Services

Just like in the WSO2 Micro Integrator case, all the Docker images for our .Net Core applications are stored in the private registry and the charts that require these images also contain the credentials to pull them.

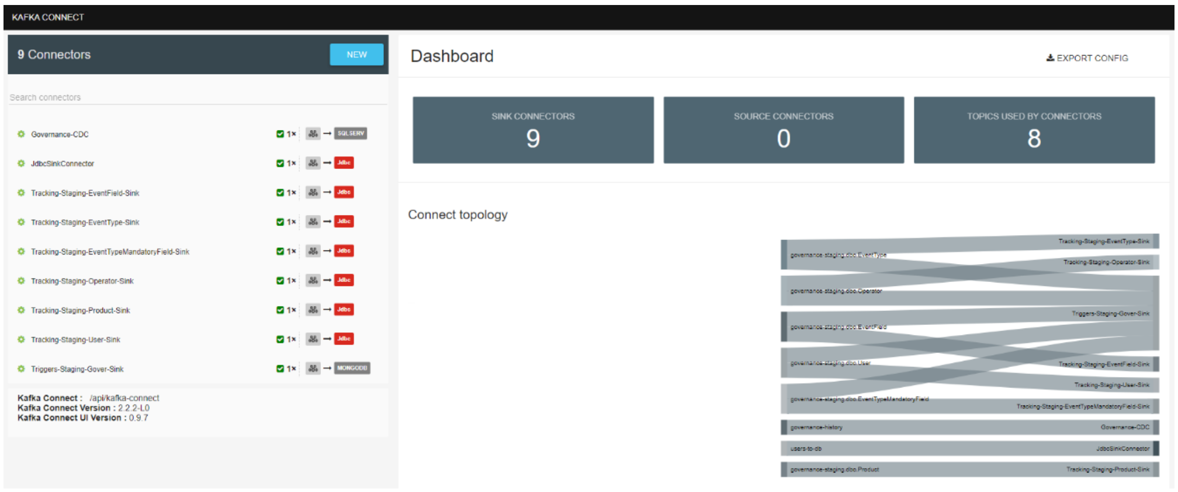
The .Net Core containers are all stateless and contain a Deployment, a Service and an Ingress Kubernetes roles along with a Config Map that contains the appsettings.json where all the application specific configuration are defined.

5.4.4 Kafka

Apache Kafka and all the auxiliary components were some of the most complicated components to deploy. Due to its stateful nature and persistence requirements the Helm charts are much more complex. Another problem was finding the most suitable open-source images for all these components as Confluent's charts are the most complete but require a license after the testing period of 30 days.

The final solution was a chart to deploy the Kafka cluster with the other components in their own charts as dependencies. Regarding Kafka and Zookeeper, bitnami's[104] images and charts were used since they seemed to be the most complete open-source options at the time.

Regarding the Schema Registry, Kafka Connect and dashboard tool, Confluent's charts were used. The plan was to emulate the Lenses setup[105] used in the staging environment which provided a web UI to configure the Kafka connect pipelines as shown in Figure 27.


KAFKA CONNECT
9 Connectors
NEW
Search connectors
Dashboard
EXPORT CONFIG
SINK CONNECTORS
9
SOURCE CONNECTORS
0
TOPICS USED BY CONNECTORS
8
Connect topology
governance-staging.dbo.EventType
governance-staging.dbo.Operator
governance-staging.dbo.EventField
governance-staging.dbo.User
governance-staging.dbo.EventTypeMandatoryField
governance-history
users-to-db
governance-staging.dbo.Product
Tracking-Staging-EventType-Sink
Tracking-Staging-Operator-Sink
Triggers-Staging-Gover-Sink
Tracking-Staging-EventField-Sink
Tracking-Staging-User-Sink
Tracking-Staging-EventTypeMandatoryField-Sink
Governance-CDC
JdbcSinkConnector
Tracking-Staging-Product-Sink

Governance-CDC	1×	SQLSERV
JdbcSinkConnector	1×	Jdbc
Tracking-Staging-EventField-Sink	1×	Jdbc
Tracking-Staging-EventType-Sink	1×	Jdbc
Tracking-Staging-EventTypeMandatoryField-Sink	1×	Jdbc
Tracking-Staging-Operator-Sink	1×	Jdbc
Tracking-Staging-Product-Sink	1×	Jdbc
Tracking-Staging-User-Sink	1×	Jdbc
Triggers-Staging-Gover-Sink	1×	MONGODB

Kafka Connect : /api/kafka-connect
Kafka Connect Version : 2.2.2-L0
Kafka Connect UI Version : 0.9.7

Figure 27 Lenses IO Fast-Data-Dev Kafka Connect Dashboard

One major issue with the development of these Kafka Connectors was that they contain a lot of database engine endpoints and credentials which change from one environment to the other. For the staging and testing environments this change was done manually with the developer having to create different connector versions for each environment.

The solution for the problem above was having environment variables used in the connectors to parameterize them and allow for the same version to be used across many environments with each connector being built as its own image and the environment variables would be inserted during the container instantiation. This feature was not fully developed and tested as the main goal of this environment was the load testing and for this purpose, the databases were pre-loaded with all the required information that would be migrated across domains, so it ended up not being prioritized.

5.4.5 Mediawiki

As stated in the portion of the development section regarding the Wiki, a custom image was created with all the required extensions. The wiki image is deployed in a Helm chart along with all the other wiki related components like Parsoid and the MI service converter.

The Wiki is stateful as all the requests from the user must be made to the same node to maintain the correct state of the user's session, therefore a StatefulSet was created for the mediawiki instanes. Connected to these pods is a shared ReadWrite volume for the storage of the images which are usually stored in the filesystem of the machine running the engine with some metadata entries in the database. The StatefulSet also has a ConfigMap that contains the LocalSettings.php file where all the wiki's configurations occur.

Parsoid is stateless and all the required configurations are handled with environment values so only a Deployment was required. Both Mediawiki and Parsoid were then exposed and load balanced using a Service and an Ingress Kubernetes resource.

5.4.6 Elastic (ELK) Stack

For the monitorization data domain, ElasticSearch and Kibana[106] clusters had to be deployed. These instances were not used for the monitorization of the Kubernetes cluster as that component should be installed in a separate environment for availability concerns. The purpose of ElasticSearch in the context of the monitorization data domain is to receive information from the Kafka broker to create an index that enables the usage of aggregation queries over the traceability events collected. Kibana would be used to create visualizations from the ElasticSearch indexes generating dynamic graphs that could be embedded in the web applications.

The Elastic Stack developers provide Helm charts for all their software and the default settings are enough for our purposes. ElasticSearch was deployed with three replicas with a minimum of two nodes having the master role and Kibana with two replicas and sticky sessions to maintain the user's dashboard session. The remaining components of the stack were not required therefore discarded as none of the data collection components was required since that role would be handled by Kafka Connect instances pushing information into ElasticSearch but they may be added in the future if suitable sources of information are available.

5.4.7 Database Engines

Several database engines were used during the development of this project either when developing new applications or as a requirement of the software. Some of them were used for collaborative development purposes so credentials and permissions had to be configured and distributed in all those cases. Some of the technology choices were made due to compatibility constrains, namely Mediawiki which dropped support for MS SQL Server with the 1.34 release, so MySQL was used instead, ending up with 2 relational database engines that are very similar to prevent issues in the future.

5.4.7.1 SQL Server

Some compatibility issues prevented SQL Server from being deployed in the Kubernetes cluster. The image for the container orchestrator supplied by Microsoft at the time of development was not compatible with the latest version of Kubernetes (v1.18) which had several changes to its management APIs.

After deciding not to downgrade the cluster due to all the other software already deployed, an attempt was made to deploy a cluster using Windows VMs but that idea was scrapped after realizing that Windows VMs were limited to 4 CPU cores.

The final solution was based on Microsoft's GitHub SQL HA examples[107] which provide Ansible playbooks to install and configure the cluster on Linux VMs. For this purpose, 4 nodes with Ubuntu 18.04 images were used with 12 CPUs and 12 GB of RAM. One of the nodes will act as the master and was configured with 2 virtual networking interfaces with one being dedicated to the availability group router component of the cluster which is responsible for routing the requests across the nodes. All the nodes also must have static routes to each other with their hostname since ansible registers the hostnames of the machines during cluster creation to establish connectivity between the nodes.

Following the official instructions, the Vault.yml file must be configured with the proper credentials and the inventory file with the roles for each VM. Before the script can be executed Microsoft's repository must be added with the following command: "sudo curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -".

After installation, some tweaks must be done to allow for the horizontal scaling for read operations. For best performance, the main node was left out of the read only routing group as it is the only node that can write to the database while the remaining three were all part of the group.

With the cluster configured databases were deployed manually using SMSS and DACPAC files with sample data to validate the cluster with load tests to the API that consumes the database to ensure that the read-only transactions were being equally distributed across the worker nodes.

5.4.7.2 MySQL

In the Kubernetes environment MySQL was only used for the WSO2 Registries so far. In Mediawiki's case the staging environment was already in use and being populated with information, so the migration was delayed. The Kubernetes wiki instance uses the same database as the staging one as it provides usable information for testing.

For the WSO2 Registry the official MySQL chart is included as a dependency in both the APIM and IS charts with all the credential and database creation process automated. During the performance tests performed there seemed to not be much load placed on the registry with the workloads applied so the default registry was kept, which does not have redundancy. A new version of the WSO2 chart may also be created to use the MS SQL server database instead, by adding the proper libraries to the WSO2 images.

