



The Role of a Microservice Architecture on cybersecurity and operational resilience in critical systems

FRANCISCO PINTO SEBASTIÃO

Outubro de 2023

**The Role of a Microservice Architecture on
cybersecurity and operational resilience in critical
systems**

Francisco Pinto Sebastião

**Dissertation to obtain the Master's degree in Informatics Engineering,
Specialization in Software Engineering**

Advisor: Isabel Praça

Co-advisor: Orlando Sousa

Dedication

To my family and those who believed in me and supported me allowing me to reach where I am today, to coffee, my pets, and the music that accompanied me during this dissertation.

Abstract

Critical systems are characterized by their high degree of intolerance to threats, in other words, their high level of resilience, because depending on the context in which the system is inserted, the slightest failure could imply significant damage, whether in economic terms, or loss of reputation, of information, of infrastructure, of the environment, or human life. The security of such systems is traditionally associated with legacy infrastructures and data centers that are monolithic, which translates into increasingly high evolution and protection challenges.

In the current context of rapid transformation where the variety of threats to systems has been consistently increasing, this dissertation aims to carry out a compatibility study of the microservice architecture, which is denoted by its characteristics such as resilience, scalability, modifiability and technological heterogeneity, being flexible in structural adaptations, and in rapidly evolving and highly complex settings, making it suited for agile environments. It also explores what response artificial intelligence, more specifically machine learning, can provide in a context of security and monitorability when combined with a simple banking system that adopts the microservice architecture.

Keywords: Microservices architecture; Critical systems; Cybersecurity; Operational resilience; Architectural evaluation; Machine learning.

Resumo

Os sistemas críticos são caracterizados pelo seu elevado grau de intolerância às ameaças, por outras palavras, o seu alto nível de resiliência, pois dependendo do contexto onde se insere o sistema, a mínima falha poderá implicar danos significativos, seja em termos económicos, de perda de reputação, de informação, de infraestrutura, de ambiente, ou de vida humana. A segurança informática de tais sistemas está tradicionalmente associada a *infraestruturas* e *data centers legacy*, ou seja, de natureza monolítica, o que se traduz em desafios de evolução e proteção cada vez mais elevados.

No contexto atual de rápida transformação, onde as variedades de ameaças aos sistemas têm vindo consistentemente a aumentar, esta dissertação visa realizar um estudo de compatibilidade da arquitetura de microserviços, que se denota pelas suas características tais como a resiliência, escalabilidade, modificabilidade e heterogeneidade tecnológica, sendo flexível em adaptações estruturais, e em cenários de rápida evolução e elevada complexidade, tornando-a adequada a ambientes ágeis. Explora também a resposta que a inteligência artificial, mais concretamente, *machine learning*, pode dar num contexto de segurança e monitorabilidade quando combinado com um simples sistema bancário que adota uma arquitetura de microserviços.

Palavras-chave: Arquitetura de microserviços; Sistemas críticos; Cibersegurança; Resiliência operacional; Avaliação arquitetural; Machine learning.

Acknowledgments

I want to firstly thank my family, mainly my parents, who always believed in me, and pushed me, and it is thanks to them that I can find myself in this position today.

Secondly, my supervisors, for their support and availability during the elaboration of this dissertation.

Finally, my friends, some of whom provided great support during this important phase of my personal and professional life.

Table of Contents

1	Introduction	1
1.1	Context	1
1.1.1	Cybersecurity	1
1.1.2	Artificial intelligence	1
1.1.3	Operational resilience	2
1.1.4	Critical systems	3
1.1.5	Microservices	4
1.2	Problem	7
1.3	Objectives	7
1.4	Planning	7
1.5	Document structure	8
2	State of the art	11
2.1	Relevant studies and papers	11
2.1.1	Evolution of research over the years	11
2.1.2	Transition of critical systems into the microservice architecture	13
2.1.3	Proposal of cybersecurity practices for microservices	15
2.1.4	Usage of artificial intelligence in microservices	17
2.1.5	Threat detection and/or labeling in cybersecurity	22
2.2	Comparison of the gathered research	23
2.2.1	Conclusions	24
2.3	Existing technologies and patterns	25
2.3.1	Technologies	25
2.3.2	Patterns	27
2.4	Summary	29
3	Analysis	31
3.1	Concepts and activities	31
3.1.1	Core context	31
3.1.2	User context	33
3.1.3	Transfer context	34
3.1.4	Payment context	35
3.1.5	Domain model	36
3.2	Processes and stakeholders	36
3.3	Use cases	39
3.4	Summary	39
4	Design	41
4.1	Level 1: Context	42

4.2	Level 2: Containers.....	43
4.3	Level 3: Components.....	47
4.3.1	Core Service.....	48
4.3.2	Other services.....	51
4.4	Level 4: Code	53
4.4.1	Transaction-related components of the Core Service	54
4.4.2	Transfer Service components	57
4.5	Database.....	59
4.5.1	Core Service.....	59
4.5.2	Other services.....	61
4.6	Summary.....	62
5	Technologies.....	63
5.1	Adopted technologies.....	63
5.2	System's services	64
5.2.1	.NET	64
5.2.2	C#.....	64
5.2.3	Serilog.....	64
5.2.4	Elastic Common Schema	65
5.2.5	Entity Framework Core	65
5.2.6	Steeltoe.....	65
5.2.7	Swashbuckle.....	65
5.2.8	Elastic APM	66
5.3	API Gateway.....	67
5.3.1	Service discovery	67
5.4	Database.....	70
5.4.1	Microsoft SQL Server	70
5.4.2	PostgreSQL Server	70
5.4.3	Elasticsearch	70
5.5	Logging, monitoring, tracing, and observability	70
5.5.1	Kibana.....	71
5.5.2	APM Server	72
5.5.3	Metricbeat.....	73
5.5.4	Heartbeat.....	73
5.6	Summary.....	74
6	Implementation	75
6.1	Particular situations.....	75
6.1.1	Service Discovery	75
6.1.2	Logging, tracing, and metrics collection	79
6.1.3	Machine learning job creation	89
6.2	Infrastructure setup.....	98
6.2.1	System	99
6.2.2	Elastic stack	102

6.3	Summary	108
7	Evaluation	109
7.1	Investigation hypothesis	109
7.2	Indicators and information sources	110
7.3	Goals, Questions, Metrics.....	110
7.4	Tests.....	112
7.4.1	Unit tests	114
7.4.2	Integration tests	119
7.5	Continuous Integration and Continuous Deployment/Development	122
7.6	Experimentation setup.....	128
7.7	Results	129
7.7.1	Availability.....	129
7.7.2	Performance.....	132
7.7.3	Scalability.....	132
7.7.4	Monitorability	133
7.7.5	Security	133
7.8	Summary	136
8	Conclusions	139
8.1	Objectives achieved.....	139
8.2	Limitations.....	140
8.3	Future work	140
8.4	Final appreciation	140
	References	143
	Annex A	154
A.1	Value analysis	154
A.1.1	New Concept Development Model.....	154
A.1.2	Value	156
A.1.3	Functional Analysis (FAST).....	158
	Annex B	159
B.1	Results per query on B-On and ScienceDirect.....	159
B.2	Container level process views with more outcomes	161
B.3	Cloud Run additional configuration options.....	163

Table of Figures

Figure 1 - The stages of cyber resilience.....	3
Figure 2 - Example of a microservice architecture [16].....	5
Figure 3 - Number of results presented by B-On and ScienceDirect on the search using all queries	12
Figure 4 - Amazon OpenSearch workflow [40].....	25
Figure 5 – Example architecture of a real-time anomaly detection solution in Google Cloud [44]	26
Figure 6 - Circuit breaker states [47].....	27
Figure 7 - Core context portion of the domain model.....	32
Figure 8 - User context portion of the domain model.....	33
Figure 9 - Transfer context portion of the domain model.....	34
Figure 10 - Payment context portion of the domain model.....	35
Figure 11 - Domain model	36
Figure 12 - Activity diagram describing process 1	37
Figure 13 - Activity diagram describing process 2	37
Figure 14 - Activity diagram describing process 3	38
Figure 15 - Activity diagram describing processes 4 through 6.....	38
Figure 16 - Use case diagram	39
Figure 17 - Logical view of the context level of the system	42
Figure 18 - Logical view of the container level of the system	43
Figure 19 - Logical view of the container level of the elastic stack.....	44
Figure 20 - Physical view of the container level of the system	45
Figure 21 - Process view of the container level of the system regarding the happy path of execution of a fund transfer	46
Figure 22 - Logical view of the component level of the Core Service container	48
Figure 23 - Process view of the component level of the Core Service regarding fetching user information	49
Figure 24 - Logical view of the component level of the Payment Service container	52
Figure 25 - Process view of the container level of the Payment Service container regarding payment execution with only the happy path.....	52
Figure 26 - Class diagram of the class level referring to the Transaction-related components of the Core Service container	54
Figure 27 - Class diagram of the class level of the Transfer Service container components	57
Figure 28 - Entity-Relationship model of the Core Service container	59
Figure 29 - Entity-Relationship model of the User Service, Transfer Service, and Payment Service containers	61
Figure 30 - Example of Swagger UI page	66
Figure 31 - Example of an Ocelot API Gateway configuration with Consul [86]	67
Figure 32 - Example of a system with client-side service discovery [90]	68
Figure 33 - Example of a system with server-side service discovery [90]	69

Figure 34 - Consul UI	69
Figure 35 - Example of searching data through analytics in Kibana	71
Figure 36 - Example of consulting Elastic Observability overview on Kibana.....	72
Figure 37 - Example of metrics collected into a dashboard on Kibana	73
Figure 38 - Example of uptime monitoring with information collected by Heartbeat on Kibana	74
Figure 39 - Consulting User Service logs in the Discover feature on Kibana	82
Figure 40 – User service APM overview tab	84
Figure 41 - User service APM transactions tab	84
Figure 42 - User service APM metrics tab	85
Figure 43 - User service APM service map tab	85
Figure 44 - Metrics Inventory overview of all the API Gateway instances CPU usage	87
Figure 45 - Metrics Inventory overview of a selected API Gateway container metrics	88
Figure 46 - Metrics Explorer overview of all the API Gateway instances average CPU usage ...	88
Figure 47 - Anomaly detection option in the Services view in the APM section of Observability	89
Figure 48 - Anomaly detection tab with the created ML job.....	90
Figure 49 - Anomaly detection job creation options	91
Figure 50 - Selection of time range for collected data for analysis in anomaly detection job creation	92
Figure 51 - Selection of field for analysis in anomaly detection job creation	92
Figure 52 - Job details configuration of in anomaly detection job creation	93
Figure 53 - Job validation in the anomaly detection job creation	93
Figure 54 - Job summary in the anomaly detection job creation	94
Figure 55 - Data frame analytics job creation	95
Figure 56 - Data frame analytics job creation included fields and training segment	96
Figure 57 - Data frame analytics job creation additional options segment	97
Figure 58 - Data frame analytics job creation job details segment	97
Figure 59 - Data frame analytics job creation validation segment	98
Figure 60 - Data frame job creation create segment.....	98
Figure 61 - Test Pyramid [108]	113
Figure 62 – CI/CD workflow [122]	122
Figure – Cloud Build trigger setup in relation to the name, region, event, and source	124
Figure – Cloud Build trigger setup in relation to the configuration and service account.....	125
Figure 65 – Cloud Build build details after pipeline execution	126
Figure 66 – Cloud Run service configuration	126
Figure 67 – Cloud Run service creation.....	127
Figure 68 – Deployed service on Cloud Run	128
Figure 69 - Uptime of the User Service during the testing period.....	130
Figure 70 - Transaction metrics of the User Service during the testing period	130
Figure 71 - Analysis of dependent libraries of the Core Service solution	134
Figure 72 - Detected anomalies in transactions by the APM ML job	135
Figure 73 - Detected anomalies in log count and ingestion rate by the logs ML job	135

Figure - The New Concept Development model.....	154
Figure - FAST diagram for this dissertation.....	158
Figure 76 - Number of results presented by B-On and ScienceDirect using the query “microservices operational resilience OR microservices resilience”.....	159
Figure 77 - Number of results presented by B-On and ScienceDirect using the query “microservices cybersecurity OR microservices cyber security”.....	160
Figure 78 - Number of results presented by B-On and ScienceDirect using the query “microservices artificial intelligence”.....	160
Figure 79 - Number of results presented by B-On and ScienceDirect using the query “artificial intelligence cybersecurity OR artificial intelligence cyber security”.....	160
Figure 80 - Process view of the container level of the system regarding the execution of a fund transfer with all possible outcomes.....	161
Figure 81 - Process view of the container level of the Payment Service container regarding payment execution with all paths.....	162
Figure 82 – Cloud Run container options regarding general and capacity options.....	163
Figure 83 – Cloud Run container options regarding execution environment, environment variables, secrets, health checks, and cloud SQL connections.....	164
Figure 84 – Cloud Run network options.....	165
Figure 85 – Cloud Run security options.....	165

Table of Tables

Table 1 - Types of Critical Systems [11]	4
Table 2 - Conditions for inclusion/exclusion criteria	12
Table 3 - Summary of papers on the transition of critical systems into the microservice architecture.....	13
Table 4 – Comparison of papers on the proposal of cybersecurity practices in microservices .	15
Table 5 - Comparison of papers on the usage of artificial intelligence in microservices	17
Table 6 – Comparison of papers on threat detection and/or labeling in cybersecurity	22
Table 7 - Comparison of researched papers in the aspects of this dissertation	23
Table 8 - Types of Machine Learning in Elastic [45]	26
Table 9 - Adopted technologies	63
Table 10 - Metrics by quality attributes related to resilience [105], [106]	110
Table 11 - Metrics by quality attributes related to security [106], [107]	112
Table 12- Goals, questions, metrics.....	112
Table 13 - Usage frequency of each microservice	132
Table 14 - Evaluation results.....	137
Table 15 - Summary of objectives.....	139

Table of Snippets

Snippet 1 - Consul configuration on the User Service's appsettings file	76
Snippet 2 - Service Discovery set up on the User Service's Program class.....	77
Snippet 3 - Implementation of discovery client and handler to interact with the core service	78
Snippet 4 - Serilog configuration on host builder	80
Snippet 5 - Elastic APM agent configuration in the appsettings file	82
Snippet 6 - Elastic APM agent usage in the Startup class	83
Snippet 7 - Metricbeat.yml file	86
Snippet 8 - Dockerfile build environment restore segment	99
Snippet 9 - Dockerfile build environment build segment.....	100
Snippet 10 - Dockerfile runtime environment entry-point segment	100
Snippet 11 - Docker-compose file.....	102
Snippet 12 - Elasticsearch YAML file	103
Snippet 13 - Kibana YAML file	103
Snippet 14 - APM Server YAML file.....	104
Snippet 15 - Partial Heartbeat YAML file	105
Snippet 16 - Elasticsearch Dockerfile.....	105
Snippet 17 - Docker-compose network and volume configuration.....	106
Snippet 18 - Elasticsearch configuration in the docker-compose file	106
Snippet 19 - Metricbeat service configuration on the docker-compose file.....	107
Snippet 20 - Kibana configuration on the docker-compose file	108
Snippet 21 - Example unit test of the debit functionality of a bank account in C#.....	114
Snippet 22 - Setup of the UserControllerTests class	116
Snippet 23 - Unit test that validates a specific behavior of the UserController class	117
Snippet 24 - Unit test that validates a specific behavior of the UserService class.....	118
Snippet 25 - Unit test that validates a specific behavior of the UserMapper class	119
Snippet 26 - Partial implementation of the class fixture developed for the UserRepository integration tests.....	120
Snippet 27 - Pair of integration tests of the UserRepository class	121
Snippet 28 - Examples of usage of Vegeta.....	128
Snippet 29 - Quality of Service options for the circuit breaker feature	131
Snippet 30 - Adding Polly to the services	131
Snippet 31 - Authentication and Authorization configuration	136
Snippet 32 - Usage of the authorize annotation in an endpoint	136

Acronyms

ACM	Association for Computing Machinery
AI	Artificial Intelligence
AL	Autonomous Learner
ANN	Artificial Neural Network
AS	Architectural Style
AWS	Amazon Web Services
CI	Critical Infrastructure
CI/CD	Continuous Integration and Continuous Delivery/Deployment
CR	Consistency Rate
CVM	Containers of Virtual Machines
DBN	Deep Believe Network
DCRNN	Diffusion Convolutional Recurrent Neural Network
DDD	Domain-Driven Design
DDoS	Distributed Denial-of-Service
DL	Deep Learning
FAST	Functional Analysis System Technique
FFNN	Feed-Forward Neural Network
FX	Foreign Exchange
GAN	Generative Adversarial Network
GCNN	Graph Convolutional Neural Network
GCP	Google Cloud Platform
GQM	Goal Questions Metrics
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IoC	Inversion of Control

IoT	Internet of Things
KNN	K-Nearest-Neighbors
LSTM	Long-Short-Term Memory
MCI	Microservice-based Critical Infrastructure
MIL	Microservices Integration Layer
ML	Machine Learning
MLaaS	Machine Learning-as-a-Service
OO	Object-Oriented
ORM	Object-Relational Mapping
QA	Quality Attribute
R&D	Research and Development
RF	Random Forest
RNN	Recurrent Neural Network
RQ	Research Question
SLA	Service Level Agreements
SMR	Systematic Mapping Review
SVM	Support Vector Machine

1 Introduction

This chapter aims to present the context, problem, and objectives associated with the development of this dissertation, as well as display the planning and the overall document structure.

1.1 Context

This section presents the important aspects inherent to this dissertation. Those are the aspects of cybersecurity, artificial intelligence (AI), operational resilience, and microservice architecture.

1.1.1 Cybersecurity

Cybersecurity is the art of protecting networks, devices, and data from unauthorized access or criminal use and the practice of ensuring confidentiality, integrity, and availability of information [1].

It's important since smartphones, computers, and the internet are now such a fundamental part of modern life, that it's difficult to imagine how society would function without them. From online banking and shopping, to email and social media, it's more important than ever to take steps that can prevent cyber criminals from getting hold of our accounts, data, and devices [2].

1.1.2 Artificial intelligence

In its simplest form, AI is a field that combines computer science and robust datasets to enable problem-solving [3]. It encompasses two disciplines that are frequently mentioned with it:

- **Machine Learning (ML):** ML is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behavior [4]. It uses algorithms

to parse data, learn from that data, and make informed decisions based on what it has learned [5];

- **Deep Learning (DL):** DL is a subfield of machine learning that structures algorithms in layers to create an Artificial Neural Network (ANN) that can learn and make intelligent decisions on its own [5]. DL is what powers the most human-like AI.

In the context of cybersecurity, AI is an interesting resource to be employed since it can be leveraged in multiple manners such as threat detection, classification, and blocking to have a higher degree of security in networks and services. It has the possibility of preventing new attacks with the usage of autonomous systems and learning patterns [6].

1.1.3 Operational resilience

Operational resilience is the ability to identify and protect from threats and potential failures, respond and adapt to, as well as recover and learn from disruptive events to minimize their impact on the delivery of critical operations through disruption [7].

The term cyber resilience refers to the ability of systems to resist and recover from or adapt to an adverse occurrence is used to refer to operational resilience of systems [8].

Cyber resilience is important due to its ability to restore a system's functionality after an attack directed at a broad range of systems, such as critical infrastructures. Cyberattacks on water supplies, energy and communication networks, and healthcare facilities bring significant consequences [9]. For example, a ransomware attack on a health service organization reduced the organization's data handling methods to be manual for a significant amount of time [10].

[9] affirms that "Such attacks can produce massive damage to the economic well-being of an organization and to our broader society, and even endanger human lives."

When such attacks occur, the systems absorb the impact, and their functionality begins to degrade. Then, for the systems to bounce back to normalcy, mechanisms and/or processes are engaged to absorb the negative impacts so that the system's functionality can be recovered.

The stages of cyber resilience are presented in Figure 1 [9].

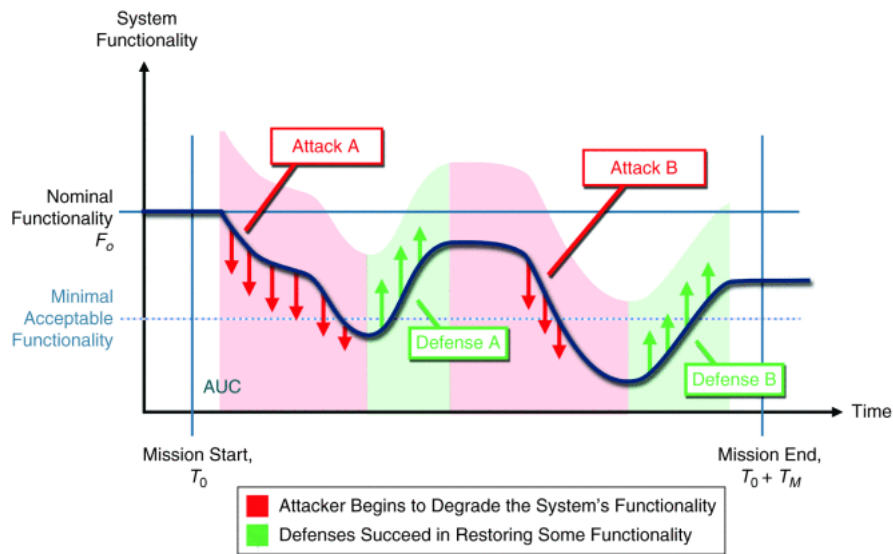


Figure 1 - The stages of cyber resilience

From what can be observed, cyber resilience is focused on recovery either partial or complete. In some cases, it could involve adaptations to improve functionality or resilience to future attacks. It will also depend on the aspects of the systems (e.g., design, controls, anticipation) before an attack occurs [9].

Ultimately, the assessment of cyber resilience begins with acknowledging the inevitability of attacks. When systems are affected, their functionality is degraded, and the focus is on the speed of recovery [9].

1.1.4 Critical systems

According to [11] in their study, “Critical systems are those in which a failure or malfunction could cause considerable negative effects.” They elaborate further by adding that critical systems “[...] may have strict requirements for security and safety, to protect the user or others.”

Recently, the Industry, Research and Energy committee of the European Parliament identified that due to the coronavirus pandemic an unforeseen acceleration in digital transformation took place in societies around the world. This means that in addition to the traditional critical sectors (e.g., energy, water plants, hospitals, etc.), other sectors outside this category began being identified as critical (e.g., e-commerce, e-banking, etc.) due to the growth in dependence of these systems by societies, meaning that the opportunities in cyber-crimes (i.e., cyber-attacks) for malicious authors increased in exponentially as well [12].

Four types of critical systems are identified, as presented in Table 1.

Table 1 - Types of Critical Systems [11]

Type of Critical	Implication for Failure
Safety-Critical	May lead to loss of life, serious personal injury, or damage to the natural environment.
Mission-Critical	May lead to an inability to complete the overall system or project objectives (e.g., loss of critical infrastructure or data).
Business-Critical	May lead to significant tangible or intangible economic costs (e.g., loss of business or damage to reputation).
Security-Critical	May lead to loss of sensitive data through theft or accidental loss.

The impact of a critical system, where a failure or malfunction occurs, will depend on the setting or context where it is inserted (i.e., Type of critical system). For example, in an e-commerce setting, the failure of such systems will always have an impact on the company's financials, at minimum, but could be elevated to the closure of the company. The cost of this impact will depend on the system that is down:

- A website and/or ordering system being unavailable for several hours could result in the loss of business to a different competitor (i.e., Business-Critical).
- Unauthorized/Illegitimate access to the customer information system could result in the loss of sensitive data of the customers (i.e., Security-Critical).
- Loss or corruption of data, power shortages, faulty hardware, or environmental disasters in the data centers would lead to the cease of functions of the overall company (i.e., Mission-Critical).

On the other hand, in a medical context, any type of failure in critical systems (e.g., Pacemakers, defibrillator machines, robotic surgery machines) could mean the loss of life (i.e., Safety-Critical).

1.1.5 Microservices

Microservices, or microservice architecture, according to [13] is an architectural style inspired by service-oriented architecture in combination with the old Unix principle of "do one thing and do it well." They are supposed to be lightweight, flexible, and easy to get started with, fitting in with modern software engineering trends such as Agile development, Domain Driven Design (DDD), cloud, containerization, and virtualization.

There are numerous ways to describe microservices, but the two most common are the following:

1. [14] defined microservices as small autonomous services built around the principles of Model (services) around business concepts, adopt a culture of automation, hide internal

implementation details, decentralize all things, isolate failure, and make services independently deployable and highly observable.

- [15] defined microservices as “an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.”

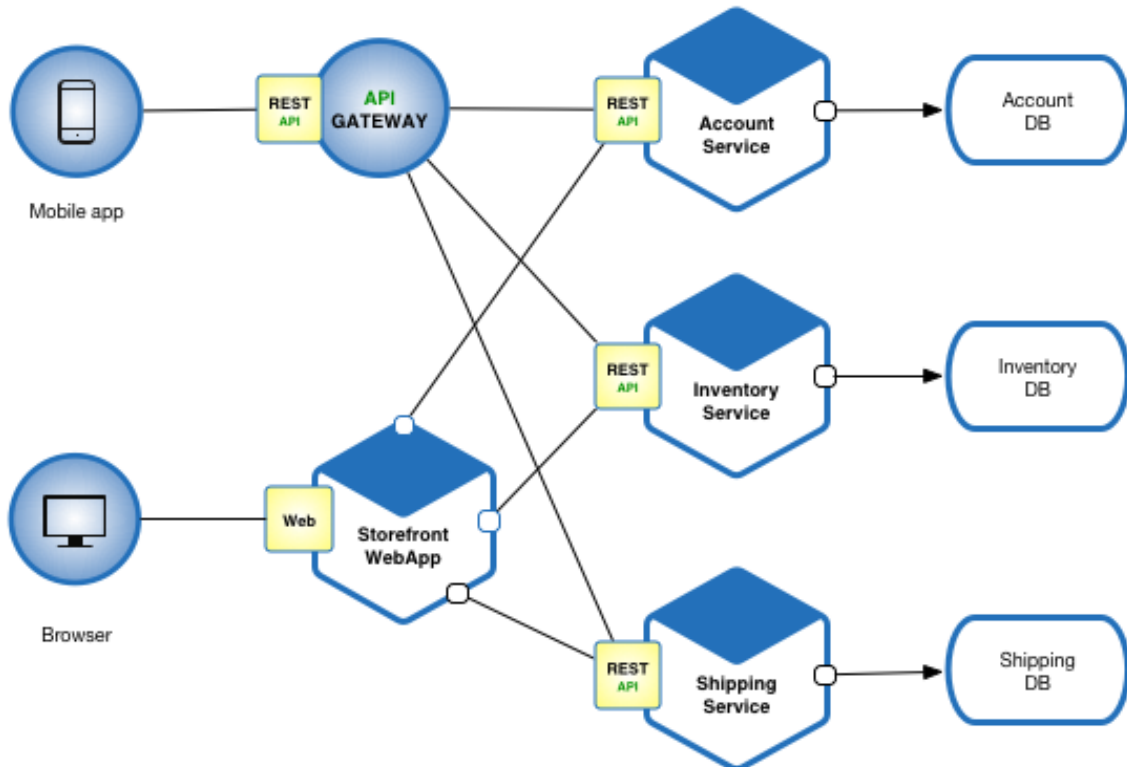


Figure 2 - Example of a microservice architecture [16]

Looking at Figure 2, we can observe an example of a simple microservice architecture of an e-commerce system. Its backend is composed of three microservices (Account, Inventory, and Shipping), each with its database, and on the front-end side, there is a web application and mobile application. The web application can be accessed through a browser and communicates directly with every single microservice through their respective REST API [17], while the mobile application communicates with an API Gateway [18] that will then call the necessary microservices to complete its request.

According to [14], the key benefits of the microservice architecture are:

- **Technology Heterogeneity:** Since the system can be composed of multiple, collaborating services, the decision to use different technologies for each can be made. This allows the choice of the right tool for each job, rather than following a one-size-fits-all approach.

- **Resilience:** If one component (i.e., service) of the system fails and it doesn't cascade, the problem can be isolated, and the rest of the system can carry on. Systems that can handle the total failure of services and degrade functionality accordingly can be built.
- **Scaling:** Only the services that need scaling can be scaled, instead of the system-as-a-whole. This allows to run other parts of the system on smaller, less powerful hardware.
- **Ease of Deployment:** A change can be made to a service and deployed independently of the rest of the system, allowing the fast deployment of code and rollback of said service in case of a problem.
- **Organization Alignment:** Allows for the better alignment of architecture for organizations, minimizing the number of people working on any one codebase to reach the optimal team size and productivity. Ownership of services can also be shifted between teams to keep people working on more than one service.
- **Composability:** Opens opportunities for the reuse of functionality. Allows functionality to be consumed in different ways for different purposes.
- **Optimizing for Replaceability:** Being small, these can be replaced with a better implementation or removed altogether. Teams are comfortable with completely rewriting or killing services when required or no longer needed.

On the other hand, there are also disadvantages to microservices:

- **Multiple design choices:** The increased complexity that it brings can be a pain to keep track of the overall system is composed of several components (i.e., services).
- **Difficulty of testing:** The larger scope tests (i.e., end-to-end tests) become a challenge as the number of moving parts increases (i.e., services). These moving parts can introduce failures that have nothing to do with the functionality under test, but some other non-related problem (e.g., services being down, network issue, incorrect version of used services being tested against).
- **Difficulty of monitoring:** Unlike a monolith where there is only the need to look at a single entity, microservices, scale with the number of components, meaning multiple servers to monitor, multiple logfiles to go through, and multiple places where network latency could cause problems. This in turn makes the task of monitoring and investigating issues substantially harder.
- **Operation overhead:** With multiple services to build, test, deploy, and run, comes a significant operation overhead. This means the number of processes also scales (e.g., the number of pipelines, and repositories to maintain).

To conclude, the microservice architecture is ideal in the current context of rapid transformation and evolution of systems (i.e., agile environment) where it is important for organizations to stay relevant in the market, especially those in the context of critical systems, but it also has some tradeoffs they need to have in mind.

1.2 Problem

The security of critical IT systems (e.g., bank systems, airports, hospitals, etc.) is traditionally associated with monolithic data centers and legacy infrastructures, which entails a set of added challenges in terms of evolution and protection. Critical systems are characterized by a high degree of intolerance to threats, where exposure to the smallest failure could cause significant damage. In the current context of rapid transformation, especially regarding the variety of attacks that are practiced today, it is important to bear in mind that legacy systems, with structural adaptation difficulties, are especially susceptible [19]–[21]. In this sense, it is essential to be aware of the advantages and solutions offered by more recent architectures, such as Microservices, where the impacts of evolution will be smaller thanks to the segmentation of the system into several components and where its security is addressed using multiple layers of protection.

1.3 Objectives

The main objective of this dissertation is to understand how microservices can respond, from an architectural point of view to a context of high demand that is from critical systems relying on concepts such as resilience, durability, adaptability, scalability, confidentiality, integrity, and availability.

The goal is to conduct a case study of the compatibility of a simple banking system that adopts the microservice architecture, from the aspects of cybersecurity, particularly how artificial intelligence and machine learning can perform in this aspect, and operational resilience to a specific field, in this case, critical systems.

To achieve these, the following research questions (RQs) were formulated:

- **RQ.1:** Does the proposed architecture offer high levels of operational resilience?
- **RQ.2:** Does the adoption of AI methods assist in security in such a context?

1.4 Planning

To plan this case study, the initial goal of researching the current developments and/or studies published on the subject at hand was set so that the conclusions taken from the existing research can be understood and worked with to have a strong contextual basis to approach this case study and to add more value to it.

Knowing this information, a value analysis was made for this project, to understand the need for it and to realize what this adds to the problem and to the context where it's inserted.

Afterward, the design and implementation of the software prototype that will be used to approach the goal of this project will be made, followed by extensive testing in various scenarios (i.e., experimentation).

Finally, the outputs of the experimentation made will be gathered to assess these, based on a series of measurable dimensions and factors.

To summarize, the execution of this case study can be split into three stages:

1. **Research and analysis:** Gathering of existing knowledge on the theme as well as the evaluation methodologies, and value analysis of the project.
2. **Design, implementation, and experimentation:** Design, supported by UML diagrams (i.e., architecture, components, system), and the development of the software prototype, followed by experimentation composed of various scenarios.
3. **Outputs and results:** Gathering of the outputs by the experimentation to compile these. The result of this compilation will be used to answer the RQs and reach a conclusion.

1.5 Document structure

To ease the reading of this document, below is the list of chapters accompanied by an explanation of their contents:

- **Introduction** (Chapter 1) – Introduction of the dissertation. Here the context, problem, objectives, and planning of this dissertation can be found.
- **State of the art** (Chapter 2) – Presents what relevant research has been developed on the subject, what this dissertation seeks to contribute to, what technologies exist, and what patterns are typically applied in such a context.
- **Analysis** (Chapter 3) – Presents an analysis of the domain's concepts and activities, its processes and stakeholders, and its use cases.
- **Design** (Chapter 4) – Presents the design choices made on the developed solution from coarse-grained to fine-grained aspects finishing on the database design.
- **Technologies** (Chapter 5) – Introduces and describes the adopted technologies for the development of the solution.
- **Implementation** (Chapter 6) – Exposes the construction of the solution, starting by describing particular situations and how the underlying infrastructure was set up.
- **Evaluation** (Chapter 7) – The solution is evaluated according to a certain methodology that aims to answer a couple of investigation hypotheses, what tests were developed and the importance of CI/CD, to understand what conclusions can be made.

- **Conclusions** (Chapter 8) – The conclusions that were taken from the work performed can be found here as well as the found limitations, the future work that can be done, and a personal appreciation of the work done by the student.
- **Annex A** – Contains the value analysis of this dissertation.
- **Annex B** – Complimentary figures and diagram that present more information related to certain parts of the dissertation.

2 State of the art

This chapter aims to explore studies and papers that have been published regarding the dissertation's aspects previously contextualized (see Context), to analyze and compare their aspects, and the evolution of research over the recent years. Then, an overall comparison of these studies is conducted, followed by a conclusion on these, based on the presented information, on how this dissertation positions itself compared to the current investigation. Finally, a description of some existing technologies and commonly applied concepts is presented.

2.1 Relevant studies and papers

This section presents the evolution of research over the years, and relevant literature, analyzes it, compares their aspects, and looks at the open questions left by these. There have been a lot of interesting studies and papers published over the last years but there isn't one that touches on all four aspects (i.e., microservices, cybersecurity, artificial intelligence, and operational resilience) that this dissertation seeks to work on. They only look at combinations of microservices with the other three aspects. Research on cybersecurity regarding cyberattacks was conducted, specifically the detection, labeling/classification with resources to artificial intelligence and machine learning to understand how it could relate in a microservices context.

2.1.1 Evolution of research over the years

To learn the evolution of the theme at hand over the years the digital libraries Association for Computing Machinery (ACM), Science Direct, and Institute of Electrical and Electronics Engineers (IEEE) Xplore were surveyed about the number of studies regarding this theme through the B-On and Science Direct digital libraries. The research was made following the Systematic Mapping Review (SMR) method [22].

The conditions in Table 2 were used as criteria of inclusion/exclusion.

Table 2 - Conditions for inclusion/exclusion criteria

Criteria	Conditions
Information topic	Migration of critical systems to the microservice architecture; Cybersecurity practices in microservices; Artificial intelligence in microservices; Threat detection and/or labeling in cybersecurity
Type of information	Academic Journals; Conference Materials; Magazines; Trade Publications; Reports; Review Articles; Research Articles
Publication date	Between 2018 and 2022
Language	English
Type of review	Peer review
Access	Accessible to ISEP teaching staff and students

Bearing in mind the stated conditions, the following queries were used:

- microservices operational resilience OR microservices resilience;
- microservices cybersecurity OR microservices cyber security;
- microservices artificial intelligence;
- artificial intelligence cybersecurity OR artificial intelligence cyber security.

The obtained overall results are presented in Figure 3.

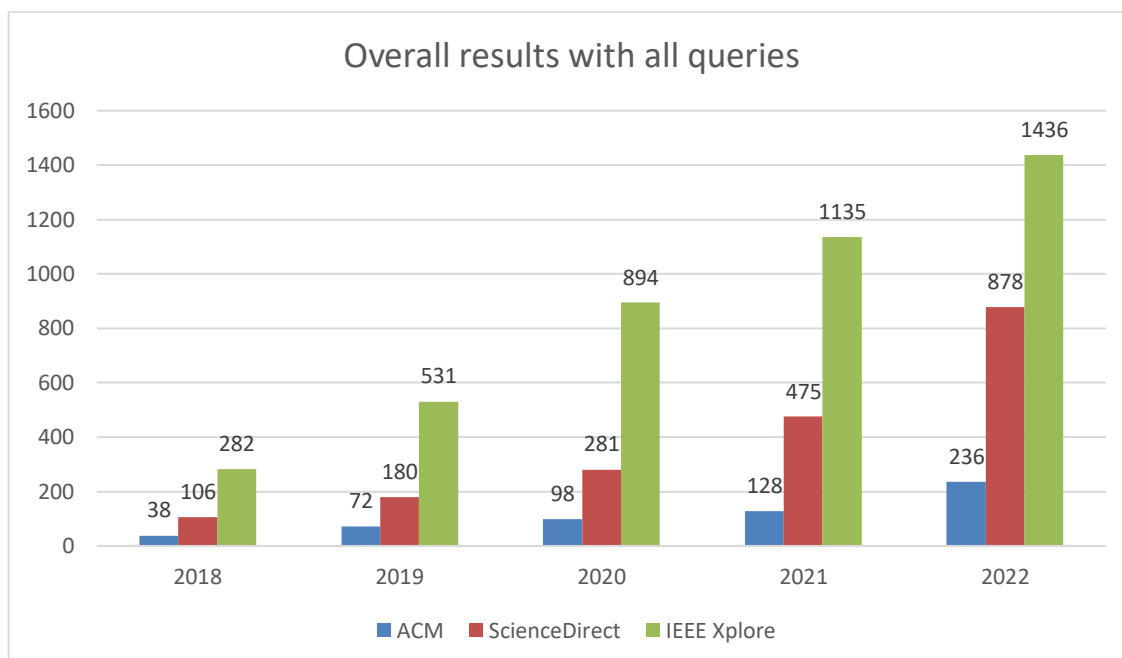


Figure 3 - Number of results presented by B-On and ScienceDirect on the search using all queries¹

¹ Obtained in 2023-02-25.

As shown in the graph above, in all digital libraries, there is an increasing trend in the number of results over the last five years with 2022 as the year with largest number of obtained results for all. This trend is most apparent in IEEE Xplore then in the other two, with Science Direct coming up afterwards and ACM consistently with the lowest number of obtained results.

It should be noted that most of the obtained results were from the last query, with the other three queries having a more conservative number of results. The results for each individual query can be observed in Annex B.

A total of 13 papers were selected for full review after the removal of duplicates and application of the criteria above. These papers are introduced and described in the following sections.

2.1.2 Transition of critical systems into the microservice architecture

In this section, papers related to the transition into a microservice architecture by critical systems are presented. In Table 3 a summary of these papers can be observed, and a more complete description is followed.

Table 3 - Summary of papers on the transition of critical systems into the microservice architecture

Paper	Publication Date	Scope of transition	Topic focus
[23]	12/2021	One core system	Complete migration of a core system from a monolith architecture to a microservice architecture
[24]	08/2022	Gradual transition of systems	Development of an integration layer on top existing legacy systems to encapsulate these, and the gradual peeling off (i.e., decommission) monolith components/systems into new microservices
[25]	08/2022	Integration layer on top of legacy systems	Development of an integration layer based on a microservice architecture on top of legacy systems and the flip of consumer/producer roles to avoid reengineering the legacy systems

In [23] and [24] detail how large northern European financial organizations migrated legacy systems (i.e., critical systems), previously in a monolithic architecture, into a microservice architecture. In the case of [23], the organization made a complete transition of one of their core systems, responsible for foreign exchange (FX), into the microservice architecture. In [24], the organization conceived and is still following a global transition plan of their legacy systems into the microservice architecture. This transition plan consists of three key components: the (1) gradual peeling off monolith's services into microservice, the (2) decommissioning of the monolithic system by identifying services to be transitioned, and the (3) encapsulation of the

legacy systems with an integration layer to allow engineers to develop new applications of top of this layer. The development teams at both organizations noted on how much faster and easier it became to introduce change and innovation into their systems, after separating their components into independent microservices. They further compare both architectures from different aspects (e.g., availability, reliability, complexity, heterogeneity) and the problems that the migration solved, minimized and persist. Overall, the systems became faster, resilient and easier to operate, even with the increased operation overhead.