5.4.7.3 MongoDB

MongoDB's deployment was done following the official documentation that contains a few options on deploying MongoDB Enterprise with Kubernetes. A Helm chart is available that deploys the MongoDB Kubernetes Operator which acts as a management daemon to the cluster. Since MongoDB supports sharding there are master and replica nodes. Each master node contains a unique shard of the cluster and each master node has a set of replica nodes mirroring all the information for availability purposes. The usual cluster topology is shown in Figure 28.

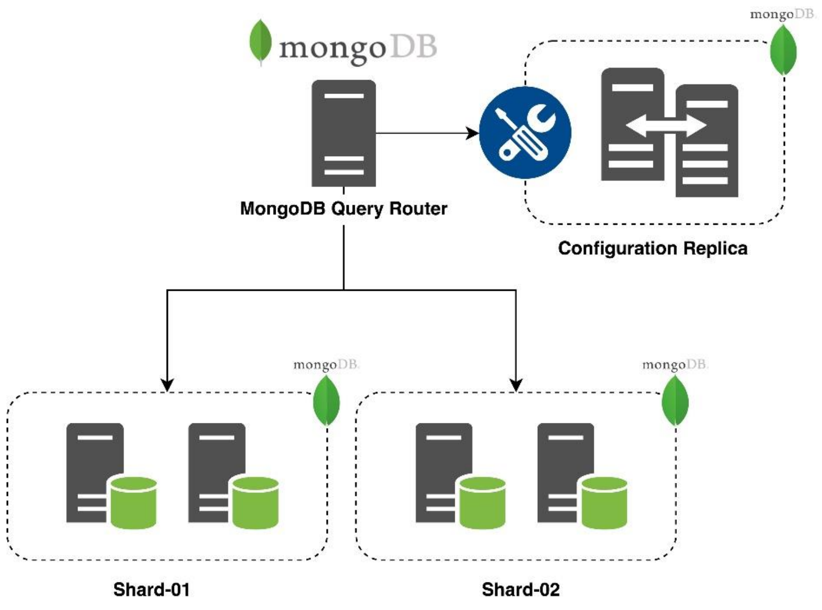


Figure 28 MongoDB Cluster Topology (source: mongodb.com)

Completing the cluster are the Query Router modules which are responsible for routing the query requests to the appropriate node and a Configuration cluster that stores all the MongoDB related configurations.

5.4.7.4 Neo4j

In Neo4j two kinds of components are deployed: core nodes and replica nodes. The Neo4j HA architecture represented in Figure 29 presents a cluster of core nodes which can handle any type of query and uses a majority voting system for master election and transaction commitment decisions.

The replica nodes replicate the information from the core cluster asynchronously via transaction log shipping and they can be used for read-only queries. In case of a complete failure of the core cluster these nodes can be used to temporarily provide read-only access to the databases.

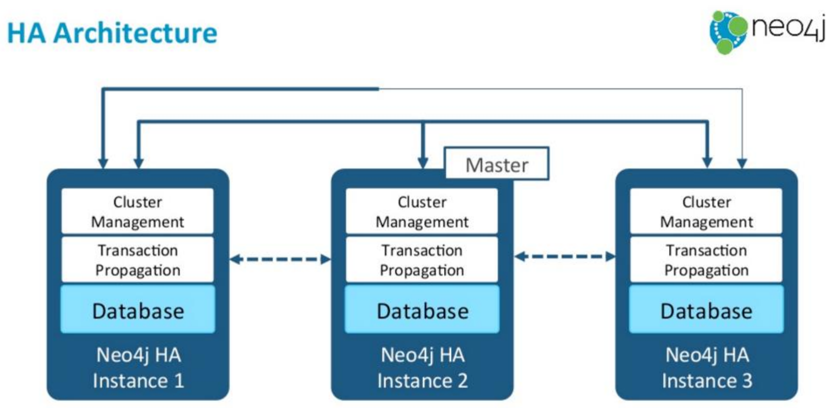


Figure 29 Neo4j HA Architecture (source: neo4j.com)

The official Helm chart allows for the installation of the cluster with configurable core nodes and auto scaling replica nodes and no further customizations were required.

5.5 *Monitorization*

Many monitorization systems were available to monitor the software deployed over the development of the middleware. The physical machines are monitored by the OpenNebula software which provides metrics dashboards for CPU and Memory allocation and utilization in real time, as exemplified in Figure 30 with the metrics of one of the hosts.

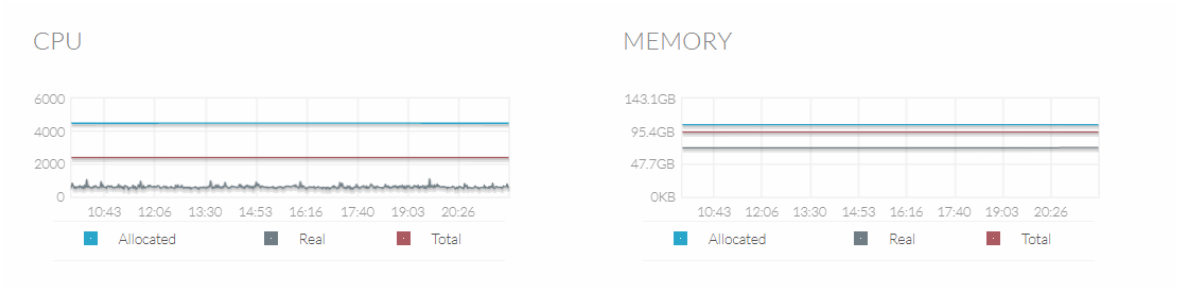


Figure 30 OpenNebula – Physical Host Metrics

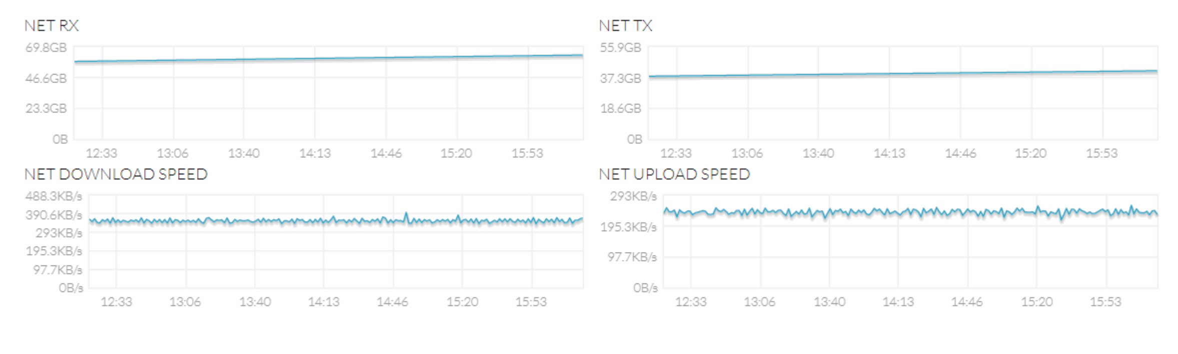
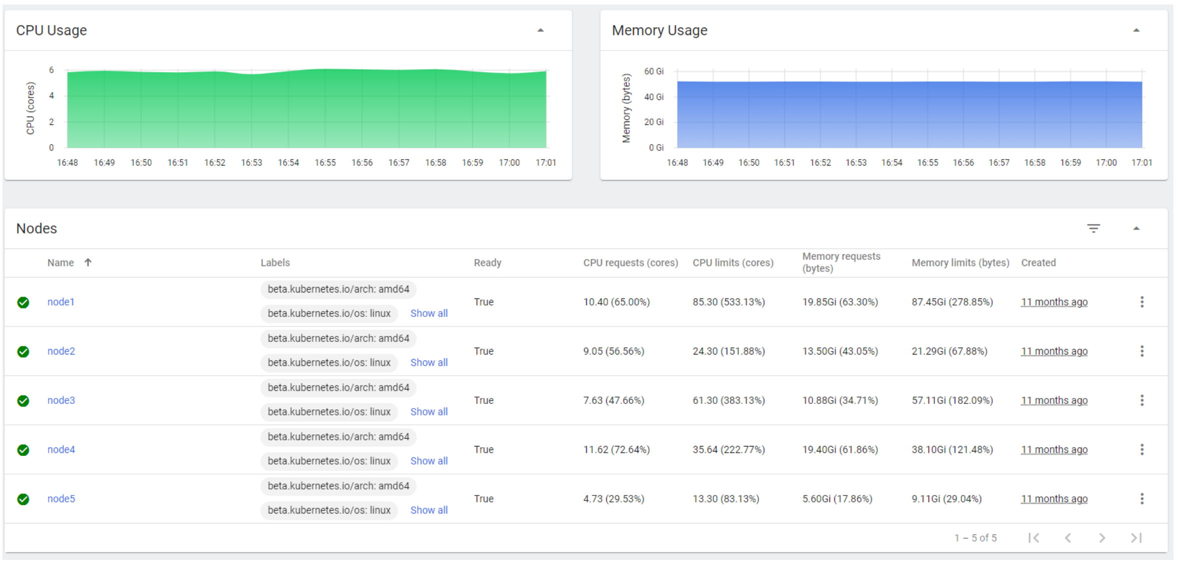


Figure 31 OpenNebula – VM Network Metrics

The virtual machines are also monitored with this dashboard with CPU and memory usage as well as the virtual network interfaces bandwidth in use like exemplified by Figure 31.