[25] documents how at Hitachi Research & Development (R&D) they developed a microservices integration layer (MIL) for next-generation rail operations centers, mission-critical systems. What is interesting about how they implemented the MIL is that instead of migrating the existing legacy systems, whose programming languages have limited native support for building microservices, they reversed the papers by turning the MIL into the service provider and the legacy systems into consumers. This eliminated the need to reengineer the legacy systems into microservices, which would be a significant challenge. So, by turning all systems into clients of the MIL, they centralized the security management and simplified the tracing of security calls, monitoring and debugging.

They highlight four lessons learned:

- **Fallback option:** Having a fallback option is a good idea when introducing new technology. Despite of extensive testing and no issues over continuous operation during a few weeks, a memory leak was found that led to failures after very long runtimes.
- **Redundancy \neq Availability:** The ability to failover is not always sufficient to guarantee availability. They exemplify that when starting up or recovery from failure, these systems need to read a lot of state information from the MIL quickly. A microservice instance failing in combination with a client causing an increased message load during recovery could cause a domino effect if the other instances are unable to handle the increased loads from failed instances and if the load balancer is not distributing loads equally.
- **Reactive systems and backpressure:** It highlights the importance of detecting failures early and recovering quickly through failover. Backpressure is a characteristic of reactive systems, where slow consumers (e.g., subscribers) are unable to consume messages at the rate they are produced. Their initial strategy of using an unlimited buffer worked until consumers became unable to recover, causing performance degradation, loss of messages and out-of-memory errors on microservice instances. So, they set a limit on the backpressure buffer size for each individual subscription. Another issue was the inability of the microservice to detect failed network connections, which they fixed with a heartbeat event. These lessons helped them develop a very resilient system that is currently in operation.
- **Simple and future-proof design:** The lesson here is that its good practice to think ahead and consider possible future requirements and maintain architectural flexibility to quickly respond to new requirements. They also note that the theoretical advantages of autonomous development, deployment, composability, and replaceability in practice can be easily negated when hidden dependencies are introduced, such as adapters or connectors from vendor or version specific external components (i.e., external

dependencies). The good practice would be to avoid using these and to rely on open standards when possible.

2.1.2.1 Open questions

[23] affirm that there will be an increase in the development of new programming languages intending to address the microservice paradigm. They also note that this paradigm is still missing a conceptual model able to support the early stages of development and list a set of research challenges to be covered by a software-engineering approach within the microservices field:

- The need for a uniform way to model autonomous and heterogeneous microservices, to allow the easy interconnection through dynamic relations, turning the engineering process more efficient;
- Each microservice should have a partial view on the system knowledge, and at the same time must be specialized and adapted to face different requirements, user needs, context changes, and missing functionalities;
- The possibility of handling at run-time improbable situations such as context changes, availability of functionalities, and trust negotiations, instead of analyzing these situations at design-time and pre-embedding the corresponding recovery activities.

[24] notes on a limitation that is the lack of quantitative metrics to give a more detailed conclusion on his work. He does provide a direction to follow for future research which is to investigate performance loss when encapsulating large legacy monolith systems containing architecture debt, and to research the effect of encapsulation when compared with rewriting systems, which would benefit organizations with legacy system that want to be competitive in a rapidly changing market.

2.1.3 Proposal of cybersecurity practices for microservices

In this section papers related to the proposal of cybersecurity practices for microservice architecture-based systems are presented. In Table 4 a comparison of these papers can be observed, and a more complete description is followed.

Table 4 – Comparison of papers on the proposal of cybersecurity practices in microservices

Paper	Publication Date	Scope of proposal	Technical approach	Security practices proposed
[13]	05/2018	Component level	Containers	Security framework for authentication between containers

Paper	Publication Date	Scope of proposal	Technical approach	Security practices proposed
[26]	08/2021	System level	Physical systems	HTTPS and authentication on the API Gateway, authentication in the orchestrator service, service responsible for authorization
[27]	01/2021	System level	Kubernetes cluster	Injection of HAProxy service in each pod to encrypt internal requests, Kubernetes to encrypt external requests

In [13], it touches on an interesting point that is that most engineers when developing microservices assume that the components are safe inside their boundaries (i.e., the security perimeter encircles the system). They instead propose that the security perimeter should be the level of each microservice. To achieve that, they designate a standard security framework for the authentication between microservices, giving Netflix as an example of the industry.

[26] on the other hand broadens the scope by applying different security mechanisms for authentication (i.e., Use of HTTPS and authentication on the API Gateway and authentication in the orchestrator microservice.) and authorization (i.e., Guard microservice that checks authorization levels to access the other components.) in the different components/levels of a microservice architecture-based system. It proposes using orchestration as a mechanism to control the components and demonstrates how the flow of a request works over the security mechanisms comparing the execution times of each security mechanism and all these combined. The results were quite interesting because while the overall execution time with the combined security mechanisms was significantly higher, it was still within an adequate range.

Another interesting proposal on how to add security in microservices, in the context of a Kubernetes [28] cluster, is shown in [27]. They propose the injection of a HAProxy [29] service in each pod, to encrypt both external (Kubernetes Service) and internal (i.e., between pods using Transport protocol – Kubernetes Headless Service) HTTP requests. In this study the system only suffered a 7% loss in performance, an acceptable value.

2.1.3.1 Open questions

[26] ends on the direction that is to replicate his study from the choreography composition of microservices, since it provides faster composition cycle times than orchestration, as well the investigation of the expansion of security mechanisms.

[27] mentions that there are still other technologies to investigate that could have a better impact, or the need to evaluate their proposal in various workload patterns, and that there is the hypothesis of using encryption to store data.

2.1.4 Usage of artificial intelligence in microservices

In this section, papers related on usage of artificial intelligence in microservices are presented. In Table 5 a comparison of papers is presented followed by a more complete description of each.

Table 5 - Comparison of papers on the usage of artificial intelligence in microservices

Paper	Publication Date	Context proposal	of	AI method	Technical approach	Proposal main points
[30]	03/2022	Security of Internet Things systems	of (IoT)	DL	Docker containers	Multiple containers per AI based microservice or cybersecurity solution at the edge of the network; AI based orchestration for the optimization of resources and secure deployment of microservices; Each microservice contributes to the AI service(s) of each application.
[31]	07/2018	Implementation of ML-as-a-Service for general purpose usage		ML	Docker containers	ML configuration pool that indicates which model and input/output nodes to use to quickly deploy a microservice with the chosen configuration; Independent trainer service that: receives training data from other microservices and feeds it to the ML microservice; evaluates the ML microservice output to update parameters in configuration pool.
[32]	12/2022	Self-improvement of systems		DL	Docker containers	One component to analyze, generate and inject errors to stress the microservice; One component that first extracts real-world data from random noise to generate synthetic data then compares both data

Paper	Publication Date	Context proposal	of	AI method	Technical approach	Proposal main points
						types until they are indistinguishable; One component that uses that data to identify opportunities and builds a checklist for possible scenarios of fragility; Virtual environment for stress testing real-world copies of the systems with countermeasures.

In the first one, [30] conducted an extensive gathering of existing literature concerning microservices, AI models such as ML and DL, data privacy and network security within IoT nodes. After this gathering, they consolidated the gathered literature into three groups (i.e., tables), a category, (1) related studies to their theme, (2) edge AI computing, and (3) microservices architectures at edge computing, while presenting what was proposed, developed, etc., in each literature. Then, they made a comparison of all three groups over various categories. Afterwards, introduced what sort of challenges exist concerning security in microservices at edge computing from four distinct aspects, (1) containers, (2) data, (3) permission, and (4) network. They then propose a secure edge AI microservices framework based on realistic implementation of IoT networks. It consists of:

- Multiple Containers of Virtual Machines (CVMs) allocated for every AI-based microservice or cybersecurity solution, such as intrusion detection and threat intelligence, at the edge of the network;
- AI-based microservices could apply an AI orchestration process that automatically configure the computing resources and securely implement microservices at the edge;
- Each microservice provides part of the overall AI service related to each application.

This framework is an interesting proposal, but it raises some questions which can be read in section 2.1.4.1.

[31] proposes on how to utilize ML as a microservice (MLaaS), for offline contexts in IoT. They present a microservice architecture where there exists a trainer microservice and ML configuration pool (i.e., a configuration designates what ML algorithm to use and the input and output data nodes).

This pool would be utilized to quickly deploy microservice with a preferred ML configuration. The trainer microservice implements the functionality to train the ANN offered by the ML microservice. Training data is sent from the other microservices to the trainer microservice, then from the trainer microservice to the ML microservice. It also evaluates the output of the ML microservice and updates the corresponding parameters in the configuration at the pool.

They justify that separating the ML microservice from the trainer microservice makes sense since both offer different functionality and depending on the ANN it offers the possibility to use different trainers or combining training approaches.

They then evaluate the performance of their solution using the same ML microservice with three different configurations, Feed-Forward Neural Network (FFNN), Deep Believe Network (DBN), and Recurrent Neural Network (RNN). Both ML and trainer microservice for the three configurations were implemented using the Google TensorFlow Python library. The performance evaluation was conducted by comparing the MLaaS approach to calling the TensorFlow library directly:

- **Runtime performance:** FFNN and DBN were slower in the ML microservice when compared to the TensorFlow library, whereas RNN was significantly faster in the ML microservice.
- **Learning performance:** They note how the processing steps of the TensorFlow library are significantly longer for classification, but when compared to the MLaaS approach this difference can be neglected due to the overhead added per library invocation by the MLaaS approach.
- **Implementation performance:** They compare for each algorithm the implementation and setup times, as well as the resulting lines of code in both library and MLaaS approach. The latter significantly reduces the development time for both implementation and setup for each of the ANNs, being identical in all.

This study demonstrates how ML implementations can be modularized to facilitate and speed up the implementation of ML functionality in microservices, therefore simplifying the use of ML significantly.

[32] proposes a concept of a microservice architecture antifragile framework for critical infrastructure (CI) systems. The antifragility concept states that systems can improve with threats and shocks [33], whereas resilience means to maintain or to recover the original state. This framework is composed of four components:

- **Stressor:** The central element of the concept. It seeks to challenge the system with randomness and disruptions while benefiting from these to gain information on improvement. It crosses the boundaries of intolerable failures, exposing the microservices' fragility, using virtual copies of these, while monitoring its behavior;
- **Autonomous learner (AL):** This component constructs a fragility list tracing to events or functions of the microservice's unexpected behavior. For that it trains a Generative Adversarial Network (GAN), consisting of two separate neural networks, to generate synthetic data based on data collected from monitoring the fragile microservices (i.e., from the Stressor component). The first neural network has the task of generating synthetic data from random noise extracted from trained real-world data. The second neural network means to compare the generated synthetic data to the real data stopping the classification when both are indistinguishable. This data will help the component to train on unclear

failures and unexpected stressful situations further helping the prediction of unexpected microservice behavior;

- **Antifragility checklist builder:** This component builds a fragility checklist whose aim is to prepare other microservices for possible future scenarios by learning from the fragility of the stressed microservice. It uses the fragility list (i.e., from the AL component) and transforms it into knowledge and opportunities by contextualizing questions for the selected microservice in how it would gain from stressors and approach detected fragilities, then the quantitative responses are aggregated. This checklist could be reused as a guideline to update similar microservices across different CI domains;
- **Virtual CI environment:** Adopting the Digital Twins concept, it seeks to create a realistic digital copy of a microservice-based CI (MCI). For this, real data from realistic MCI is collected. It aims to create a safe virtual CI environment where the collection of data is done over time and under different conditions. It tests the impact of antifragility changes on the behavior of microservices across different virtual CIs after randomly stressing a virtual microservice and conducting an antifragility analysis. This allows to identify antifragility countermeasures to be applied in real MCIs.

This framework is experimented on a microservices application that is a train ticket system. A set of fault types were identified along with fault cases associated with each. For this experiment an open-source AI tool, Gretel.AI [34], was used to generate synthetic data and comparative analysis was performed on various models for the data classifier. These were k-Nearest-Neighbors (KNN), Logistic Regression, Random Forest (RF), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP).

The evaluated metrics were accuracy, recall, precision, and F1-score. In terms of results, the GAN model with the resampling of training data was observed to aid classifiers in outperforming the evaluation results obtained on the original training data (e.g., KNN achieved improved scores on all metrics – 90.17% accuracy, 88.13% precision, 90.17% recall, 88.44% F1-score – with the resampled training data when compared with original training data – 86.02% accuracy, 80.63% precision, 86.02% recall, 82.21% F1-score). These obtained results reveal that the GAN model can increase fault detection with the creation of virtual data like the original data, allowing an increase in anomaly detection capabilities which enable the creation of a more accurate fragility list. Overall, this paper proposes an interesting approach on how AI could be leveraged to improve the behavior of microservices in unexpected scenarios in a context of CIs.

2.1.4.1 Open questions

[30] lists several open research challenges and future research directions:

- **Edge lightweight microservice algorithms:** In the context of IoT, the increasing number of systems deploying nodes with limited resources pose a serious challenge on how these systems are performing. To tackle that, the direction would be to invest into more research on lightweight algorithms to operate on these constrained systems, since these would ideally consider the resource scarcity to maximize the systems operations, performance, etc.;

- **Resource provision at edge nodes:** They identify that DL processes are hard to implement due the limited specifications and sheer amount of data that each device has, that the time required to train a deep network plays a significant role, and that time-critical and real-time applications would not be able to benefit from DL at the IoT edge. The direction would be to investigate how load balancing and scheduling mechanisms could optimize resources at edge nodes and servers;
- **Microservices construction and deployment in IoT devices:** They identify that the execution of AI algorithms could require a set of software dependencies, and for that a solution to isolate the various AI services within shared resources is needed. This brings multiple challenges since it is a new field, one of which is how to flexibly handle AI deployment and management. Another would be how to achieve live migration of microservices to minimize migration times and the unavailability of AI services when faced with user mobility. Finally, the challenge of resource orchestration to obtain the best possible performance;
- **Security-related challenges:** Traditional security practices and solutions cannot cope with the novel requirements brought by IoT networks. There is a need for intelligent methods that would analyze and automate security reactions. The best methods would be to investigate intrusion detection, threat intelligence and digital forensics-based techniques at the network edge;
- **Development of AI algorithms as microservices:** They suggest the development of AI-based micro-algorithms as services. This would allow for the efficient running and quick deployment of distributed AI micro-algorithms, especially ML and DL, for cybersecurity applications at the edge nodes. This paradigm could be a design of an agile and secure architecture for AI models, such as data protection, privacy-preserving, and intrusion detection algorithms.

Finally, they end on that future direction of this paper would be implement their proposal on several testbeds and applying it on different scenarios where AI models are to be deployed and managed, as well as to measure accuracy, efficiency and time metrics.

While [31] do not mention any directions for future research, an important one can be inferred: the application of their proposal for different contexts. The contexts could be the usage of more distinct AI models, such as Long-Short Term Memory (LSTM) or RF, in various fields of application, such as cybersecurity that this dissertation seeks to look at.

[32] leaves some open questions since it is an early concept of its proposed framework and that is also aware the in the current CIs context there are still very few microservice-based systems with many of them being still in monolithic systems, limiting the cloning of real physical MCIs. Future research would be in conducting more extensive evaluations in other distinct CI scenarios while looking at other aspects of the GAN.

2.1.5 Threat detection and/or labeling in cybersecurity

In this section, papers related on threat detection and/or labeling in cybersecurity are presented. In Table 6, a comparison of papers is presented, followed by a more complete description of each.

Table 6 – Comparison of papers on threat detection and/or labeling in cybersecurity

Paper	Publication Date	AI method	Number of AI models	Technical approach	Data dimensionality	Containerized microservices
[35]	02/2021	×	0	Docker containers	Network traffic data	✓
[36]	07/2022	ML	1	Docker containers	Network traffic data	✓
[37]	11/2019	ML	One per system component	Docker containers	Network traffic data	✓
[38]	12/2021	ML	3	Sequential analysis	System generated events	×

In [35] they attempted to detect attacks in two scenarios, (1) password guessing and (2) NoSQL injection, using distributed tracing, a method of tracking application requests as they flow from frontend devices to backend services and databases [39]. This study can be considered limited since it only made use of sequential API calls to simulate attacks and used a small dataset. For the first scenario, they were able to detect the attacks looking at the generated distributed traces, whereas on the second scenario they argue that it is not possible to detect this type of attack since it would result in multiple objects being returned from a NoSQL database instead of a single object, meaning there wouldn't be any substantial changes to the distributed logging data and hence would not be detectable.

To follow up on the previous study, in [36], they broadened the scope by using a neural network, specifically a graph convolutional neural network (GCNN). In short, they designed, developed and trained a Diffusion Convolutional Recurrent Neural Network (DCRNN) model, a state-of-the-art GCNN, by capturing existing spatial data and temporal dynamics within the tracing data. They intended to use the DCRNN to model application topology and predict ongoing traffic, the irregular microservice traffic caused by various types of cyberattacks (i.e., attack detection).

Three distinct types of attacks were used: Brute force password, batch registration of bot accounts, and distributed denial of service (DDoS). All attacks were detected, making this study a success, but there were some limitations. For example, the fact that only a single DCRNN model was used means that this model was trained around the entire systems behaviors. The

tradeoff in this is that it's expensive to maintain a model in this manner due to the implication that whenever a new change is introduced in the system, something frequent in microservices, then it is necessary to retrain the model which is costly both in time and resources. Another study by [37] is mentioned where multiples models were used, meaning less time in training but as a tradeoff this method is not able to detect attacks spanning multiple subsystems.

[38] proposes an interesting tool that contains a ML component (ML Engine) whose models analyze alerts and compute a probability score for each alert. The experimented ML algorithms were RF, MLP and LSTM. In this study, ML was used from a supervised perspective, being split in two phases, learning and predicting. In the learning phase, events, alerts and incidents were collected to be preprocessed into a clean dataset. The alerts include features such as severity and are enriched from related events. Data from an external component was used to label the alerts accordingly to identify which scaled into incidents and those that did not. The dataset was then split for evaluation purposes. After training the model it can be deployed into the ML Engine. Following up in the predicting phase, when the ML Engine is executed, it fetches the alerts not yet inspected and its model calculates an incident probability score for each one, and stores these in a database. To visualize this information, they used Kibana where two dashboards were built, one for alerts and one for incidents. In these dashboards, various elements of visualization such as graphs and gauges were assembled.

2.1.5.1 Open questions

[38] finishes on how the future direction of the application of their tool would be to see how it would respond different specific attack scenarios.

2.2 Comparison of the gathered research

In this section a comparison of the research presented in the previous section is made. This comparison is made by categorizing the aspects that this dissertation seeks to investigate and presents which aspects each paper covers. In Table 7, this comparison is presented.

Table 7 - Comparison of researched papers in the aspects of this dissertation

Category \ Paper	Microservices	Cybersecurity	Operational resilience	Artificial Intelligence
[23]	✓	×	✓	×
[24]	✓	×	✓	×
[25]	✓	×	✓	×
[13]	✓	✓	×	×
[26]	✓	✓	×	×
[27]	✓	✓	×	×
[35]	✓	✓	×	✓

Category Paper	Microservices	Cybersecurity	Operational resilience	Artificial Intelligence
[36]	✓	✓	×	✓
[37]	✓	✓	×	✓
[38]	×	✓	×	✓
[30]	✓	✓	×	✓
[31]	✓	×	×	✓
[32]	✓	×	×	✓

As observed in the table above, current literature does not cover all four aspects this dissertation seeks to expand on. What can be observed is that all papers cover some combinations of these aspects.

2.2.1 Conclusions

In this section, conclusions are made based on the current research and how it relates to this dissertation.

As explained and presented in the previous sections (see Relevant studies and papers and Comparison of the gathered research), it can be concluded current literature does not cover all four aspects that this dissertation seeks to investigate. Summarized, current literature covers one of the following:

- The transition of critical systems from a monolith architecture to a microservice architecture highlights the several benefits that came with this transition;
- Different approaches on how to implement security mechanisms or technologies at different levels on microservices to turn these more secure and in one of those papers highlights the (minor) performance loss that came with those mechanisms;
- How the usage of AI in microservices can be taken advantage of in microservices either in a cybersecurity or a general-purpose context;
- The different techniques that can be used to detect and/or label attacks in cybersecurity with a look at how the usage of AI can be leveraged in this aspect in most of these papers.

The open questions left by the gathered literature that apply to the context of the dissertation are to be considered in its development since it will investigate how the four aspects can complement each other in the hopes of making a great contribution to research on its topic, and these open questions can provide a path on hypothesis for design, implementation, and experimentations to conduct.

2.3 Existing technologies and patterns

This section presents what technologies currently exist and which patterns are widely used in microservices when it comes to making systems more secure with AI, and resilient.

2.3.1 Technologies

This section describes some of the technologies that are widely used for security (and more) in microservices that provide rich AI feature sets for anomaly detection and possibly prevention.

2.3.1.1 Amazon OpenSearch

Amazon OpenSearch is an Amazon Web Services (AWS) service that allows to perform interactive log analytics, real-time application monitoring, and more, for applications hosted on AWS [40], as it's built on top of OpenSearch, an open-source distributed search and analytics suite derived from Elasticsearch [41].

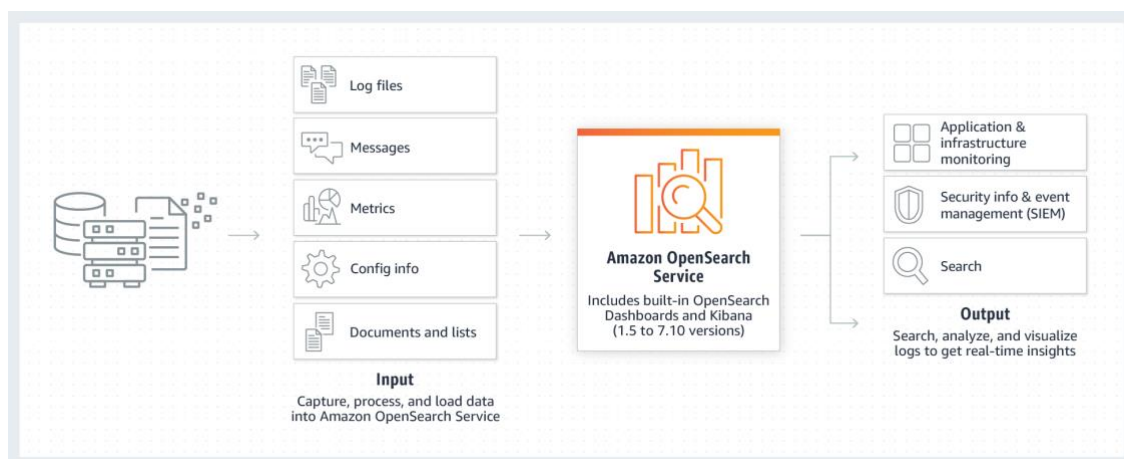


Figure 4 - Amazon OpenSearch workflow [40]

More importantly, it provides ML features for real-time anomaly detection to proactively detect anomalies in real-time streaming data. It can detect anomalies such as unusually high error rates or sudden changes in the number of requests. This feature uses the Random Cut Forest algorithm. It's an unsupervised algorithm that constructs decision trees from numeric input data points in order to detect outliers (i.e., anomalies) [42].

2.3.1.2 Google Dataflow

Google Dataflow is a Google Cloud service that provides a unified stream and batch data processing at scale. It can create data pipelines that read from one or more sources to transform and write data to a destination. It has multiple use cases such as data movement (i.e., data ingestion and replication across subsystems), extract-transform-load (ETL) workflows, and most importantly, applying ML in real-time streaming data [43].

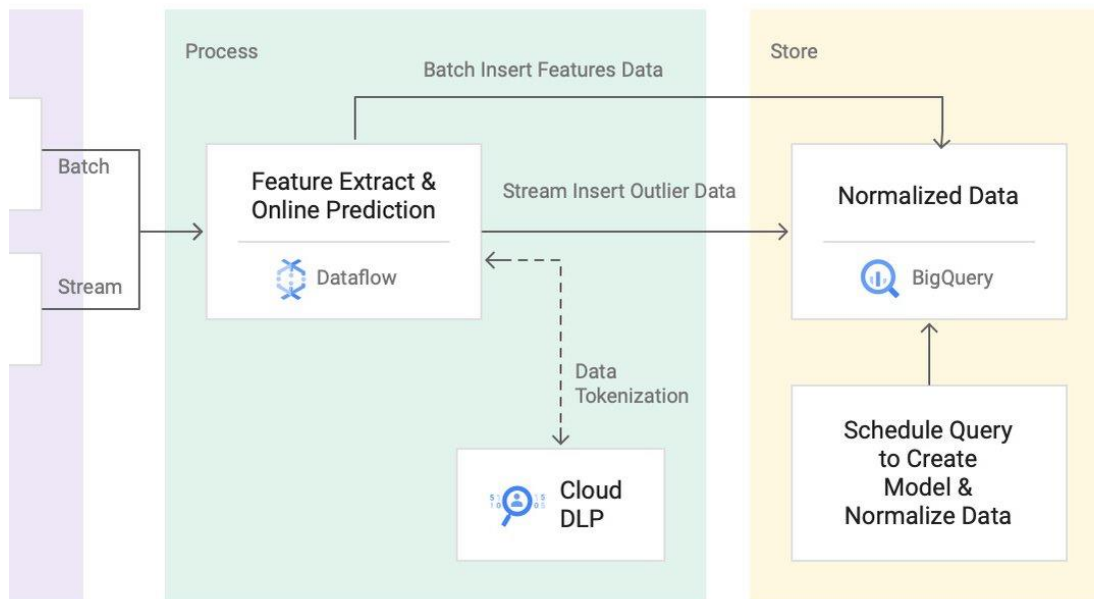


Figure 5 – Example architecture of a real-time anomaly detection solution in Google Cloud [44]

Dataflow allows for anomaly detection by building streaming analytics pipelines that detect anomalies by analyzing and extracting features from real-time logs [44].

2.3.1.3 Elastic Machine Learning

Elastic offers ML features that work seamlessly with its stack. These allow us to analyze and generate models for the collected data's behavior patterns. It contains two categories of ML with two types each as presented in Table 8.

Table 8 - Types of Machine Learning in Elastic [45]

Category	Type	Description
Unsupervised	Anomaly detection	Requires time series data as it builds a probability model that is run continuously to identify unusual events as they occur allowing it to evolve over time proving insights for forecasting future behavior
	Outlier detection	Performs data frame analytics that identifies unusual points in datasets by analyzing the proximity and density of datapoints. It does not run continuously and generates a copy of the dataset that has each data point annotated with an outlier score that indicates the outlier extent of the data point compared to others
Supervised	Classification	Performs data frame analytics that learns relationships between data points to predict discrete categorical values such as if requests have a malicious domain or not

Category	Type	Description
	Regression	Performs data frame analytics that learns relationships between data points to predict continuous numerical values such as response times for requests

These ML features are simple to configure and start since the Elastic Stack provides an easy-to-use UI and plenty of configuration options for each of the types above through Kibana.

2.3.2 Patterns

This section describes what patterns are most commonly used in terms of resilience in microservices.

2.3.2.1 Circuit breaker

Circuit breaker, as its name suggests, is a pattern designed to stop interacting with a service if it's not responding. It provides stability while the system is recovering from failure by quickly rejecting a request for an operation that's likely to fail rather than waiting for it to timeout or never return. In the case of distributed systems, which is the case with microservices, it also prevents the cascade of failures across the systems [46]. It is comprised of three different states, as shown in Figure 6.

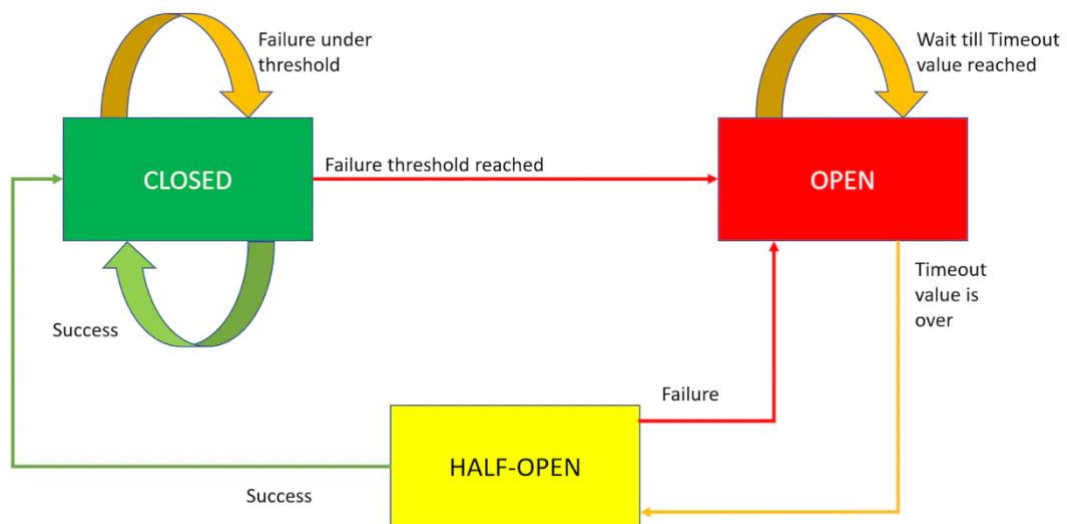


Figure 6 - Circuit breaker states [47]

From what can be observed, the three states are [46]:

- **Closed:** the service receives requests and either succeeds or fails a number of times before overcoming its threshold;

- **Open:** the circuit breaker opens when the failure count is greater than its threshold and the service will not receive requests while it's in timeout;
- **Half-Open:** after the timeout, the circuit breaker will enter this state where it will send a request to the service and depending on the response will either go to the open or closed state where it will go through timeout again.

2.3.2.2 Timeout and retry

Timeout and retry is a pattern that is employed as a resiliency mechanism by setting a specific timeout for a service's operations. If an operation times out then it's considered a failure, and with the retry logic then it can reattempt a certain number of times before stopping being deemed an error [48].

2.3.2.3 Rate limiter

Rate limiter is a throttling pattern to control the consumption of resources and protect services from excessive load by imposing limits on the rate at which services can be accessed (i.e., incoming requests). This ensures that services remain stable, responsive, and available to users under varying load conditions. There are three common rate limiting strategies used in microservices [49]:

- **Fixed window:** a fixed number of requests are allowed within a specific time window. Once this limit is reached it starts rejecting requests until the next time window;
- **Sliding window:** known as the token bucket algorithm, it allows by continuously refilling a bucket of tokens that represent the number of requests during a time period. A token is consumed by request and if the bucket is empty then it's rejected. It allows for more flexible handling of varying traffic conditions;
- **Leaky bucket:** similar to sliding window, this algorithm imposes rate limits by emptying the bucket at a fixed rate. Incoming requests are added to a bucket and if it overflows, the requests are rejected. It enforces a consistent processing pace.

2.3.2.4 Health check

Health check is a design pattern that detects if a service is available (i.e., able to handle requests) by periodically pinging a specific endpoint (e.g., HTTP /health) that returns the health of the service [50].

2.3.2.5 Fallback

Fallback is a design pattern that allows services to continue the execution of requests in the scenario of a failed request to another service by defining a structured behavior to be executed upon failure, such as cached data, default values, or a user-friendly error message [51].

2.3.2.6 Bulkhead

Bulkhead is an applicational design pattern that is failure tolerant that derives from compartmentalization and it achieves this isolation by segregating resources to maintain stability and availability. There are two types of bulkhead isolation [52]:

- **Resource-level isolation:** manages the allocation of resources such as threads and connection pools across different services ensuring that these do not affect each other;
- **Process-level isolation:** segregates services into different processes or containers ensuring that if a service goes down the other continue to function without being impacted.

2.4 Summary

This chapter presented and described:

- How the research and collection of current and recent publications has evolved over the recent years, and described some of these by category to highlight the work they performed and what open points they left to be answered;
- The comparison of the collected research by categories and what conclusions can be drawn from these;
- What technologies currently exist in making systems more secure with AI features and what are the most commonly used patterns in turning microservices more resilient.

3 Analysis

This chapter presents an analysis of the type of business chosen to prototype/emulate a critical system – in this case, a simple banking system. It starts by describing its main concepts accompanied by the domain model and lastly, presents the main processes and stakeholders.

3.1 Concepts and activities

In this section, the business concepts are described by context while accompanied by their respective portion of the domain model with the complete domain model being presented at the end. It's important to mention that domain was designed according to Domain-Driven Design (DDD) concepts to maintain the business logic as simple as possible as to not make the software development process more complex than it could be. This is important since one of the security problems associated with critical systems is their highly complex domain and when the software grows, its complexity tends to grow as well leading to non-ideal conditions when it comes to tasks such as maintenance and development [53]. The first applied concept of DDD that is mentioned is the usage of contexts (i.e., bounded contexts). Bounded contexts are a central pattern in DDD that represent boundaries in which certain subdomains are defined and applicable. When subdomains change within bounded contexts, the entire system doesn't have to change as well [54].

“The domain model is the organized and structured knowledge of the problem. The domain model should represent the vocabulary and key concepts of the problem domain and it should identify the relationships among all of the entities within the scope of the domain” [55].

3.1.1 Core context

The core context of the business is composed of users (i.e., clients) who possess at least one bank account that can execute transactions to either another bank account or a utility account.

A user is composed of a (i) unique identifier (i.e., Guid/Uuid), a (ii) first name, (iii) last name, (iv) email address, and an (v) identification number. The identification number is the user’s official identification (e.g., citizen number) It possesses at least one bank account.

A bank account is composed of a (i) unique identifier, the (ii) user’s unique identifier, the (iii) account number, its (iv) available balance, and (v) actual balance, its (vi) status, (vii) type, and executed (viii) transactions. The account status indicates its status as one of three: active, pending, or blocked. The account type indicates its type: savings, fixed, or loan. A valid bank account needs to have a user associated, a type, and a status.

A transaction is composed of a (i) unique identifier, a (ii) bank account’s unique identifier, the (iii) transacted amount, a (iv) reference number, and a (v) transaction type. The transaction type indicates if the transaction was a fund transfer between accounts or a utility payment to a provider (i.e., utility account). The bank account’s unique identifier indicates which bank account is associated with the transaction (i.e., the transaction associated with the sender account has its unique identifier, and the one associated with the recipient account has its unique identifier). The amount is how much is sent from the sender’s account available balance. The reference number depends on the transaction type: in the case of a fund transfer, it indicates on both accounts (i.e., the transaction associated with each account) the recipient account’s number, and in the case of a utility payment its value is inputted by the user.

A utility account is composed of a (i) unique identifier, the (ii) provider’s name, and the (iii) account’s number. It has no associations.

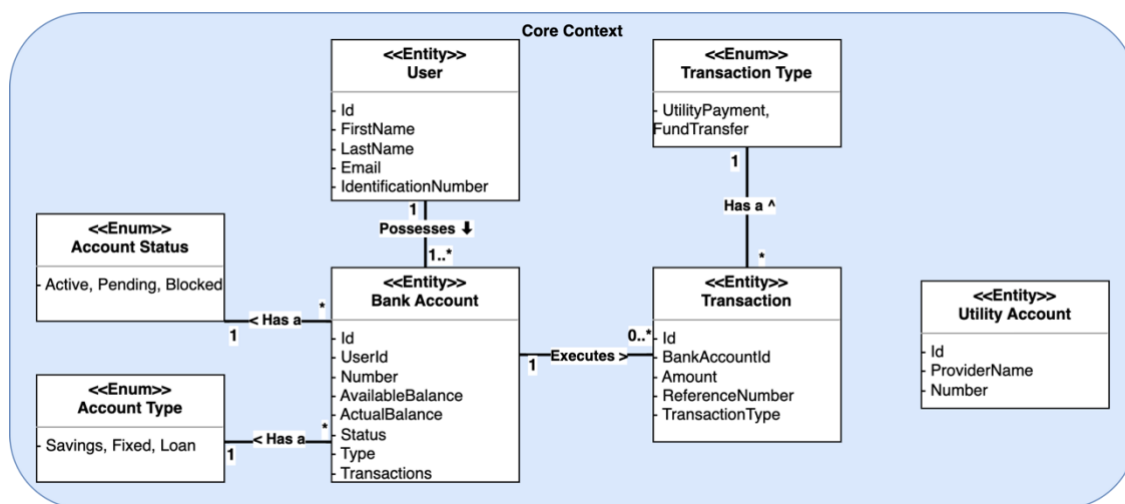


Figure 7 - Core context portion of the domain model

3.1.2 User context

The user context is composed of users that are also present in the core context (i.e., have a bank account) and can interact with the banking system (i.e., register, log in, execute transactions, etc.).

A user is composed of a (i) unique identifier, an (ii) email address, a (iii) identification number, a (iv) password salt, the (v) hashed password, and a (vi) status. The password salt is used to further encrypt the user's already hashed password. The hashed password is the user's password already encrypted to not be visible to those who can consult the database where it is stored. The user status indicates its account status: pending, approved, disabled, or blacklist. It references the user in the core context through the identification number which means that a user can only register if they are presented in the core context (i.e., a client of the bank).

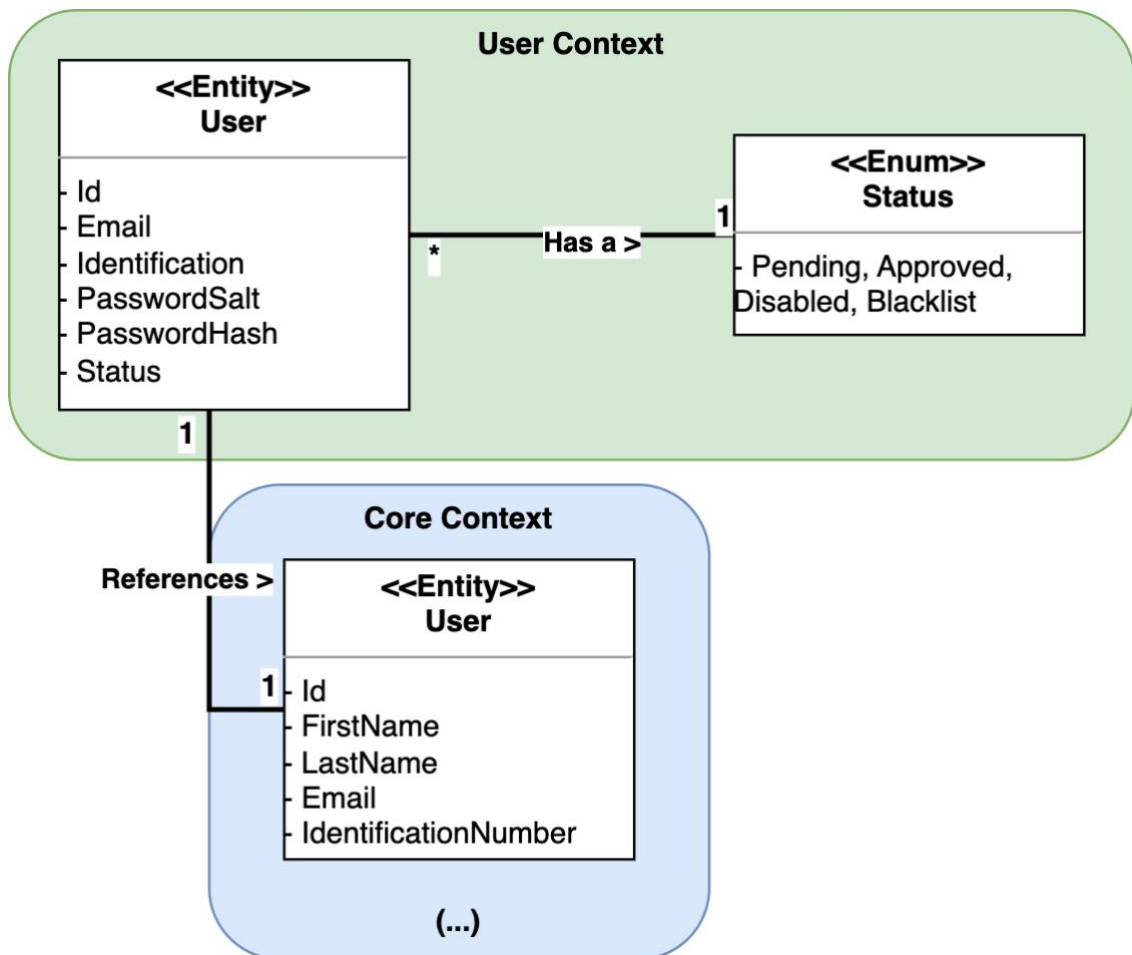


Figure 8 - User context portion of the domain model

3.1.3 Transfer context

The transfer context is composed of fund transfers that are requested by users to execute transactions between bank accounts.