Many iterations of monitorization systems were designed for the project software infrastructure but there was never a decision on a final implementation. For the docker swarm approach a cAdvisor[108], Prometheus[109] and Grafana[110] approach was planned and tested. The cAdvisor agents would expose metrics regarding all the containers and host machines through a REST API. Prometheus would be then setup to pull the information and store it as it can act as a timeseries database and Grafana would be used to create web dashboards to display the information and configure alerts with many communication options like email or calls to a REST API.

With the switch to Kubernetes for the container orchestration the monitorization system choice was put on hold as there was the option to use Kubernetes' web-based dashboard[111] which allows the user to view the status of every node in the cluster as well as view, edit and monitor the metrics of all resource types in a Kubernetes environment.



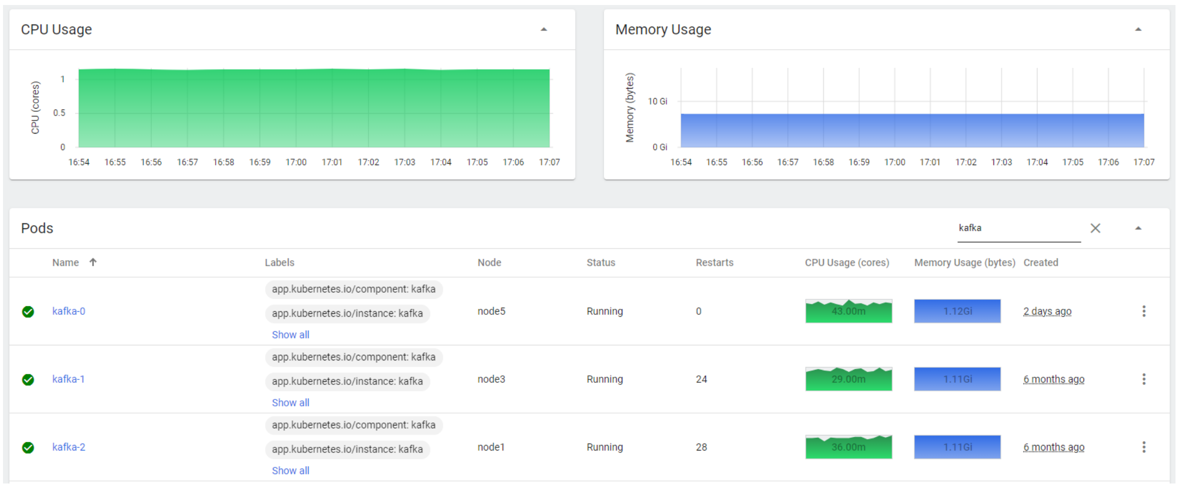
Nodes

	Name	Labels	Ready	CPU requests (cores)	CPU limits (cores)	Memory requests (bytes)	Memory limits (bytes)	Created
✓	node1	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux Show all	True	10.40 (65.00%)	85.30 (533.13%)	19.85Gi (63.30%)	87.45Gi (278.85%)	11 months ago
✓	node2	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux Show all	True	9.05 (56.56%)	24.30 (151.88%)	13.50Gi (43.05%)	21.29Gi (67.88%)	11 months ago
✓	node3	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux Show all	True	7.63 (47.66%)	61.30 (383.13%)	10.88Gi (34.71%)	57.11Gi (182.09%)	11 months ago
✓	node4	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux Show all	True	11.62 (72.64%)	35.64 (222.77%)	19.40Gi (61.86%)	38.10Gi (121.48%)	11 months ago
✓	node5	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux Show all	True	4.73 (29.53%)	13.30 (83.13%)	5.60Gi (17.86%)	9.11Gi (29.04%)	11 months ago

1 – 5 of 5

Figure 32 Kubernetes UI – Nodes

In Figure 32 the view of the cluster's nodes is present with the status of each node and metrics regarding the current resource occupation in the table with graphs displaying the metrics of the entire cluster above. Figure 33 shows a more product specific example, in this case Kafka where each pod's state is displayed with the corresponding metrics. The top graphs display total metrics for the namespaces those pods belong to. In this case all the Kafka cluster components were deployed in that namespace, so it displays the cumulative metrics of all the pods running that comprise the cluster.



	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
✓	kafka-0	app.kubernetes.io/component: kafka app.kubernetes.io/instance: kafka Show all	node5	Running	0	43.00m	1.12Gi	2 days ago
✓	kafka-1	app.kubernetes.io/component: kafka app.kubernetes.io/instance: kafka Show all	node3	Running	24	29.00m	1.11Gi	6 months ago
✓	kafka-2	app.kubernetes.io/component: kafka app.kubernetes.io/instance: kafka Show all	node1	Running	28	36.00m	1.11Gi	6 months ago

Figure 33 Kubernetes UI – Kafka Pods

Although the Kubernetes UI is easy to setup and provides plenty of information about the current state of the cluster it still lacks in features such as persistence of the information as it only shows real time metrics and alerts. With the goal of improving overall monitorization of the cluster the planned monitorization architecture for docker swarm was re-evaluated and adapted to fit the new Kubernetes setup. Prometheus was maintained and deployed using a chart available in a community repository[112]. This chart also includes instructions for the deployment of Node Exporter[113] which acts as a daemon in every node to expose the machine's metrics that Prometheus pulls. Grafana was deployed using the official charts[114] and some sample dashboards from the public dashboard library were imported like the example in Figure 34 with host CPU, memory, storage and networking metrics.

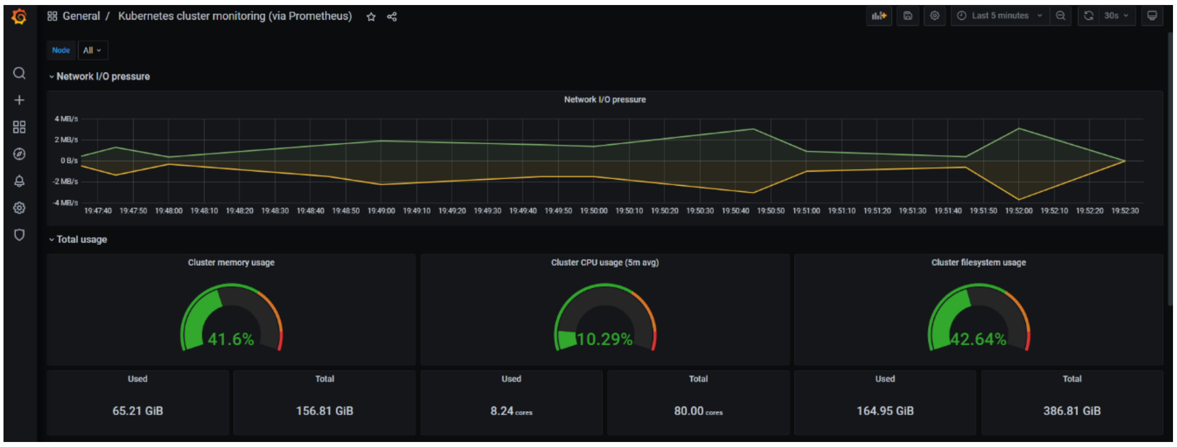
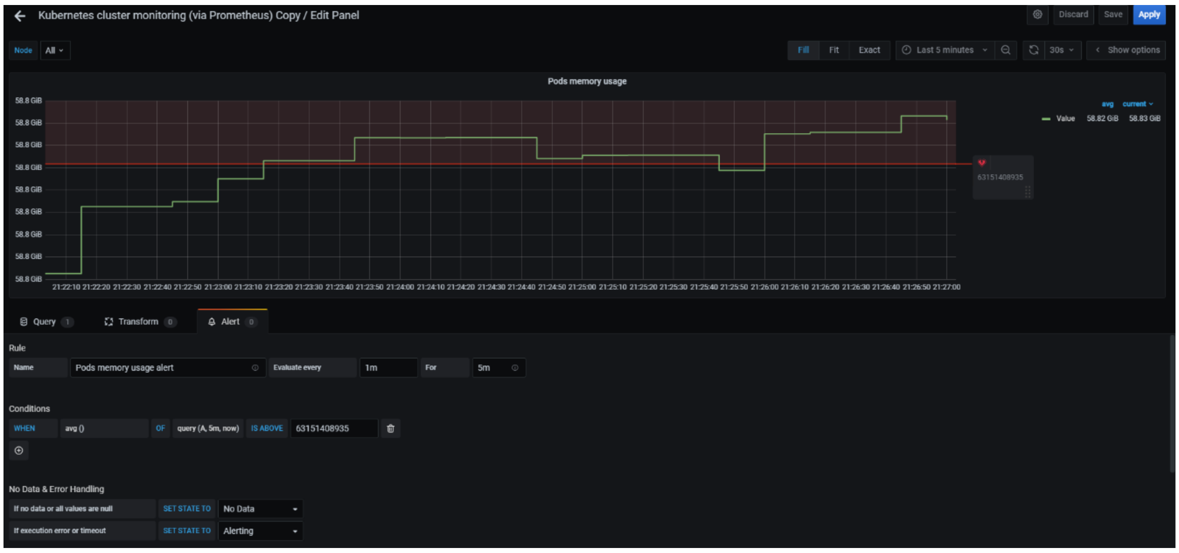


Figure 34 Grafana – Kubernetes Cluster Dashboard

As stated before, Grafana can also be used to set thresholds and configure alerts. Figure 35 shows an example, configuring an alert threshold for when the used memory reaches a certain point. The alerts are generated and can be sent via multiple channels with some examples present in Figure 36.



Rule

Name: Pods memory usage alert | Evaluate every: 1m | For: 5m

Conditions

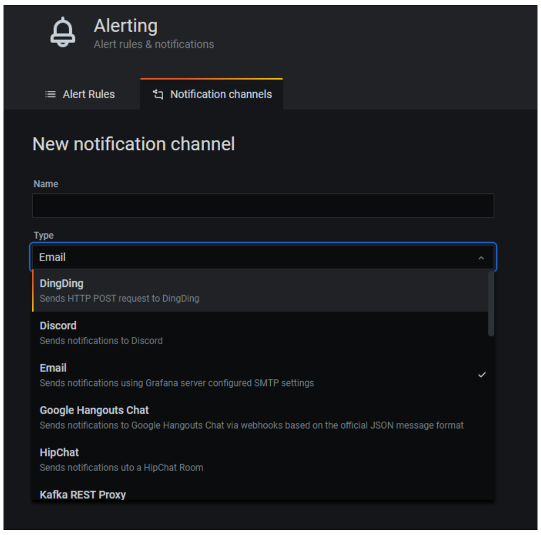
WHEN avg () OF query (A, 5m, now) IS ABOVE 63151408935

No Data & Error Handling

If no data or all values are null SET STATE TO No Data

If execution error or timeout SET STATE TO Alerting

Figure 35 Grafana – Alert Configuration



New notification channel

Name

Type: Email

- DingDing — Sends HTTP POST request to DingDing
- Discord — Sends notifications to Discord
- Email — Sends notifications using Grafana server configured SMTP settings ✓
- Google Hangouts Chat — Sends notifications to Google Hangouts Chat via webhooks based on the official JSON message format
- HipChat — Sends notifications uto a HipChat Room
- Kafka REST Proxy

Figure 36 Grafana – Alerting Channels

More monitorization features and dashboards can be added to this setup by adding more information collection systems and integrating them to Prometheus like software specific monitorization for components like the database engines and Kafka which might also have some pre-made dashboards available.

Since the ELK stack is also installed, other types of information useful to a system's monitorization can be stored and viewed there. This tool is best suited to aggregate information like logs but could also be used for metrics, although not as good as Grafana for the metrics dashboards. Unlike Prometheus which functions in a pull metrics monitoring architecture, Elasticsearch is a push-based system and agents must be configured to push the desired information to this analytics engine. While the log aggregation could be a nice addition, the pod's logs are already available centrally to use with the kubectl tool and via the Kubernetes dashboard so, while planned, ELK was not used for this purpose yet.

5.6 External Network Access

After the development of the prototype APIs and wiki installation these were deployed in the staging environment described previously. To enable the other developers' access to these APIs some networking configurations and administrative requests had to be placed as the private cloud used is only accessible from the university's network with a NGINX instance handling the routing of traffic from outside the network. DNS names and corresponding certificate-key pair were requested from the IT department to be used with the NGINX proxies for all the components that were to be exposed with HTTPS (with auto redirects when attempting to use HTTP). These components were Wiki, the Backoffice Dashboard, the API Gateway and the APIM management related web applications. These proxy configurations were then sent to the cloud administrator to be added to the running environment.

5.7 Load Tests

Load tests were applied to the Kubernetes middleware version to validate the integration of all the components and to evaluate performance gains when scaling the components horizontally and vertically.

For these load tests JMeter[115] was employed using 4 CentOS VMs as worker nodes with 8 CPUs and 8 GB of RAM. Due to networking constraints on JMeter's clusters all the nodes must be in the same subnetwork, so a Windows VM was used as a master to enable the usage of the JMeter UI to fine tune the tests and data collected.

A key element to obtaining accurate results was to sync the clocks of all the VMs before starting the tests since JMeter uses the timestamps of the requests to calculate the throughput. This was done using CentOS' ntpdate tool which updated the clock using the university's local NTP server.

To configure the cluster all the nodes must have the JMeter binaries as well has the proper configuration with all the nodes' IPs. To get around the security configuration step, certificate validation was disabled.

For the best results, the test batch should be executed from the command line and with the least amount of data collection possible to ensure the best possible performance from the workers, allowing for a higher load to be applied to the tested components.

The tests were applied to the Tracking API which allows for the retrieval of traceability information about a product lot. The database was loaded with sample data with some short event chains and a JMeter test collection was created with a request to retrieve the history of a product lot which included a valid authorization token in the header. Multiple runs of the tests were executed with different amounts of concurrent users and tracking application containers with the goal of evaluating average response times and throughput under load. These tests validate the integration and performance of the API management, application and persistence layers. Each configuration was stress tested for at least five minutes and the results are presented in Table 2 and Table 3.

Table 2 Load test results - Throughput and bandwidth

Test Configuration	Throughput	Network (KB/sec)

Threads	Nodes	Transactions/s	Received	Sent
2000	1	329	478.87	72.46
2000	3	909	1321.16	199.91
2000	5	1156	1679.09	254.07
2000	8	1308	1899.82	287.46
400	1	342	497.84	75.33
400	3	921	1337.39	202.37
400	5	1130	1641.05	248.32
400	8	1248	1813.03	274.34
200	1	342	497.84	75.33
200	3	517	751.76	113.75
200	5	533	775.16	117.29
200	8	574	833.77	126.16

Table 3 Load test results – Response time

Test Configuration		Response Times (ms, lower = better)						
Threads	Nodes	Average	Min	Max	Median	90th pct	95th pct	99th pct
2000	1	5927	29	17068	5784.00	7092.00	7245.00	8876.97
2000	3	2183	26	11139	1225.00	1803.00	1922.00	2501.00
2000	5	1712	26	12423	594.00	1565.80	2024.00	3149.99
2000	8	1469	26	7019	1026.00	2114.90	2824.95	3928.99
400	1	568	26	3006	552.00	716.00	799.00	1102.00
400	3	425	27	2910	399.00	565.00	614.00	761.00
400	5	344	26	2454	321.00	526.00	585.95	794.98
400	8	302	26	3456	285.00	445.00	500.00	611.00

200	1	568	26	3006	552.00	716.00	799.00	1102.00
200	3	371	26	1860	492.00	640.00	693.00	876.00
200	5	363	25	2130	466.00	667.00	721.00	838.99
200	8	335	25	2366	420.00	582.00	642.00	762.99

Regarding throughput, the results depicted in Figure 37 were slightly lower than expected with the resources allocated. Some previous tests executed without data and a draft of the application yielded a much higher transaction throughput. This indicates that even with multiple database nodes the information retrieval portion of the processing of the request will most likely be the bottleneck in this scenario. During the tests that yielded 1300 requests/second of throughput the average throughput of transactions/s in the database worker nodes was over 6000 per second, in total 18000 which is an average of over 13 database transactions per request.

The differences observed with varying amounts of application nodes were much closer to what was expected with a nearly linear increase with the increase of resources. This increase is not perfect though and this might be justified due to the other components being the limiting factor or due to the hardware configuration used, as this private cloud contains physical machines with varying specs so some of the containers might have been deployed in slightly slower nodes than others.

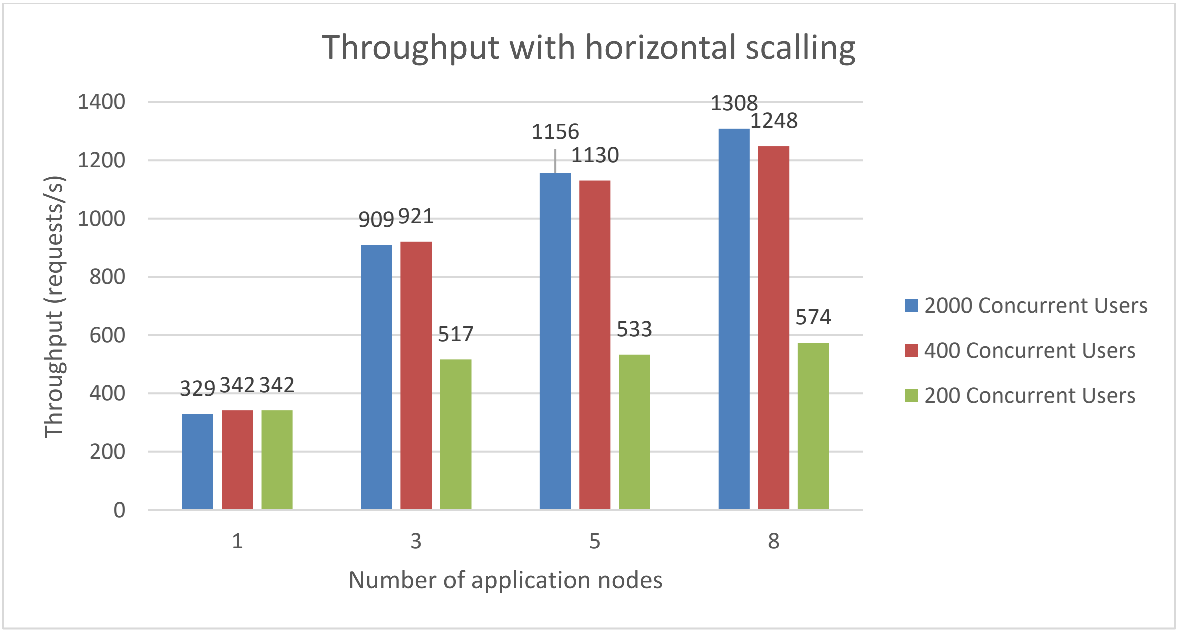


Figure 37 Tracking API Load test results – throughput

Figure 38 reports the results regarding response times which were similar for both 200 and 400 concurrent users but with a much higher load of 2000, response times the increase was much higher which might be noticeable to end users meaning that the system would require more resources for handle that workload or that one of the other components is the bottleneck.