A fund transfer is composed of a (i) unique identifier, the (ii) sender’s account number, the (iii) recipient’s account number, the (iv) transaction amount, the (v) transaction status, and the (vi) transaction reference. The sender’s account number and recipient’s account number reference the users’ accounts in the core context. The transaction status is shared with the payment context and indicates the transfers status: pending, processing, success, or failed. The transaction reference references the transaction’s unique identifier from the core context.

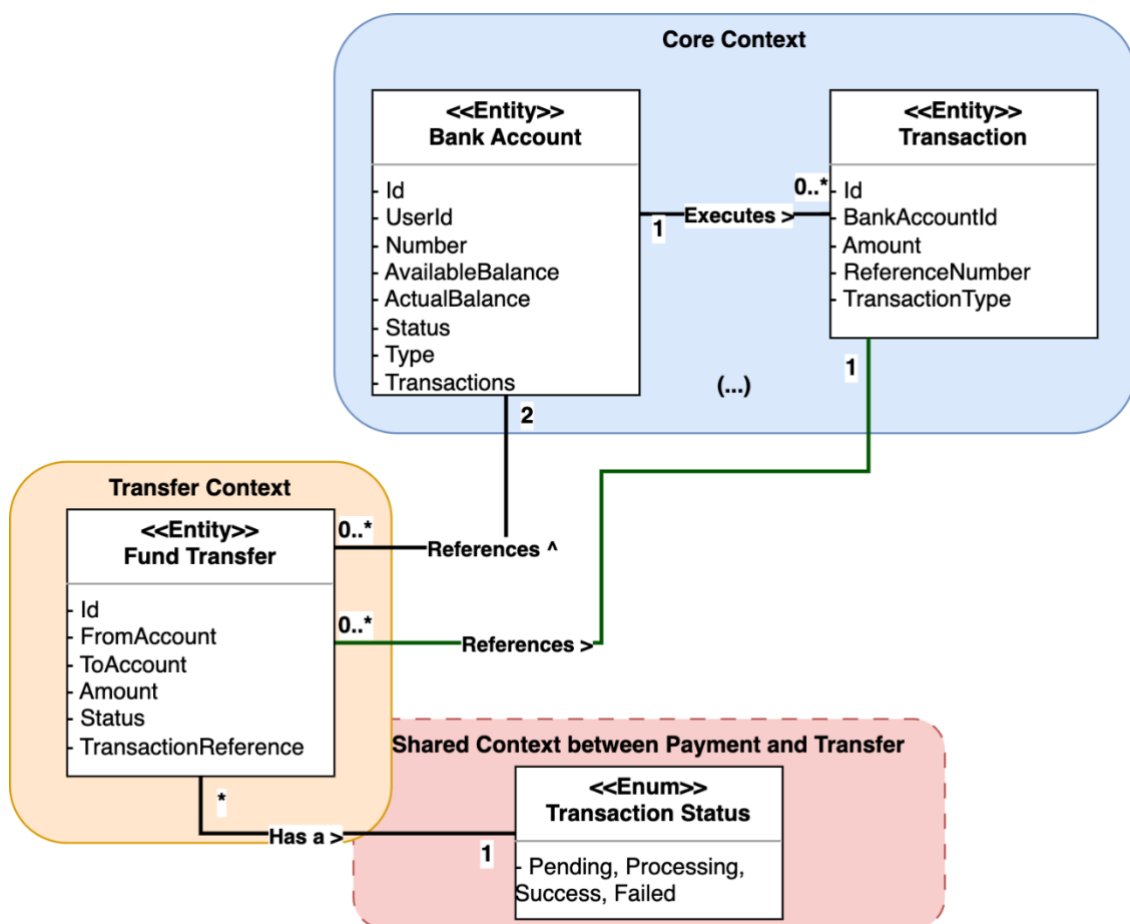


Figure 9 - Transfer context portion of the domain model

3.1.4 Payment context

The payment context is composed of utility payments that are requested by users to perform payments from their accounts to the providers' accounts (i.e., utility accounts).

A utility payment is composed of a (i) unique identifier, the (ii) provider account identifier, the (iii) bank account number, the (iv) transaction amount, the (v) transaction status, the (vi) transaction identifier, and the (vii) reference number. The provider account identifier, bank account number, and transaction identifier reference the utility account, bank account, and transaction in the core context. The transaction status is shared with the transfer context and indicates the payment's status: pending, processing, success, or failed. The reference number is provided by the user's request.

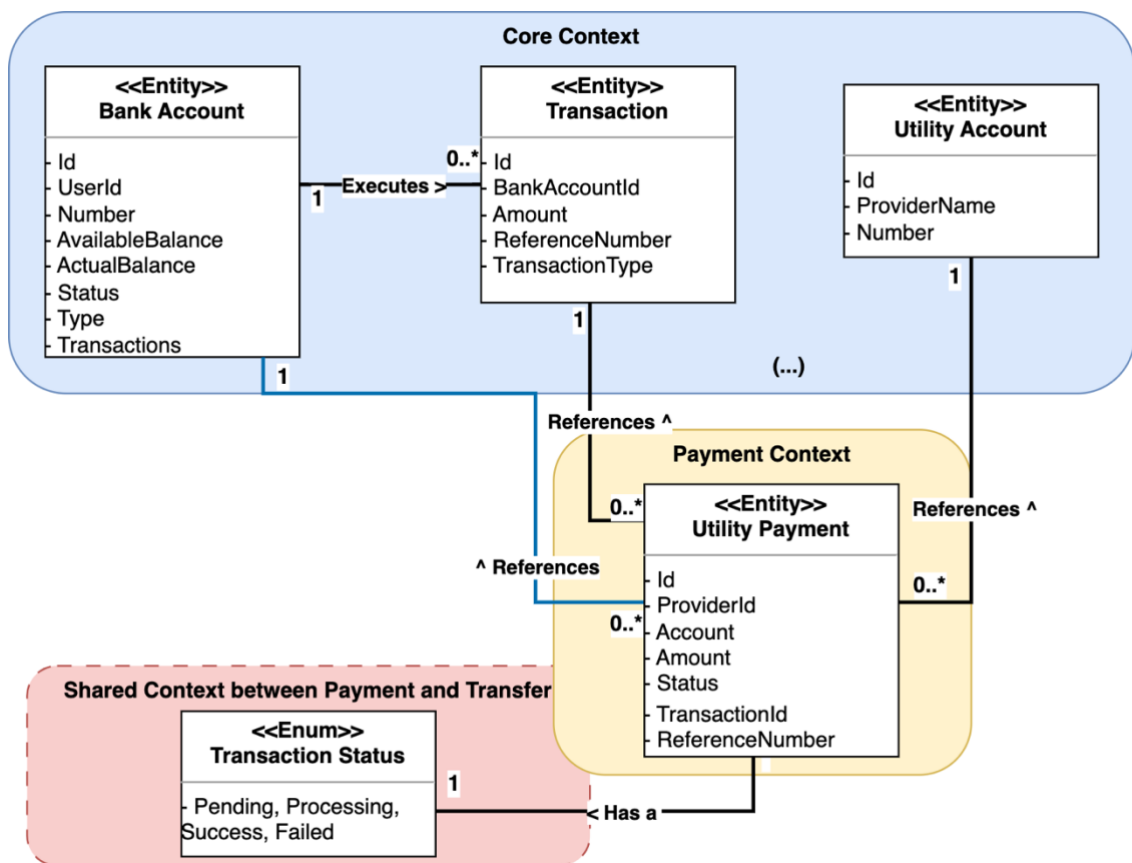


Figure 10 - Payment context portion of the domain model

3.1.5 Domain model

The complete domain model encompassing all previously described contexts can be observed below (cf. Figure 11).

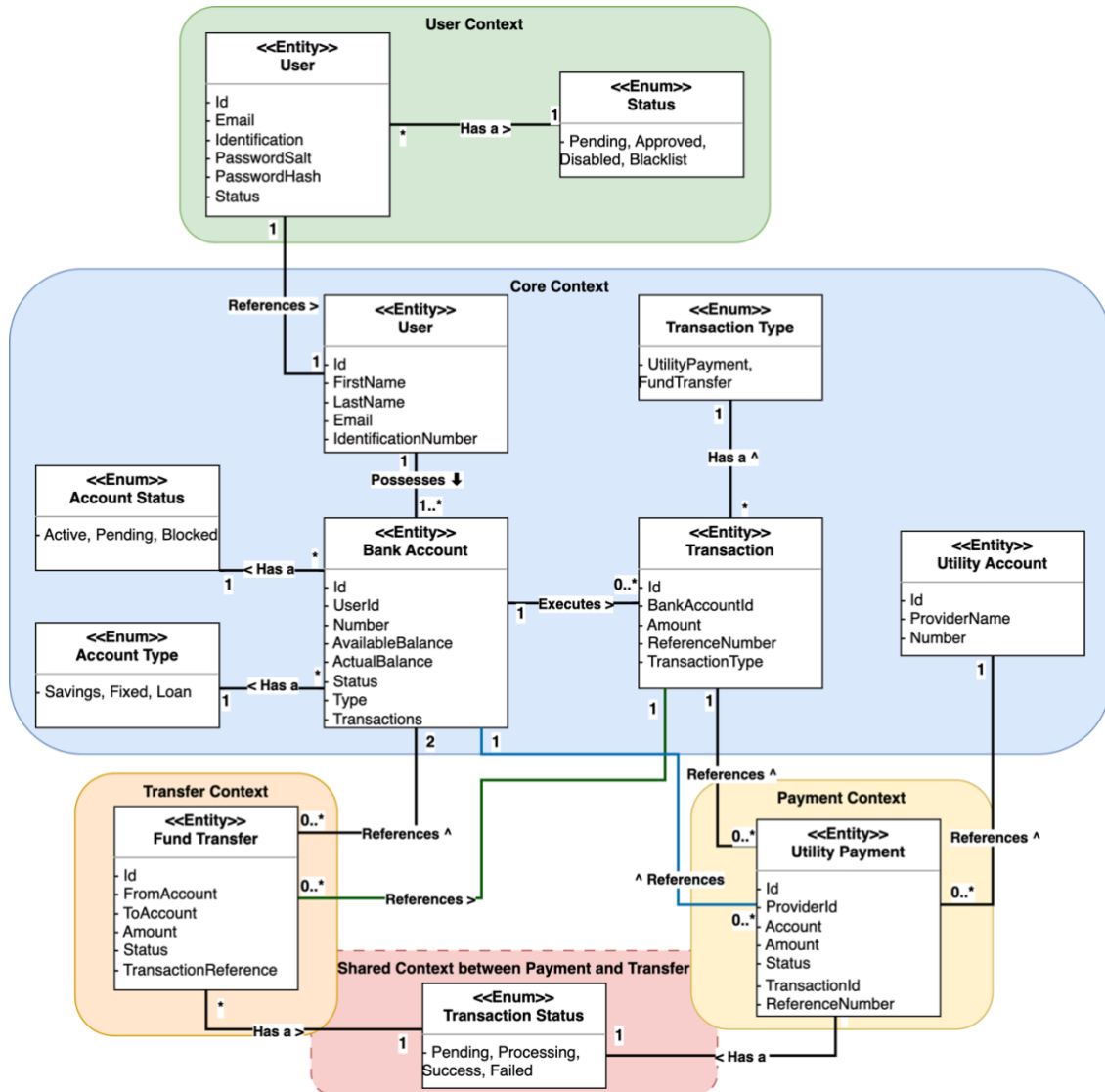


Figure 11 - Domain model

3.2 Processes and stakeholders

Since the business intends to allow users to register, log in, and execute transactions, that involves a set of processes to follow:

1. User and bank account creation: It involves the user creating a client profile and their respective bank account(s)

2. Utility account creation: the utility provider (e.g., water supply, electrical supply, etc.) can open a utility account to receive payments from their customers (i.e., users);
3. User registration: the user can register to be able to log into the banking system;
4. User login: the user can log into the banking system and request transactions;
5. Execute transfer between bank accounts: the user can request fund transfers between bank accounts;
6. Execute payment from a bank account to a utility account: the user can request a payment from their account(s) to a utility provider.

It is noted that processes 1 and 2 are not handled by the system but are relevant to the overall comprehension of the business.

The users (i.e., stakeholders) of the banking system are the bank's clients (i.e., private clients – users – and business clients – utility providers).

The following activity diagrams (cf. Figure 12, Figure 13, Figure 14, Figure 15) present a visual description of the processes above.

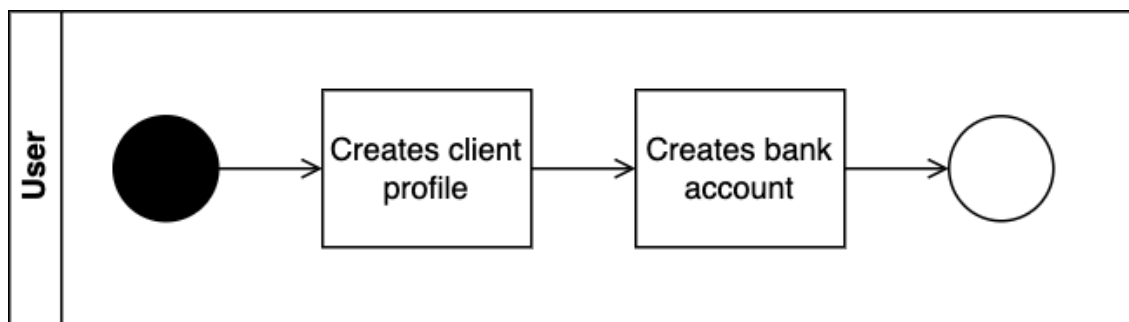


Figure 12 - Activity diagram describing process 1

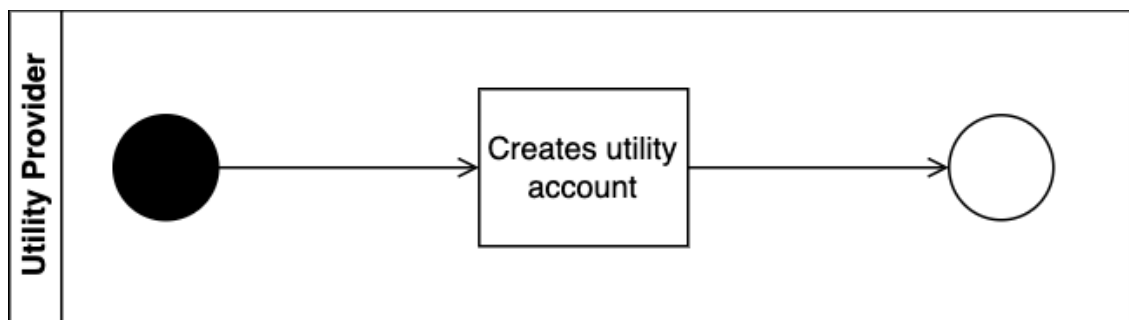


Figure 13 - Activity diagram describing process 2

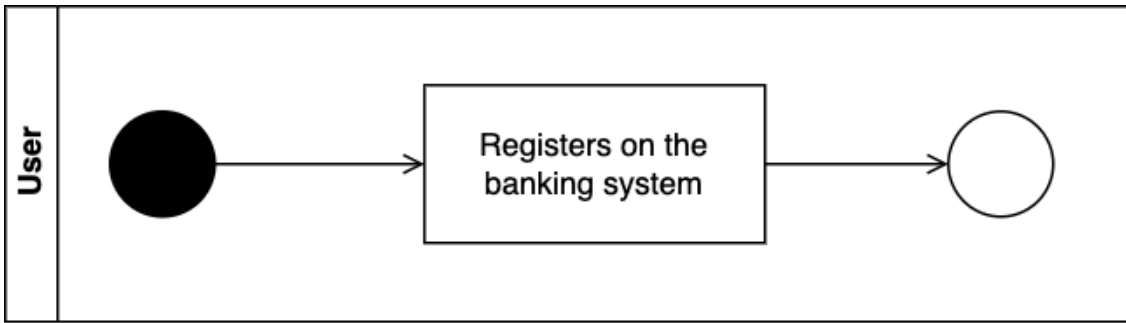


Figure 14 - Activity diagram describing process 3

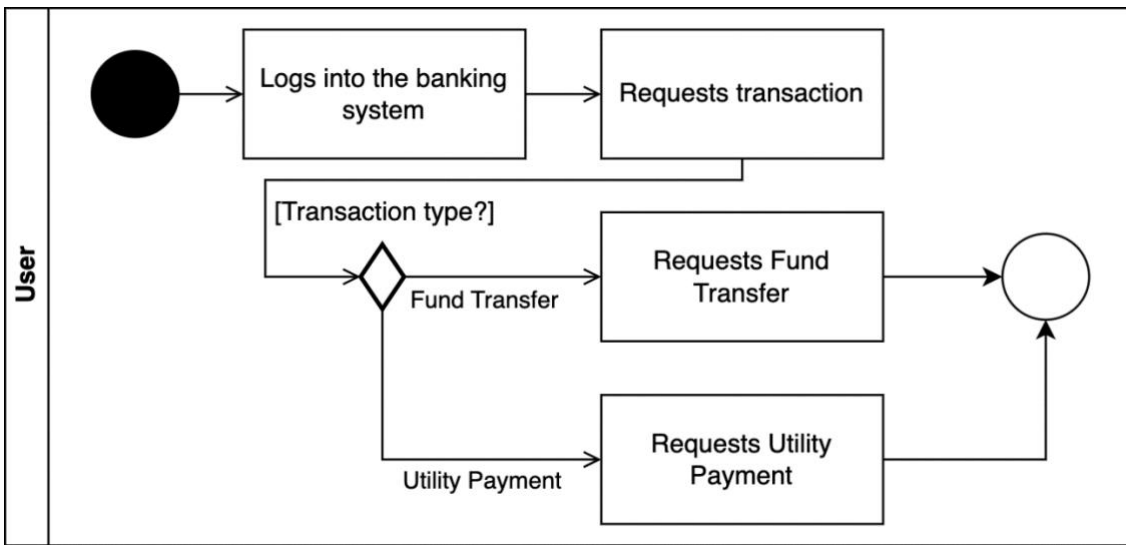


Figure 15 - Activity diagram describing processes 4 through 6

3.3 Use cases

With the information from the previous section, the use cases for the system are identified and presented in the use case diagram below (cf. Figure 16).

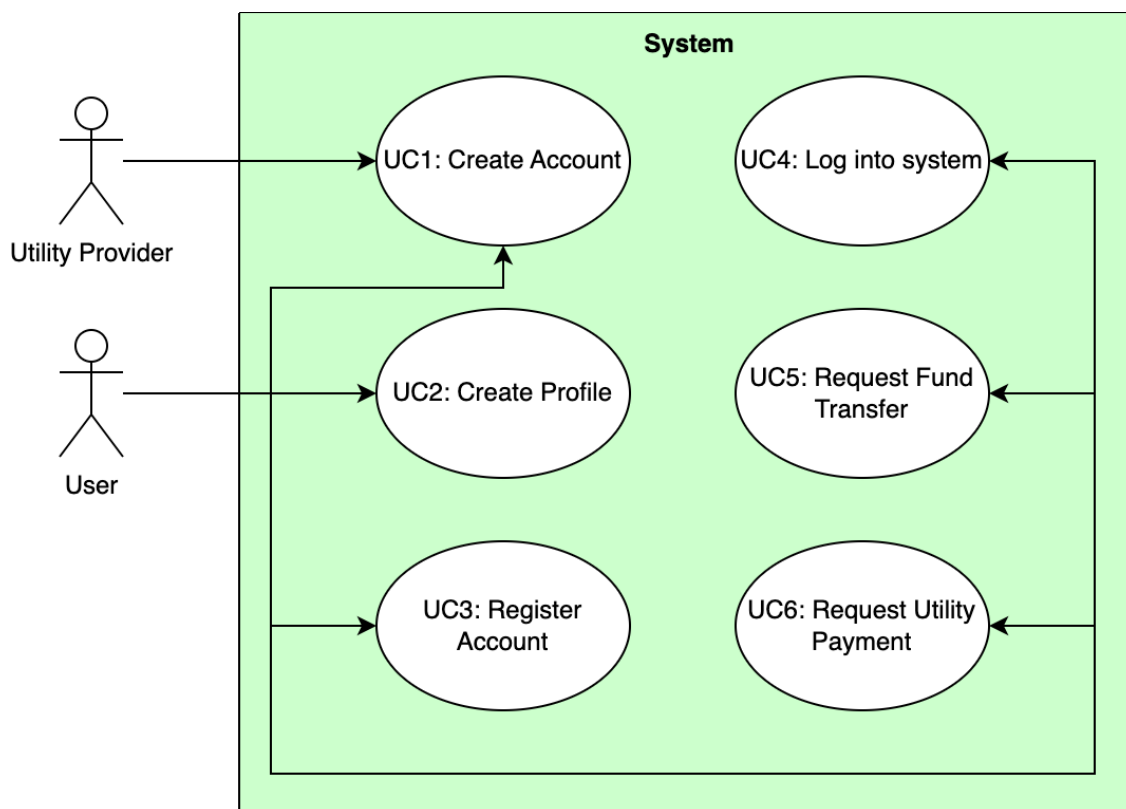


Figure 16 - Use case diagram

By analyzing it, it can be observed that the use cases are a direct translation of the business processes. As noted in the previous section, UC1 and UC2 – that address processes 1 and 2 – are not processed by the system.

3.4 Summary

This chapter presented and described:

- What the concepts and activities of the business are by context with a description of these accompanied with portions of the domain model, ending on the complete domain model;
- What are the processes and stakeholders of the business accompanied by activity diagrams for a visual representation of these processes and their respective stakeholders;
- What are the use cases of the business with a visual representation by use case diagram.

4 Design

This chapter presents the most important decisions on the design of the solution. It starts by approaching the design on a high level of abstraction and then goes to the succeeding lower levels, more detailed, where it presents each component of the solution in a more specific context finishing on the database design.

Knowing that the solution must emulate a critical system (cf. section 1.3), it's important to adhere to the Secure-by-Design approach for it be as secure as possible [56] and its methods applications are mentioned in this chapter and in the following ones as well.

The designed solution imitates a banking system's core features such as user management and transactions such as funds transfers and utility payments. The solution is composed of four microservices: (1) a core service, (2) a user service, (3) a transfer service, and (4) a payment service.

A description of the system's architecture is now presented. Its description follows the C4 model, which suggests four levels of granularity [57]:

1. **Context:** The coarser-grained level (i.e., the highest possible abstraction) where the system's functions and external services are described;
2. **Containers:** Describes the independent parts – the containers – that compose the system;
3. **Components:** Describes the internal components of the containers;
4. **Code:** The finer-grained level where it presents the components' details.

Different views are adopted as needed in each level's description inspired by the 4+1 model that proposes the following [58]:

1. **Logical view:** Presents information about the various parts of the system (i.e., its design);
2. **Process view:** Describes the behavior of the system's processes (i.e., the system's behaviors);
3. **Physical view:** Presents the system's physical implantation (i.e., where it's implanted, how many nodes it occupies, and what each node contains);

4. **Implementation view:** Focuses on the organization of the software modules;
5. **Use case view:** Describes the features of the system from a user perspective.

4.1 Level 1: Context

The following component diagram (cf. Figure 17) presents the designed system and its context, namely the available interfaces and the systems that it depends on and the systems that depend on it.

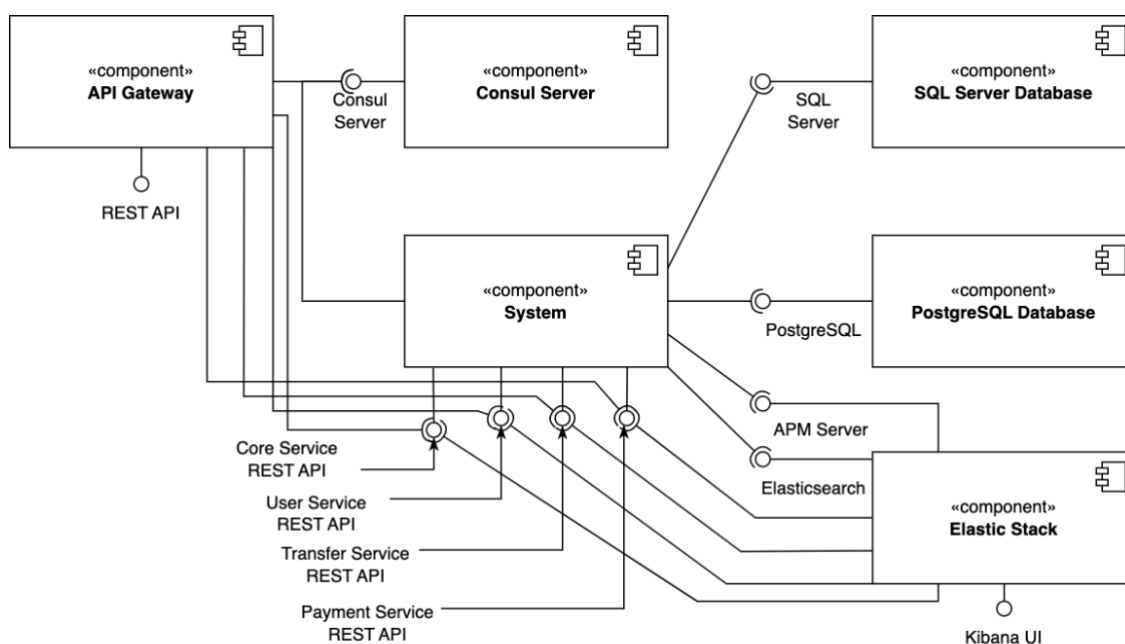


Figure 17 - Logical view of the context level of the system

Starting from the left, the point of entry to the system is through the REST API made available by the API Gateway. The API Gateway consumes the Consul Server and each of the REST APIs of the microservices from the System. The Consul Server provides a Service Discovery service for the System's microservices and the API Gateway, and it is through it that the microservices communicate with each other and how the API Gateway communicates with the microservices. The system consumes three external services besides the API Gateway: SQL Server Database, PostgreSQL Database, and the Elastic Stack. The SQL Server Database and PostgreSQL Database are where the data is persisted. The Elastic Stack is where the microservices logs and traces are stored via Elasticsearch and APM Server. The Elastic Stack also consumes the microservices REST API for health checks and has a Kibana UI available where logging, tracing, metrics and other relevant information can be consulted.

4.2 Level 2: Containers

The following component diagram (cf. Figure 18) presents the system's internal architecture where we can observe the microservices (i.e., the containers) that compose the system and how these interact. The DDD concepts of subdomain and bounded context can be observed here since each component of the system represents a subdomain and bounded context previously described in section 3.1.

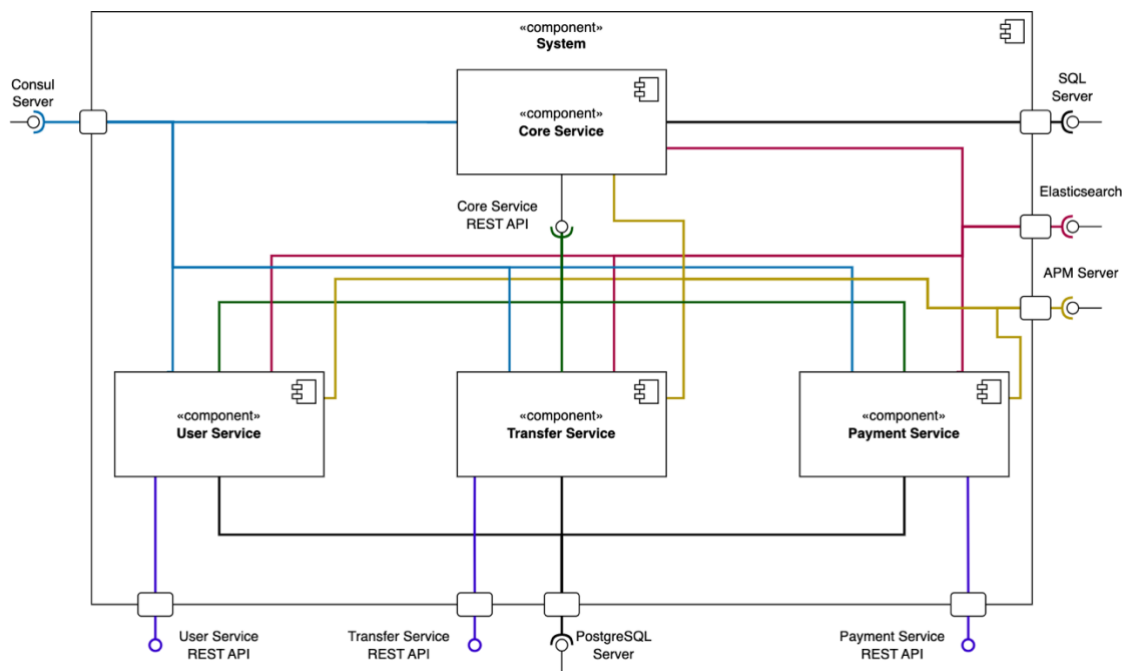


Figure 18 - Logical view of the container level of the system

As observed:

- The Core Service component makes its REST API available internally for the other components. This component is responsible for managing and manipulating all the business components, users, bank accounts, utility accounts, and transactions. It follows the Interface Segregation Principle (ISP) from the SOLID principles [59] where these features were separated into smaller cohesive interfaces increasing its maintainability and adoption. This component consumes the SQL Server to read and write data;
- The User Service component makes its REST API available externally. This component is responsible for fetching user information and registering and logging users to the system. It consumes the Core Service REST API which will validate user information when registering;
- The Transfer Service component makes its REST API available externally. This component is responsible for fetching transfer information and registering new transfer transactions. It consumes the Core Service REST API which will validate the bank accounts and the sender account's funds as well as register the new transaction if the information is valid when registering new transfers;

- The Payment Service component makes its REST API available externally. This component is responsible for fetching payments and registering new payment transactions. It consumes the Core Service REST API which will validate the bank account, its funds, and the utility account as well as register the new transaction if the information is valid when registering new payments;
- All components except the Core Service consume the PostgreSQL Server where they read and write data;
- All components consume the Consul Server where they register themselves and obtain the service registry when sending requests to other components;
- All components consume the Elasticsearch and APM Server where they send their logs and traces;
- Following the REST architectural style [60], the client-server architecture was adopted between the presented REST APIs (i.e., the servers) and the components that consume them (i.e., clients) as evidenced in the previous bullet points.

The following component diagram (cf. Figure 19) presents the Elastic Stack's internal architecture where we can observe the services (i.e., the containers) that compose it and how these interact.

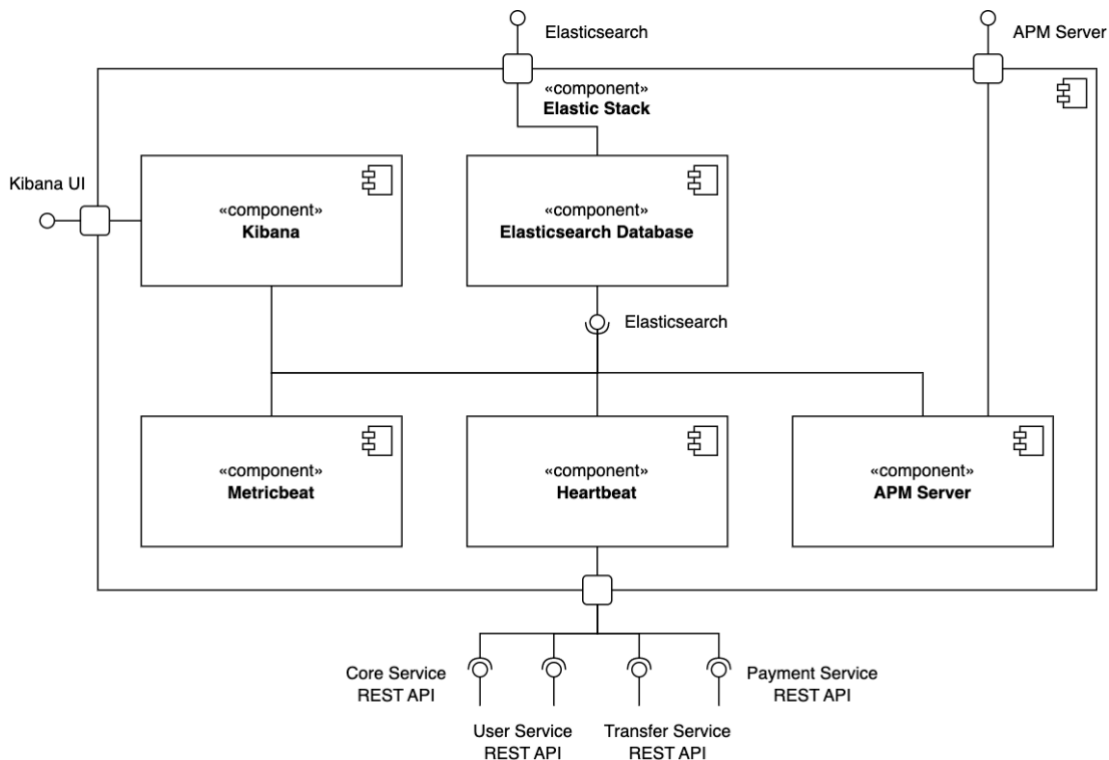


Figure 19 - Logical view of the container level of the elastic stack

As observed:

- The Elasticsearch Database component is available for internal and external consumption. It's responsible for storing data regarding the system's logs, traces, and metrics;

- The APM Server component is available for external consumption. It's responsible for collecting tracing data that is sent to it. It consumes Elasticsearch where it writes the tracing data in its indexes;
- The Heartbeat component is responsible for performing health checks. It consumes the microservices REST API where it pings them in a specific path to verify their health and sends that data to the Elasticsearch Database;
- The Metricbeat component is responsible for collecting the systems metrics. These are metrics related to CPU Usage, RAM Usage, Inbound traffic, and Outbound traffic. It collects this data from the infrastructure where the system is implanted and then stores it in the Elasticsearch Database;
- The Kibana component makes its UI available externally. It is where we can consult all the collected logs, traces, and metrics from the system via dashboards and other graphics.

The Elastic Stack is important since it allows the system to adhere to the monitoring and logging method of the Secure-by-Design approach since these are necessary for detecting abnormal patterns and log auditing of the system [56].

The following node diagram (cf. Figure 20) presents the systems containers implantation and the used communication protocols (i.e., physical view).

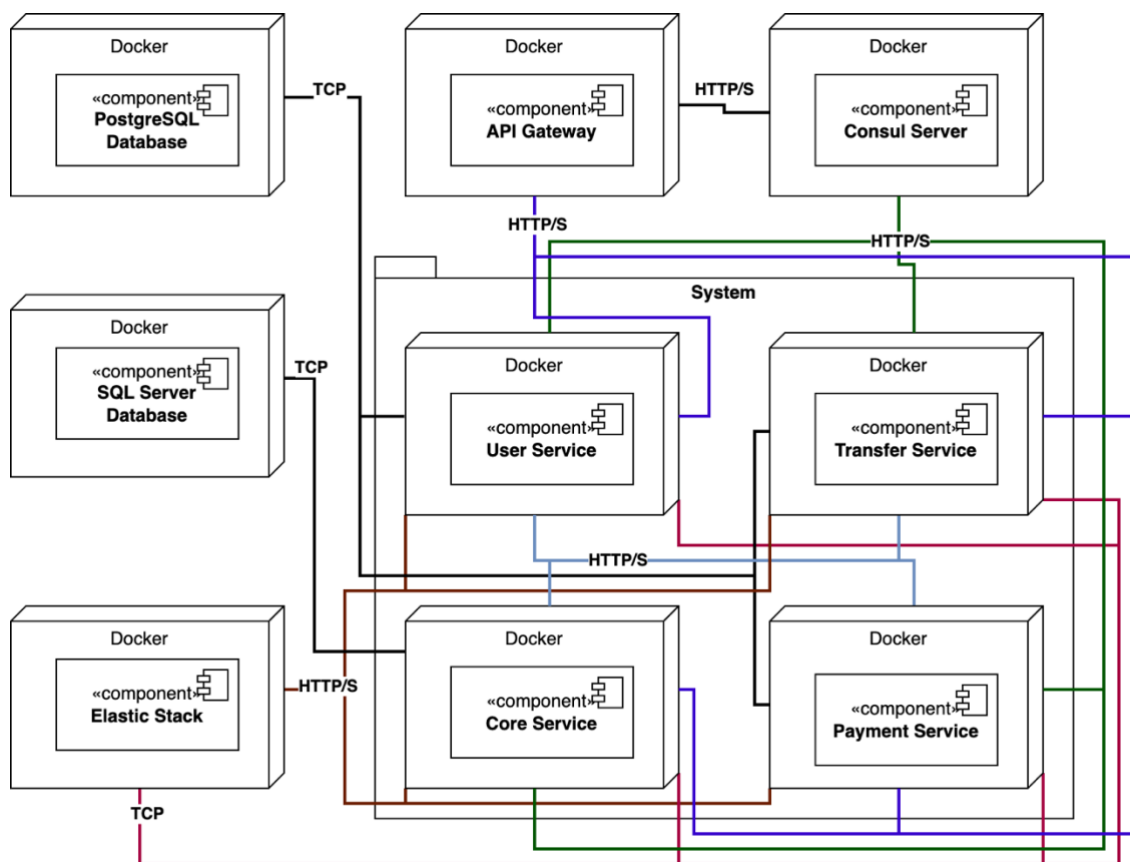


Figure 20 - Physical view of the container level of the system

The systems containers are implanted in Docker containers [61]. Docker is a technology that enables the creation, implantation, and execution of applications through the employment of containers (i.e., isolated environments). With the creation of a container, an application can run in an isolated manner on any machine with the same operating system [62].

The containers are running on Linux virtual machines on the student’s laptop. These share the following resources:

- 6 logical CPUs, each correspondent to an Intel i7 8750H CPU core thread;
- 8 gigabytes (GB) of memory;
- 75 GB of storage.

Most interactions are executed via HTTP/S except for the databases and some of the Elastic Stack’s containers where these are executed via TCP.

In the following sequence diagram (cf. Figure 21) the information flow between containers regarding the execution of a fund transfer between bank accounts can be observed. This flow can be described as a happy path and the flow with all possible paths/outcomes can be consulted in Annex B (cf. Figure 80).

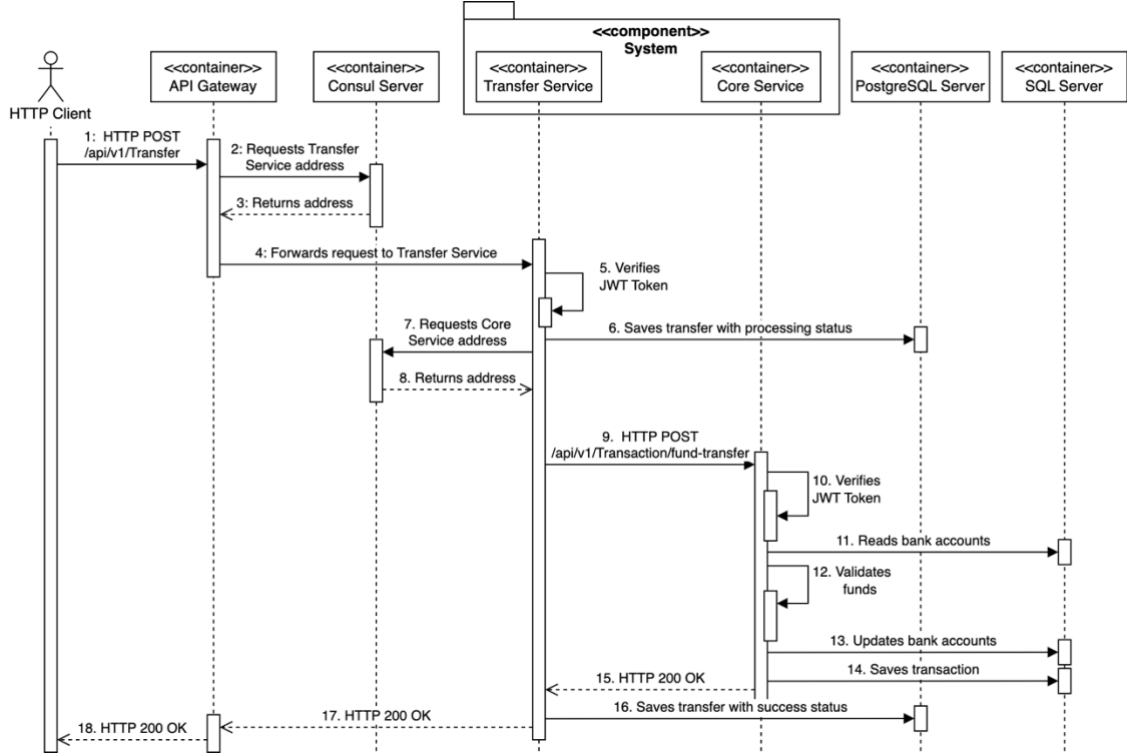


Figure 21 - Process view of the container level of the system regarding the happy path of execution of a fund transfer

To explain:

1. The HTTP Client sends an HTTP POST request to the `/api/v1/Transfer` endpoint with the necessary information;
2. The API Gateway requests the Transfer Service's address to the Consul Server's Service Registry;
3. The Consul Server returns the address;
4. The API Gateway then forwards the request to the Transfer Service;
5. The Transfer Service will validate the sent JWT Token;
6. The Transfer Service will save the requested transfer with the processing status in the PostgreSQL Server database;
7. The Transfer Service requests the Core Service's address to the Consul Server's Service Registry;
8. The Consul Server returns the address;
9. The Transfer Service sends an HTTP POST request to the Core Service to execute the fund transfer with the necessary information;
10. The Core Service validates the sent JWT Token;
11. The Core Service queries the SQL Server database for the given bank accounts;
12. The Core Service validates if the sender's bank account has the necessary funds to execute the transfer;
13. The Core Service will then update the bank accounts' balance with the given amount;
14. The Core Service registers the transaction in the SQL Server database;
15. The Core Service responds to the Transfer Service with an HTTP 200 OK response with the transaction number and a success message;
16. The Transfer Service takes the received transaction number, adds it to the transfer as a reference, and updates it with success status in the PostgreSQL Server database;
17. The Transfer Service responds to the API Gateway with an HTTP 200 OK response with the transaction number and success message;
18. The API Gateway forwards the response to the HTTP Client.

The Elastic Stacks containers are not represented in the diagram above due to the complexity and size it would introduce but it can be assumed that all actions (i.e., traces) and logs are sent to the APM Server and Elasticsearch database.

4.3 Level 3: Components

This section presents the architecture of each of the system's containers. These are presented according to the C4 model, which means the internal components and how these relate.

4.3.1 Core Service

This container is the most complex in terms of responsibility as it's seen as the source of truth for the system's features that are executed by the other containers (i.e., user, bank account, and utility account information).

The following component diagram (c.f. Figure 22) presents the internal components of the Core Service and how these interact internally and externally.

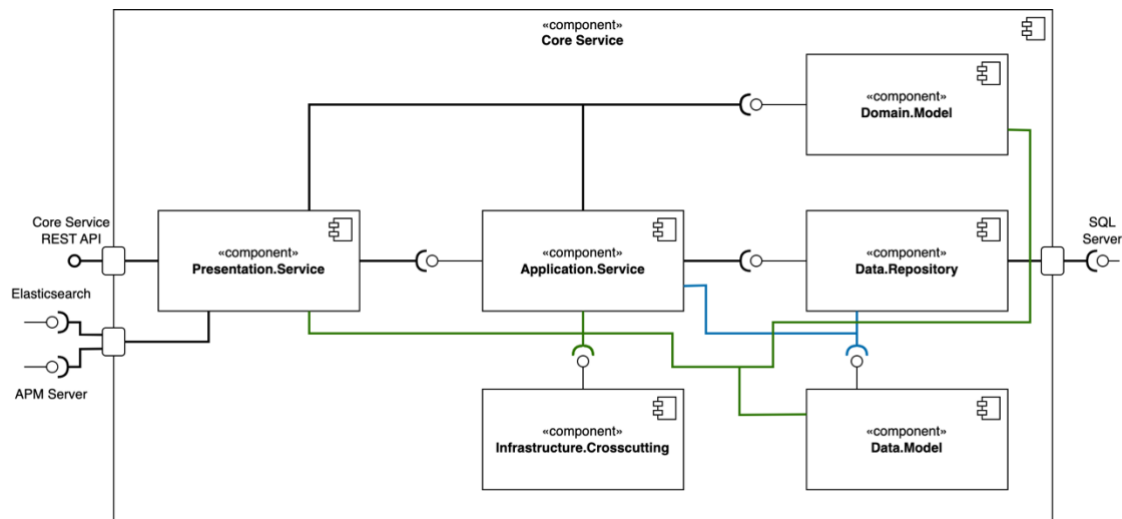


Figure 22 - Logical view of the component level of the Core Service container

Analyzing the diagram, the requests are received by the Presentation Service component that contains the Controllers for each entity (i.e., user, accounts, and transactions) that will coordinate the responses through the Application Service component. These are responsible for interacting with the domain. This is done through the Data Repository component to access the persisted data and Mappers – contained by the Application Service component – are employed as intermediaries to map data between the Data Model and Domain Model components that are called by Services – contained by the Application Service component. The Infrastructure Crosscutting component contains logic that is transversal to all components such as exceptions, guards, generic interfaces, and configurations. The Presentation Service component also consumes the external services Elasticsearch and APM Server to send its logs and traces, respectively.

To better understand the functioning of the architectural design, the following sequence diagram (cf. Figure 23) presents the information flow regarding the fetching of user information.

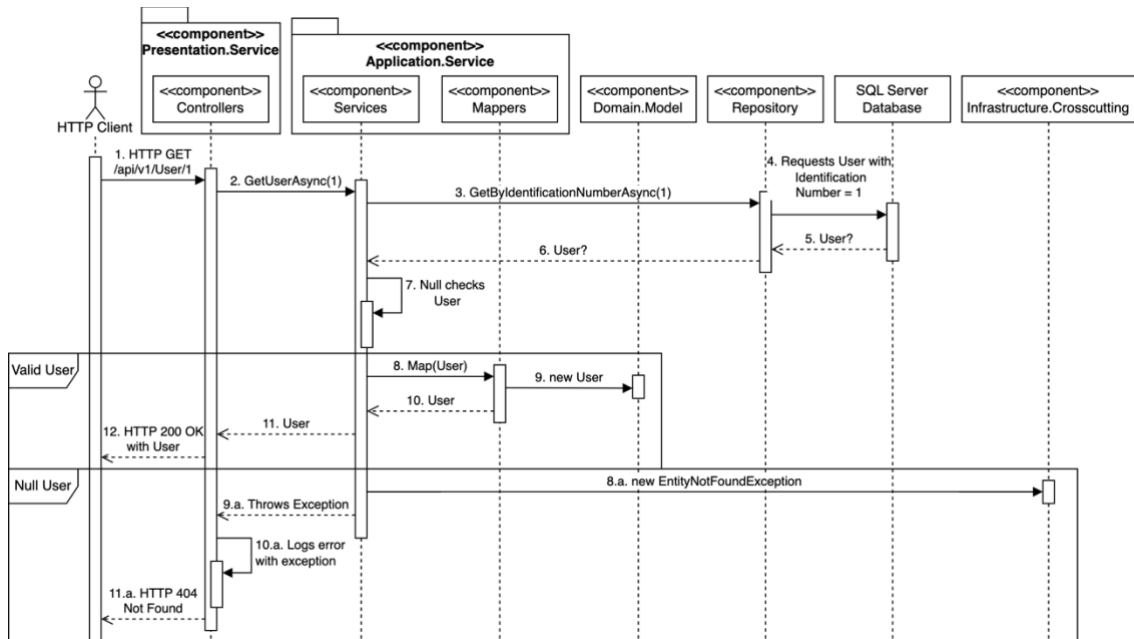


Figure 23 - Process view of the component level of the Core Service regarding fetching user information

By analyzing the diagram, we can conclude that the information flow fits the description of the previous component diagram (cf. Figure 22), which means:

1. The HTTP Client sends an HTTP GET request to consult a User with the Identification Number 1;
2. The Controller receives this request and asks the Service for the requested User;
3. The Service asks the Repository for the User;
4. The Repository asks the SQL Server Database for the User;
5. The SQL Server Database returns something;
6. The Repository returns that to the Service;
7. The Service performs a null check on what it received;
8. If the User is valid then it calls the mapper to map the User;
9. The Mapper maps the User from a Data Model User to a Domain Model User;
10. The Mapper returns the mapped User to the Service;
11. The Service returns the mapped User to the Controller;
12. The Controller returns an HTTP 200 OK response with the mapper User.

On the other hand, if it doesn't pass the null check then:

- A. The Service creates a new Entity Not Found Exception;
- B. The Service throws it to the Controller;

- C. The Controller catches the Exception and logs an error with the exception;
- D. The Controller returns an HTTP 404 Not Found response.

Regarding the design itself, the Onion architectural style was adopted. Onion is a software architectural style introduced in a series of publications by Jeffrey Palermo [63]. It has the goal of facing other architectural style's challenges (e.g., 3-Tier, N-Tier) and providing solutions to common issues such as coupling and separation of interests. It contains a set of concepts that lead to the construction of software with better testability, maintainability, and reliability in infrastructures such as databases and services [64]. By looking at the design again it's possible to observe the four layers it suggests:

1. **Domain model layer:** the Data Model component;
2. **Domain Services layer:** the Domain Model component;
3. **Applicational Services layer:** the Application Service component;
4. **Infrastructure layer:** The Presentation Service, Infrastructure Crosscutting, and Data Repository components.

Several software design patterns were considered. The adoption of these is important during the development of a system since they promote modularity, code reusability, better code readability, and better safety. The following patterns were adopted in this architectural design:

- **Service Layer:** a domain logic pattern that defines the boundary of an application with a services layer that establishes a set of available operations and coordinates the application's response to each operation. It encapsulates the application's business logic by controlling its transactions and coordinating responses in its operations implementations. The benefit of implementing this pattern is that it establishes an application's common set of available operations for various types of clients and coordinates the application's response [65].
 - It was implemented to abstract the component's applicational logic making it responsible for managing interactions with the domain;
- **Mapper:** a data mapping pattern that establishes communication between two independent objects that need to stay ignorant of each other. It may be possible to modify the objects but to create dependencies between these is not intended. It acts as an insular layer between two objects which controls the communication details among these without them being aware [66].
 - It was implemented to abstract the mapping of domain objects into data objects and vice-versa;
- **Strategy:** a behavioral pattern that transforms a set of behaviors into objects making these permutable inside the original context's object. The original object (i.e., context) has a reference to a Strategy object to which it delegates the behavior's execution. To switch how the context executes its behaviors execution, other objects may replace the currently linked Strategy object with another [67].

- It was implemented to abstract application logic into interfaces making it so that there are no dependencies to the implementations. It allows the replacement of an interface's implementation for another with ease;
- **Dependency Injection:** a design pattern used to implement Inversion of Control (IoC). IoC is a design principle to invert control in an Object-Oriented (OO) design with the purpose of achieving low coupling among classes [68]. Dependency Injection allows the creation of independent objects outside of a class and provides these to a class through various means. It abstracts from the classes the process of creation and binding dependent objects on which it depends. This promotes low coupling among components by not exposing the code's implementation and increases the code's testability through the simulation of these with mocks and behavior injection [69].
 - It was implemented to inject classes with their necessary dependencies making these not dependent on implementation, only on interfaces;
- **Repository:** an object-relational metadata mapping pattern that serves as a mediator between domain and persistence (i.e., database) by using a collection-like interface to access the domain's objects. A system with a complex domain model benefits from a layer that isolates domain objects from persistence access details. Conceptually, it encapsulates a set of persistent objects on the database and the operations performed on them, providing a more OO view of the persistence layer [70].
 - It was implemented to abstract the persistence layer and its access.
- **Service Registry:** "(..) a design pattern commonly used in microservice architecture to enable service discovery and dynamic load balancing. The microservices register themselves with a service registry which acts as a central repository for service metadata"[71].
 - It was implemented to enable interaction between services and external requests from the API Gateway via Consul.

4.3.2 Other services

The other containers (i.e., User Service, Transfer Service, and Payment Service) follow the same architectural design with the difference between them being the entity they represent and manage. Their architectural design is very similar to the Core Service container's architectural design and follows the same design patterns.

The following component diagram (cf. Figure 24) presents the architectural design of the Payment Service container (i.e., its internal components and how they relate).

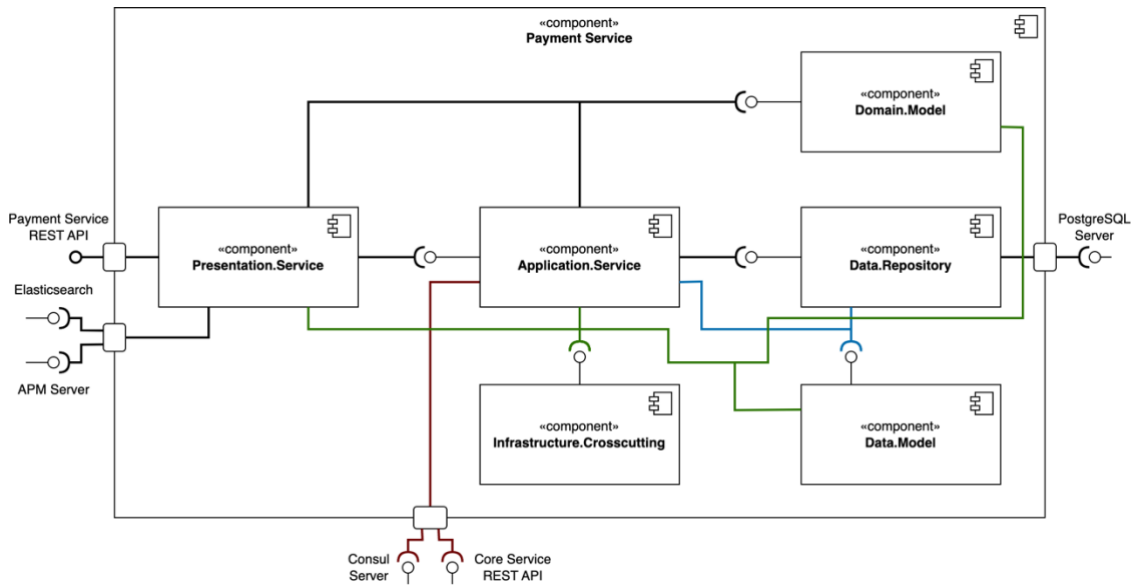


Figure 24 - Logical view of the component level of the Payment Service container

By analyzing it, it can be noted that it's indeed similar to the Core Service container's architectural design. It follows the same logic as the Core Service with the exception being that it interacts with the Consul Server to get the Core Service address to then request it to validate payments so that it can execute and register these. Regarding the consultation of payments, it does not interact with the Core Service as it queries its database (i.e., PostgreSQL Database).

To better understand this difference in the information flow, the following sequence diagram (cf. Figure 25) can be observed regarding the execution of a new payment. The diagram only represents the happy path of this process since its complexity and size increase significantly with all paths. Regardless, the sequence diagram with all paths can be consulted in Annex B (cf. Figure 81).

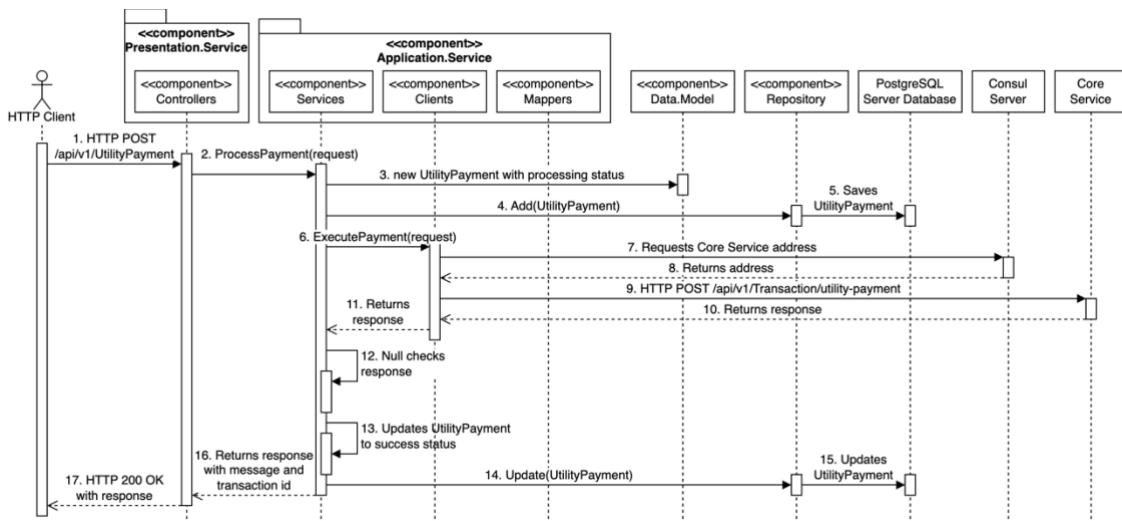


Figure 25 - Process view of the container level of the Payment Service container regarding payment execution with only the happy path

By analyzing the diagram, we can conclude that the information flow fits the description of the previous component diagram (cf. Figure 24), which means:

1. The HTTP Client sends an HTTP POST request to process a new Utility Payment;
2. The Controller receives this request and asks the Service to process this new Utility Payment;
3. The Service creates a new Utility Payment with the processing status;
4. The Service asks the Repository to add this new Utility Payment;
5. The Repository adds it to the PostgreSQL Server database;
6. The Service asks the Client to execute the Utility Payment;
7. The Client asks the Consul Server for the Core Service's address;
8. The Consul Server returns the address;
9. The Client sends an HTTP POST request to the Core Service to execute the Utility Payment;
10. The Core Service returns a response;
11. The Client deserializes the response and returns it to the Service;
12. The Service null checks the response;
13. Since the response is valid it will update the Utility Payment status to success and add the transaction reference to it;
14. The Service asks the Repository to update the Utility Payment;
15. The Repository updates the Utility Payment on the PostgreSQL Server database;
16. The Service returns a response with a success message and transaction reference to the Controller;
17. The Controller returns this response to the HTTP Client.

4.4 Level 4: Code

Knowing the high complexity of the business model, this section only presents the fine-grain design of the Transaction-related components of the Core Service container and the Transfer Service container components. The DDD concept of ubiquitous language application is noticed here since classes and their methods in the different layers are named according to the entity these represent.

4.4.1 Transaction-related components of the Core Service

The following class diagram (cf. Figure 26) presents the classes that compose the Transaction-related components of the Core Service container and their dependencies.

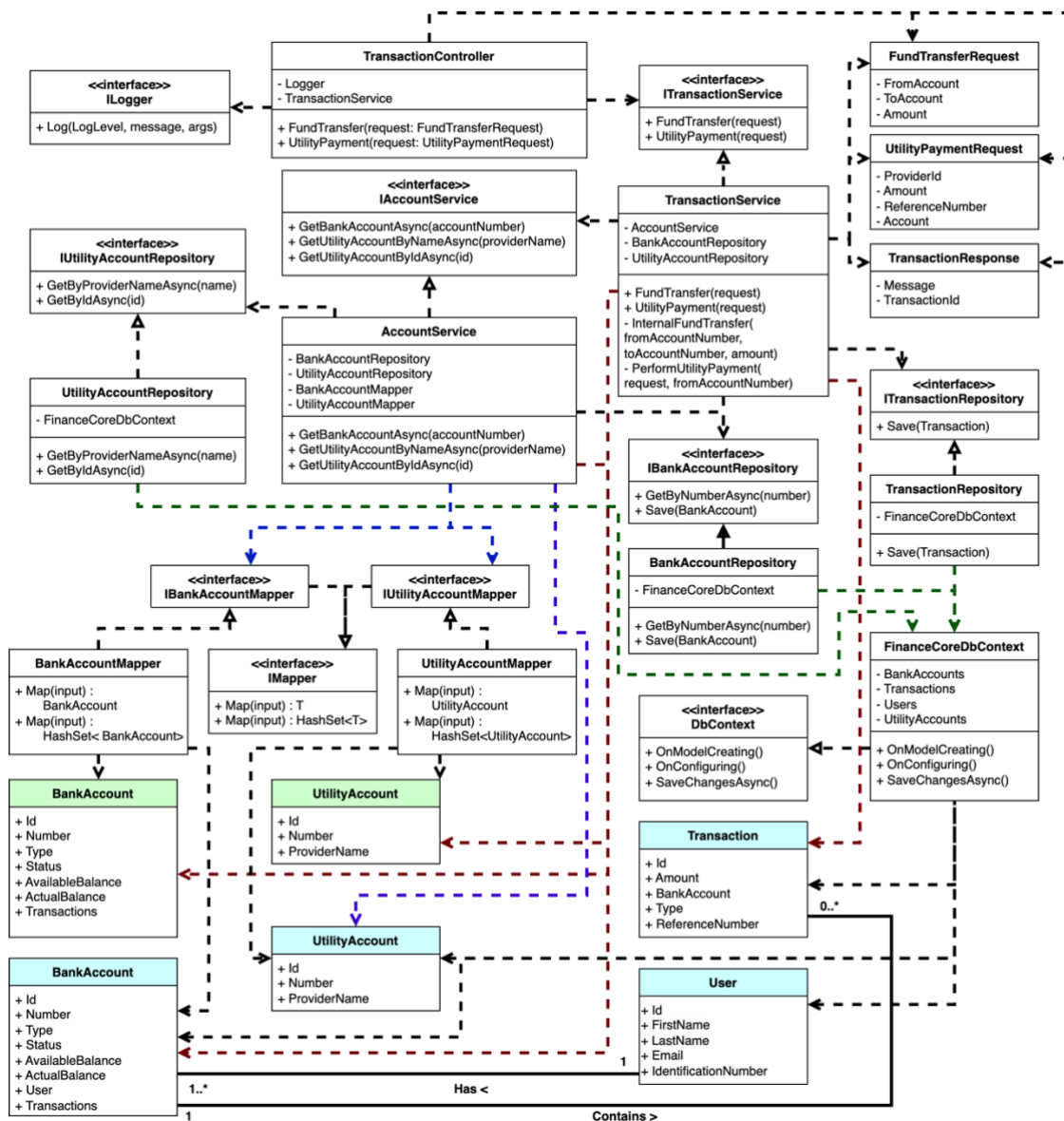


Figure 26 - Class diagram of the class level referring to the Transaction-related components of the Core Service container

Analyzing it from the top:

- The TransactionController:
 - Contains two endpoints. The first, FundTransfer, executes a fund transfer between bank accounts and returns a message and the transaction reference, and the second, UtilityPayment, executes a utility payment between a bank account (i.e., user account)

- and a utility account (i.e., provider account) and returns a message and the transaction reference;
- Depends on the `ITransactionService` and `ILogger` interfaces, and the `FundTransferRequest`, `UtilityPaymentRequest`, and `TransactionResponse` classes.
 - The `ILogger` interface sets multiple behaviors since it's provided by .NET but for this context, only one – `Log` – is employed;
 - The `Logger` object is injected by .NET into the `TransactionController` class;
 - The `ITransactionService` interface sets two behaviors, `FundTransfer` and `UtilityPayment`;
 - The `TransactionService` class:
 - Implements the `ITransactionService` interface (i.e., its behaviors) and is injected in the `TransactionController` class;
 - Depends on the `ITransactionRepository`, `IAccountService`, and `IBankAccountRepository` interfaces and the `FundTransferRequest`, `UtilityPaymentRequest`, and `TransactionResponse` classes.
 - The `IAccountService` interface sets three behaviors, `GetBankAccountAsync`, `GetUtilityAccountByNameAsync`, and `GetUtilityAccountByIdAsync`;
 - The `AccountService` class:
 - Implements the `IAccountService` interface (i.e., its behaviors) and is injected in the `TransactionService` class;
 - Depends on the `IUtilityAccountRepository`, `IBankAccountRepository`, `IUtilityAccountMapper`, and `IBankAccountMapper` interfaces and `BankAccount` and `UtilityAccount` classes.
 - The `IUtilityAccountMapper` and `IBankAccountMapper` interfaces extend the `IMapper` interface;
 - The `IMapper` interface sets two behaviors to map the state between domain objects (i.e., classes with a green fill) and data objects (i.e., classes with a blue fill), `Map` and `Map` where the difference is one returns a single object, and the other a set of objects;
 - The `BankAccountMapper` and `UtilityAccountMapper` classes:
 - Implement the `IBankAccountMapper` and `IUtilityAccountMapper` interfaces respectively, and are injected in the `AccountService` class;
 - Depend on the two `BankAccount` classes and the two `UtilityAccount` classes respectively.
 - The `IUtilityAccountRepository` and `IBankAccountRepository` set two behaviors each, `GetByProviderName` and `GetByIdAsync`, and `GetByNumberAsync` and `Save`, respectively;
 - The `UtilityAccountRepository` class:
 - Implements the `IUtilityAccountRepository` interface and is injected in the `AccountService` Class;
 - Depends on the `FinanceCoreDbContext` class.
 - The `BankAccountRepository` class:
 - Implements the `IBankAccountRepository` interface and is injected in the `AccountService` and `TransactionService` classes;
 - Depends on the `FinanceCoreDbContext` class.

- The `ITransactionRepository` interface sets the `Save` behavior;
- The `TransactionRepository` class:
 - Implements the `ITransactionRepository` interface and is injected in the `TransactionService` class;
 - Depends on the `FinanceCoreDbContext` class.
- The `DbContext` interface:
 - Belongs to the ORM EF Core;
 - Set three methods, `OnModelCreating`, `OnConfiguring`, and `SaveChangesAsync`. The first is to map the classes/data objects and the persistence so that EF Core may generate their relational schema. The second has to configure the persistence access (i.e., type of persistence, where it is found, etc.). The third has the objective of permanently saving state changes in the persistence.
- The `FinanceCoreDbContext` class:
 - Implements the `DbContext` interface and is injected in the `BankAccountRepository`, `UtilityAccountRepository`, and `TransactionRepository` classes;
 - Depends on the `Transaction`, `User`, `UtilityAccount`, and `BankAccount` classes so that it can map them and access their tables in the persistence (i.e., database).
- The `BankAccount`, `UtilityAccount`, `Transaction`, and `User` classes are presented as described in section 3.1:
 - `BankAccount`: Contains 8 attributes. An identifier (i.e., `Id`), an account number (i.e., `Number`), an account type (i.e., `Type`), a status (i.e., `Status`) – that indicates if the account is active, blocked, or pending –, the available balance (i.e., `AvailableBalance`), the actual balance (i.e., `ActualBalance`), and the transactions associated (i.e., `Transactions`);
 - `UtilityAccount`: Contains 3 attributes. An identifier (i.e., `Id`), an account number (i.e., `Number`), and the provider's name (i.e., `ProviderName`);
 - `Transaction`: Contains 5 attributes. An identifier (i.e., `Id`), the transacted amount (i.e., `Amount`), the associated bank account (i.e., `BankAccount`), the transaction type (i.e., `Type`) – that is if it's a transfer or a payment –, and a reference number (i.e., `ReferenceNumber`);
 - `User`: Contains 5 attributes. An identifier (i.e., `Id`), the first name of the user (i.e., `FirstName`), the last name of the user (i.e., `LastName`), the user's email (i.e., `Email`), and its identification number (i.e., `IdentificationNumber`).

4.4.2 Transfer Service components

The following class diagram (cf. Figure 27) presents the classes that compose the Transfer Service container and their dependencies.

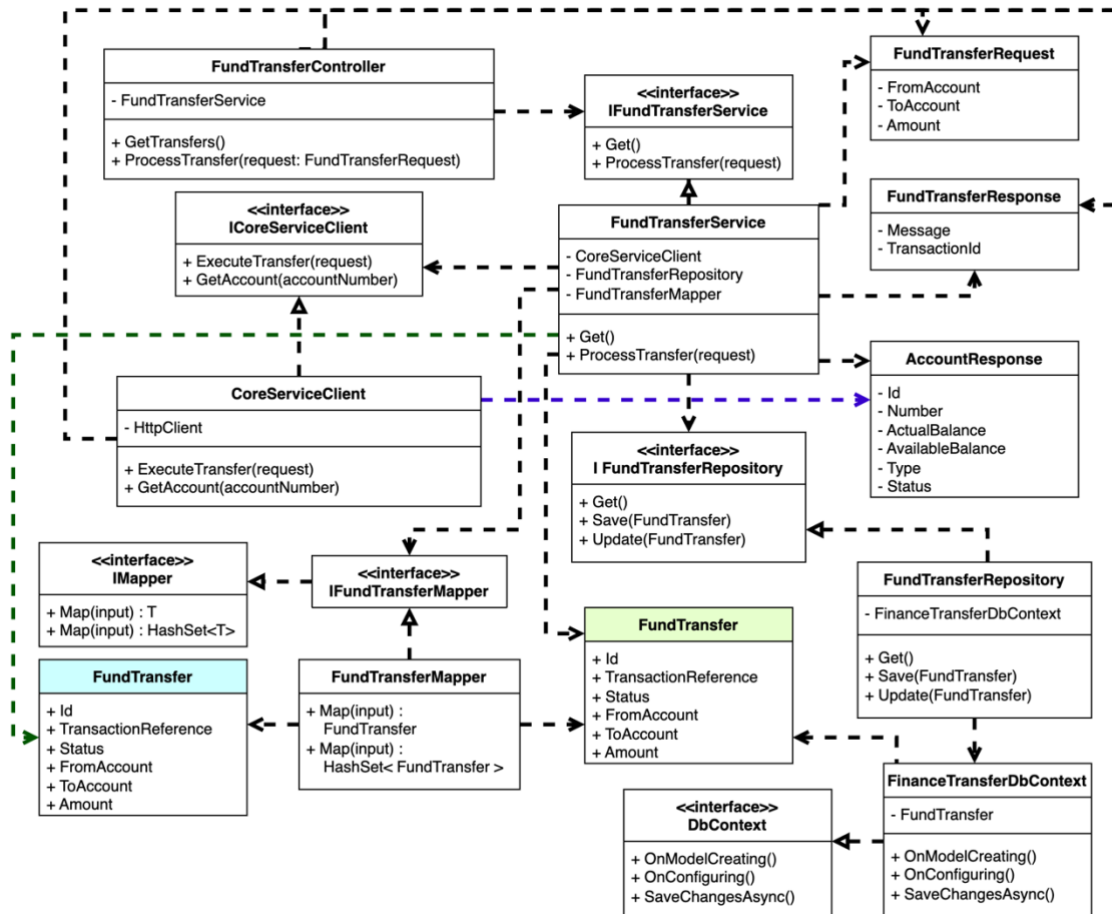


Figure 27 - Class diagram of the class level of the Transfer Service container components

Analyzing it from the top:

- The FundTransferController class:
 - Contains two endpoints. GetTransfers – to obtain all existent transfers – and ProcessTransfer – that receives a transfer request to process;
 - Depends on the IFundTransferService interface, and the FundTransferRequest and FundTransferResponse classes.
- The IFundTransferService interface sets two behaviors, Get and ProcessTransfer;
- The FundTransferService class:
 - Implements the IFundTransferService interface and is injected in the FundTransferController class;

- Depends on the ICoreServiceClient, IFundTransferRepository, and IFundTransferMapper interfaces, and FundTransferRequest, FundTransferResponse, AccountResponse, and both FundTransfer (i.e., domain and data) classes.
- The ICoreServiceClient interface sets two behaviors, ExecuteTransfer and GetAccount;
- The CoreServiceClient class:
 - Implements the ICoreServiceClient interface and is injected in the FundTransferService class;
 - Depends on the HttpClient, AccountResponse, FundTransferRequest, FundTransferResponse, and AccountResponse classes.
- The IFundTransferMapper interface extends the IMapper interface that sets two behaviors to map the state between domain objects (i.e., classes with a green fill) and data objects (i.e., classes with a blue fill), Map and Map where the difference is one returns a single object, and the other a set of objects;
- The FundTransferMapper class:
 - Implements the IFundTransferMapper interface and is injected in the FundTransferService class;
 - Depends on both FundTransfer classes.
- The IFundTransferRepository interface sets three behaviors, Get, Save, and Update;
- The FundTransferRepository class:
 - Implements the IFundTransferRepository interface and is injected in the FundTransferService class;
 - Depends on the FinanceTransferDbContext class.
- The DbContext interface:
 - Belongs to the ORM EF Core;
 - Set three methods, OnModelCreating, OnConfiguring, and SaveChangesAsync. The first is to map the classes/data objects and the persistence so that EF Core may generate their relational schema. The second has to configure the persistence access (i.e., type of persistence, where it is found, etc.). The third has the objective of permanently saving state changes in the persistence.
- The FinanceTransferDbContext class:
 - Implements the DbContext interface and is injected in the FundTransferRepository class;
 - Depends on the FundTransfer class so that it can map it and access its tables in the persistence (i.e., database).
- The FundTransfer class is presented as described in section 3.1:
 - Contains 6 attributes. An identifier (i.e., Id), a transaction reference (i.e., TransactionReference), a status (i.e., Status) – that if a transfer is pending, processing, successful, or failed –, the sender’s account number (i.e., FromAccount), the receiver’s account number (i.e., ToAccount), and the transaction amount (i.e., Amount).

4.5 Database

Regarding the database design, a code-first approach was adopted with the employment of the object-relational mapper (ORM) Entity Framework Core (EF Core). The entities were designed through the adoption of the OO paradigms (i.e., classes), and the relational schema was generated automatically through EF Core.

4.5.1 Core Service

The following entity-relationship (ER) model (cf. Figure 28) presents the entities that are in the Core Service container and persisted in the SQL Server database.

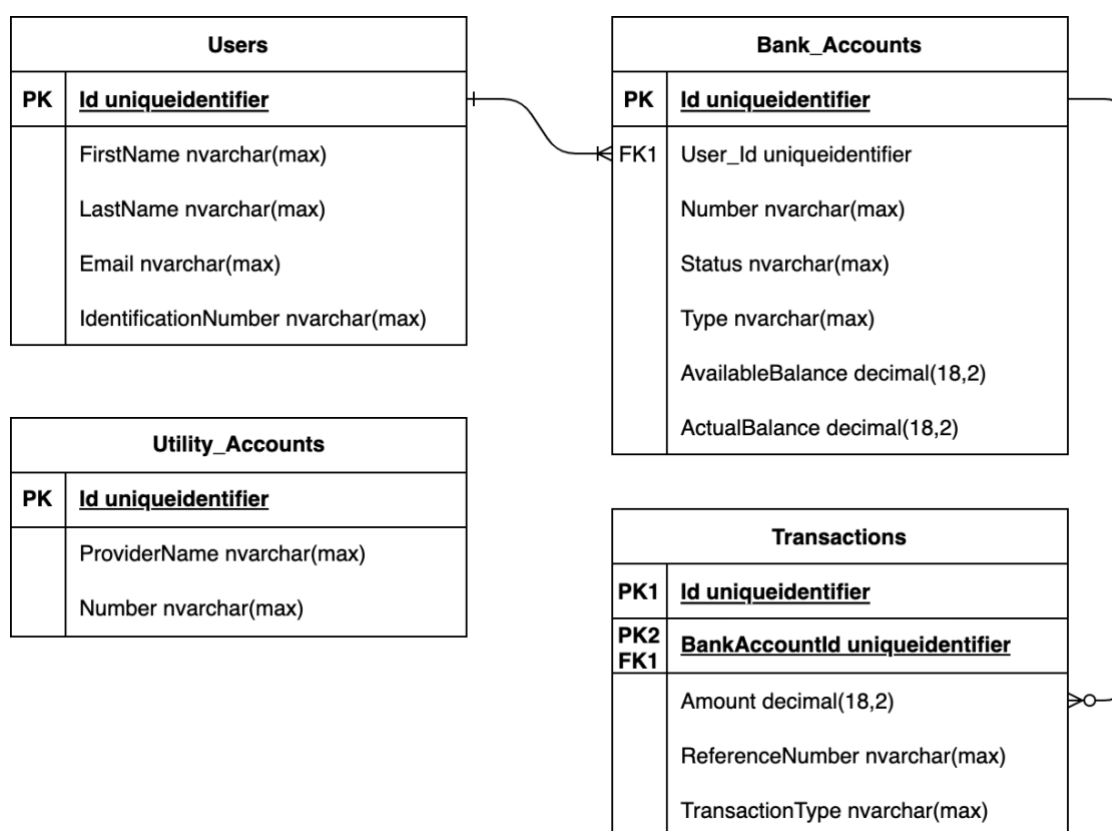


Figure 28 - Entity-Relationship model of the Core Service container

When comparing the tables (i.e., entities) of the model with the domain classes in the class diagram in the previous section (cf. Figure 26) it is noted that these are virtually identical. That is the result of the class mapping by EF Core in the relational schema that can be observed in the model above.

Four entities can be observed by analyzing the model:

- Users:
 - Composed by an (i) unique identifier of the type uniqueidentifier (i.e., GUID/UUID) that serves as the primary key, a (ii) first name, (iii) last name, and (iii) email of the type nvarchar, and an (iv) identification number of the type nvarchar since it can be alphanumeric;
 - Possesses a one-to-many relationship with Bank Accounts.
- Bank Accounts:
 - Composed by an (i) unique identifier of the type uniqueidentifier that serves as the primary key, a (ii) user identifier to whom the bank account belongs of the type uniqueidentifier and is a foreign key, an (iii) account number of the type nvarchar, a (iv) status of the account that indicates if it's active, blocked, or pending of the type nvarchar, the (v) type of account that indicates if it's a fixed, savings, or loan account of the type nvarchar, and the (vi) available balance and (vii) actual balance of the account of the type decimal;
 - Possesses a many-to-one relationship with Users and a one-to-zero-or-more relationship with Transactions.
- Transactions:
 - Composed by an (i) unique identifier of the type uniqueidentifier, and the (ii) bank accounts' primary key that is a foreign key and composes the primary key with the unique identifier, the (iii) transaction amount of the type decimal, the (iv) reference number of the transaction of the type nvarchar, and the (v) type of transaction that indicates if it's a fund transfer or a utility payment of the type nvarchar;
 - The unique identifier is generated by the Core Service and is shared by transactions that are of the fund transfer type in order to identify the same transaction for both accounts. That is also why the bank account's primary key is part of the composed primary key as it allows for both records to have the same unique identifier;
 - Possesses a zero-or-more-to-one relationship with Bank Accounts.
- Utility Accounts:
 - Composed by an (i) unique identifier of the type uniqueidentifier, the (ii) provider name of the type nvarchar, and an (iii) account number of the type nvarchar.

All unique identifiers except the Transactions are automatically generated by the database that are assigned when inserting new records.

4.5.2 Other services

The following entity-relationship (ER) model (cf. Figure 29) presents the entities that are in the User Service, Transfer Service, and Payment Service containers and are persisted in the PostgreSQL Server database.

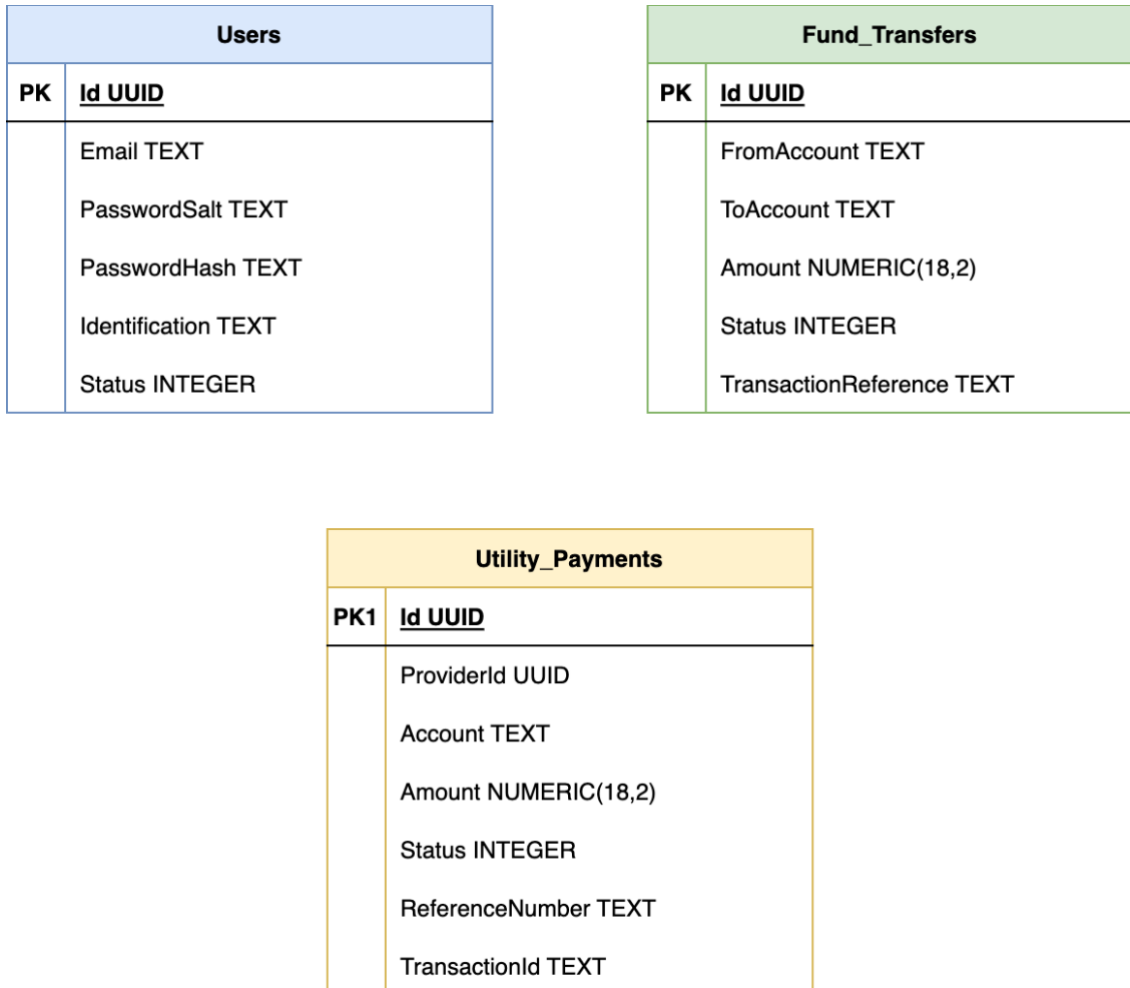


Figure 29 - Entity-Relationship model of the User Service, Transfer Service, and Payment Service containers

Comparing the Fund Transfers table (i.e., entity) with the domain class in the class diagram in the previous section (cf. Figure 27) confirms that these are virtually identical. This is the result of the class mapping by EF Core in the relational schema that can be observed in the model above. The same can be affirmed for the other entities in spite of not being presented in a class diagram.