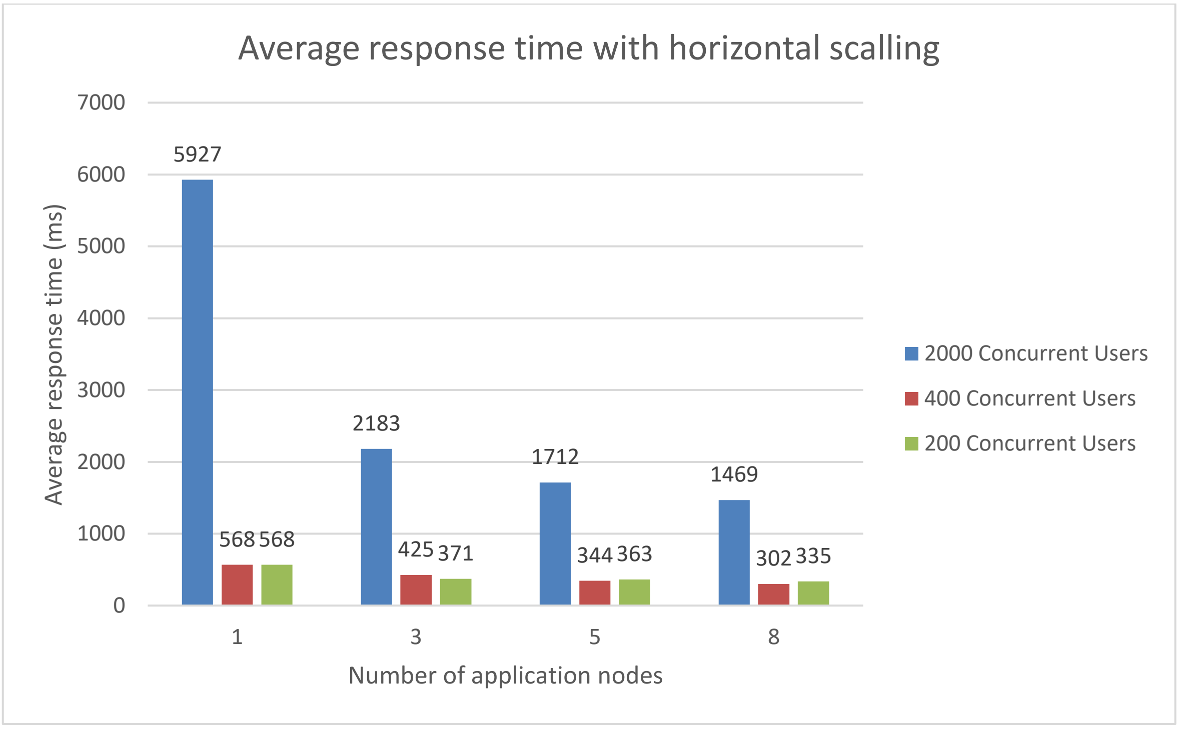


Figure 38 Tracking API Load test results – average response times

6 Development Process and CI/CD

As I was working with a team of other developers on the middleware involving daily collaboration and due to the architecture choice of microservices it was beneficial to adopt agile methodologies for the development of the middleware and DevOps toolchains to automate component integration processes.

6.1 Scrum Methodology

The agile framework we chose was Scrum[116] as it is lightweight and adequate for software development. Not all the principles were followed religiously but we tried our best to adhere to the key principles of small increments.

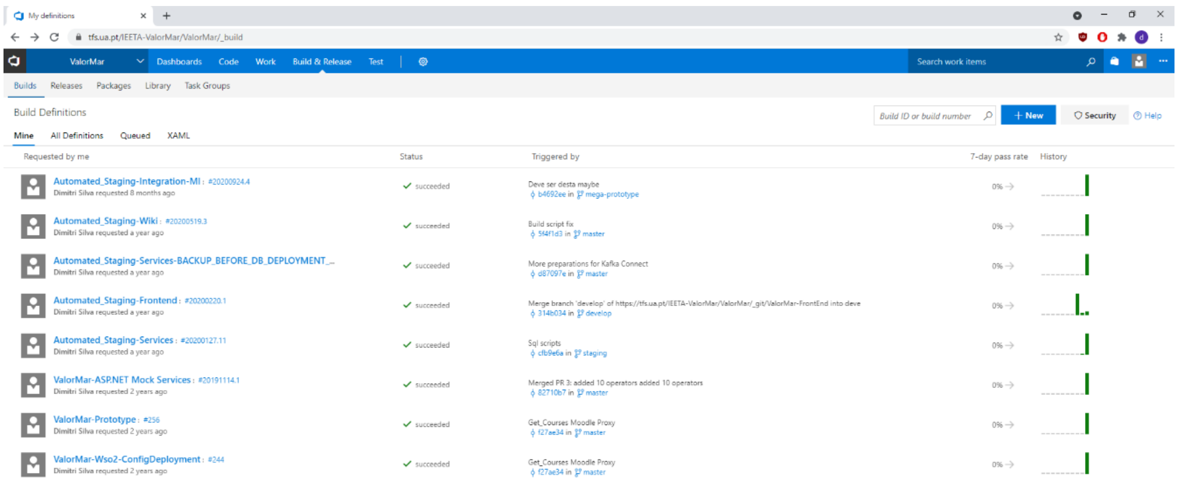
In our team we were all considered Developers with our advising professor being the Product Owner. There was no explicit Scrum Master which is supposed to be the person in charge of the implementation of the scrum methodology as each of us would take that role in turns.

For the sprint planning in the later phases of development we were doing two-month sprints for development with other tasks in parallel like writing reports and preparing presentations. At the start of the middleware's development this was a much harder as we were not as good at estimating the time it would take for a certain feature to be implemented and there was a long learning process of all the technologies before a simple but complete prototype with all the components integrated was ready for us to add small increments to and in those occasions the Scrum methodology was often ignored for some time.

6.2 Team Foundation Server

The CI/CD tool used for this project was Microsoft's Team Foundation Server 2017. This choice was made due to already being installed and maintained by the university's IT department. This tool allows for the creation and usage of git repositories where the source code of all the components and even some configurations were stored.

This platform allows for the creation of continuous integration pipelines called Build Definitions which can pull from local TFS repositories or externals ones if desired and they allow for triggers on branch changes or depending on schedules to automate the CI process. Some examples developed and using during the project are present in Figure 39.

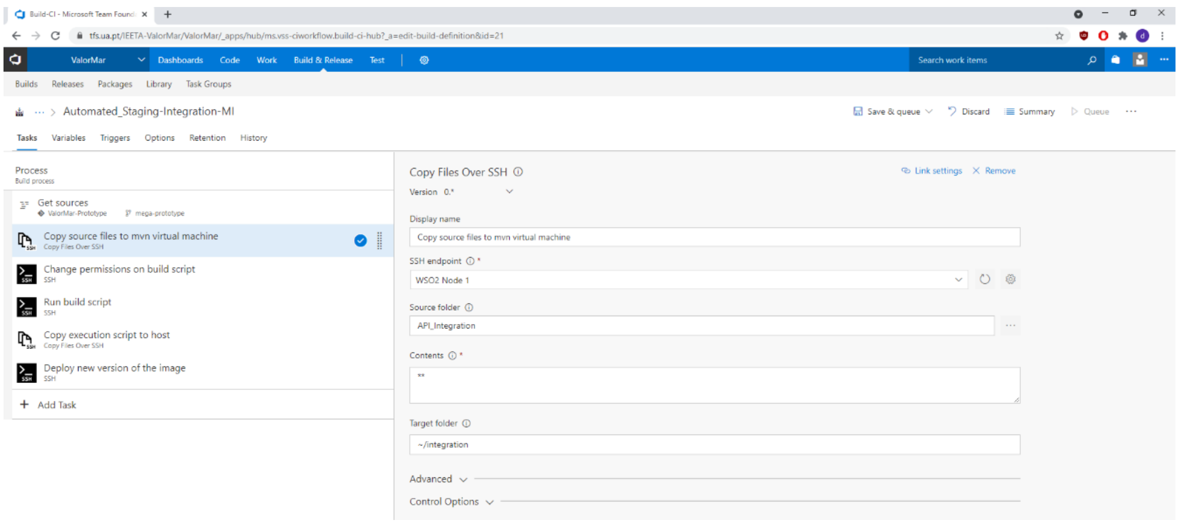


Build Definitions

Requested by me	Status	Triggered by	7-day pass rate
Automated_Staging-Integration-MI : #20200924.4 Dimitri Silva requested 8 months ago	✓ succeeded	Deve ser desta maybe b4692ee in mega-prototype	0%
Automated_Staging-Wiki : #20200519.3 Dimitri Silva requested a year ago	✓ succeeded	Build script fix 5f4f1d3 in master	0%
Automated_Staging-Services-BACKUP_BEFORE_DB_DEPLOYMENT_... Dimitri Silva requested a year ago	✓ succeeded	More preparations for Kafka Connect d87097e in master	0%
Automated_Staging-Frontend : #20200220.1 Dimitri Silva requested a year ago	✓ succeeded	Merge branch 'develop' of https://tfs.ua.pt/IEETA-ValorMar/ValorMar/_git/ValorMar-FrontEnd into deve 314b034 in develop	0%
Automated_Staging-Services : #20200127.11 Dimitri Silva requested a year ago	✓ succeeded	Sql scripts cfb9e6a in staging	0%
ValorMar-ASP.NET Mock Services : #20191114.1 Dimitri Silva requested 2 years ago	✓ succeeded	Merged PR 3: added 10 operators added 10 operators 82710b7 in master	0%
ValorMar-Prototype : #256 Dimitri Silva requested 2 years ago	✓ succeeded	Get_Courses Moodle Proxy f27ae34 in master	0%
ValorMar-Wso2-ConfigDeployment : #244 Dimitri Silva requested 2 years ago	✓ succeeded	Get_Courses Moodle Proxy f27ae34 in master	0%

Figure 39 TFS CI Pipelines

TFS also allows for the definition of release pipelines triggered by the publishing of new artifacts which were not used in this project due to the artifacts being packaged straight away into docker containers. This was done because the automated testing that was being done required docker images for deployment as many components were being tested at once and, for that reason we opted to use only CI pipelines as very little would be left to do after the test results as the code was already packaged in the images in another repository and this option reduced complexity.