Three entities can be observed in the model above:

- Users:
 - Composed by an (i) unique identifier of the type uuid that serves as the primary key, a (ii) email of the type text, a (iii) password salt that is used to make the password harder to crack of the type text, the (iv) password that is hashed to be encrypted of the type text, an (v) identification number of the type text since it can be alphanumeric, and a (vi) status that indicates if the user's account is pending, active, or blocked of the type integer;
- Fund Transfers:
 - Composed by an (i) unique identifier of the type uuid that serves as the primary key, the (ii) sender's account number of the type text, the (iii) recipient's account number of the type text, the (iv) transaction amount of the type numeric, the (v) transfer's status that indicates if it's processing, successful, or failed, of the type integer, and the (vi) transaction's reference number of the type text.
- Utility Payments:
 - Composed by an (i) unique identifier of the type uuid that serves as the primary key, the (ii) provider's unique identifier of the type uuid, the (iii) payer's account number of the type text, the (iv) transaction amount of the type numeric, the (v) payment's status that indicates if it's processing, successful, or failed, of the type integer, the (vi) payment's reference number that is supplied by the payment request of the type text, and the (vii) transaction identifier that is received by the Core Service of the type text.

All unique identifiers are automatically generated by the database that are assigned when inserting new records. The reason why the column types are different from the ones in the Class Service is due to the database being different (i.e., PostgreSQL) from the one that is being used in the Core Service (i.e., SQL Server).

4.6 Summary

This chapter presented and described:

- The design choices of the solution in different granularities were complemented by different views of them, thanks to the adoption of the C4 model, which allowed in the form of diagrams to have a clear vision of the interactions and necessary precautions to adopt for the development of the solution;
- The database design presented all persisted classes in the different databases.

5 Technologies

This section describes the adopted technologies for the development of the system and its dependencies.

5.1 Adopted technologies

Various technologies were adopted to develop this system. The latest stable versions of these were selected which allows to adhere the patch management method of the Secure-by-Design approach as its important to maintain the system's components up to date [56]. Table 9 enumerates the adopted technologies.

Table 9 - Adopted technologies

Component	Technologies
System's services	<ul style="list-style-type: none">• .NET• C#• Serilog• Elastic Common Schema• Entity Framework Core• Steeltoe• Swashbuckle• Elastic APM
API Gateway	<ul style="list-style-type: none">• The same technologies as the System's services• Ocelot
Service discovery	<ul style="list-style-type: none">• Consul
Database	<ul style="list-style-type: none">• Microsoft SQL Server• PostgreSQL Server• Elasticsearch
Logging, monitoring, tracing, and observability	<ul style="list-style-type: none">• Kibana• APM Server• Metricbeat

Component	Technologies
	<ul style="list-style-type: none"> • Heartbeat

An introduction to each technology is found in following sections. Some are explained in more detail than others, assuming that the reader has the necessary basic knowledge.

5.2 System's services

This section describes the adopted technologies for the development of the system's services.

5.2.1 .NET

.NET (dot NET), formerly known as .NET Core, is a software development open-source multiplatform framework [72] that is developed and maintained by Microsoft [73]. It is seen as the successor of the widely known .NET Framework [74]. It has the development of services, websites, and console applications as its goal [72]. It fully supports C#, Visual Basic, and F# as its main programming languages [75].

This framework was adopted for the development of the system and the API Gateway due to the fact that its multiplatform, is widely supported (i.e., has a massive community), has rich documentation, and is the framework that the student has most experience with.

5.2.2 C#

C# (C Sharp) is a multiparadigm programming language with a strong type system that can be utilized for imperative, declarative, functional, generic, OO, and component-oriented programming. Its purpose is to be a simple, modern, and general-purpose OO programming language [76].

This language was adopted for the development of the system and the API Gateway.

5.2.3 Serilog

Serilog is a diagnostic logging library for .NET applications. It runs on all .NET platforms, is easy to set up, and has a clean API. It provides robust support for structured logging, particularly when instrumenting complex, distributed, and asynchronous applications and systems. It supports diagnostic logging to files, the console, and many other outputs [77].

This library was adopted to better structure the system's logs on the console and to send these to the Elasticsearch database, through an Elasticsearch sink (i.e., writer) provided by Serilog [78].

5.2.4 Elastic Common Schema

Elastic Common Schema (ECS) is an open-source specification developed with support from the Elastic user community. It defines a common set of fields to be used when storing event data in Elasticsearch, such as logs and metrics that can then be consulted on Kibana [79]. ECS has a collection of formatters for .NET logging libraries that are used to format log events according to their specification, one of which is Serilog [80].

ECS was adopted to format log events being written from Serilog into Elasticsearch with its format/specification.

5.2.5 Entity Framework Core

Entity Framework Core (EF Core) is a light, extensible, and multiplatform open-source ORM framework. It allows for .NET developers to work with databases using .NET objects by providing a data persistence access layer thus eliminating the need to write a lot of database access code that normally would have been written otherwise [81].

EF Core was adopted to implement the data persistence access layer and to perform the relational mappings between the domain objects since it is standard in .NET development when it comes to relational models.

5.2.6 Steeltoe

Steeltoe is a collection of libraries that help .NET developers build production-grade cloud-native applications with externalized configuration, service discovery, circuit-breakers, distributed tracing, application management, and more. It provides a seamless way to build, configure, and run event-driven microservice applications and stream-based data processing applications [82].

The service discovery library was adopted for the system and the API Gateway interactions, specifically the library with Consul support.

5.2.7 Swashbuckle

Swashbuckle is Swagger tooling for APIs built with .NET. It generates API documentation following the OpenAPI specification, providing an easy-to-use UI to explore that can test operations directly from the API's routes, controllers, and models. It is customizable, works in any environment, and in any browser [83].

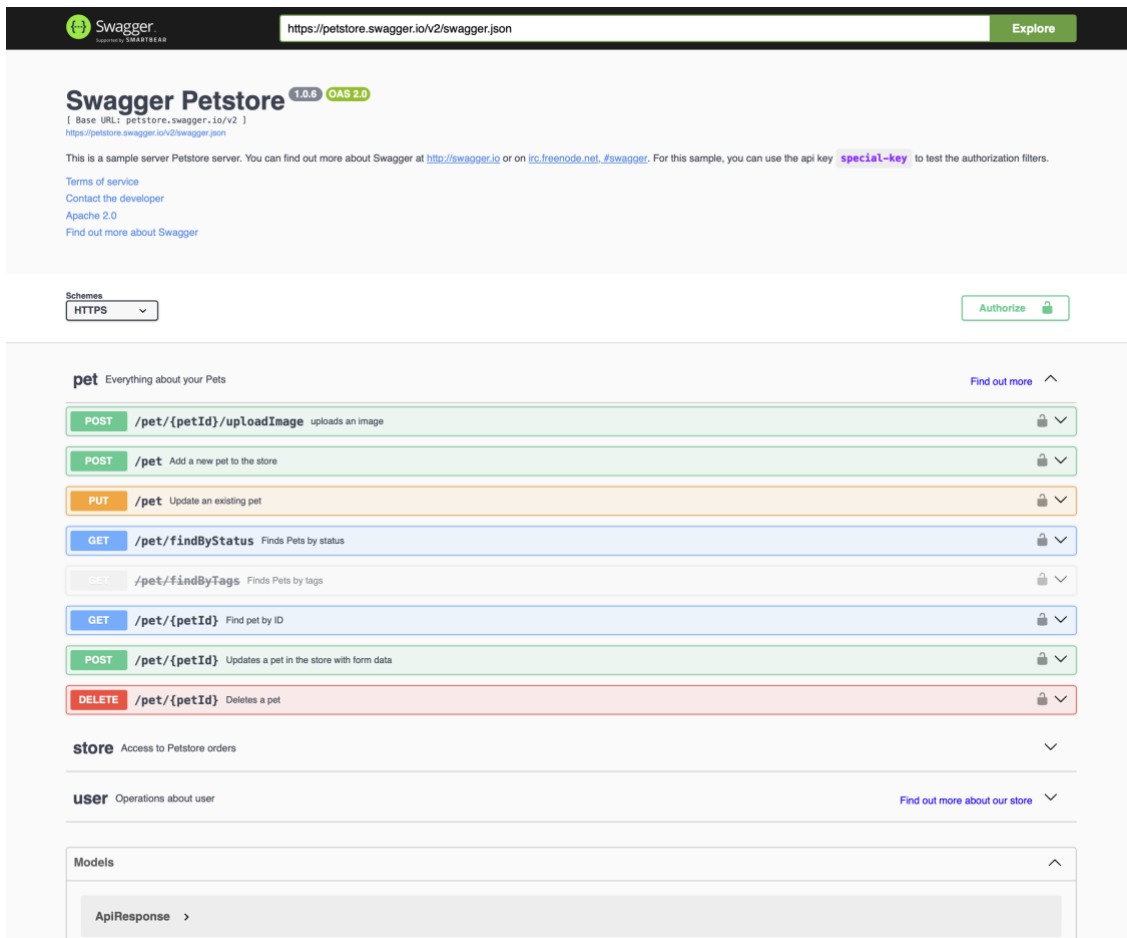


Figure 30 - Example of Swagger UI page

This tooling was adopted to document the system's endpoints for ease of comprehension and reading over each.

5.2.8 Elastic APM

Elastic Application Performance Monitoring (APM) is an APM system built on the Elastic Stack. It allows the monitoring of software services and application in real-time by collecting detailed performance information on response time for incoming requests, database queries, call to caches, external HTTP requests, and more [84]. Elastic has a collection of libraries that allow the set-up of APM agents in various languages, including .NET [85].

Elastic APM was adopted to automatically measure the system's performance by and track its errors.

5.3 API Gateway

This section describes the adopted technologies for the development of the API Gateway. Since the majority of the technologies are the same as the system's services, only Ocelot is described.

Ocelot is a .NET API Gateway. It's aimed at .NET microservices or systems using service-oriented architecture that need a unified point of entry into their system. It is composed of middlewares that follow a specific order: it essentially manipulates external requests until these reach a request builder middleware that maps them into internal requests that are sent to the downstream services, and then maps the internal responses into external responses which are sent to the client. It also contains more features such as caching, quality of service (i.e., circuit breaker), service discovery, load balancing, rate limiting, and more [86].

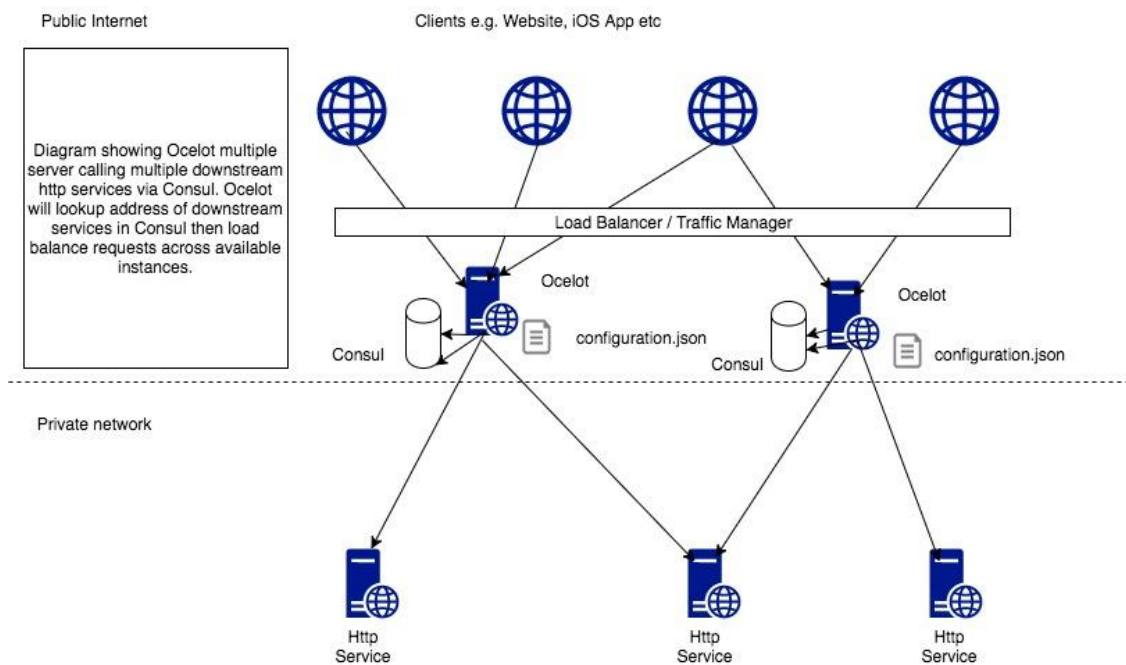


Figure 31 - Example of an Ocelot API Gateway configuration with Consul [86]

Ocelot was adopted along with the features mentioned above to develop the API Gateway that serves as the entry point to the system due to its ease of use and rich feature set.

5.3.1 Service discovery

This section describes the adopted technology for service discovery, Consul.

Service discovery is how applications and services locate each other on a network. It functions through a central registry (i.e., service registry) that maintains a global view of addresses and clients that connect to the central registry to update and retrieve addresses [87]. There are two types of service discovery: client-side and server-side. Client-side service discovery is when

client applications obtain the location of the service instance by querying the service registry to make a request to that service [88]. Server-side service discovery is when client applications make a request via router (i.e., load balancer) that queries the service registry for the location of the service instance to forward its request to [89].

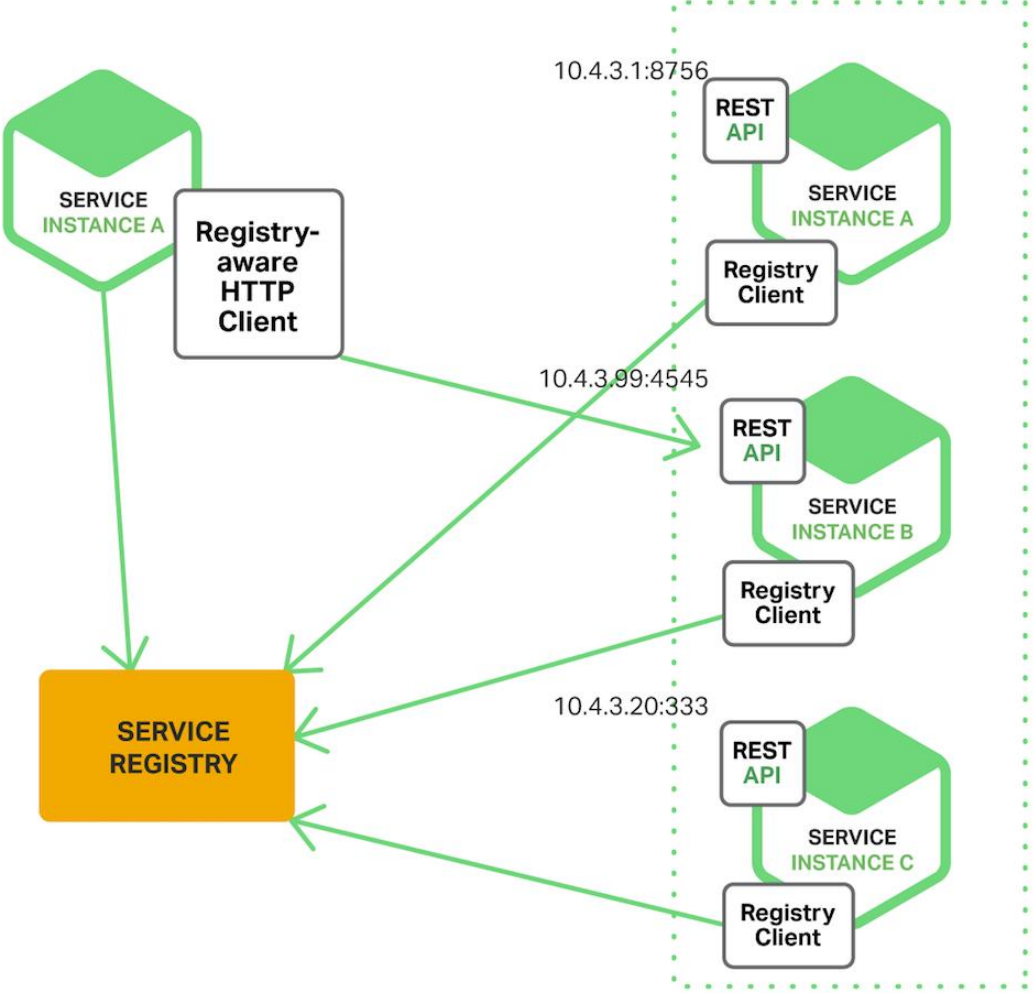


Figure 32 - Example of a system with client-side service discovery [90]

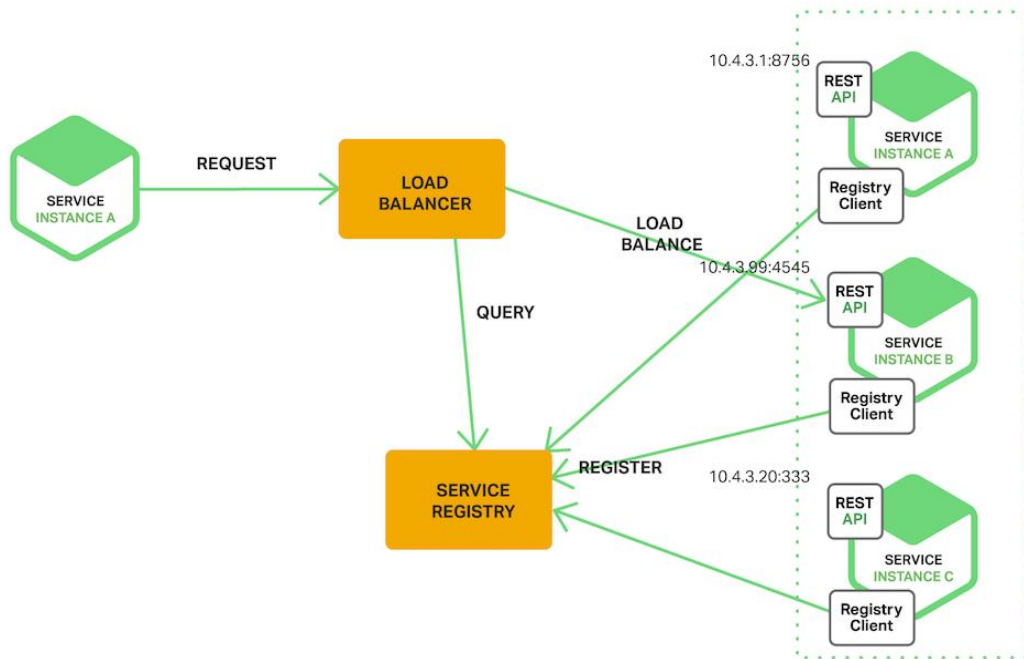


Figure 33 - Example of a system with server-side service discovery [90]

Consul is a service networking solution that enables the management of secure network connectivity between services across different environments and runtimes. It offers features such as service discovery, service mesh, and traffic management [91].

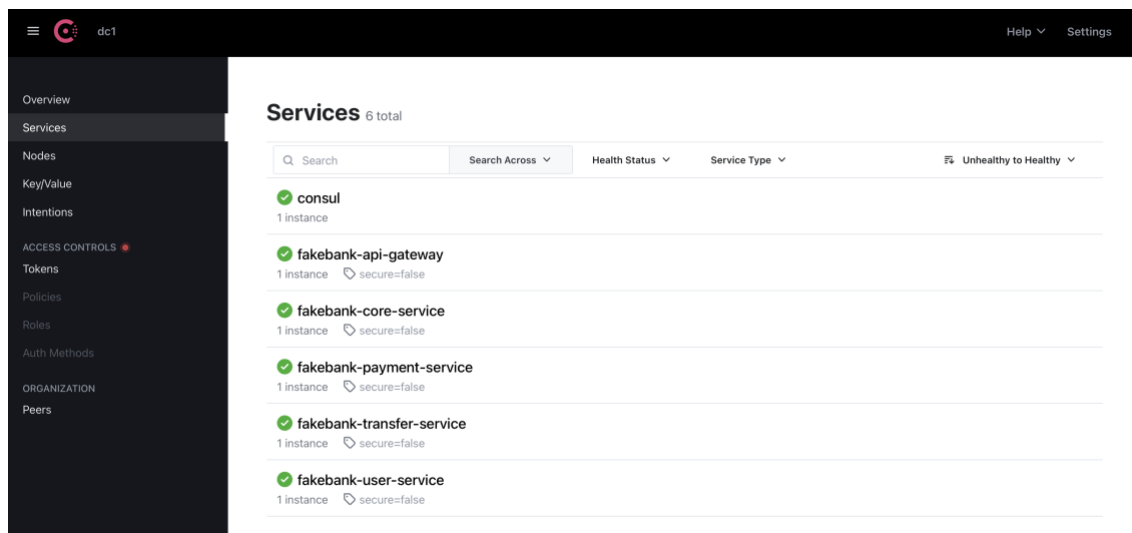


Figure 34 - Consul UI

Consul is adopted for its service discovery feature and ease of use.

5.4 Database

This section describes the adopted technologies for persisting data (i.e., databases).

5.4.1 Microsoft SQL Server

Microsoft SQL Server (SQL Server) is a relational database management system (RDMS) developed by Microsoft [92]. The main objective of an RDMS is to persist and return data when queried [93]. SQL Server contains different editions for different needs [94].

The Express version of SQL Server was adopted for both local development and testing since it only includes the database engine and is ideal for small applications [94]. It is used by the Core Service for data persistence.

5.4.2 PostgreSQL Server

PostgreSQL is a powerful popular open-source RDMS developed by the University of California at Berkeley. It has a strong reputation for its architecture, reliability, data integrity, robust feature set, extensibility, and a large, dedicated community [95].

PostgreSQL was adopted for both local development and testing. It is used by the User, Transfer, and Payment services for data persistence.

5.4.3 Elasticsearch

Elasticsearch is the distributed search and analytics engine that is the core of the Elastic Stack. It provides near real-time search and analytics for all types of data whether these are structured or unstructured text, numerical data, or geospatial data. It can efficiently store and index it in a way that supports fast searches. Other than data retrieval and information aggregation it allows to discover trends and patterns in data. It supports a wide variety of use cases such as storage and analysis of logs, metrics, and security event data, the usage of machine learning models to automatically model the data's behavior in real-time, and many more [96].

Elasticsearch was adopted to store logs, traces, and metrics from the system and the API Gateway.

5.5 Logging, monitoring, tracing, and observability

This section describes the adopted technologies for logging, monitoring, tracing, and observability of the system.

5.5.1 Kibana

Kibana is a UI that allows the visualization of Elasticsearch data and to navigate the Elastic Stack. It allows us to analyze, monitor, and observe applications. It has features such as [97]:

- **Elastic Observability:** enables monitoring and applying analytics in real events happening across environments. It can analyze log events, monitor performance metrics for the host or container that it ran in, trace transactions, and verify overall service availability;
- **Analytics:** enables the search of data for hidden insights and relationships with queries and filters, to create visualizations of data in many ways and perspectives with dashboards, to use machine learning models to model the behavior of data to forecast unusual behavior and perform outlier detection, regression, and classification analysis;
- **Manage, monitor, and secure the Elastic Stack:** enables to refresh, flush, and clear the cache of indices, to define the lifecycle of an index as it ages, to roll up data from one of more indices into a new, compact index, to replicate indices from a remote cluster to a local cluster. It can detect and act on significant shifts and signals in the data such as memory, CPU, and storage usage through email, Slack, and other 3rd party integrations. It allows to organize content into spaces for different needs without impacting others. It provides a range of security features such as logging in with 3rd party authentication providers – single sign-on (SSO) providers –, role-based access to features, and audit logging to maintain logging of who did what, when.

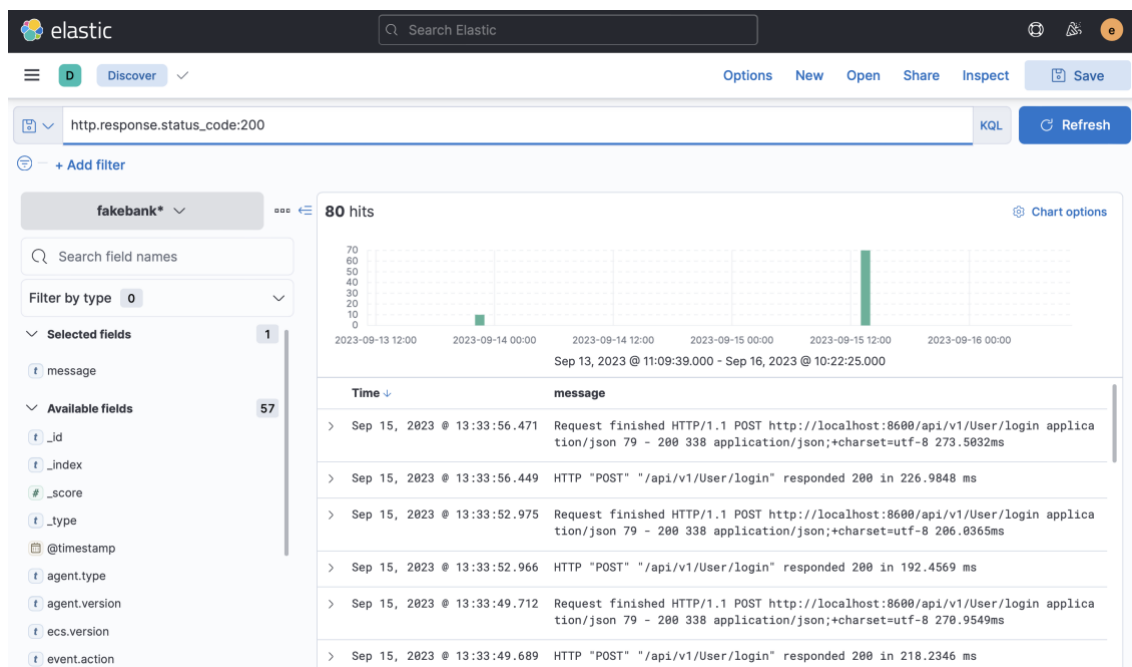


Figure 35 - Example of searching data through analytics in Kibana

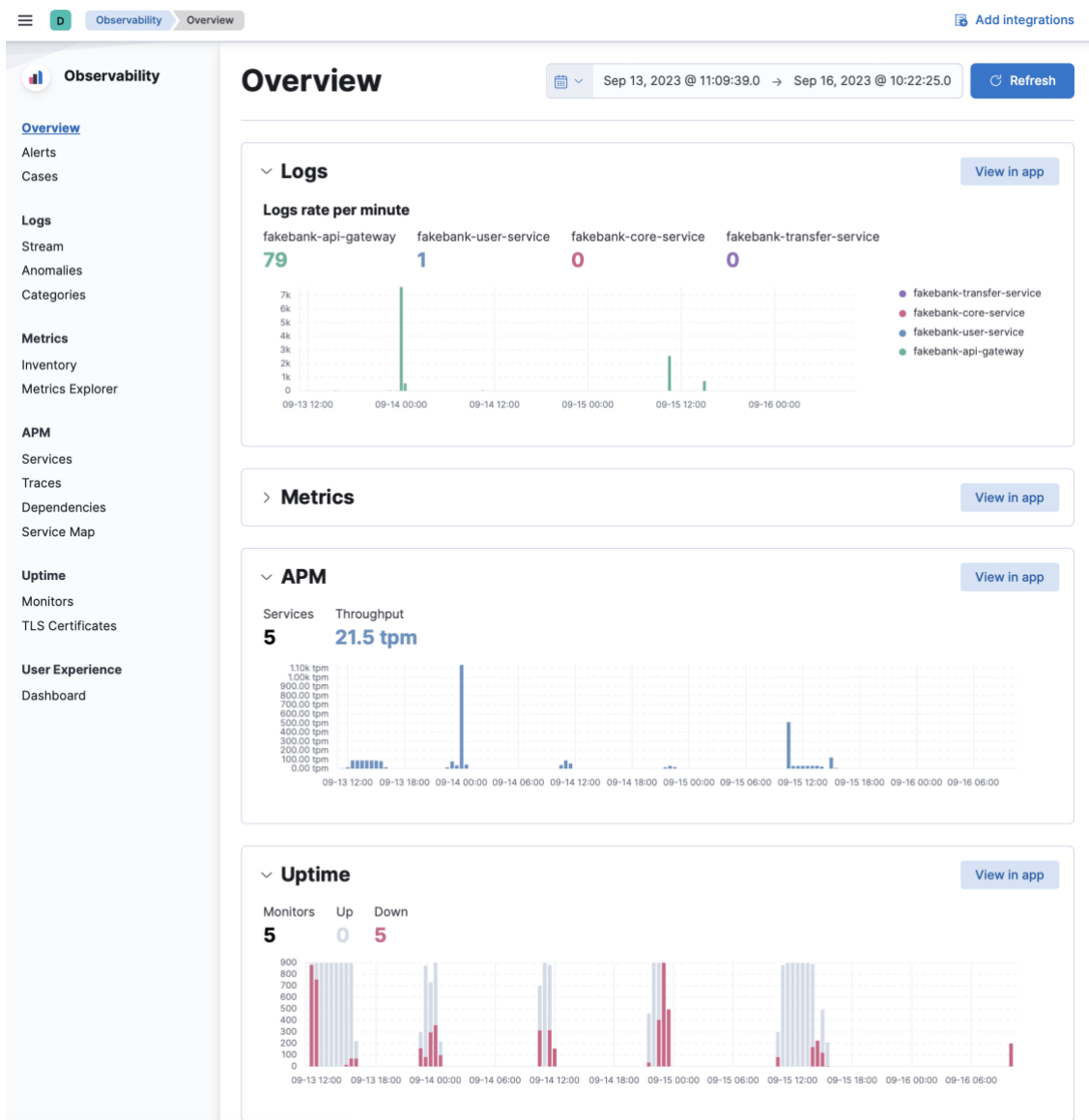


Figure 36 - Example of consulting Elastic Observability overview on Kibana

Kibana was adopted to consult, visualize, analyze, and train machine learning models on logs, traces, and metrics that were stored in Elasticsearch.

5.5.2 APM Server

Elastic APM is already described in section 5.2.8. The APM Server is where the data is sent from the APM agents in the system and in the API Gateway, and then after processing and enriching it, stores it in Elasticsearch.

5.5.3 Metricbeat

Metricbeat is part of the Beats, which is a collection of open-source data shippers that act as agents to send operation data to Elasticsearch. Metricbeat specifically collects information on systems metrics such as CPU, memory, disk usage, and network traffic. It also has modules for collection metrics from services such as PostgreSQL, Prometheus, and many more [98].

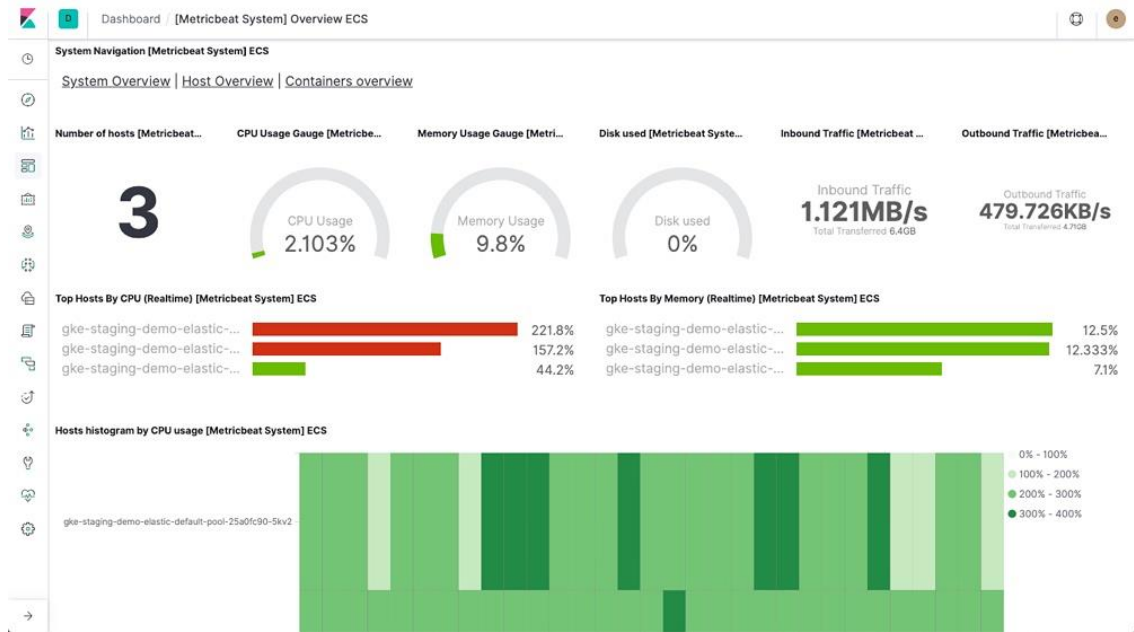


Figure 37 - Example of metrics collected into a dashboard on Kibana

Metricbeat was adopted to collect, visualize, and analyze the entire infrastructure metrics (i.e., system, API Gateway, databases, and the Elastic Stack).

5.5.4 Heartbeat

Heartbeat is another data shipper from Beats that specifically collects information on systems availability by periodically checking their status with a ping to check if the services that are load-balanced are all available. It can also monitor with echo requests to check if a service is available, with TCP to verify if an endpoint is sending and/or received a custom payment, or with HTTP to verify if the services return the expected response (i.e., status code, response header, or body). It also allows to verify if systems are acting according to their expected service level agreement (SLA) levels [99].

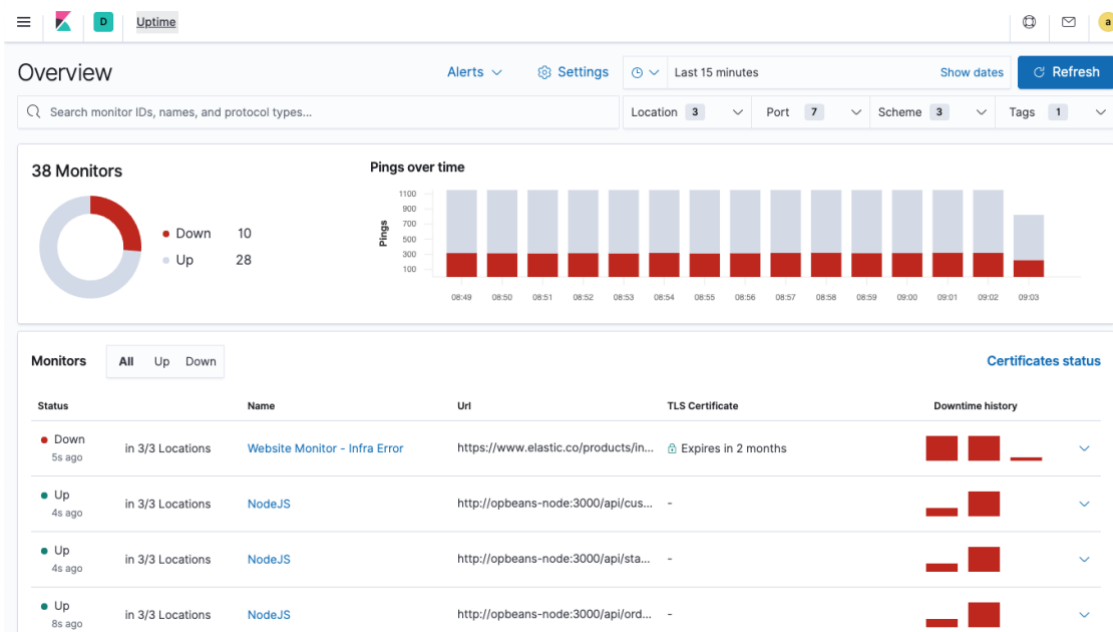


Figure 38 - Example of uptime monitoring with information collected by Heartbeat on Kibana

Heartbeat was adopted to collect data on the system's and API Gateway's availability by pinging them periodically.

5.6 Summary

This chapter presented and described:

- What were the adopted technologies for the development of the system and its surrounding infrastructure accompanied by figures for better comprehension of some of these as well as the justification of their choosing by the student.

6 Implementation

This chapter presents what technologies were adopted to develop the system, describes particular development situations, and how the overall infrastructure was set up.

6.1 Particular situations

This section presents particular situations relevant to the developed system and the surrounding infrastructure (i.e., Elastic Stack) where the implementation process and applied technologies are described accompanied by code snippets for better comprehension.

6.1.1 Service Discovery

Setting up and running service discovery on the system was a relevant situation for the implementation of the system. It's what enables the system's services to communicate with each other and with the API Gateway for handling external requests, allowing the fulfillment of all use cases.

As explained in section 5.3.1, Consul was employed for this role. A Consul image was used to run instances of a Consul server to allow for the system and the API Gateway to register on and query its registry to locate each other's instances for further interaction amongst them. How those instances were set up and started is presented in section 6.2.

After starting the Consul server it's necessary for the system's services and API Gateway to register in its registry. For that process to happen Steeltoe's Consul service discovery library [100] (i.e., NuGet package [95]) was added for them to become clients of the Consul server. To use this functionality, it's necessary to add configurations to the system's services and API Gateway settings file (cf. Snippet 1) and then setup the discovery client through the startup process (cf. Snippet 2).

```
{ (...)  
  "Consul": {  
    "Host": "localhost",  
    "Discovery": {  
      "HealthCheckUrl": "/health",  
      "ServiceName": "fakebank-user-service",  
      "IpAddress": "localhost",  
      "PreferIpAddress": true,  
      "Port": 8100  
    }  
  }, (...)  
}
```

Snippet 1 - Consul configuration on the User Service's appsettings file

Analyzing the Consul configurations above, in a JSON format, indicates us that:

- **Host:** where the Consul server is hosted (i.e., it's address) for the client to interact with;
- **Discovery.HealthCheckUrl:** the endpoint/path that the Consul server must ping to perform health checks to the service;
- **Discovery.ServiceName:** the name that client wants to be known as and registered under and to be aggregated with in the case of more instances;
- **Discovery.IpAddress:** the client's address. This is an optional setting but necessary for local development of the service when the Consul server is running in a container and needs to reach the service outside the docker network;
- **Discovery.PreferIpAddress:** indicates if the clients should be reached by its IP address or its hostname. It is set as false by default but was set to true in all services;
- **Discovery.Port:** the port exposed by the client that is to be registered.

Besides these configurations there are many more for other purposes such as tags to be associated with the instance(s), or what scheme should be registered (i.e., HTTP or HTTP/S) [101].

```

namespace Fakebank.Finance.User.Presentation.Service
{
    (...)

    using Steeltoe.Discovery.Client;
    using Steeltoe.Discovery.Consul;

    public class Program
    {
        (...)

        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        private static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                (...)
                .AddServiceDiscovery(options => options.UseConsul());
    }
}

```

Snippet 2 - Service Discovery set up on the User Service's Program class

Analyzing the Program class above indicates to us that:

- The Steeltoe service discovery packages were imported and are necessary to be able to add the service discovery functionality to the service (i.e., using block);
- The functionality is added by extending the host builder method with the AddServiceDiscovery extension at the end with the option to specify that it's a Consul client;
- The configurations that were presented in the previous are automatically loaded by the host builder and passed through to the Consul client.

With the system's services and API Gateway configured and set up they can register on the Consul server's registry and can now query it and begin interacting with each other. Steeltoe offers that functionality as well through a client interface that is registered with the service container that is inject in a HTTP client handler that intercept requests and evaluates the URL to see if the host portion of the URL can be resolved by Consul's service registry [102]. The following snippet (cf. Snippet 3) presents how this is implemented in the User Service.

```

namespace Fakebank.Finance.User.Application.Service.Clients.REST
{(...)
    using Steeltoe.Common.Discovery;
    using Steeltoe.Discovery;

    public class CoreServiceClient : ICoreServiceClient
    {
        private readonly HttpClient HttpClient;
        public CoreServiceClient(IDiscoveryClient client)
        {(...)
            var handler = new DiscoveryHttpClientHandler(client);
            this.HttpClient = new HttpClient(handler, false);
        }

        public async Task<UserResponse?> GetUser(string identification)
        {
            try
            {
                var response = await this.HttpClient
                    .GetAsync($"http://fakebank-core-
service/api/v1/User/{identification}")
                    .ConfigureAwait(false);
                (...)
            }
        }
    }
}

```

Snippet 3 – Implementation of discovery client and handler to interact with the core service

Analyzing the Core Service Client class above indicates to us that:

- The Steeltoe service discovery packages were imported and are necessary to be able to add the client and handler functionality to the service (i.e., using block);
- The client is injected through the constructor which is then passed as an argument to instantiate the discovery client handler, and then is passed as an argument to serve as the HTTP Client's handler with a false as a second argument that lets the HTTP Client know to not dispose of the handler so that it can be reused;

- With the HTTP Client using the discovery handler, it can now perform requests to other services by providing their name as the host since the Consul's service registry will translate it into an address so that the client can send its requests.

6.1.2 Logging, tracing, and metrics collection

How logs, traces, and metrics were collected is quite the relevant particular implementation situation since it's this data that is necessary for us to create machine learning jobs on Kibana and to analyze, learn, predict, and receive alerts of the system on near real-time these are trained and process data on a continual basis as data is collected.

As explained in sections 5.2 and 5.5, the system's and API Gateway's logs and traces are collected by being sent from the adopted logging library – Serilog –, complemented by the Elastic Common Schema (ECS) to format the logs, to the Elasticsearch server, and the Elastic APM agent to the APM Server that handles the tracing data before sending it to the Elasticsearch server as well. The metrics are collected from the infrastructure on which they run (i.e., docker) by Metricbeat which also stores them in the Elasticsearch server. This reinforces the adoption of the monitoring and logging method of the Secure-by-Design approach.

6.1.2.1 Log collection

To use Serilog with the ECS and its Elasticsearch sink it is necessary to add the necessary configurations to the system's services and API gateway on their host builder in the Program class. This is presented in the following snippet (cf. Snippet 4).


```

namespace Fakebank.Finance.User.Presentation.Service {(...)
    using Elastic.CommonSchema.Serilog;
    using Serilog;
    using Serilog.Filters;
    using Serilog.Sinks.Elasticsearch;
    public class Program {(...)
        private static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)(...)
                .UseSerilog((context, services, configuration) => {
                    var elasticAddress =
Environment.GetEnvironmentVariable("ELASTIC_ADDRESS");
                    var elasticUri = new Uri(elasticAddress);
                    var httpAccessor = context.Configuration.Get<HttpContextAccessor>();
                    var textFormatterConfiguration = new EcsTextFormatterConfiguration {
                        MapCustom = (document, @event) => {
                            document.Event!.Dataset = ServiceName;
                            return document;});
                    configuration.ReadFrom.Services(services)
                        .Enrich.FromLogContext()
                        .Enrich.WithEcsHttpContext(httpAccessor!)
                        .Filter.ByExcluding(Matching.WithProperty<string>("RequestPath", v
=>
                            "/health".Equals(v, StringComparison.InvariantCultureIgnoreCase)
|| "/actuator/prometheus".Equals(v,
StringComparison.InvariantCultureIgnoreCase)))
                        .WriteTo.Console()
                        .WriteTo.Elasticsearch(new ElasticsearchSinkOptions(elasticUri) {
                            CustomFormatter = new
EcsTextFormatter(textFormatterConfiguration),
                            IndexFormat = ServiceName,
                            ModifyConnectionSettings = connectionConfiguration =>

```

Snippet 4 - Serilog configuration on host builder

Analyzing the Program class and host builder above indicates to us that:

- The necessary Serilog and ECS packages were imported (i.e., using block);
- Serilog is added through the AddSerilog extension method for the host builder and is configured with the following:
 - Group of variables that contain Elasticsearch server address, the HTTP Accessor to enrich the logs with HTTP Context, and the text formatter configuration to map a custom field (i.e., Event.Dataset) with the service name that is necessary for log streaming and machine learning purposes. These are to be used in the configuration of Serilog;
 - Serilog's configuration, which reads from the services configuration to see if there are options that were added in the Startup class, enriches logs with log context, and HTTP context with the previous HTTP Accessor, filters logs to exclude from two specific paths as these are quite frequent and "pollute" the logs, writes the logs to the console so that these can be observed in the console while in running locally from the IDE or from the container logs, locally or remotely, and writes logs to Elasticsearch with the previous address, custom text formatter configuration, to the index name with the service's name and basic connection configuration.

Serilog can also be configured from the appsettings configuration files as well, in which case it would only need to read from the configuration and not have an extensive code-based configuration that is presented above.

The logs can then be consulted by Elasticsearch index on Kibana through its Discover feature on the Analytics tab. An example of consulting the user service can be observed in Figure 39.

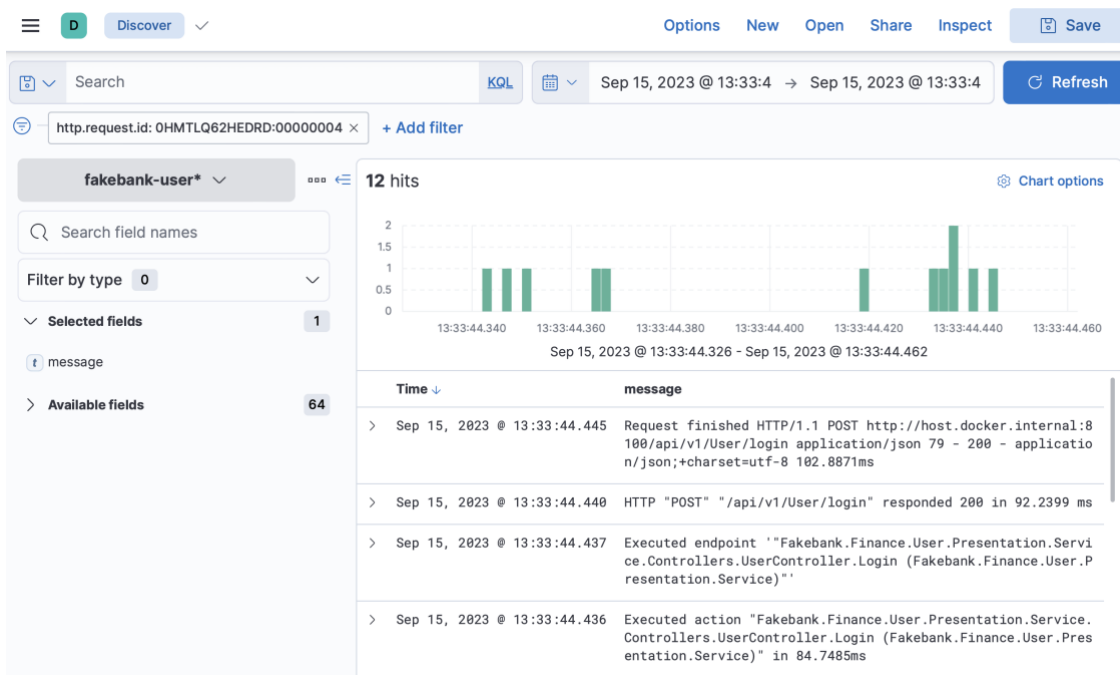


Figure 39 - Consulting User Service logs in the Discover feature on Kibana

6.1.2.2 Traces collection

To configure and set up the APM agent in the service the first thing to do is to add its configuration to the appsettings file (cf. Snippet 5), and then add it to the application builder in the Startup class on the Configure method (cf. Snippet 6).

```
{ (...)
  "ElasticApm": {
    "ServerUrl": "http://localhost:7900",
    "TransactionSampleRate": 1.0,
    "ServiceName": "fakebank-user-service"
  }, (...)
}
```

Snippet 5 - Elastic APM agent configuration in the appsettings file

Analyzing the configuration above indicates to us that:

- **ServerUrl**: the APM Server's address, where the agent will send the collected traces and associated metrics;
- **TransactionSampleRate**: the rate/percentage of traces to be sampled. The value is set between 0.0 and 1.0 (i.e., 0% to 100%);
- **ServiceName**: the service's name to be associated with the collected traces.

```

namespace Fakebank.Finance.User.Presentation.Service
{
    (...)
    using Elastic.Apm.NetCoreAll;

    public class Startup
    {
        (...)
        public void Configure(IApplicationBuilder app, IHostEnvironment env)
        {
            (...)
            app.UseAllElasticApm(this.Configuration);
            (...)
        }
        (...)
    }
}

```

Snippet 6 - Elastic APM agent usage in the Startup class

Analyzing the snippet above indicates to us that:

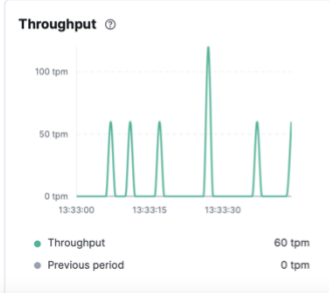
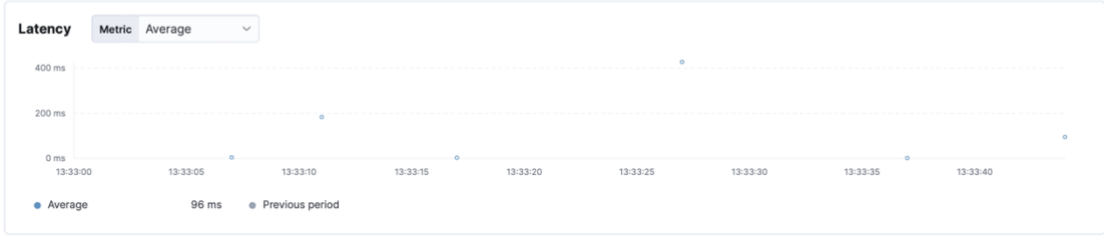
- The Elastic APM agent package is imported (i.e., using);
- The APM agent is added to the application builder through the UseAllElasticApm extension method, where the previously presented configuration is passed as an argument to be loaded so that when the application starts it can begin collecting and sending traces and its associated metrics to the APM server.

With the APM agent collecting and sending tracing data it is now possible to consult this information through Kibana with different visualizations and information. Figures 30 through 33 present some of these. It is possible to consult:

- Overview of the collected data for the service (cf. Figure 40);
- Transaction-specific information, that can be segregated by request type (cf. Figure 41);
- System metrics that are collected at the time of the traces (cf. Figure 42);
- Service map that displays the transaction flow between the different services (cf. Figure 43);
- And more.

request Search transactions, errors and metrics (E.g. transaction.duration.us > 300000 AND http.response.status_code >= 400)

Comparison Day before Sep 15, 2023 @ 13:33:44.32 → Sep 15, 2023 @ 13:33:44.46



Name	Latency (avg.)	Throughput	Failed transaction rate	Impact ↓
POST User/Login	375 ms	4.0 tpm	0%	<div style="width: 100%;"></div>
GET /health	5.7 ms	5.4 tpm	0%	<div style="width: 100%;"></div>

[View transactions](#)

< 1 >

Figure 40 – User service APM overview tab

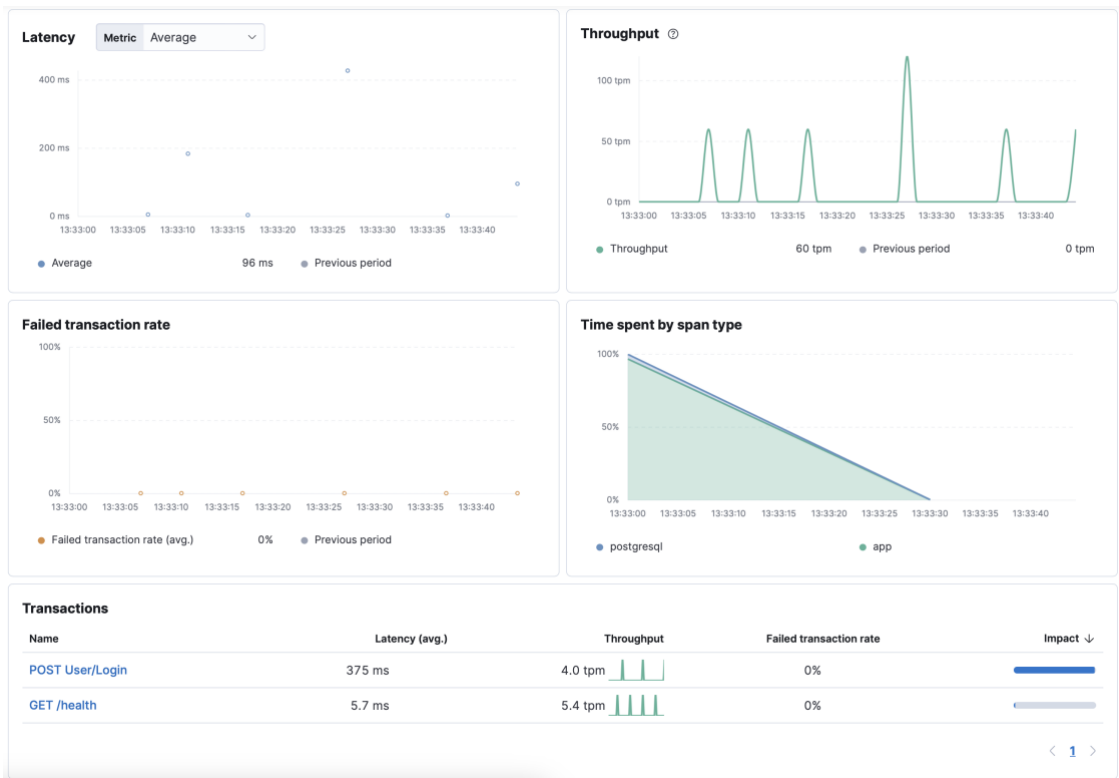


Figure 41 - User service APM transactions tab

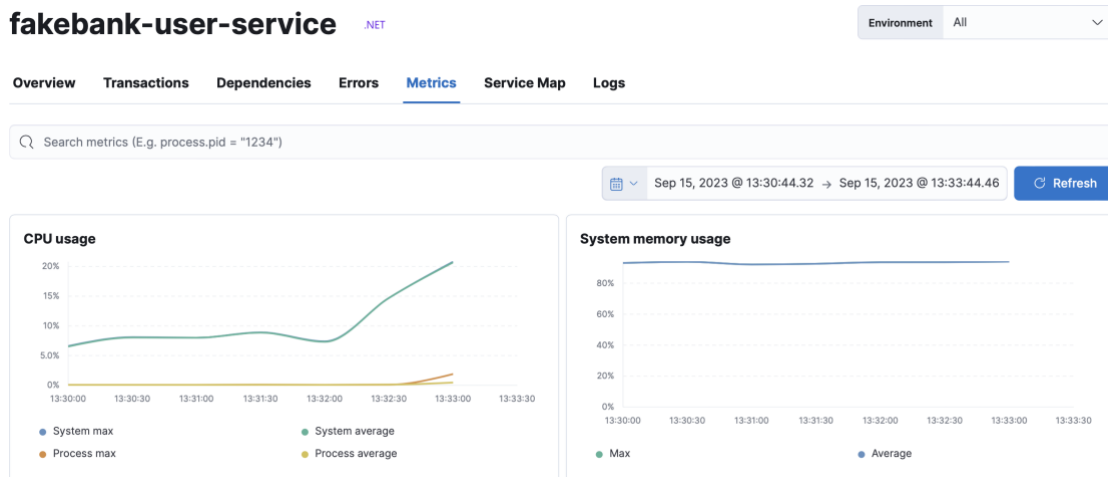


Figure 42 - User service APM metrics tab

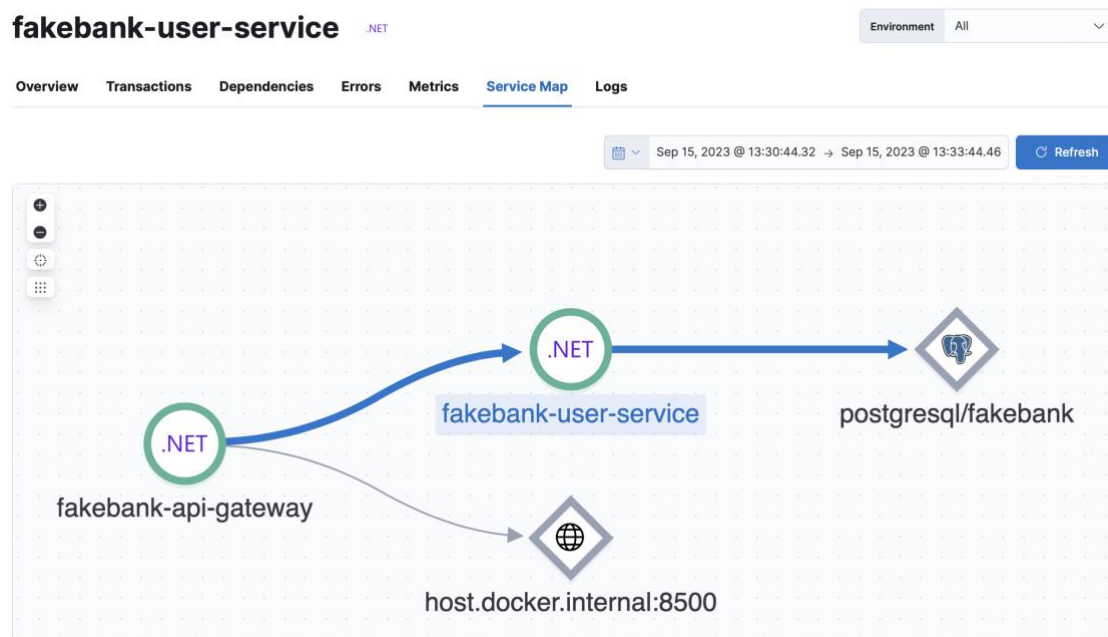


Figure 43 - User service APM service map tab

6.1.2.3 Metrics collection

How metrics are collected is different from the implementations above since it is not “implemented” into the services. It is a separate Metricbeat instance that looks at docker, collects a diverse set of metrics, and indexes these into Elasticsearch. The difference between how Metricbeat and the APM agent collect this type of information is that Metricbeat collects periodically while the APM agent only collects when there are transactions occurring.

To configure Metricbeat it's necessary to create a YAML [96] file with specific settings (cf. Snippet 7) that is copied to the image, in a Dockerfile, that will be used to raise a container. How and where the container is raised will be presented in section 6.2.

```
output.elasticsearch:
  hosts: ["http://elasticsearch:9200"]
  username: "elastic"
  password: "elastic"
setup.kibana:
  host: "http://kibana:5601"
  username: "elastic"
  password: "elastic"
metricbeat.autodiscover:
  providers:
    - type: docker
      hints.enabled: true
metricbeat.modules:
- module: docker
  metricsets:
    - "container"
    - "cpu"
    - "diskio"
    - "healthcheck"
    - "info"
    - "memory"
    - "network"
  hosts: ["unix:///var/run/docker.sock"]
  period: 10s
  enabled: true
```

Snippet 7 - Metricbeat.yml file

Analyzing the YAML file by setting indicates to us that:

- **Output.Elasticsearch:** where the Elasticsearch server is, and the authentication needed for Metricbeat to send its data;

- **Setup.Kibana:** where the Kibana instance is, and the authentication needed for Metricbeat to load its Kibana dashboards via Kibana API;
- **Metricbeat.Autodiscover:** a feature that allows tracking containers as they are started and shut down since these are “moving targets” for the monitoring system. It uses providers to know where to look, in this case, Docker.
- **Metricbeat.Modules:** modules are a large collection of known applications/services that Metricbeat has dashboards for and a set of metrics that it can collect (i.e., Azure, Kafka, PostgreSQL, etc.). The docker module is the only one enabled and it is configured to collect metrics on the container, CPU, disk I/O, health check, memory, network, and general information every 10 seconds to the docker daemon that is located under the hosts configuration.

Once Metricbeat starts collecting information it can be consulted on Kibana. Metrics observability provides two overviews:

- **Inventory:** where we can consult all containers whose metrics got collected, filter these, look at specific metrics (cf. Figure 44), and have the option of consulting details of single containers (cf. Figure 45);
- **Metrics Explorer:** where we can explore more types of collected metrics over a period of time in a time graph (cf. Figure 46).

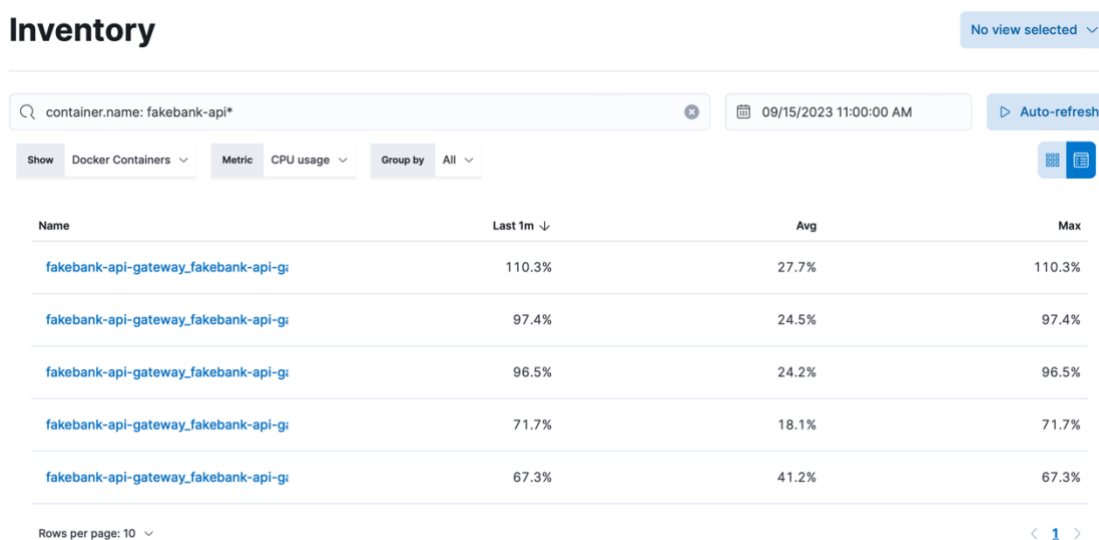


Figure 44 - Metrics Inventory overview of all the API Gateway instances CPU usage

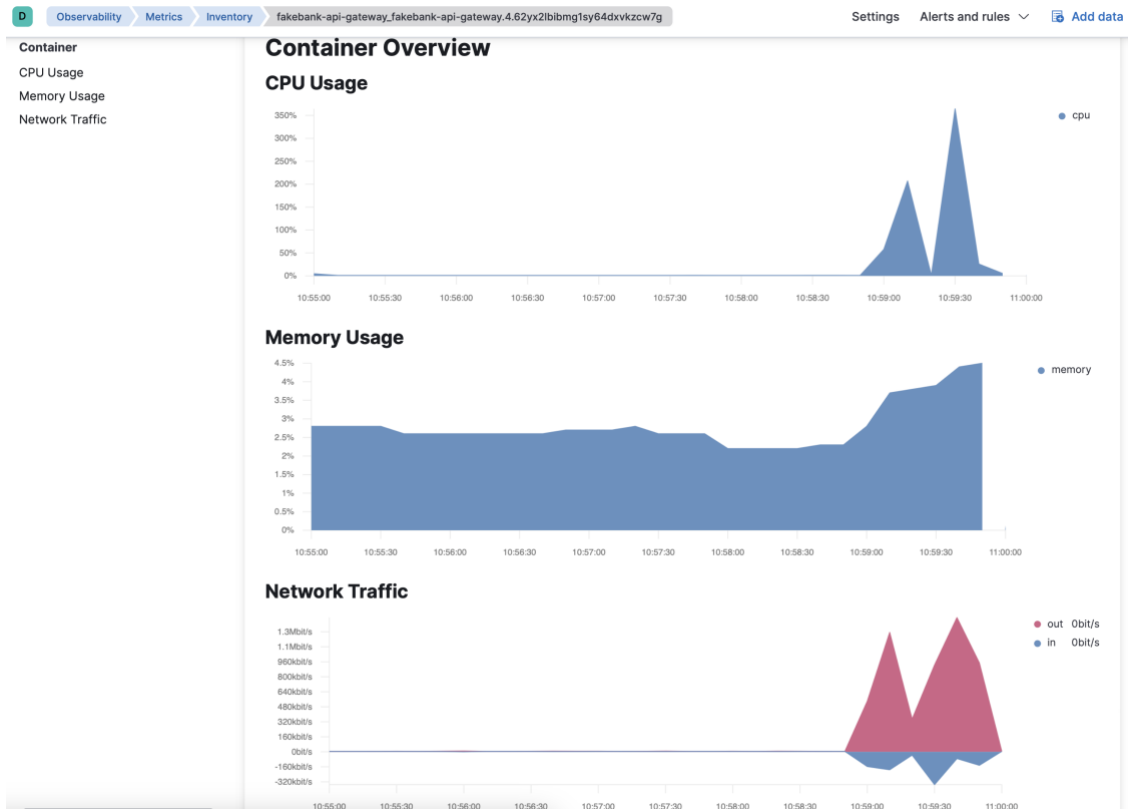


Figure 45 - Metrics Inventory overview of a selected API Gateway container metrics

Metrics Explorer



Figure 46 - Metrics Explorer overview of all the API Gateway instances average CPU usage

6.1.3 Machine learning job creation

As previously described in section 2.3.1.3, Elastic offers ML features that work seamlessly with its stack. These are anomaly detection, outlier detection, regression, and classification. These are all configurable through Kibana as presented in the next sections.

6.1.3.1 Anomaly detection

Regarding anomaly detection, it can also be easily set up from the different views on Kibana as it is integrated with them with a pre-defined configuration (i.e., ML model) for each of them.

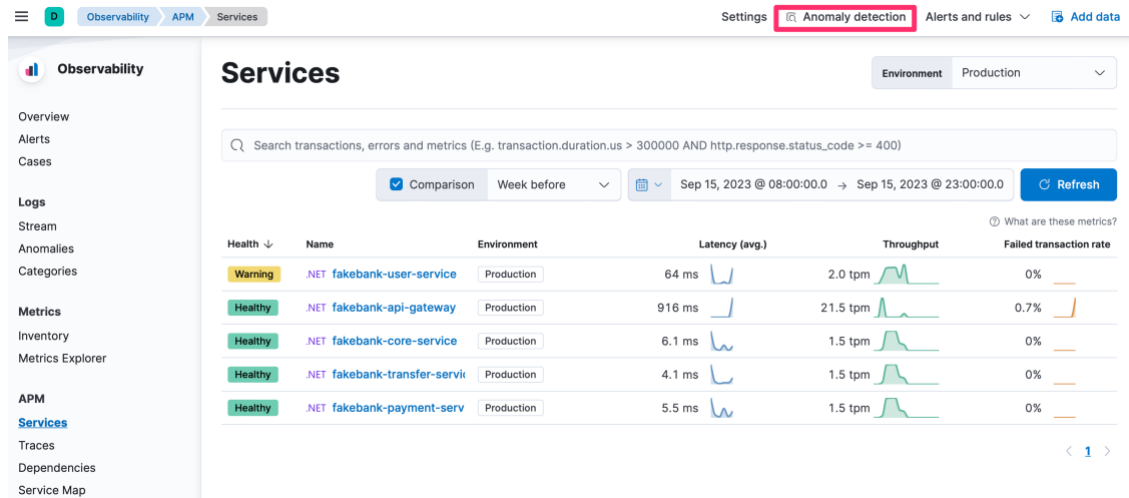


Figure 47 - Anomaly detection option in the Services view in the APM section of Observability

The view above (cf. Figure 47) presents the anomaly detection option in one of the sections of the Observability feature of Kibana. What it does when selected is that it requests the user to select one of the environments where the services run (e.g., Production, Development) and to start. Its pre-defined configuration is to analyze transactions and to alert when transactions are taking longer than expected by looking at the high mean of the transaction durations by transaction type. This job after being created can be consulted in the anomaly detection tab of the ML section of the Analytics feature as presented in Figure 48.

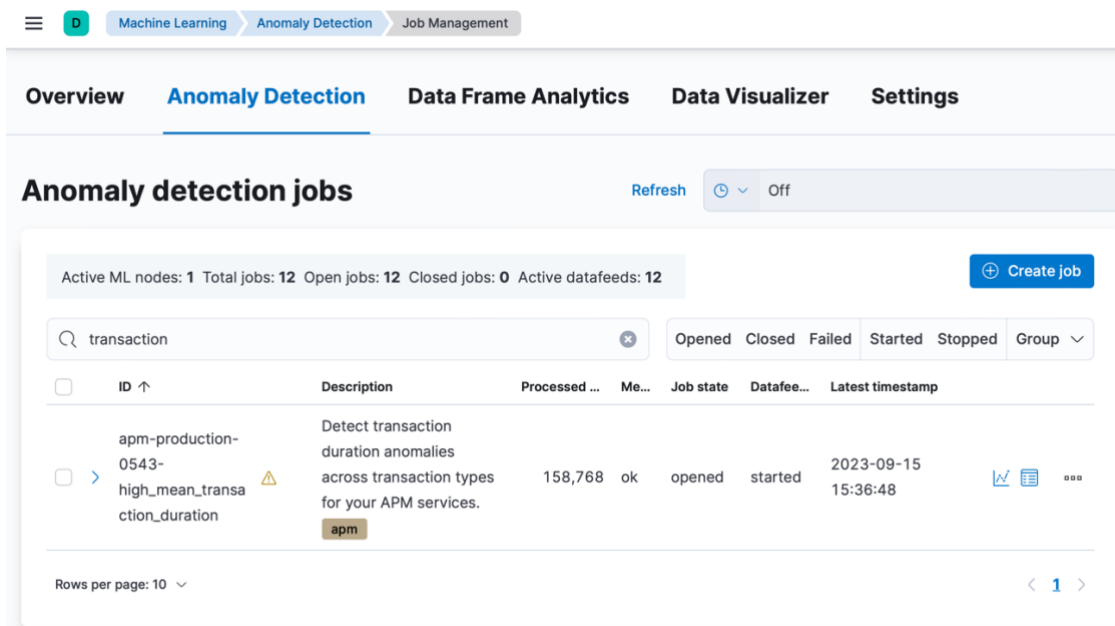


Figure 48 - Anomaly detection tab with the created ML job

Regarding the other types of default anomaly detection jobs:

- In the case of metrics, its default configuration is to create three jobs after selecting the infrastructure (e.g., Host or Kubernetes pods). Each has the job of identifying unusual spikes in a different metric (i.e., CPU, memory, and network);
- In the case of uptime, the default configuration for each monitor is to identify periods of increased latency;
- In the case of logs, its default configuration is to detect anomalies in log entries by category and in the long entry ingestion rate.

Other than default jobs, anomaly detection jobs can also be created manually via the “Create job” option shown above. It allows the options presented in Figure 49.

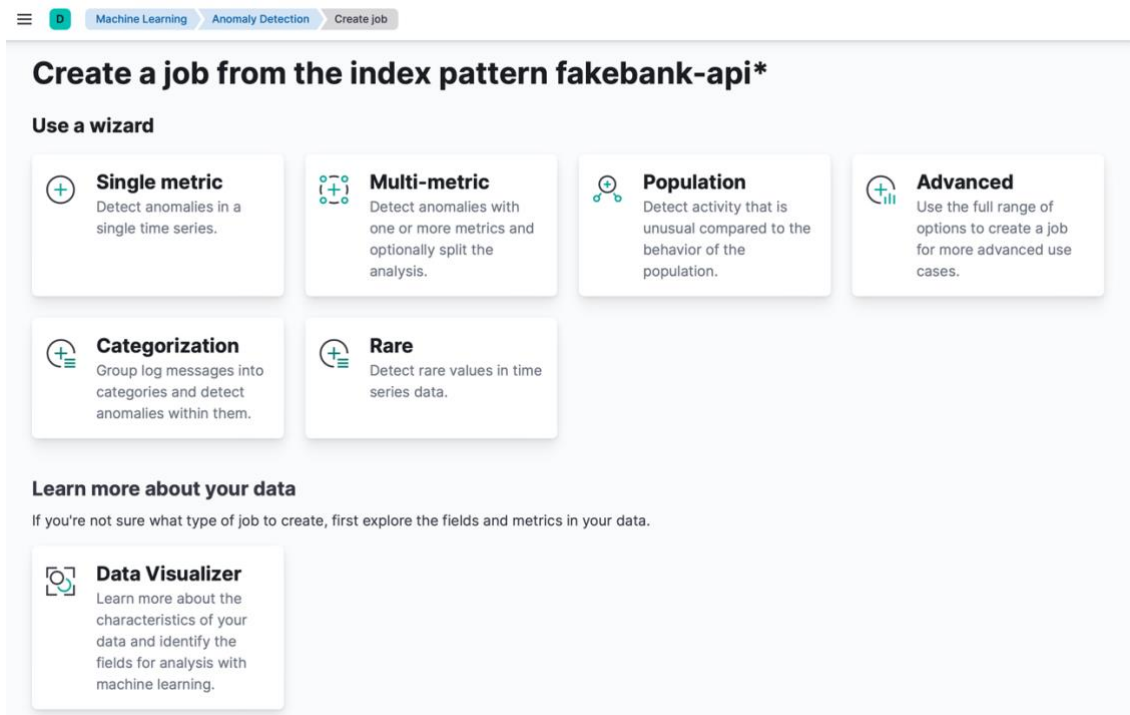


Figure 49 - Anomaly detection job creation options

As inferred, there are a lot of options regarding anomaly detection, from “simple” jobs such as Single Metric and complex jobs such as Advanced that can combine the other options and are aimed at more specific and complex use cases, and it also allows to learn more about the collected data through the Data Visualizer to identify fields for analysis with ML. Due to the wide range of options, only the Single Metric job creation is presented. Creating a new anomaly detection job is comprised of 5 steps:

1. Figure 50 **Time range**: selection of data for the job. The data could be selected from a specific time interval or all of the available data. This is presented in Figure 50;

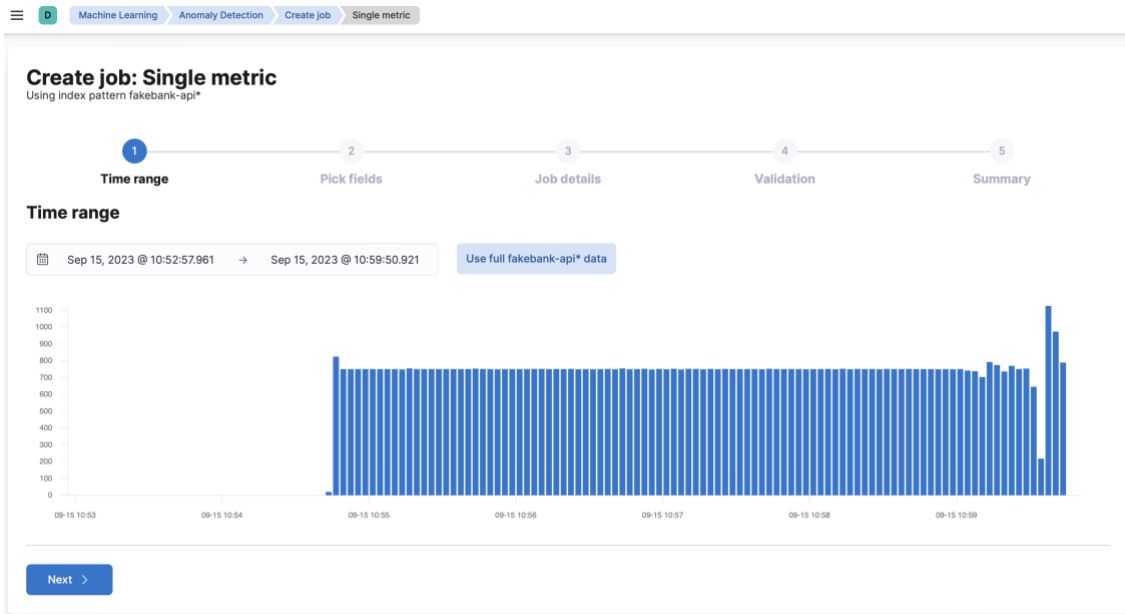


Figure 50 - Selection of time range for collected data for analysis in anomaly detection job creation

2. **Picking fields:** selection of field for analysis. A field from the data is selected for analysis along with what type of analysis (i.e., count, mean, distinct count, max, min, etc.), what the bucket span is (i.e., frequency of analysis), and if the data should be sparsed (i.e., ignore empty buckets). This is presented in Figure 51;

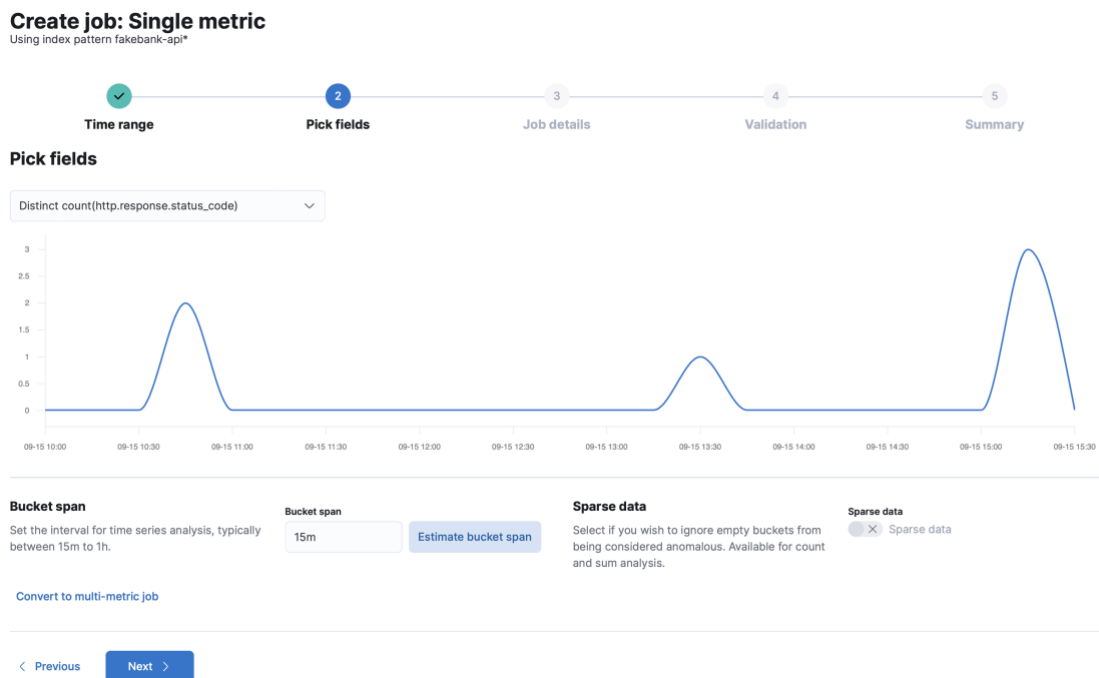


Figure 51 - Selection of field for analysis in anomaly detection job creation

- Job detail configuration:** inputting the job id, if it's associated with a group of jobs, or inputting a description. It has advanced settings such as model plotting, generating annotations when the model changes significantly, use of a dedicated Elasticsearch index, and manually configuring the memory limit for the model. This is presented in Figure 52;

The screenshot shows the 'Job details' configuration step in an anomaly detection job creation process. The interface features a progress bar at the top with five steps: 'Time range', 'Pick fields', 'Job details' (the current step, marked with a blue circle and the number 3), 'Validation', and 'Summary'. Below the progress bar, the 'Job details' section is divided into several sub-sections:

- Job ID:** A unique identifier for the job. Spaces and the characters /, ?, *, <, > are not allowed. The input field contains 'http-status-code-distinct-count-api-gateway'.
- Job description:** Optional descriptive text. The input field is empty.
- Groups:** Optional grouping for jobs. New groups can be created or picked from the list of existing groups. The dropdown menu shows 'Select or create groups'.
- Advanced settings:**
 - Enable model plot:** Select to store additional model information used for plotting model bounds. This will add overhead to the performance of the system and is not recommended for high cardinality data. The toggle is turned on.
 - Enable model change annotations:** Select to generate annotations when the model changes significantly. For example, when step changes, periodicity or trends are detected. The toggle is turned on.
 - Use dedicated index:** Store results in a separate index for this job. The toggle is turned off.
 - Model memory limit:** Set an approximate upper limit for the amount of memory that can be used by the analytical models. The input field contains '11MB'.