Automated_Staging-Integration-MI

Process
Build process

- Get sources — ValorMar-Prototype, mega-prototype
- Copy source files to mvn virtual machine — Copy Files Over SSH
- Change permissions on build script — SSH
- Run build script — SSH
- Copy execution script to host — Copy Files Over SSH
- Deploy new version of the image — SSH

Copy Files Over SSH
Version 0.*
Display name: Copy source files to mvn virtual machine
SSH endpoint *: WSO2 Node 1
Source folder: API_Integration
Contents *: **
Target folder: ~/integration

Figure 40 TFS Micro Integrator pipeline

The pipelines are executed in TFS Agents which run on Windows and are managed by the SysAdmin responsible platform. TFS comes pre-configured with plugins for many of their own tools but many of the technologies like Docker and Maven used in this project were not supported and the agents had to be changed to accommodate for that. The responsible SysAdmin was not available all the time to make this type of changes, so some alternatives had to be used like the pipeline in Figure 40 where the code is transferred to a Linux virtual machine tailored to our project needs and uses scripts from the repository itself to create the remaining of the pipeline leaving as little tasks as possible for the TFS Agents.

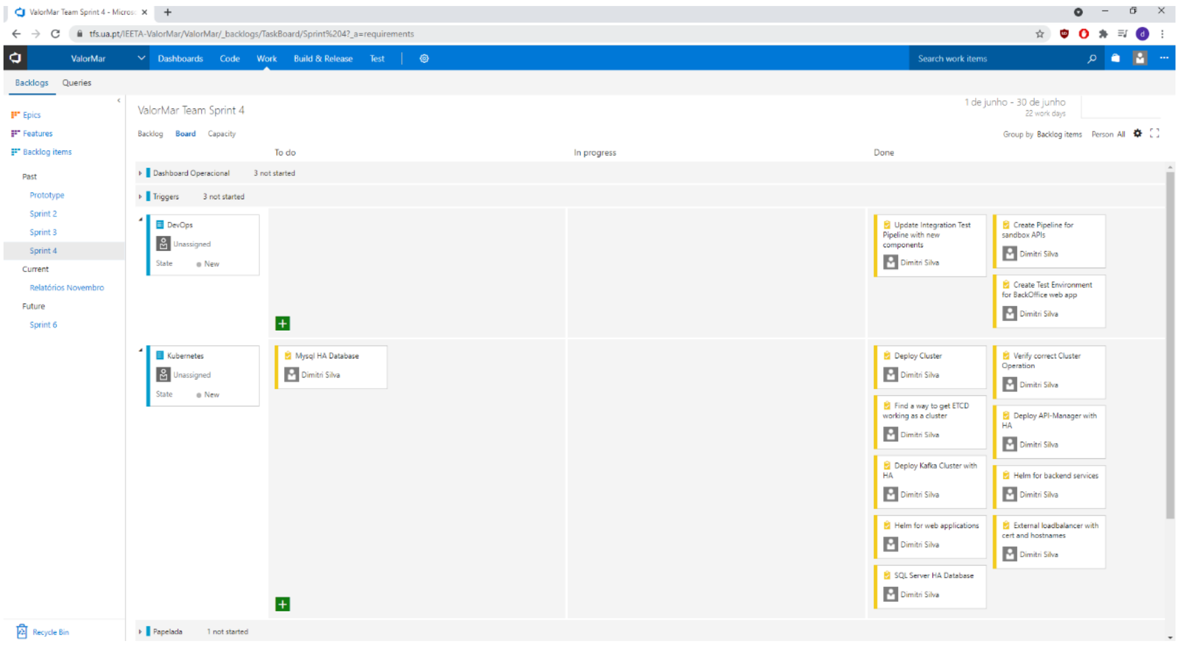


Figure 41 TFS Scrum sprint task board

Figure 41 is an example of a sprint task board that was occasionally used in TFS to implement the Scrum methodology. Every major task or application had a backlog item associated where tasks would get created and completed.

6.3 Tests

The test developer developed Integration Tests and Acceptance Tests. In some of these cases some automation using pipelines was required to assemble the testing environment with the latest developed middleware components.

Most of the applications that exposed REST APIs had Integration Tests to validate all the methods and could be executed with ease as the only dependencies required were database engines which were emulated automatically by .NET's testing frameworks.

In the Tracking domain the event integration component was more difficult test due to the asynchronous nature of the event insertion and that component being dependent on Kafka. .Net core's Kafka support at the time was very limited. Other frameworks like Spring have much better testing support with Kafka brokers embedded into the testing environment to allow for the injection of test messages into consuming components and the evaluation of messages inserted into topics by producers. In our case a whole integration testing environment had to be created for the tracking domain to test the components.

Lastly, Acceptance Tests were developed using Selenium[117] for the Backoffice web application that were executed using the cross browser testing tool LambdaTest[118]. This tool is only available as a service, so the testing environment had the requirement of being accessible from a public network which meant having DNS registrations and certificates created for a reverse proxy for this version of the application.

6.4 CI Pipelines

The pipelines we created had two main purposes: automating the deployment of the staging environment and automating the execution of the tests described in the previous subsections. For this purpose, four main pipelines were development:

1. Tracking Integration Testing Pipeline:

This pipeline was responsible for compiling and packaging into images the Event Integration, Event Notifier and Event Retrieval components. These images were then deployed to the testing environment which also had a database, Kafka cluster and micro integrator instance to complete the core of that domain required to run the integration tests. The information necessary for the operation of these components from the governance domain were injected into the testing database before every test from a script in the repository where the tester could create any information needed for the test with the database behind cleared before a new test run.

2. Acceptance Testing Pipeline:

In these acceptance tests the important component to test is the Backoffice. The web application is built and packaged into a docker image and deployed to the testing environment. Before the tests are executed the databases for this environment are cleared and re-populated using scripts from the repository that the test developer can customize. Since the LambdaTest tool required either a tunnel, which was not practical with the TFS pipelines tool, or the website to the accessible publicly. To have a fully functioning application a full clone of almost the entire middleware had to be created as the application made calls to many of the APIs. To avoid some configuration hassle, the sandbox gateway environment of the API Manager was used instead of having to create a whole new setup for testing purposes. In this sandbox environment the latest versions of all the APIs are currently deployed and use the databases mentioned previously. To access this sandbox environment instead of the production one, the web application’s credentials are switched with ones proper for this environment as the API Manager handles the routing to the different environments based on the application's credentials. Lastly, the tester was provided with testing accounts for each of the user profiles that allowed the tests to login with the Identity Provider to have the proper claims and permissions for all the application's usage scenarios.

3. Staging Services Deployment Pipeline

All the .Net Core applications are stored in the same solution as of now and are deployed at once. The pipeline compiles all the code and creates the corresponding docker images for each component and deploys them to both the production and sandbox environments. In the sandbox environments the databases are cleared, and the schema is always freshly deployed while for the production environment the intention was to evolve the schema using migration scripts. This migration feature was never tested as migration scripts were never created. Finalizing the pipeline is the update of the swagger definitions of each API in the API Manager which is done using a REST API with the new swagger file auto-generated by the .Net Core application using annotations.

The tests are executed in a separate pipeline from this one because the learning and development of the tests was at best going at the same rate of the middleware development and in many situations the number of false positives regarding errors identified by the tests preventing a new version from being deployed outweighed the gain obtained from automated tests.

4. Information Domain Pipeline

This is probably one of the closest examples that were developed for pipelines in a microservice-oriented scenario as it only deals with components in the information data domain. Here the latest images for the Mediawiki application and the micro-integrator instance get built, pushed to the image registry and deployed to the production environment.

5. Monitoring Domain Pipeline

The monitoring domain also follows a more microservice-oriented approach, once again with docker images being built and pushed to the registry to then be deployed to the necessary environments.

6. Backoffice Application Pipeline

This pipeline builds and deploys to the production environment the Backoffice web application. The reason for a pipeline separate from the acceptance tests is that it predates those tests and we already had some automation needs and more recently it started being used again due to the licenses to the LambdaTest being limited to a set number of executions which sometimes would run out.

In all these pipelines the compilation of the source codes was always done inside the docker image creation phase as this decreased the need to manage dependencies in project specific virtual machines or request changes and additions to the TFS build agents. In the case of the micro-integrator images for the sake of efficiency the packaging of the solutions was done before the creation of the images as the only dependency was Maven and due to lack of ability of using the image layer caching to our advantage the process would take too long as most changes would require hundreds of packages to be downloaded from the repositories.

7 Conclusion

After so many months associated with this project the feeling regarding conclusions it that a whole new document could be written just for this purpose, but I will try to be concise.

I participated in the development of the middleware from the end of the design phase to the delivery phase. A lot of the work I did was related to adapting and integrating software solutions in the project, using tools like Mediawiki and WSO2 to implement the desired features. I also was responsible for integrating and deploying all the internally developed middleware components into a staging environment which I had to maintain for the external developers to use (Kiosk and Client applications). In the final phases of the project, I prepared and tested Helm charts to deploy all the required components in a Kubernetes for a production environment and validated some of its performance with load tests. During my participation in this project, I also co-authored two scientific publications and presented one of them at a conference.

Regarding the overall development of the Middleware I think the results are acceptable with the main caveats being that a lot of the components could use some further refinement and even some planned features did not get implemented. More integrated client applications would have helped with that refinement as it would have placed more pressure on the developers to refine the current features and improve motivation by seeing the system working but instead, we focused a lot on adding new things over improving and completing what was already being worked on.

For the operational management side of things, I am very happy with the results of orchestrating such a big system using Kubernetes (over 100 containers deployed) and having the chance of using some pretty powerful hardware with some regrets of being sceptic of adopting such technology earlier on in the project due to its perceived complexity but it could have saved a lot of time with a lot of tasks.

One of the big lessons learned in this project was working around constrains with all the software choices imposed but perhaps the biggest lesson was the importance of technical communication between developers and even with the project manager as many misunderstandings caused a lot of inefficiencies during development. In a lot of situations, especially in the earlier phases, the lack of understanding of what was asked from the team caused us to go down the wrong path, wasting a lot of time. Other times not being able to effectively communicate to others the state of my work or what was needed from them also led to a lot of frustration and slower progress.

Overall, I think this was a very enriching experience and the perfect way to finish my masters, as I was unsure on what field to work on and this project allowed me to do a bit of everything.

References

[1] “Sea Rondus.” [Online]. Available: http://www.seaaroundus.org/data/#/eez. [Accessed: 18-May-2021].

[2] “Valormar.” [Online]. Available: https://valormar.pt/en/home-2/. [Accessed: 24-May-2021].

[3] “Docapesca.” [Online]. Available: http://www.docapesca.pt/. [Accessed: 30-May-2021].

[4] “Sonae.” [Online]. Available: https://www.sonae.pt/en/. [Accessed: 30-May-2021].

[5] “IPVC - Intituto Politécnico de Viana do Castelo.” [Online]. Available: https://www.ipvc.pt/en/. [Accessed: 30-May-2021].

[6] “FlowTech.” [Online]. Available: https://flowtech.pt/en/home-3/. [Accessed: 30-May-2021].

[7] S. Charlebois, B. Sterling, S. Haratifar, and S. K. Naing, “Comparison of Global Food Traceability Regulations and Requirements,” *Compr. Rev. Food Sci. Food Saf.*, vol. 13, no. 5, 2014.

[8] M. M. Aung and Y. S. Chang, “Traceability in a food supply chain: Safety and quality perspectives,” *Food Control*. 2014.

[9] J. Barnett *et al.*, “Consumers’ confidence, reflections and response strategies following the horsemeat incident,” *Food Control*, 2016.

[10] T. King *et al.*, “Food safety for food security: Relationship between global megatrends and developments in food safety,” *Trends in Food Science and Technology*. 2017.

[11] Y. Duroc and S. Tedjini, “RFID: A key technology for Humanity,” *Comptes Rendus Physique*. 2018.

[12] D. Pigini and M. Conti, “NFC-based traceability in the food chain,” *Sustain.*, 2017.

[13] M. J. Hardt, K. Flett, and C. J. Howell, “Current Barriers to Large-scale Interoperability of Traceability Technology in the Seafood Sector,” *J. Food Sci.*, 2017.

[14] H. A. Rothan and S. N. Byrareddy, “The epidemiology and pathogenesis of coronavirus disease (COVID-19) outbreak,” *Journal of Autoimmunity*. 2020.

[15] S. A. Sarkodie and P. A. Owusu, “Global assessment of environment, health and economic impact of the novel coronavirus (COVID-19),” *Environ. Dev. Sustain.*, 2020.

[16] S. J. Daniel, “Education and the COVID-19 pandemic,” *Prospects,* 2020.

[17] S. MIN, C. XIANG, and X. heng ZHANG, “Impacts of the COVID-19 pandemic on consumers’ food safety knowledge and behavior in China,” *J. Integr. Agric.*, 2020.

[18] A. Kassahun, R. J. M. Hartog, and B. Tekinerdogan, “Realizing chain-wide transparency in meat supply chains based on global standards and a reference architecture,” *Comput. Electron. Agric.*, 2016.

[19] “GS1 Standards.” [Online]. Available: https://www.gs1.org/standards/epcis. [Accessed: 22-May-2021].

[20] M. P. Papazoglou and W. J. Van Den Heuvel, “Service oriented architectures: Approaches, technologies and research issues,” *VLDB J.*, 2007.

[21] SAP, “SAP Software & Solutions | Technology & Business Applications,” *SAP.com,* 2019. [Online]. Available: https://www.sap.com/index.html.

[22] Y. C. Hsu, A. P. Chen, and C. H. Wang, “A RFID-enabled traceability system for the supply chain of live fish,” in *Proceedings of the IEEE International Conference on Automation and Logistics, ICAL 2008*, 2008.

[23] L. M. Abenavoli, F. Cuzzupoli, V. Chiaravalloti, and A. R. Proto, “Traceability system of olive oil: A case study based on the performance of a new software cloud,” *Agron. Res.*, 2016.

[24] K. Korpela, J. Hallikas, and T. Dahlberg, “Digital Supply Chain Transformation toward Blockchain Integration,” in *Proceedings of the 50th Hawaii International Conference on System Sciences (2017)*, 2017.

[25] A. Sill, “The Design and Architecture of Microservices,” *IEEE Cloud Comput.*, 2016.

[26] K. Indrasiri and P. Siriwardena, “The Case for Microservices,” in *Microservices for the Enterprise*, 2018.

[27] “WSO2.” [Online]. Available: https://wso2.com/. [Accessed: 24-May-2021].

[28] “IBM API Connect.” [Online]. Available: https://www.ibm.com/uk-en/cloud/api-connect. [Accessed: 24-May-2021].

[29] "Azure's API Management." [Online]. Available: https://azure.microsoft.com/en-us/services/api-management/. [Accessed: 24-May-2021].

[30] "Apigee." [Online]. Available: https://docs.apigee.com/. [Accessed: 21-Jun-2021].

[31] "3scale." [Online]. Available: https://www.3scale.net/. [Accessed: 21-Jun-2021].

[32] "Kong." [Online]. Available: https://konghq.com/. [Accessed: 21-Jun-2021].

[33] "Apache Kafka." [Online]. Available: https://kafka.apache.org/. [Accessed: 24-May-2021].

[34] "ActiveMQ." [Online]. Available: https://activemq.apache.org/. [Accessed: 24-May-2021].

[35] "RabbitMQ." [Online]. Available: https://www.rabbitmq.com/. [Accessed: 24-May-2021].

[36] "AMQP." [Online]. Available: https://www.amqp.org/. [Accessed: 24-May-2021].

[37] "MQTT." [Online]. Available: https://mqtt.org/. [Accessed: 24-May-2021].

[38] "STOMP." [Online]. Available: https://stomp.github.io/. [Accessed: 24-May-2021].

[39]. "Net Framework." [Online]. Available: https://dotnet.microsoft.com/. [Accessed: 24-May-2021].

[40] "JakartaEE." [Online]. Available: https://jakarta.ee/. [Accessed: 24-May-2021].

[41] "Spring." [Online]. Available: https://spring.io/. [Accessed: 24-May-2021].

[42] "Azure Functions." [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed: 24-May-2021].

[43] "AWS Lambda." [Online]. Available: https://aws.amazon.com/lambda/.

[44] "SQL Server." [Online]. Available: https://www.microsoft.com/en-us/sql-server/sql-server-2019. [Accessed: 25-May-2021].

[45] "MySQL." [Online]. Available: https://www.mysql.com/. [Accessed: 21-Jun-2021].

[46] "Neo4j." [Online]. Available: https://neo4j.com/. [Accessed: 21-Jun-2021].

[47] "MongoDB." [Online]. Available: https://www.mongodb.com/. [Accessed: 30-May-2021].

[48] "Docker." [Online]. Available: https://www.docker.com/. [Accessed: 25-May-2021].

[49] "OpenShift rkt." [Online]. Available: https://www.openshift.com/learn/topics/rkt. [Accessed: 24-May-2021].

[50] "Apache Mesos." [Online]. Available: https://mesos.apache.org/. [Accessed: 24-May-2021].

[51] "Docker Swarm." [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed:

25-May-2021].

[52] "Kubernetes." [Online]. Available: https://kubernetes.io/. [Accessed: 25-May-2021].

[53] "Ansible." [Online]. Available: https://www.ansible.com/. [Accessed: 25-May-2021].