Figure 52 - Job details configuration of in anomaly detection job creation

- Validation:** the previous steps are validated by Kibana and indicate if the selections are okay for the job or if there is any with something that could be improved. This is presented in Figure 53;

The screenshot shows the 'Validation' step in an anomaly detection job creation process. The interface features a progress bar at the top with five steps: 'Time range', 'Pick fields', 'Job details', 'Validation' (the current step, marked with a blue circle and the number 4), and 'Summary'. Below the progress bar, the 'Validation' section displays two messages:

- Time range:** A warning message (indicated by a yellow triangle icon) stating: 'The selected or available time range might be too short. The recommended minimum time range should be at least 2 hours and 25 times the bucket span.'
- Model memory limit:** A success message (indicated by a green checkmark icon) stating: 'Valid and within the estimated model memory limit. Learn more'

At the bottom of the page, there are navigation buttons: '< Previous' and 'Next >'.

Figure 53 - Job validation in the anomaly detection job creation

- Summary:** A summary of the job is presented before finalizing the creation process along with the option for starting it right away. If this option is disabled, the job can be started afterward from the jobs list. This is presented in Figure 54.

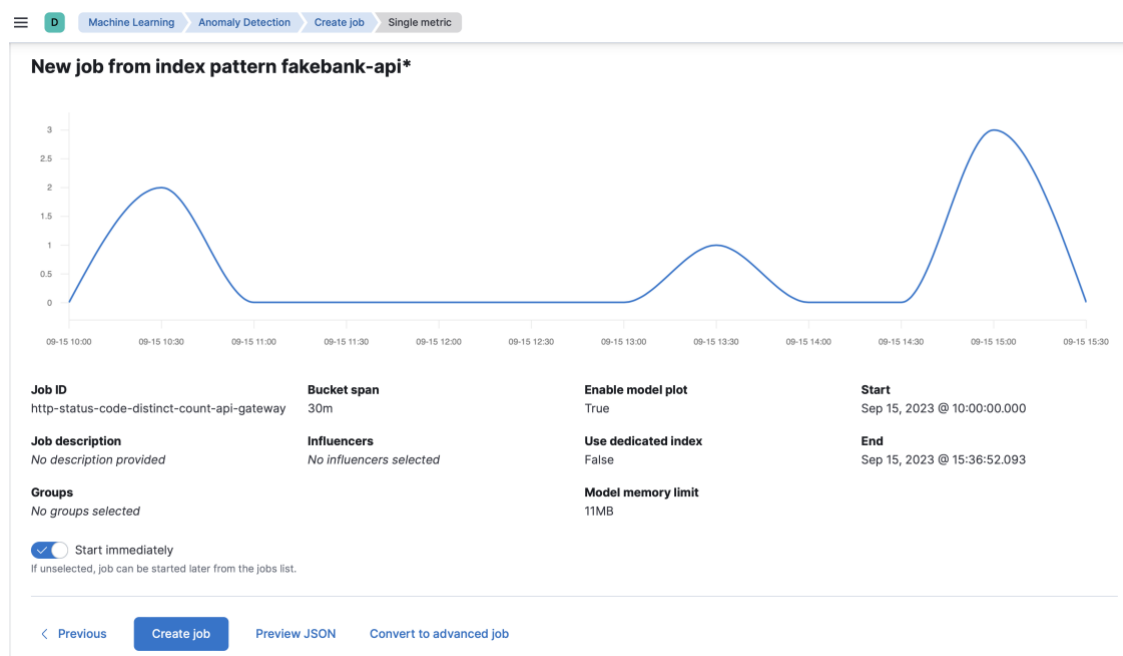


Figure 54 - Job summary in the anomaly detection job creation

6.1.3.2 Outlier detection, regression, and classification

Regarding the other three types of ML jobs, since these perform data frame analysis, they are all configured under the same job creator (i.e., data frame analytics job creator). Unlike anomaly detection, which also offers creation on Kibana views, this is the only way to create these jobs.

To create a data frame analytics job, we need to go to the data frame analytics tab that is presented in Figure 48 and select the "Create job" button. After selecting which Elasticsearch index to use, the next choice falls on the type of job we want (i.e., outlier detection, regression, or classification). Since these are all similar in configuration, the classification job creation is presented.

Creating a data frame analytics job is comprised of five steps:

- Configuration:** selection of job type, filtering of data by query if necessary, choosing of the dependent variable (i.e., field for classification prediction), which other fields to include for analysis, and what percentage of the data should be used for training the model. This is presented in Figure 55 and Figure 56;

Create job

Source index pattern: fakebank-api*

Switch to json editor

1 Configuration

Outlier detection

Outlier detection identifies unusual data points in the data set.

Select

Regression

Regression predicts numerical values in the data set.

Select

Classification

Classification predicts classes of data points in the data set.

Selected

Query

event.duration > 0 and metadata.ContentLength > 0

KQL

Runtime fields

No runtime field

Edit runtime fields

fakebank-api*

@timestamp	event.created	event.dataset	event.duration	event.severity	http.response.status_...	metadata.Conte
Sep 15, 2023 @ 15:24:5...	Sep 15, 2023 @ 15:24:5...	fakebank-api-gateway	8,767,900	2	2	400
Sep 15, 2023 @ 15:24:5...	Sep 15, 2023 @ 15:24:5...	fakebank-api-gateway	10,638,800	2	2	400
Sep 15, 2023 @ 15:24:5...	Sep 15, 2023 @ 15:24:5...	fakebank-api-gateway	11,967,200	2	2	400
Sep 15, 2023 @ 15:24:5...	Sep 15, 2023 @ 15:24:5...	fakebank-api-gateway	11,185,500	2	2	400
Sep 15, 2023 @ 15:24:5...	Sep 15, 2023 @ 15:24:5...	fakebank-api-gateway	9,185,200	2	2	400

Rows per page: 5

< 1 2 3 4 5 ... 2000 >

Dependent variable

http.response.status_code

Figure 55 - Data frame analytics job creation

Included fields
5 fields included in the analysis

is_included:true Is included Is not included

<input checked="" type="checkbox"/> Field name	Mapping	Is included	Is required	Reason
<input checked="" type="checkbox"/> event.duration	long	Yes	No	
<input checked="" type="checkbox"/> http.request.method.keyword	keyword	Yes	No	
<input checked="" type="checkbox"/> http.response.status_code	long	Yes	Yes	
<input checked="" type="checkbox"/> metadata.ContentLength	long	Yes	No	
<input checked="" type="checkbox"/> url.path.keyword	keyword	Yes	No	

Rows per page: 50 < 1 >

Scatterplot matrix
Visualizes the relationships between pairs of selected included fields.

Fields Sample size Random scoring Off

Training percent
1 100
Defines the percentage of eligible documents that will be used for training.

[Continue](#)

2 **Additional options**

Figure 56 - Data frame analytics job creation included fields and training segment

2. **Advanced options:** further configure the model with feature importance values (i.e., which fields have the largest impact on each prediction), custom prediction field name instead of the default name that is the dependent variable, randomize seed for the random generator used for picking data, the number of categories for the predicted probabilities to report to, the model memory limit, the number of threads to be used for analysis, and hyperparameters (e.g., lambda for multiplying leaf weights is loss calculation, maximum number of decision trees, etc.). This is presented in Figure 57.

Machine Learning > Data Frame Analytics

2 Additional options

Advanced configuration

Feature importance values
 Specify the maximum number of feature importance values per document to return.

Prediction field name
 Defines the name of the prediction field in the results. Defaults to <dependent_variable>_prediction.

Randomize seed
 The seed for the random generator used to pick training data.

Top classes
 The number of categories for which the predicted probabilities are reported. If you have a large number of classes there could be a significant effect on the size of your destination index.

Model memory limit
 Use estimated model memory limit
The approximate maximum amount of memory resources that are permitted for analytical processing.

Maximum number of threads
 The maximum number of threads to be used by the analysis. The default value is 1.

> **Hyperparameters**

Figure 57 - Data frame analytics job creation additional options segment

- 3. Job details:** to input the job id, description, and other options such as the destination index have the same name as the job id, or the creation of an index pattern. This is presented in Figure 58;

Machine Learning > Data Frame Analytics

3 Job details

Job ID

Job description

Destination index same as job ID

Use results field default value "ml"

Create index pattern

Figure 58 - Data frame analytics job creation job details segment

- 4. Validation:** the previous steps are validated by Kibana and indicate if the selections are okay for the job or if there is any with something that could be improved. This is presented in Figure 59;



Figure 59 - Data frame analytics job creation validation segment

5. **Create:** all that's left is to create the model and start the job. It can be started immediately or afterward from the jobs list. This is presented in Figure 60.

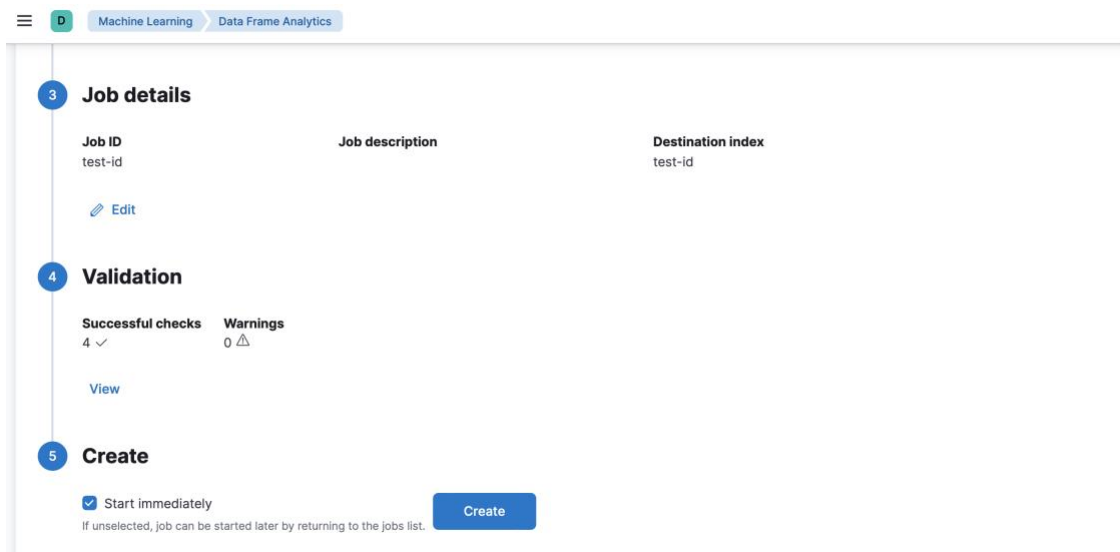


Figure 60 - Data frame job creation create segment

6.2 Infrastructure setup

This section presents how the overall infrastructure where the system and other services, particularly the Elastic Stack, ran was set up accompanied by code snippets for better comprehension. The other infrastructure components (i.e., PostgreSQL Server, SQL Server, and Consul) were started from available official images with minimal configuration as extensive/specific configurations were not necessary.

6.2.1 System

From what can be inferred from the previous chapter, Docker was used to run instances of the system on containers. For each component (i.e., service) of the system a Dockerfile and a docker-compose file were created to build an image and to start containers. All Dockerfiles and docker-compose files are similar, so the ones presented in this section represent all. The Dockerfile is presented in segments to explain the image-building process.

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build_env
WORKDIR /app

# Copy csproj and restore
COPY *.sln ./
COPY Fakebank.Finance.Core.Application.Service/*.csproj
  ./Fakebank.Finance.Core.Application.Service/
COPY Fakebank.Finance.Core.Data.Model/*.csproj
  ./Fakebank.Finance.Core.Data.Model/
COPY Fakebank.Finance.Core.Data.Repository/*.csproj
  ./Fakebank.Finance.Core.Data.Repository/
COPY Fakebank.Finance.Core.Domain.Model/*.csproj
  ./Fakebank.Finance.Core.Domain.Model/
COPY Fakebank.Finance.Core.Infrastructure.CrossCutting/*.csproj
  ./Fakebank.Finance.Core.Infrastructure.CrossCutting/
COPY Fakebank.Finance.Core.Presentation.Service/*.csproj
  ./Fakebank.Finance.Core.Presentation.Service/

RUN dotnet dev-certs https
RUN dotnet restore
```

Snippet 8 - Dockerfile build environment restore segment

The above segment can be summarized in 3 steps: Defines the .NET SDK [97] 7.0 image to be used as the build environment and the work directory where the process will occur; Copies all the solution's projects .csproj files. These declare each project's dependencies; Run a couple of commands. First to generate a self-signed certificate to enable HTTPS and second to restore the dependencies of each project.

```

# Copy everything else and build
COPY Fakebank.Finance.Core.Application.Service/.
./Fakebank.Finance.Core.Application.Service/
COPY Fakebank.Finance.Core.Data.Model/. ./Fakebank.Finance.Core.Data.Model/
COPY Fakebank.Finance.Core.Data.Repository/.
./Fakebank.Finance.Core.Data.Repository/
COPY Fakebank.Finance.Core.Domain.Model/.
./Fakebank.Finance.Core.Domain.Model/
COPY Fakebank.Finance.Core.Infrastructure.CrossCutting/.
./Fakebank.Finance.Core.Infrastructure.CrossCutting/
COPY Fakebank.Finance.Core.Presentation.Service/.
./Fakebank.Finance.Core.Presentation.Service/

WORKDIR /app/Fakebank.Finance.Core.Presentation.Service
RUN dotnet publish -c Release -o out

```

Snippet 9 - Dockerfile build environment build segment

After restoring the solutions dependencies, it copies everything else, then changes the working directory to the presentation service project (i.e., the solutions main project) and runs the publish command with the release flag. This means that it will output a production-ready application.

```

# Sets entrypoint
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS runtime
WORKDIR /app

COPY --from=build_env /app/Fakebank.Finance.Core.Presentation.Service/out ./
COPY --from=build_env /root/.dotnet/corefx/cryptography/x509stores/my/*
/root/.dotnet/corefx/cryptography/x509stores/my/
EXPOSE 8000
ENTRYPOINT [ "dotnet", "Fakebank.Finance.Core.Presentation.Service.dll" ]

```

Snippet 10 - Dockerfile runtime environment entry-point segment

Finally, it moves to the runtime environment where it defines the aspnet 7.0 image, a smaller optimized runtime image aimed at running .NET application in a production environment,

changes the working directory to the initial one, copies the generated output from the previous segment to the current directory and the generated self-signed certificate to a specific directory, exposes port 8000 and sets the entry-point for the image, which is the presentation service DLL.

With the image built, the docker-compose file (cf. Snippet 11) that is used to run the image declares the following:

- Docker-compose version to use. Version 3.8 is the one used, the most recent;
- The services it will raise. It declares the Core Service by specifying:
 - **Image:** the latest image of the Core Service;
 - **Container name:** The name to assign to the container;
 - **Build:** Where to build the image if it hasn't been built. In this case, the same directory as the docker-compose file and the Dockerfile to run;
 - **Ports:** What ports to expose externally and the internal port to connect to;
 - **The network mode:** specifies bridge, which is the docker host network, meaning it can interact with other containers in the docker network;
 - **Environment variables:** specifies values to inject/override in the image's settings such as the connection string to the SQL Server.

```

version: "3.8"
services:
  fakebank-core-service:
    image: tmdei1171245/fakebank-core-service:latest
    container_name: fakebank-core-service
    build:
      context: .
      dockerfile: ./Dockerfile
    ports:
      - "8000:8000"
    network_mode: bridge
    environment:
      - ELASTIC_ADDRESS=http://host.docker.internal:9200
      - ElasticApm__ServerUrl=http://host.docker.internal:7900
      - JwtOptions__Issuer=host.docker.internal
      - JwtOptions__Audience=host.docker.internal
      - Consul__Host=host.docker.internal
      - Consul__Discovery__IpAddress=host.docker.internal
      - Consul__Discovery__PreferIpAddress=true
      - Consul__Discovery__Port=8000
      - ConnectionStrings__SqlServer=(...)

```

Snippet 11 - Docker-compose file

6.2.2 Elastic stack

Regarding the Elastic Stack, each of its services requires specific configurations. For that, a YAML file and Dockerfile were created for each. Finally, a docker-compose file where these were all declared was built.

6.2.2.1 Services' configuration

Metricbeat's YAML file was already presented in section 6.1.2.3 so it's not presented here. Snippets 12 through 15 present the remaining services' YAML files.

```

cluster.name: "docker-cluster"
network.host: 0.0.0.0
xpack.license.self_generated.type: basic
xpack.security.enabled: true
xpack.security.authc.api_key.enabled: true
xpack.monitoring.collection.enabled: true
http.cors.enabled: true
http.cors.allow-origin: "*"
http.cors.allow-headers: X-Requested-With,Content-Type,Content-
Length,Authorization

```

Snippet 12 - Elasticsearch YAML file

Observing the Elasticsearch configuration, it defines the cluster name, and then the address to bind itself to, configures the license to basic that allows to enable basic X-Pack [98] features, enables security and authentication with API Key, as well as monitoring data collection of Elasticsearch. Finally, it enables CORS [99] so that Elasticsearch can be consulted via browser.

```

server.name: kibana
server.host: "0"
elasticsearch.hosts: [ "http://elasticsearch:9200" ]
elasticsearch.username: elastic
elasticsearch.password: elastic
xpack.monitoring.ui.container.elasticsearch.enabled: true
xpack.reporting.encryptionKey: "<key>"
xpack.encryptedSavedObjects.encryptionKey: "<key>"

```

Snippet 13 - Kibana YAML file

Observing the Kibana configuration, it defines the server's name, the server host/address to bind itself to, which Elasticsearch hosts to query – in this case, the local instance – and its authentication information, which enables Elasticsearch monitoring that is specific to instances running in containers and sets the encryption keys that are necessary for reporting and saving encrypted objects (i.e., dashboards, visualizations, alerts, actions, and advanced settings) in a dedicated, internal Elasticsearch index.


```
apm-server:
  host: "0.0.0.0:7900"
ilm:
  enabled: "auto"
  setup:
    enabled: true
    overwrite: true
setup.template.enabled: true
setup.template.name: "apm-server"
setup.template.pattern: "apm-*"
output.elasticsearch:
  hosts: ["elasticsearch:9200"]
  username: elastic
  password: elastic
```

Snippet 14 - APM Server YAML file

Observing the APM Server configuration, it defines the address and port to bind itself to, enables ILM² setup and overwrite which allows the creation of unmanaged custom indices, enabled the template loading and defines the template's name and pattern (i.e., where the data will be stored). Finally, it defines the Elasticsearch host to send its data to and its authentication information.

² ILM – Index Lifecycle Management – are policies that have the responsibility to manager indices according to specific performance, resiliency, and retention requirements.

```

output.elasticsearch:
  hosts: ["http://elasticsearch:9200"]
  username: "elastic"
  password: "elastic"
setup.kibana:
  host: "http://kibana:5601"
heartbeat.monitors:
- type: http
  id: fakebank-core-service-status
  name: Fakebank-Core-Service Status
  urls: ["http://host.docker.internal:8000/health"]
  check.response.status: [200]
  schedule: "@every 10s"
(...)

```

Snippet 15 - Partial Heartbeat YAML file

Observing the Heartbeat configuration, it defines the Elasticsearch host to send its data to and its authentication information, the Kibana host to load its dashboards via Kibana API, and the monitors to execute. The snippet above only presents one monitor since the others are similar. The monitors are of HTTP type, have an ID and name for each of the system's services, ping at the health endpoint of each expecting an HTTP 200 response status, and are executed every 10 seconds.

6.2.2.2 Docker configuration

Regarding Dockerfiles, only one is presented below to represent all since they are all similar.

```

FROM elastic/elasticsearch:7.17.13
COPY --chown=elasticsearch:elasticsearch ./elasticsearch.yml
/usr/share/elasticsearch/config/

```

Snippet 16 - Elasticsearch Dockerfile

What all the Dockerfiles essentially do is declare the image and copy the YAML file to the image's configuration directory so that when a container is started it loads the correct settings.

Snippets 17 through 20 present the docker-compose network and volumes, two of the services with the most configuration (i.e., Elasticsearch and Metricbeat), and one service to represent the other three since these are all similar.

```
networks:  
  monitoring_network:  
    driver: bridge  
  
volumes:  
  elasticsearch:  
  metricbeat:
```

Snippet 17 – Docker-compose network and volume configuration

The docker-compose defines a network that operates on the bridge driver, meaning that it can interact with other services (i.e., containers) other than the ones it declares, and two volumes, one for Elasticsearch and one for Metricbeat.

```
elasticsearch:  
  build:  
    context: elasticsearch/  
  volumes:  
    - type: volume  
      source: elasticsearch  
      target: /usr/share/elasticsearch/data  
  ports:  
    - "9200:9200"  
    - "9300:9300"  
  environment:  
    ES_JAVA_OPTS: "-Xmx512m -Xms512m"  
    ELASTIC_PASSWORD: elastic  
    discovery.type: single-node  
  networks:  
    - monitoring_network
```

Snippet 18 - Elasticsearch configuration in the docker-compose file

Elasticsearch, like the other services, defines that its build/image (i.e., its Dockerfile) is in the `elasticsearch` folder located in the same directory as the `docker-compose` file, binds the previously declared volume to the images data folder, two ports, defines environment variables for its initial and maximum heap size, password, and discovery type (i.e., if Elasticsearch should be multi-node or single-node), and bind itself to the previously defined network.

```
metricbeat:
  build:
    context: metricbeat/
  user: root
  command:
    - -e
    - --strict.perms=false
    - --system.hostfs=/hostfs
  volumes:
    (...)
    - type: bind
      source: /var/run/docker.sock
      target: /var/run/docker.sock
      read_only: true
    - metricbeat:/usr/share/metricbeat/data
  ports:
    - "7700:7700"
  networks:
    - monitoring_network
  depends_on:
    - elasticsearch
    - kibana
    - apm-server
    - heartbeat
```

Snippet 19 - Metricbeat service configuration on the `docker-compose` file

Metricbeat defines its user, commands to log errors, to disable the configuration file permission checks (i.e., allows for mounting configuration that are not owned by the root), sets the

mounting point of the host's filesystem (i.e., necessary for monitoring the host from within a container), binds a set of folders related to host necessary for monitoring, not shown above, the docker daemon for monitoring all containers, the previously declared volume to its data, declares the port, binds itself to the previously defined network, and which of the declared services it depends on to start.

```
kibana:
  build:
    context: kibana/
  ports:
    - "5601:5601"
  networks:
    - monitoring_network
  depends_on:
    - elasticsearch
```

Snippet 20 - Kibana configuration on the docker-compose file

Kibana and the other services only define their respective ports, the network, and what other services they depend on (i.e., in this case, Elasticsearch).

6.3 Summary

This chapter presented and described:

- Which technologies were adopted for the implementation of the system and the surrounding infrastructure while referencing for more information about these on the Annexes;
- Particular implementation situations for the system and infrastructure with the implementation process complemented by code snippets and screenshots of Kibana;
- How the overall infrastructure was set up with code snippets.

7 Evaluation

This chapter details the process of evaluation to be conducted ensuring that the prototype falls in line with the objectives. To fulfill this, investigation hypotheses associated with the problem are formulated, the indicators and information sources are presented, the specification of the evaluation methodology used to assess the prototype, what tests were developed and the importance of Continuous Integration and Continuous Deployment/Development (CI/CD) in evaluating systems, and how the experimentation was setup. Finally, an assessment of the system is made.

7.1 Investigation hypothesis

A hypothesis is a possible answer to a research question. It is a hunch or a guess to determine “what is going on”. It is assessed for potential approval or rejection. In the case of approval, then that hunch was correct [103].

The following formulated investigation hypothesis should meet the RQs listed in section 1.3. These are:

- **H.1:** The proposed architecture offers high levels of operational resilience;
- **H.2:** The usage of AI methods in conjunction with the proposed architecture provides a great advantage in its security.

The goal of these hypotheses is to prove that the proposed microservice architecture, in this context, presents a great number of benefits when combined with the usage of AI methods to make it more robust and secure.

It is to be noted that these hypotheses are linked with the initially formulated research questions found in section 1.3: **H.1** aims to answer **RQ.1**, and **H.2** aims to answer **RQ.2**.

7.2 Indicators and information sources

From the previous section, two aspects can be taken to be evaluated with these being specific characteristics.

The two characteristics to be evaluated are the **resilience** and **security** offered by the prototype when combined with AI methods. Both aspects will be evaluated following the **Goals, Questions, Metrics** (GQM) methodology.

These aspects were chosen as indicators due to the fact of being directly related to the investigation hypotheses previously formulated.

The first, **resilience**, evaluates the overall robustness of the proposed prototype and seeks to confirm hypothesis **H.1**, while the latter, **security**, verifies how much more secure the prototype is, and seeks to confirm hypothesis **H.2**.

7.3 Goals, Questions, Metrics

The GQM methodology is a proven method for implementing goal-oriented metrics throughout a software project. It starts by defining the goals to achieve, and then clarifying the questions to answer with the data to collect. Finally, mapping business objectives and goals to data-driven metrics allows the formation of a holistic picture of the environment created [104].

The two characteristics can be associated with a few quality attributes (QA) for an architectural style (AS). These QAs are the ones that will be used by the GQM method. The QAs will be measured following the dynamic analysis technique to evaluate its metrics. This technique was established by [105] to connect QAs with measurable metrics, and it needs the execution of the software to be assessed so it can identify gaps and flaws in the software's behavior and logic.

The resilience characteristic is linked with the concept of operational resilience, presented in section 1.1.3. A resilient system is characterized by the QAs of availability, performance, and scalability, since it needs to be able to recover quickly when failing over or experiencing fault, maintaining its high availability and service at all times (i.e., handling user/external requests) with acceptable response times and the efficient use of system resources, and also to be able to scale up and down according to the system's needs (i.e., when experiencing significant increase or decrease of traffic) [105], [106]. In Table 10, the metrics of each QA related to resilience can be observed.

Table 10 - Metrics by quality attributes related to resilience [105], [106]

Quality Attribute	Metric	Description
Availability	Uptime percentage	The proportion of time the microservice is accessible within a required period. Most of the contemporary

Quality Attribute	Metric	Description
		cloud platforms Service Level Agreements (SLA) guarantee availability of 99.9999% or above
	Successful execution rate	The ability of a service provider to successfully fulfil the requests within a given period
	Fault detection	To identify or predict the occurrence of a defect before the system may take action to recover from faults. The usage of tools such as fault monitors can help in the systems immediate reaction.
	Resilience	Characterized by the microservice's capacity to cope with failures (i.e., resilience to failure). A microservice complies to this property by saving the internal state, and restarting automatically while loading the most up-to-date state prior to the failure
Performance	Response time	The anticipated delay between the time when a request to a microservice is issued and the time when the result is delivered. It only considers the execution time, excluding the network delay time. When measuring synchronous calls, it considers the longest response time, while for asynchronous calls, it considers the average time spent.
	Average CPU utilization	Average usage of CPU compared between each service
Scalability	Usage frequency	The ratio between the requests made to the assessed microservice and all the requests made in the entire system
	Horizontal/vertical scalability	The ability of a microservice to function correctly regardless of the changes in size, horizontally or vertically, without inquiring performance penalties
	Isolation	The isolation of the microservice with respect to others with which it should only communicate via the exposed interfaces

The security characteristic can be linked with the concept of cybersecurity, presented in section 1.1.1. A secure system can be characterized by the QAs of monitorability and security, since systems need to be able to generate and store logs, distributed tracing, and applicational metrics, so these can be presentable through monitoring solutions and analytics. It should also be secure against dependencies (i.e., third-party weaknesses), monitor against security threats at various levels (i.e., anomalous behavior and attacks), as well as have authentication and authorization security mechanisms implemented [106], [107]. For the context of this dissertation, the monitoring will employ the usage of AI methods and be evaluated in that regard. In Table 11, the metrics of each QA related to security can be observed.

Table 11 - Metrics by quality attributes related to security [106], [107]

Quality Attribute	Metric	Description
Monitorability	Data generation and storage	The system should be capable of generating and storing logs, distributed tracing and applicational metrics in a storage system
	Data presentation	The store data must be presentable through monitoring solutions and analytics
Security	Third-party weaknesses	The security of each dependency, which can impact the security of the service
	Security monitor	Security Monitor is a tactic that places monitor to observe abnormal behaviors or attacks of microservice applications at different levels
	Authentication and authorization	Authentication is a process by which to confirm the identity a user or a party and authorization is a mechanism by which a principal is mapped to the action allowing an identity to do

With all the metrics tabled and described the GQM can be observed in Table 12.

Table 12- Goals, questions, metrics

Characteristic	Quality Attribute	Goals	Questions
Resilience	Availability	The prototype offers high availability and fault tolerance	Indicated in Table 10
	Performance	The prototype should be able to perform under heavier workloads	Indicated in Table 10
	Scalability	The prototype should be easily scalable	Indicated in Table 10
Security	Monitorability	The prototype provides high monitorability	Indicated in Table 11
	Security	The prototype, assisted by an AI model, offers security regarding its API and data management	Indicated in Table 11

7.4 Tests

Although tests were not the focus of this dissertation, these are still important as they are vital in validating a system's behaviors at different granularities to ensure that it's functioning as expected. The classic test pyramid suggests three levels of granularity [108]. These are:

1. **Unit tests:** software is composed of a set of units with distinct functionalities that together achieve the desired result. It is imperative that these operate correctly to validate and verify the software's quality according to its requirements. To ensure the correct functioning of each unit of functionality, unit tests are carried out [109]. These must be isolated from

external interactions to be considered unit tests. To test functionality, it is necessary to test all its possible outcomes or representative cases of every outcome, either of success or failure, to ensure its expected behavior and to identify and reduce the number of unexpected outcomes (i.e., errors or thrown exceptions) [110];

2. **Integration tests:** type of testing where units of functionality are integrated and tested as a group. Its purpose is to guarantee the integration of different units and expose unwanted outcomes/defects when the different units are integrated. It allows for verifying the interaction between components, ensuring compatibility, detecting issues early, improving the overall reliability of the system, and improving the quality of the system by identifying and fixing issues before they become more difficult and expensive to resolve [111].
3. **End-to-end tests:** a method of testing software that involves testing a system's workflow from start to finish. Its goal is to replicate real-world usage scenarios in order to validate the system in testing, and its components for data integrity and integration. It essentially validates all of the system's operations and how it communicates with the hardware, network, external dependencies, databases, and other systems. Any application is connected and integrated with multiple systems and databases outside of its environment. This makes its workflow significantly complicated. End-to-end tests verify if the application works as expected at all integration levels [112].

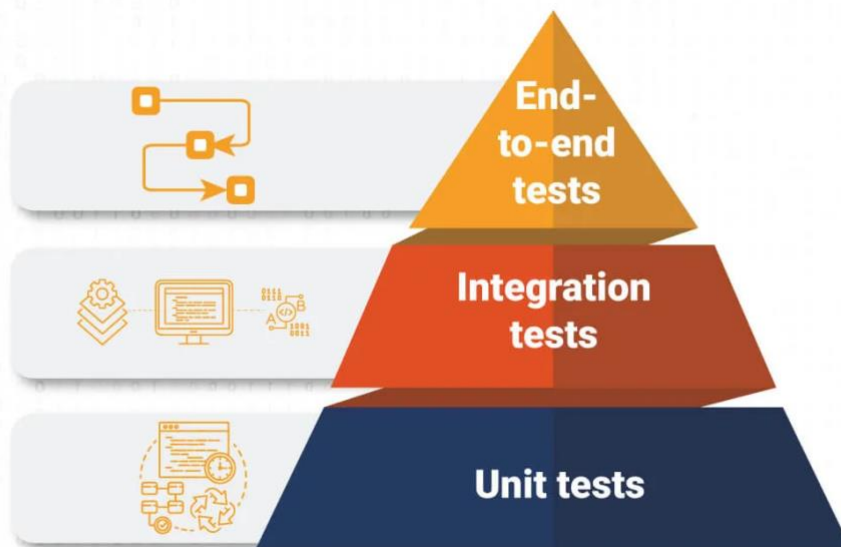


Figure 61 - Test Pyramid [108]

In the context of the developed system, unit tests and integration tests were developed to validate that the microservices' components were functioning as expected when in isolation and in integration.

7.4.1 Unit tests

Before presenting how the unit tests were developed it is important to mention how these are composed. A unit test is made up of three sections [113]:

1. **Arrange:** initializes the objects and sets that data's value to be used by the method (i.e., behavior/functionality) under test;
2. **Act:** invokes the method with the set values from the previous section as its parameters;
3. **Assert:** verifies that the method is working as expected.

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);
    // Act
    account.Debit(debitAmount);
    // Assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

Snippet 21 - Example unit test of the debit functionality of a bank account in C#

To ensure isolation, mocking is a technique that is typically used. This is a process used in unit tests when the unit under testing has external dependencies. The goal of mocking is to isolate and focus on the code under testing and not on the behavior or state of the external dependencies. In mocking, these are replaced by substitution objects that are controlled to simulate real behavior. There are three types of substitution objects [114]:

1. **Fakes:** an object that replaces real code by implementing the same interface without interacting with other objects. Typically, a fake is hard-coded to return fixed results;
2. **Stubs:** an object that returns specific results for specific inputs, and generally does not respond to other requests that it was not programmed for;
3. **Mocks:** a more sophisticated version of a stub. Return values like a stub but can also be programmed with expectations to how many times a method should be called, in what order, and with what input.

Unit tests were developed for the microservices using [115]–[118]:

- **xUnit**: unit testing tool for .NET;
- **Moq**: mocking library for .NET;
- **AutoFixture**: a tool that removes the need to hand-code variables for tests in .NET. It creates any type of object without the need to explicitly define which values should be used;
- **FluentAssertions**: a set of .NET extension methods that allow the specification of the expected result of tests in a more natural and intuitive way.

The developed unit tests were made to cover and validate the presentation and application layers. The setup of one of the unit test classes is presented as an example of the configuration made on the classes and is followed by a test of each class. The following snippets are from the User Service microservice unit tests as these represent the unit tests from the other microservices.

```

public class UserControllerTests {
    private readonly UserController Controller;
    private readonly Mock<IUserService> MockService;
    private readonly Mock<ILogger<UserController>> MockLogger;
    private readonly Mock<IJwtOptions> MockOptions;
    private readonly Fixture Fixture;

    private const string Issuer = "issuer";
    private const string Audience = "audience";
    private const string SigningKey = "signingKey-signingKey-signingKey";
    (...)
    public UserControllerTests()
    {
        this.MockService = new Mock<IUserService>();
        this.MockLogger = new Mock<ILogger<UserController>>();
        this.MockOptions = new Mock<IJwtOptions>();
        this.Fixture = new Fixture();

        this.MockOptions.Setup(m => m.Issuer).Returns(Issuer);
        this.MockOptions.Setup(m => m.Audience).Returns(Audience);
        this.MockOptions.Setup(m => m.SigningKey).Returns(SigningKey);
        this.MockOptions.Setup(m => m.ExpirationSeconds).Returns(60);

        this.Controller = new UserController(
            this.MockService.Object,
            this.MockLogger.Object,
            this.MockOptions.Object);
    }
    (...)
}

```

Snippet 22 - Setup of the UserControllerTests class

In Snippet 22 the setup of the `UserControllerTest` class is presented. It shows that the external dependencies of the class are declared as mocks, have behaviors setup for one of them with specific responses that will be common in all tests, and initializes all of these, the `UserController` instance with the mocks, and the `Fixture` instance that allows to quickly create new objects that can be used for input and output.

```
[Fact]
public async Task UserController_GetById_ReturnsNotFound() {
    // Arrange
    this.MockService.Setup(m => m.GetAsync(InvalidIdentification))
        .ThrowsAsync(new EntityNotFoundException());

    // Act
    var result = await this.Controller.GetById(InvalidIdentification)
        .ConfigureAwait(false);

    // Assert
    result.Should().NotBeNull();
    result.Should().BeOfType<NotFoundObjectResult>();
    var castedResult = (NotFoundObjectResult) result;
    castedResult.StatusCode.Should().Be(404);
    castedResult.Value.Should()
        .BeEquivalentTo(EntityNotFoundMessage);
}
```

Snippet 23 - Unit test that validates a specific behavior of the `UserController` class

In Snippet 23 we can observe a unit test that validates a specific behavior of the `GetById` endpoint of the `UserController` class which is to return a `Not Found` response. It does this by setting up the mock of the `UserService` interface that when a specific value is given then it throws a specific exception in the `arrange` section, then calling the controller endpoint with that specific value in the `act` section. What should happen when the exception is thrown is that the endpoint will handle it and return `Not Found` response that should contain a specific message which is what it validates in the `assert` section.

```

[Fact]
public async Task UserService_GetAsync_ShouldThrowException() {
    // Arrange
    this.MockRepository.Setup(m => m.Get(It.IsAny<string>()))
        .ReturnsAsync(null as Data.User);
    // Act
    var result = () => this.Service.GetAsync(UserId);
    // Assert
    await result.Should().ThrowAsync<EntityNotFoundException>();
    this.MockRepository
        .Verify(m => m.Get(It.IsAny<string>()), Times.Once);
    this.MockMapper
        .Verify(m => m.Map(It.IsAny<Data.User>()), Times.Never);
}

```

Snippet 24 - Unit test that validates a specific behavior of the UserService class

In Snippet 24 we can observe a unit test of the UserService class that validates a specific behavior of the GetAsync method which is to throw a specific exception. It does this by setting up the mock of the UserRepository interface that when given a specific value it returns a null response in the arrange section, then calling the method with a given argument in the act section. It then validates that it threw that specific exception and that one of the mocks was invoked a single time and the other was never invoked in the assert section.

```

[Fact]
public void UserMapper_Map_ShouldReturnUser() {
    // Arrange
    var dataUser = this.GetDataUser();

    // Act
    var user = this.Mapper.Map(dataUser);

    // Assert
    user.Should().NotBeNull();
    user.Id.Should().Be(dataUser.Id);
    user.Email.Should().BeEquivalentTo(dataUser.Email);
    user.Identification.Should().Be(dataUser.Identification);
    user.Status.Should().Be(dataUser.Status);
}
(...)
private Data.User GetDataUser() =>
    this.Fixture.Create<Data.User>();

```

Snippet 25 - Unit test that validates a specific behavior of the UserMapper class

In Snippet 25 we can observe a unit test that validates a specific behavior of the UserMapper class which should return a valid object. It does this by generating the input value with a Fixture instance, via private method, in the arrange section, then invoking the method with that input value in the act section, then finally it validates that the returned object's properties are as expected.

7.4.2 Integration tests

Like in unit testing, the composition of integration tests typically follows the same proposed structure (i.e., arrange, act, and assert sections). Since these validate the integration/interaction between different units of functionality there is no longer the need to isolate these by mocking their external dependencies since these will also be running for their integrated behaviors to be tested. In this case, the integration between the repository and the database is tested across the microservices. The following snippets present the configuration and a couple of integration tests of the UserRepository of the User Service microservice as these represent the other integration tests in the other microservices.


```

public class UserRepositoryFixture : IDisposable {
    public readonly UserRepository Repository;
    public readonly Fixture Fixture;
    public readonly FinanceUserDbContext Context;
    public readonly List<User> Users;
    private string DatabaseName = "InMemoryTestDB";
    public const string UserValidIdentification = "identification";
    public const string UserInvalidIdentification = "invalidIdentification";
    public const string UserValidEmail = "test@email.com";
    public const string UserInvalidEmail = "other.test@email.com";

    public UserRepositoryFixture() {
        this.Fixture = new Fixture();

        var options = new DbContextOptionsBuilder<FinanceUserDbContext>()
            .UseInMemoryDatabase(this.DatabaseName)
            .Options;

        this.Context = new FinanceUserDbContext(options);
        this.Users = this.GetDbSetUsers();
        this.Context.Users.AddRange(this.Users);
        this.Context.SaveChanges();
        this.Repository = new UserRepository(this.Context);
    }

    (...)
    private List<User> GetDbSetUsers() =>
        this.Fixture.Build<User>()
            .With(u => u.Identification, UserValidIdentification)
            .With(u => u.Email, UserValidEmail)
            .With(u => u.Status, Status.Pending)
            .CreateMany(1).ToList();
}

```

Snippet 26 - Partial implementation of the class fixture developed for the UserRepository integration tests

In Snippet 26 we can observe a partial implementation of the class fixture developed for the UserRepository integration tests. A class fixture is an xUnit feature that is a shared context to be used by a test class's tests [119]. It defines common variables to be used by the tests and

sets up the database (i.e., DbContext) in an InMemory context with some data, finally starting a new instance of the UserRepository with the database.