[54] "Chef." [Online]. Available: https://www.chef.io/. [Accessed: 25-May-2021].

[55] "Puppet." [Online]. Available: https://puppet.com/. [Accessed: 25-May-2021].

[56] "OpenNebula." [Online]. Available: https://opennebula.io/. [Accessed: 25-May-2021].

[57] "OpenStack." [Online]. Available: https://www.openstack.org/. [Accessed: 25-May-2021].

[58] "Azure DevOps." [Online]. Available: https://azure.microsoft.com/en-us/services/devops/. [Accessed: 25-May-2021].

[59] "Jenkins." [Online]. Available: https://www.jenkins.io/. [Accessed: 25-May-2021].

[60] E. F. Cruz, A. M. Rosado Da Cruz, and R. Gomes, "Analysis of a traceability and quality monitoring platform for the fishery and aquaculture value Chain," in *Iberian Conference on Information Systems and Technologies, CISTI*, 2019.

[61] M. Chinosi and A. Trombetta, "BPMN: An introduction to the standard," *Comput. Stand. Interfaces*, 2012.

[62] A. M. Rosado Da Cruz *et al.*, "On the design of a platform for traceability in the fishery and aquaculture value chain," in *Iberian Conference on Information Systems and Technologies, CISTI*, 2019.

[63] "Node.js." [Online]. Available: https://nodejs.org/en/. [Accessed: 30-May-2021].

[64] D. Da Silva, J. Costa, B. Assuncao, V. Kuprych, and C. Teixeira, "Microservice-based Middleware for Collaborative Supply Chain Tracing," in *Iberian Conference on Information Systems and Technologies, CISTI*, 2020.

[65] K. Indrasiri and P. Siriwardena, "Inter-Service Communication," in *Microservices for the Enterprise: Designing, Developing, and Deploying*, Berkeley, CA: Apress, 2018, pp. 63–88.

[66] K. Indrasiri and P. Siriwardena, "APIs, Events, and Streams," in *Microservices for the Enterprise: Designing, Developing, and Deploying*, Berkeley, CA: Apress, 2018, pp. 293–312.

[67] K. Indrasiri and P. Siriwardena, "Designing Microservices," in *Microservices for the Enterprise: Designing, Developing, and Deploying*, Berkeley, CA: Apress, 2018, pp. 19–61.

[68] J. Oliveira *et al.*, "Traceability system for quality monitoring in the fishery and aquaculture

value chain," *J. Agric. Food Res.*, vol. 5, p. 100169, 2021.

[69] "OpenAPI 3.0.0 Specification." [Online]. Available: https://spec.openapis.org/oas/v3.0.0. [Accessed: 30-May-2021].

[70] "Swagger." [Online]. Available: https://swagger.io/. [Accessed: 30-May-2021].

[71] "Debezium." [Online]. Available: https://debezium.io/. [Accessed: 30-May-2021].

[72] "Avro." [Online]. Available: https://avro.apache.org/. [Accessed: 30-May-2021].

[73] "ElasticSearch." [Online]. Available: https://www.elastic.co/elasticsearch/. [Accessed: 30-May-2021].

[74] "Mediawiki." [Online]. Available: https://www.mediawiki.org/wiki/MediaWiki. [Accessed: 29-May-2021].

[75] "Wiki.js." [Online]. Available: https://js.wiki/. [Accessed: 30-May-2021].

[76] "Scribunto." [Online]. Available: https://www.mediawiki.org/wiki/Extension:Scribunto. [Accessed: 30-May-2021].

[77] MediaWiki, "Extension:Cargo/Cargo and Semantic MediaWiki - MediaWiki." [Online]. Available: https://www.mediawiki.org/wiki/Extension:Cargo/Cargo_and_Semantic_MediaWiki#Faster_performance. [Accessed: 01-Dec-2019].

[78] M. Krötzsch, D. Vrandečić, M. Völkel, H. Haller, and R. Studer, "Semantic Wikipedia," *Web Semant.*, 2007.

[79] "Mediawiki Extension: OpenID Connect." [Online]. Available: https://www.mediawiki.org/wiki/Extension:OpenID_Connect. [Accessed: 30-May-2021].

[80] "Mediawiki Extension: Pluggable Auth." [Online]. Available: https://www.mediawiki.org/wiki/Extension:PluggableAuth. [Accessed: 30-May-2021].

[81] "Tweeki." [Online]. Available: https://www.mediawiki.org/wiki/Skin:Tweeki. [Accessed: 27-May-2021].

[82] "DACPAC." [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/data-tier-applications/data-tier-applications?view=sql-server-ver15. [Accessed: 17-Jun-2021].

[83] "SCIM2 Standard." [Online]. Available: https://tools.ietf.org/html/rfc7644. [Accessed: 30-May-2021].

[84] "Kafka REST Proxy." [Online]. Available: https://docs.confluent.io/platform/current/kafka-rest/index.html. [Accessed: 30-May-2021].

[85] "MITM Attack." [Online]. Available: https://en.wikipedia.org/wiki/Man-in-the-middle_attack. [Accessed: 30-May-2021].

[86] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "Openid connect core 1.0," *OpenID Found.*, 2014.

[87] "WSO2 IS OpenId Connect Flows." [Online]. Available: https://is.docs.wso2.com/en/5.9.0/learn/openid-connect-authentication/ . [Accessed: 30-May-2021].

[88] "SAML2." [Online]. Available: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html. [Accessed: 25-Jun-2021].

[89] "Shibboleth Identity Provider." [Online]. Available: https://wiki.shibboleth.net/confluence/display/IDP4/Home. [Accessed: 25-Jun-2021].

[90] "ReactJS." [Online]. Available: https://reactjs.org/. [Accessed: 27-May-2021].

[91] "Material UI." [Online]. Available: https://material-ui.com/. [Accessed: 27-May-2021].

[92] J. Oliveira, A. M. R. Da Cruz, and P. M. Faria, "Zeroconf Network Retail Kiosk for Fish Products Traceability," in *Iberian Conference on Information Systems and Technologies, CISTI*, 2020.

[93] J. Oliveira, P. M. Faria, and A. M. Rosado da Cruz, "User Experience in Kiosk Application for Traceability of Fishery Products," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2020.

[94] "Docker Hub." [Online]. Available: https://hub.docker.com. [Accessed: 27-May-2021].

[95] "Portainer." [Online]. Available: https://www.portainer.io/. [Accessed: 30-May-2021].

[96] "Helm." [Online]. Available: https://helm.sh/. [Accessed: 30-May-2021].

[97] "Ansible one_vm module." [Online]. Available: https://docs.ansible.com/ansible/latest/collections/community/general/one_vm_module.html . [Accessed: 29-May-2021].

[98] "Kubespray." [Online]. Available: https://github.com/kubernetes-sigs/kubespray. [Accessed: 29-May-2021].

[99] "etcd." [Online]. Available: https://etcd.io/. [Accessed: 30-May-2021].

[100] "VM Performance Optimization for OpenNebula." [Online]. Available: https://storpool.com/blog/vm-performance-optimization-for-opennebula-with-kvm. [Accessed: 29-May-2021].

[101] "Kubernetes APIM Charts." [Online]. Available: https://github.com/wso2/kubernetes-apim. [Accessed: 29-May-2021].

[102] "Kubernetes IS Charts." [Online]. Available: https://github.com/wso2/kubernetes-is. [Accessed: 29-May-2021].

[103] "WSO2 Micro Integrator - Kubernetes." [Online]. Available: https://wso2.com/library/articles/deploying-wso2-micro-integrator-on-kubernetes/. [Accessed: 25-Jun-2021].

[104] "Bitnami." [Online]. Available: https://bitnami.com/. [Accessed: 30-May-2021].

[105] "LensesIO Fast Data Dev." [Online]. Available: https://github.com/lensesio/fast-data-dev. [Accessed: 30-May-2021].

[106] "Kibana." [Online]. Available: https://www.elastic.co/kibana. [Accessed: 30-May-2021].

[107] "Microsoft's SQL Server HA Ansible Playbook Examples." [Online]. Available: https://github.com/microsoft/sql-server-samples/tree/master/samples/features/high availability/Linux/Ansible Playbook. [Accessed: 28-May-2021].

[108] "cAdvisor." [Online]. Available: https://github.com/google/cadvisor. [Accessed: 29-May-2021].

[109] "Prometheus." [Online]. Available: https://prometheus.io/. [Accessed: 29-May-2021].

[110] "Grafana." [Online]. Available: https://grafana.com/. [Accessed: 29-May-2021].

[111] "Kubernetes Web UI." [Online]. Available: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/. [Accessed: 29-May-2021].

[112] "Prometheus Community Charts." [Online]. Available: https://github.com/prometheus-community/helm-charts/tree/main/charts. [Accessed: 29-May-2021].

[113] "Node Exporter." [Online]. Available: https://github.com/prometheus/node_exporter. [Accessed: 29-May-2021].

[114] "Grafana Charts." [Online]. Available: https://github.com/grafana/helm-charts. [Accessed: 29-May-2021].

[115] “JMeter.” [Online]. Available: https://jmeter.apache.org/. [Accessed: 30-May-2021].

[116] “Scrum.” [Online]. Available: https://www.scrum.org/resources/what-is-scrum. [Accessed: 27-May-2021].

[117] “Selenium.” [Online]. Available: https://www.selenium.dev/. [Accessed: 27-May-2021].

[118] “LambdaTest.” [Online]. Available: https://www.lambdatest.com/. [Accessed: 27-May-2021].