```
[Fact]
public async Task UserRepository_Get_ReturnUser() {
    // Act
    var result = await this.UserRepositoryFixture
        .Repository.Get(UserRepositoryFixture.UserValidIdentification)
        .ConfigureAwait(false);

    // Assert
    result.Should().NotBeNull();
    result.Email.Should().Be(UserRepositoryFixture.UserValidEmail);
}

[Fact]
public async Task UserRepository_Get_ReturnNull() {
    // Act
    var result = await this.UserRepositoryFixture
        .Repository.Get(UserRepositoryFixture.UserInvalidIdentification)
        .ConfigureAwait(false);

    // Assert
    result.Should().BeNull();
}
```

Snippet 27 - Pair of integration tests of the UserRepository class

In Snippet 27 we can observe two integration tests that validate the two behaviors of the Get method of the UserRepository class in which it either returns a user or a default response (i.e., null). Since both tests use the existing database (i.e., shared data) there is no need for an arrange section. In the first test it invokes the Get method with a valid information in the act section, then asserts that its response is not null and that it has expected data in one of its properties. In the second test it does the same but with invalid data in the act section, then validates that the response is null since said object with said data does not exist.

7.5 Continuous Integration and Continuous Deployment/Development

Continuous Integration and Continuous Deployment/Development (CI/CD) is vital component of the process of automating software testing and delivery [120]. Although CI/CD was not adopted in the system, since it ultimately was developed and tested locally, it needs to be mentioned since it's an important component of any software that can be accessed/used by anyone, whether internally/privately by enterprises or publicly by other individuals. It can be said that any software today without this process in place is doomed for failure since teams cannot deliver value fast enough to market and by consequence are beaten by the competition [121]. CI/CD is also important to adhere to the Secure-by-Design approach in which automated security tests can be executed as part of the pipeline [56].

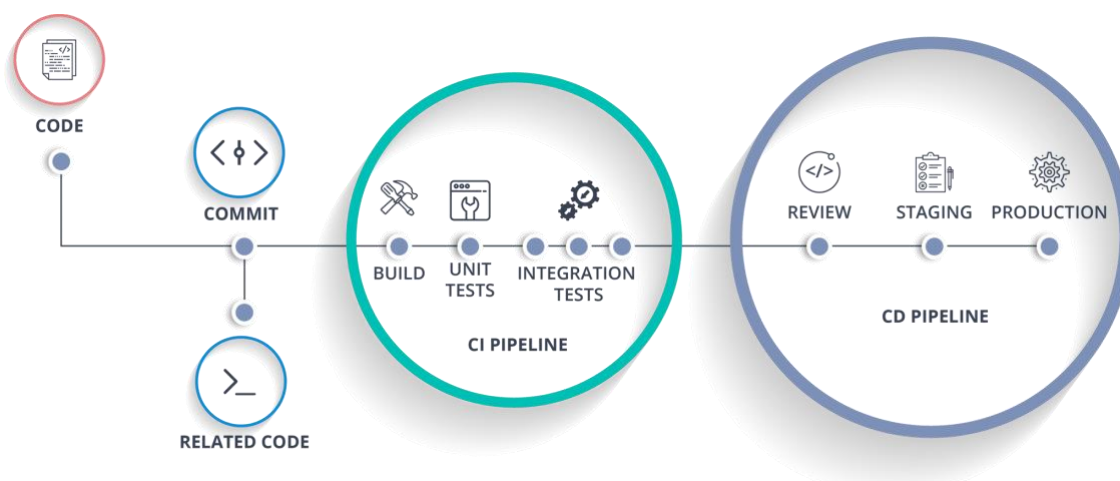


Figure 62 – CI/CD workflow [122]

From what can be understood, CI/CD is composed of two components [123], [124]:

1. **CI pipeline:** the first component in action, that is triggered when developers push changes to their code. It typically performs two different sets of actions which are to build the code and then to execute its tests (e.g., unit and integration tests) before moving onto the CD pipeline. It's usually complemented by static code analysis tools (e.g., SonarQube) that scans for potential issues, bugs, or violations of code standards which helps maintain code quality and consistency. It then generates reports and notifications to the developers about the build status, test results, and any code analysis findings allowing for early identification and addressing problems quickly;
2. **CD pipeline:** after the CI pipeline is done, the CD pipeline is triggered. What it does is to deploy the software gradually into different environments (e.g., testing, staging, production) that allows to perform further testing until it ultimately reaches production typically addressed as live. CD can be performed in two different ways, manually (i.e., Continuous Delivery) or automatically (i.e., Continuous Deployment). The difference is that in the latter the software is promoted straight to production automatically whereas the first requires for a developer to manually execute this promotion.

To implement CI/CD there are a wide range of tools for engineers to rely on such as Jenkins, CircleCI, Azure DevOps, and many more [125]–[127].

Even though the usage of CI/CD was not ultimately implemented as previously mentioned, it was still experimented on through Google Cloud Platform's Cloud Build [128] for CI and Cloud Run [129] for CD. The pipeline was setup through the platform's UI which allows various configuration options for triggering.

Google Cloud TMDEI-1171245 cloud build

← Create trigger

Name *
fakebank-corre-service-pipeline
Must be unique within the project's region

Region *
europe-west4 (Netherlands)

Description

Tags ?

Event
Repository event that invokes trigger

Push to a branch
 Push new tag
 Pull request
Not available for Cloud Source Repositories

Or in response to

Manual invocation
 Pub/Sub message
 Webhook event

Source
Repository generation

1st gen
 2nd gen

Repository *
fpsebastiao/fakebank-finance-core-service (Bitbucket)
Select the repository to watch for events and clone when the trigger is invoked

Branch *
^master\$
Trigger only for a branch that matches the given regular expression [Learn more](#)

Invert Regex
Matches the branch: master

Figure 63 – Cloud Build trigger setup in relation to the name, region, event, and source

Setting up a CI pipeline on Cloud Build is relatively easy and straightforward through its UI. First, the trigger must be named, a region must be selected and then to choose what type of event will trigger the pipeline and what is the source of the repository (cf. Figure).

Google Cloud TMDEI-1171245 cloud build

Create trigger

Configuration

Type

- Cloud Build configuration file (yaml or json)
- Dockerfile
- Buildpacks

Location

- Repository
fpsebastiao/fakebank-finance-core-service (Bitbucket)
- Inline
Write inline YAML

/ Dockerfile directory ?

The directory will also be used as the Docker build context

Dockerfile name
Dockerfile

The filename is relative to the Dockerfile directory

Image name * ?

gcr.io/macro-mile-396518/bitbucket.org/fpsebastiao/fakebank-finance-core-ser ?

Supported variables: \$PROJECT_ID, \$REPO_NAME, \$BRANCH_NAME, \$TAG_NAME, \$COMMIT_SHA, \$SHORT_SHA

Docker command preview

The command will be executed at the root of your repository. For more advanced usage, configure a cloudbuild.yaml file. [Learn More](#)

```
$ docker build \
  -t gcr.io/macro-mile-396518/bitbucket.org/fpsebastiao/fa
  -f Dockerfile \
```

Timeout seconds

The default timeout is 10 minutes

Use private pool

Advanced

Approval

Require approval before build executes

Service account

Trigger a build with the following service account [Learn more](#)

Service account email
665533743668-compute@developer.gserviceaccount.com ?

Figure 64 – Cloud Build trigger setup in relation to the configuration and service account

Then the actual configuration of the pipeline must be selected (i.e., how the pipeline will work) and what the service account will trigger the build (cf. Figure). In this case, it's triggered by

commits to the master branch and will build the Dockerfile and publish the result image to a registry.

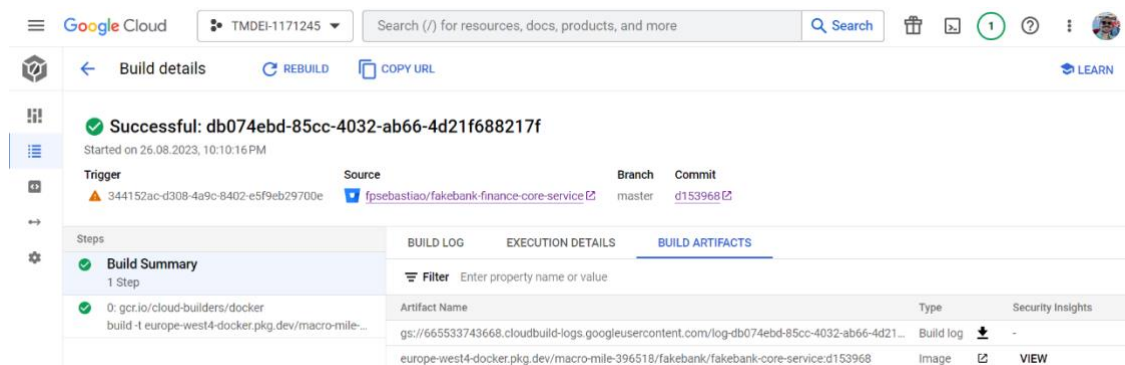


Figure 65 – Cloud Build build details after pipeline execution

Once a pipeline is triggered and successfully run, its status (i.e., build details) can be consulted as well as its build logs, execution details, and generated artifacts (i.e., Docker image) (cf. Figure 65).

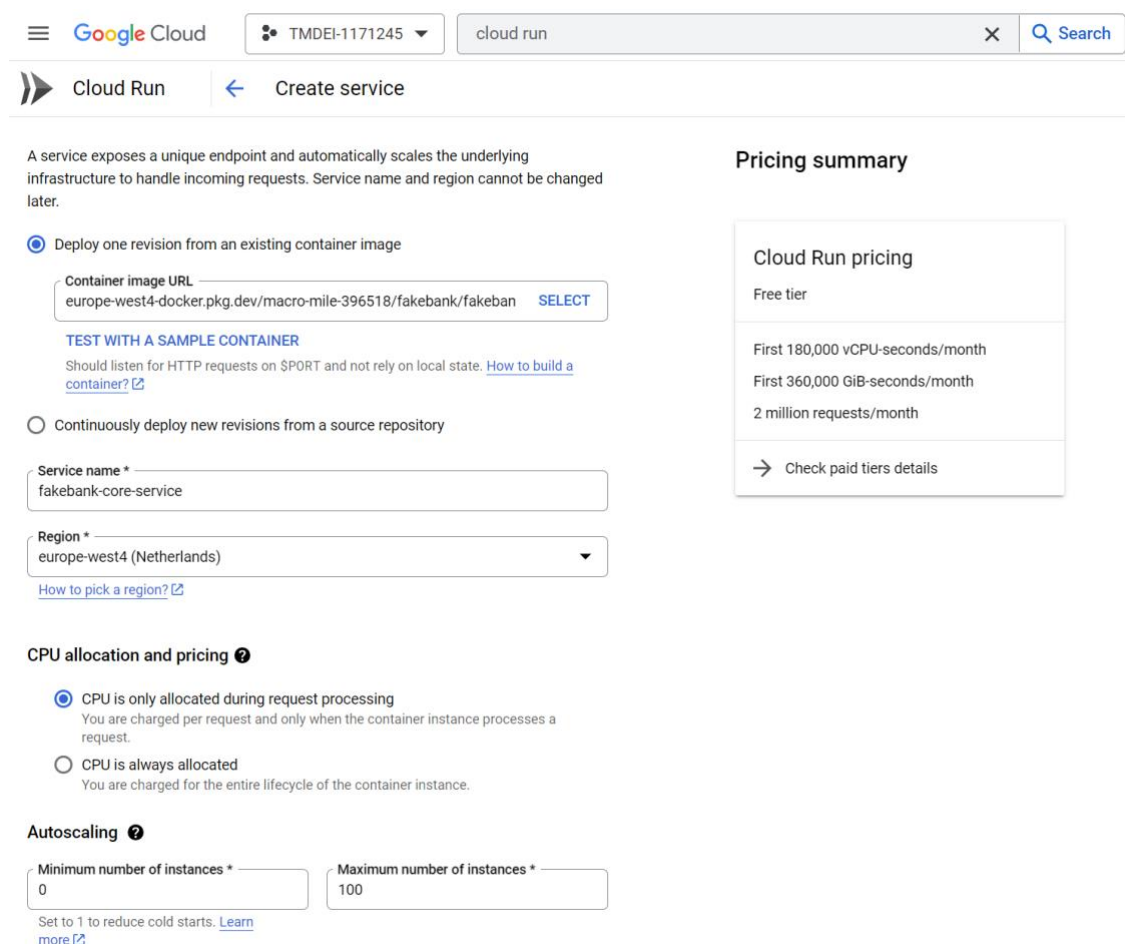



Figure 66 – Cloud Run service configuration

Once we have artifacts that can be published/deployed in a live environment, the service can be created through Cloud Run's UI. There are two options regarding its configuration, a specific container image can be selected for deployment which means that for future releases it must be changed to more recent versions (i.e., Continuous Delivery) or it can be integrated with Cloud Build for automatic releases (i.e., Continuous Deployment). Once that part is setup, the service must be named as well as a region to host it, choose what is needed regarding CPU allocation (i.e., dynamic allocation or fixed allocation), and select what autoscaling options are required (cf. Figure 66).

Ingress control

- Internal
Allow traffic from your project, shared VPC, and VPC service controls perimeter. Traffic from another Cloud Run service must be routed through a VPC. Limitations apply. [Learn more](#) 
- All
Allow direct access to your service from the internet

Authentication *

- Allow unauthenticated invocations
Check this if you are creating a public API or website.
- Require authentication
Manage authorized users with Cloud IAM.

Container, Networking, Security

Figure 67 – Cloud Run service creation

Finally, the ingress control and authentication are setup, and additional container, networking and security options are also available before creating the service (cf. Figure 67). The additional options can be consulted in Annex B (cf. Figure 84 and Figure 85).

The screenshot shows the Google Cloud Run console for a service named 'fakebank-core-service' in the 'europe-west4' region. The URL is 'https://fakebank-core-service-5ve75ejoq-ez.a.run.app'. The 'REVISIONS' tab is active, showing a table with one revision: 'fakebank-core-service-00001-9s4' with 100% traffic and deployed 'Just now'. The right-hand panel shows the configuration for this revision, including 'General' settings (CPU allocation, Startup CPU boost, Concurrency, Request timeout, Execution environment), 'Autoscaling' (Min instances: 1, Max instances: 5), and 'Image URL' (europe-west4-docker.pkg.dev/macro-mile-396518/fak...). Other settings like Port (8080), Build, Source, Command and args, CPU limit (1), and Memory limit (512MiB) are also visible.

Figure 68 – Deployed service on Cloud Run

After the service has been created it can be consulted through Cloud Run along with other information such as metrics, SLOs (Service-level objective), logs, revisions, networking, security, triggers, and more (cf. Figure 68).

7.6 Experimentation setup

To perform experimentations on the developed system to validate the metrics set by GQM, the usage of Vegeta – an HTTP load testing tool – was employed. Vegeta allows through the command line to execute HTTP requests at specific rates of requests per second for a given period, and more [130].

```
echo "GET http://:80" | vegeta attack -rate=50 -duration=5m | vegeta encode >
results.json

jq -ncM '{method: "POST", url: "http://:80", body: "{\'user\': \'user\',
\'password\': \'password\'}" | @base64, header: {"Content-Type":
["application/json"]}}' |
  vegeta attack -format=json -rate=20 -duration=1m | vegeta encode >
results.json
```

Snippet 28 - Examples of usage of Vegeta

Snippet 28 exemplifies a couple of ways that Vegeta can be used: In the first example, Vegeta is going to load test the localhost on port 80 with the HTTP GET method with a rate of 50 requests/second for 5 minutes, then it saves the results on a JSON file. In the second example, it's going to load test the same host and port with the HTTP POST method and a JSON body with a rate of 20 requests/second for 1 minute, then it saves the results in a JSON file.

Vegeta was used to load test the system's endpoints to generate logs, traces, and metrics for what could be considered a normal operating context to establish a baseline of "normal conditions". With a considerable number of collected data, ML jobs were created for anomaly detection and classification for prediction. Finally, Vegeta was used to execute tests to validate each of the GQM's metrics.

7.7 Results

This section presents the results obtained for each quality attribute's metric presented in section 7.3.

7.7.1 Availability

This section presents the evaluation of the availability quality attribute's metrics.

7.7.1.1 Uptime percentage

Regarding uptime percentage, this is a metric that requires continuous assessment over several months and production-level environmental conditions for it to be verified. Regardless, an assessment was made through Vegeta to guarantee a stable supply of requests to the system and consulting the uptime percentage in Kibana, as described in Annex A.1.5.4, during the period of the execution of requests. This is presented in Figure 69, and it also represents the whole system, that during the assessment the microservice maintained its availability at 100% fulfilling what would be most cloud platforms SLA. This allowed to verify the metric as compliant.

Fakebank-User-Service Status

Sep 15, 2023 @ 11:00:00.00 → Sep 15, 2023 @ 13:00:00.00 Refresh

Enable status alerts
 Define a default connector in [Settings](#) to enable monitor status alerts.

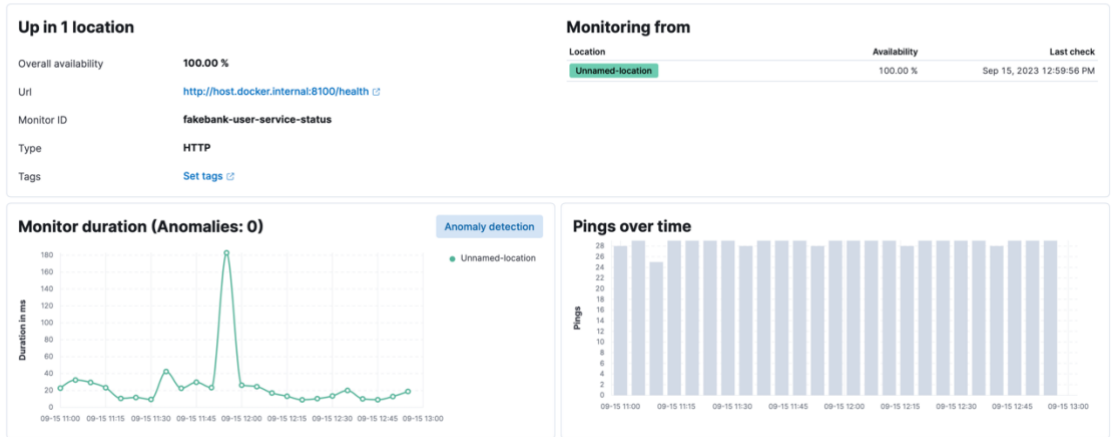


Figure 69 - Uptime of the User Service during the testing period

7.7.1.2 Successful execution rate

Like the uptime percentage, the successful execution rate metric requires the same conditions for it to be verified, but an assessment was made through request execution by Vegeta and the result can be consulted on Figure 70. As observed, the latency and throughput were relatively stable, with latency typically below 100ms with a few spikes, throughout the load testing period with 0% failed transaction rate, ensuring 100% successful execution rate. This validates the metric as compliant.

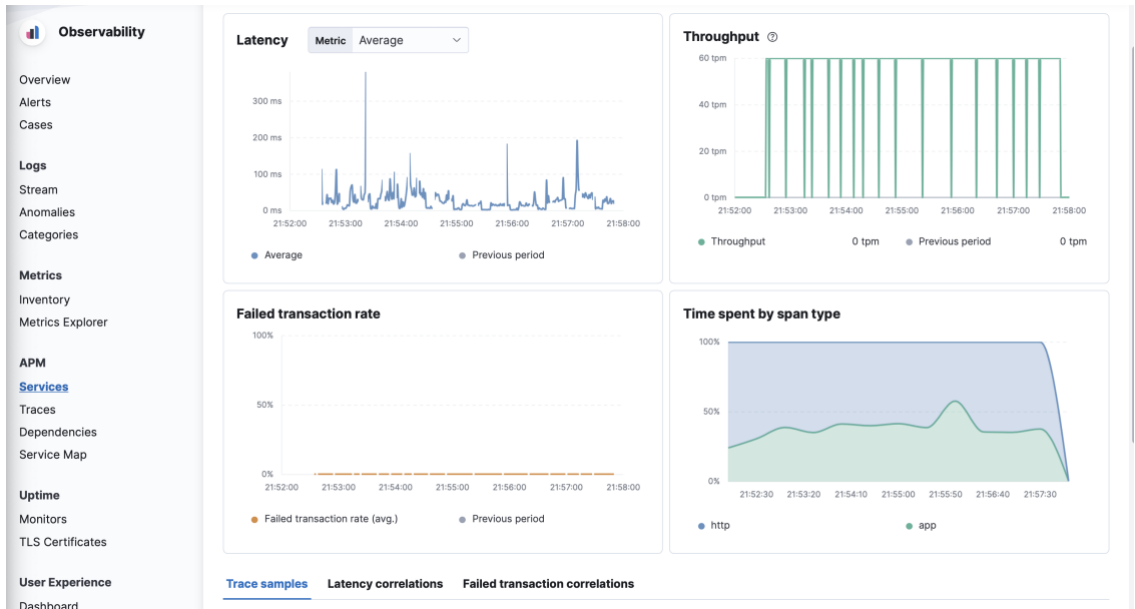


Figure 70 - Transaction metrics of the User Service during the testing period

7.7.1.3 Fault detection

Regarding fault detection, a circuit breaker was implemented on the API Gateway since Ocelot provides that feature, as described in Annex A.1.2, through the Polly library [131]. To use this feature, a configuration must be added to the declared routes in Ocelot's configuration (cf. Snippet 29), and then it must be added to the service collection (cf. Snippet 30) [132].

```
"QoSOptions": {  
  "ExceptionsAllowedBeforeBreaking": 3,  
  "DurationOfBreak": 5000,  
  "TimeoutValue": 10000  
},
```

Snippet 29 - Quality of Service options for the circuit breaker feature

To explain the options above: Exceptions allowed before breaking is the number of exceptions that can occur before opening the circuit (i.e., breaking). Duration of break is how long the circuit will stay open after breaking. The timeout value is how long a request is allowed to live before being timed out.

```
public void ConfigureServices(IServiceCollection services) {  
  services.AddPolly();  
  (...)  
}
```

Snippet 30 - Adding Polly to the services

With logs and metrics stored in Elasticsearch, Kibana can transform these into alarms for situations, via anomaly detection, such as unexpected downtime of a service that sends automatic notifications to the developer, reducing the time to correct unwanted situations. With these tools, fault detection can be guaranteed for each microservice. This validates the metric as compliant.

7.7.1.4 Resilience

Regarding resilience, this was ensured by Docker Swarm since it maintains the configured number of instances at all times [133]. Whenever an instance experiences failure, it starts a new one to replace it, and since its state is saved onto the database the new instances are always up to date. This verifies the metric as compliant.

7.7.2 Performance

Regarding performance, even though the real-time applicational performance metrics such as response time, average CPU utilization, average memory usage, and network traffic could be consulted and analyzed in Kibana, as presented in sections 6.1.2.2 and 6.1.2.3, this was not possible to verify since testing in a local environment (i.e., personal computer) does not guarantee a stable test environment where the system's performance is not impacted by external factors (i.e., third party systems). This means that the metrics related to this quality attribute are unable to be verified.

7.7.3 Scalability

This section presents the evaluation of the scalability quality attribute's metrics.

7.7.3.1 Usage frequency

Regarding usage frequency, Elastic's APM Server was the chosen tool to track distributed tracing and evaluate the percentage of requests made to the evaluated microservice (i.e., Transfer Service) when compared to the whole system. A single request was made to each available operation (i.e., endpoint) in the exposed REST APIs to uncover the subsequent requests, presented in Table 13.

Table 13 - Usage frequency of each microservice

Microservice	Number of operations	Number of invocations to other Microservices
Core Service	6	0
User Service	5	1
Transfer Service	2	2
Payment Service	2	1
Total	15	4

The ratio between requests made to the evaluated microservice and the total requests made to the whole system is 13.33% meaning that, when supported by the fact that the subsequent requests are all asynchronous, it guarantees the scalability of the system (i.e., compliant with the metric).

7.7.3.2 Horizontal/vertical scalability

Regarding horizontal and vertical scalability: horizontal scalability was achieved by using Docker Swarm. It allows the manipulation of the number of instances of the same image under the same name, and this number can be changed on the fly [133]. Vertical scalability can also be achieved through Docker but not in a way that would ensure this metric, since it requires to manually start each container with the option of CPU and memory and to change the assigned values implies starting another instance with these values updated and then to stop the previous instance. This means that this metric is partially compliant.

7.7.3.3 Isolation

Regarding the isolation of the microservices, these can only communicate through the exposed REST API interfaces. This verifies the metric as compliant.

7.7.4 Monitorability

This section presents the evaluation of the monitorability quality attribute's metrics.

7.7.4.1 Data generation and storage

Regarding data generation and storage, as previously described in sections 6.1.2 and 6.2.2:

- Logs are generated by Serilog following the ECS format and are then written in Elasticsearch which manages the data storage;
- Traces are sent to the APM Server which in turn writes these in Elasticsearch;
- Metrics are collected by Metricbeat and written in Elasticsearch.

This allows to verify this metric as compliant.

7.7.4.2 Data presentation

Regarding data presentation, the Kibana instance consumes this data and provides the user with multiple views and monitors as shown in section 6.1.2 and in Annex A.1.5. This verifies that the metric is compliant.

7.7.5 Security

This section presents the evaluation of the security quality attribute's metrics.

7.7.5.1 Third-party vulnerabilities

Regarding third-party vulnerabilities, the Snyk [134] plugin was used to analyze the whole dependency tree of the project for known vulnerabilities. The plugin achieves this by looking at the Common Weakness Enumeration (CWE) [135], Common Vulnerabilities and Exposures (CVE) [136], and Common Vulnerability Scoring System (CVSS) [137] databases, and Snyk's own vulnerability database [138]. The usage of this tool adheres to the static and dynamic analysis method of the Secure-by-Design approach in which it identifies vulnerabilities in the application [56]. Figure 71 presents the results of the conducted analysis on the Core Service solution and these represent the other services as well since they share the same dependencies on their presentation service project except the data model vulnerability.

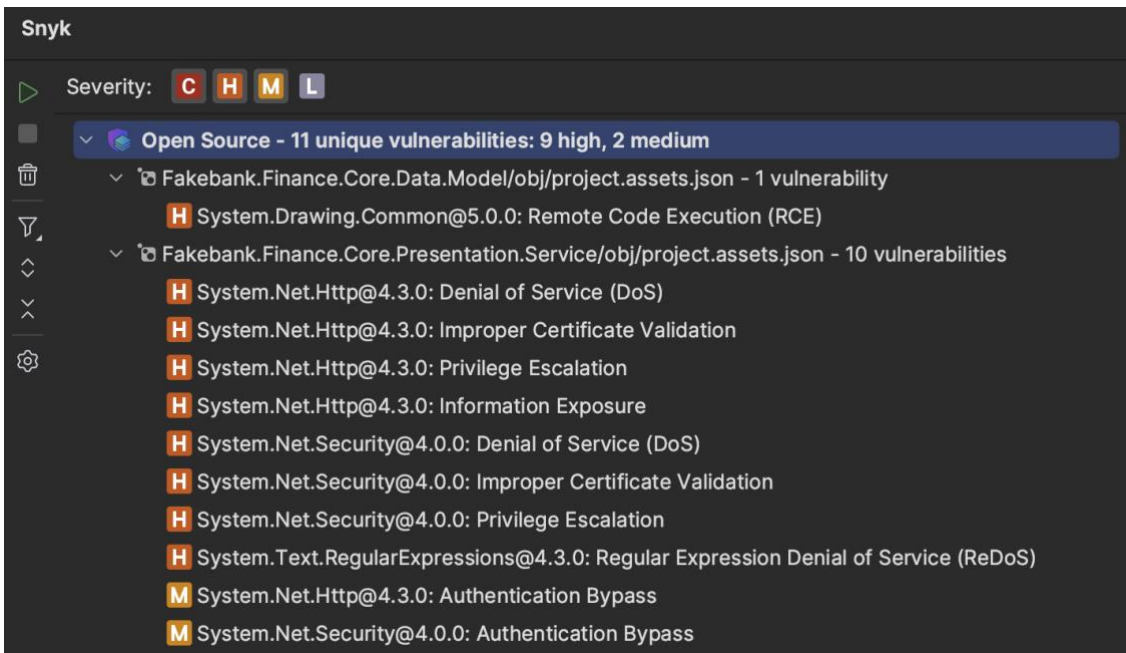


Figure 71 - Analysis of dependent libraries of the Core Service solution

Although an effort was made to combat the number of vulnerabilities of the projects, by upgrading and downgrading some of the dependencies, this small number remained. For this reason, this metric is not compliant. Is it worth noting that these vulnerabilities are related to the chosen libraries and framework.

7.7.5.2 Security monitor

When it comes to the security monitor, Kibana allows the generation of detailed graphics and monitors at various levels to observe and monitor as well as alert of the anomalous behavior of each microservice through the created ML jobs, as described in section 0. This means this metric is compliant.

Anomalies								
Severity	warning	Interval	Auto					
Time	Severity	Detector	Found for	Influenced by	Actual	Typical	Description	Actions
> September 13th 2023	70	high duration by transaction type for an APM service	request	service.name: fakebank-core-service transaction.type: request	3708.7	501.4	↑ 7x higher	⚙️
> September 15th 2023	28	high duration by transaction type for an APM service	request	service.name: fakebank-user-service transaction.type: request	184880	13991	↑ 13x higher	⚙️
> October 1st 2023	15	high duration by transaction type for an APM service	request	service.name: fakebank-user-service transaction.type: request	314380	13992	↑ 22x higher	⚙️
> September 14th 2023	15	high duration by transaction type for an APM service	request	service.name: fakebank-api-gateway transaction.type: request	2081721	9041.7	↑ More than 100x higher	⚙️

Rows per page: 25

Figure 72 - Detected anomalies in transactions by the APM ML job

Anomalies										
Severity		Interval		Auto						
Time	Severity	Detector	Found for	Influenced by	Actual	Typical	Description	Job ID	Category examples	Actions
> September 14th 2023	18	count	fakebank-api-gateway	event.dataset : fakebank-api-gateway	91670	9.69	↑ 100x higher	kibana-logs-ui-default-log-entry-rate		
> September 15th 2023	63	count	fakebank-user-service	event.dataset : fakebank-user-service	2569	101.1	↑ 25x higher	kibana-logs-ui-default-log-entry-rate		
> October 1st 2023	55	count	fakebank-transfer-service	event.dataset : fakebank-transfer-service	195	9.32	↑ 21x higher	kibana-logs-ui-default-log-entry-rate		
> September 15th 2023	10	count	fakebank-transfer-service	event.dataset : fakebank-transfer-service	180	22.9	↑ 8x higher	kibana-logs-ui-default-log-entry-rate		
> October 1st 2023	10	count by learned log entry category	mlcategory 159	event.dataset : fakebank-user-service mlcategory: 159	22	2.41	↑ 9x higher	kibana-logs-ui-default-log-entry-categories-count	Executing action me...	
> October 1st 2023	10	count by learned log entry category	mlcategory 161	event.dataset : fakebank-user-service mlcategory: 161	22	2.41	↑ 9x higher	kibana-logs-ui-default-log-entry-categories-count	Executed DbComma...	
> October 1st 2023	10	count by learned log entry category	mlcategory 162	event.dataset : fakebank-user-service mlcategory: 162	22	2.41	↑ 9x higher	kibana-logs-ui-default-log-entry-categories-count	Executed action met...	

Figure 73 - Detected anomalies in log count and ingestion rate by the logs ML job

7.7.5.3 Authentication and authorization

Finally, regarding authentication and authorization, this was implemented via JSON Web Token (JWT) [139] on each microservice by employing .NET's internal implementation. Snippet 31 exemplifies how it is configured on each microservice. The JWT options are loaded from the app settings configuration file and are then used to set up the authentication. To start using authorization the "authorize" annotation must be added to the endpoints on the Controllers. The requirement of roles can be added through the annotation as exemplified in Snippet 32.


```

private static void ConfigureAuth(IServiceCollection services, IConfiguration
configuration) {
    var jwtOptions = configuration.GetSection("JwtOptions").Get<JwtOptions>();
    services.AddSingleton(jwtOptions);
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options => {
            var signingKeyBytes =
Encoding.UTF8.GetBytes(jwtOptions!.SigningKey);
            options.TokenValidationParameters = new TokenValidationParameters
{
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ValidIssuer = jwtOptions.Issuer,
                ValidAudience = jwtOptions.Audience,
                IssuerSigningKey = new SymmetricSecurityKey(signingKeyBytes)
            };
        });
    services.AddAuthorization();
}

```

Snippet 31 - Authentication and Authorization configuration

```

[HttpGet("{identification}")]
[Authorize(Role = "Admin")]
public async Task<IActionResult> GetById(string identification)
{(...) }

```

Snippet 32 - Usage of the authorize annotation in an endpoint

7.8 Summary

Rounding up the evaluation made in Table 14, 15 metrics were considered, with 10 compliant, 1 partially compliant, 2 unable to be verified, and 1 non-compliant.

Table 14 - Evaluation results

Characteristic	Quality Attribute	Metric	Compliance degree
Resilience	Availability	Uptime percentage	Complete
		Successful execution rate	Complete
		Fault detection	Complete
		Resilience	Complete
	Performance	Response time	Unable to verify
		Average CPU utilization	Unable to verify
	Scalability	Usage frequency	Complete
		Horizontal/vertical scalability	Partial
Isolation		Complete	
Security	Monitorability	Data generation and storage	Complete
		Data presentation	Complete
	Security	Third-party weaknesses	Does not comply
		Security monitor	Complete
		Authentication and authorization	Complete

It can be concluded that the designed and implemented system accomplished completely the goals associated with the availability and monitorability QAs, but there is still a need to execute further testing to ensure its performance and scalability, and to resolve the weaknesses found in its dependencies to guarantee the system's security.

Regarding the investigation hypotheses:

- **H.1:** The system provides high levels of availability, and some of scalability, but its performance cannot be verified. This means that the proposed architecture offers some levels of operation resilience, but it requires further testing to ensure it provides high levels of resilience.
- **H.2:** The system provides high levels of monitorability and some of security, meaning that the usage of AI methods, particularly ML, in conjunction with the proposed architecture does offer a great advantage in its security, but it needs to solve the weaknesses stemming its dependencies or at least keep them to an acceptable level.

8 Conclusions

This chapter describes which objectives were achieved, and those that were not, the limitations that were found during the development, and the future work for other possible approaches for further investigation. Finally, a final appreciation of the work done and the obtained knowledge on a personal level is presented.

8.1 Objectives achieved

The primordial objective associated with this dissertation is to understand how microservices can respond to the context of critical systems when looking at resilience and security when paired with machine learning. The compliance of the objectives is summarized in the form of RQs with their respective investigation hypotheses in Table 15.

Table 15 - Summary of objectives

Research question	Investigation hypothesis	Compliance degree
Does the proposed architecture offer high levels of operational resilience?	The proposed architecture offers high levels of operational resilience	Partial
Does the adoption of AI methods assist in security in such a context?	The usage of AI methods in conjunction with the proposed architecture provides a great advantage in its security	Partial

In short, it can be stated that the objectives of this dissertation were partially accomplished. As explained in section 7.8, the system provided certain levels of operational resilience and the usage of ML provided a great advantage in its security but there are certain aspects that need further testing such as its performance and scalability, and further refinement regarding the weaknesses found on its dependencies.

8.2 Limitations

Certain difficulties were detected throughout this dissertation that raised concerns regarding the system's validation and applicability to other situations. These are:

- Due to the limited resources available, it was not possible to develop an isolated testing environment, not ensuring the validity of some tests. This is because the available Azure and Google Cloud student credits were not sufficient to setup a relatively robust infrastructure with the system and its external dependencies for long enough to perform experimentations, even with the lowest tier pricing;
- This dissertation focuses on a single system and framework so the results cannot be generalized. Further case studies with other systems and different frameworks would be required to highlight the pros and cons of each situation.

8.3 Future work

Although most goals were accomplished, there are still other possible approaches that can be followed for further investigation that allow to make the system more robust and for further testing regarding performance, anomaly detection, and prevention. Those are:

- **Remove dependency vulnerabilities:** it is necessary to ensure the system's safety by performing a careful analysis of the detected vulnerabilities to validate and assess possible fixes since these are possible points of breach for malicious external authors;
- **Usage of a cloud-based environment as a stable testing environment:** with the system secured, the usage of a cloud-base environment as stable testing environment such as Azure, Amazon Web Services (AWS), or Google Cloud Platform (GCP) would follow to ensure its stability and security;
- **Execute remaining tests:** with robust infrastructure capable of boasting sufficient resources (i.e., CPU and RAM) for extensive testing due to the cloud-based environment, the system could now be tested on its performance and remaining scalability metrics, perhaps public testing to have a broader user base;
- **Experimentation with other machine learning technologies:** with a more robust infrastructure boasting more resources than the typical local development machine, the system can generate more logs, traces and metrics that can be used for further ML experimentation in anomaly detection and prevention with the use of other technologies such as Amazon CloudWatch, or Google Dataflow (cf. section 2.3.1).

8.4 Final appreciation

This dissertation allowed the student to consolidate and apply the knowledge obtained throughout the master's course in an investigational context. This also allowed the student to acquire experience in the development and investigation of technologies, software

architectural design, and concepts that are the Elastic Stack, microservices, security, and machine learning.

One of the main goals of the dissertation was the adaptation of the student's knowledge to a context that is the further deepening of the knowledge of a specific subject. The proposed project presented itself as complex and challenging, but at the same time attractive, since it allowed the student to cement the themes studied during the degree, to touch on themes outside the degree that are security and machine learning, and to acquire competencies in an investigational context.

Finally, the study of new technologies, namely the Elastic Stack and its many features, allowed to expand the student's technical knowledge

References

- [1] C. and I. S. Agency, “What is Cybersecurity?” Accessed: Jan. 22, 2023. [Online]. Available: <https://www.cisa.gov/uscert/ncas/tips/ST04-001>
- [2] National Cyber Security Centre, “What is cyber security?” Accessed: Jan. 22, 2023. [Online]. Available: <https://www.ncsc.gov.uk/section/about-ncsc/what-is-cyber-security>
- [3] IBM, “What is Artificial Intelligence (AI) ? | IBM.” Accessed: Feb. 25, 2023. [Online]. Available: <https://www.ibm.com/topics/artificial-intelligence>
- [4] S. Brown, “Machine learning, explained | MIT Sloan.” Accessed: Feb. 25, 2023. [Online]. Available: <https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained>
- [5] P. Grieve, “Deep learning vs. machine learning: What’s the difference?” Accessed: Feb. 25, 2023. [Online]. Available: <https://www.zendesk.com/blog/machine-learning-and-deep-learning/>
- [6] K. Joshi, “How AI is Changing Cybersecurity: New Threats & Opportunities.” Accessed: Feb. 25, 2023. [Online]. Available: <https://emeritus.org/blog/cybersecurity-how-ai-is-changing-cybersecurity/>
- [7] B. Committee on Banking Supervision, “Principles for Operational Resilience,” Aug. 2020.
- [8] D. Cio, “DoDI 8500.01, March 14, 2014, Incorporating Change 1 on October 7, 2019,” 2014.
- [9] A. Kott and I. Linkov, “To improve cyber resilience, measure it,” *Computer (Long Beach Calif)*, vol. 54, no. 2, pp. 80–85, Feb. 2021, doi: 10.1109/MC.2020.3038411.
- [10] “UHS breach shows the dangers facing hospitals with growing ransomware threats | Fierce Healthcare.” Accessed: Feb. 12, 2023. [Online]. Available: <https://www.fiercehealthcare.com/tech/uhs-breach-shows-dangers-facing-hospitals-growing-cyber-threats>
- [11] M. Hinchey and L. Coyle, “Evolving critical systems: A research agenda for computer-based systems,” *17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2010*, pp. 430–435, 2010, doi: 10.1109/ECBS.2010.56.
- [12] “The NIS2 Directive: A high common level of cybersecurity in the EU | Think Tank | European Parliament.” Accessed: Oct. 13, 2023. [Online]. Available: [https://www.europarl.europa.eu/thinktank/en/document/EPRS_BRI\(2021\)689333](https://www.europarl.europa.eu/thinktank/en/document/EPRS_BRI(2021)689333)
- [13] T. Yarygina and A. H. Bagge, “Overcoming Security Challenges in Microservice Architectures,” *Proceedings - 12th IEEE International Symposium on Service-Oriented*

System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018, pp. 11–20, May 2018, doi: 10.1109/SOSE.2018.00011.

- [14] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O'Reilly Media, Inc., 2015.
- [15] M. Fowler and J. Lewis, "Microservices." Accessed: Feb. 11, 2023. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [16] "What are microservices?" Accessed: Feb. 11, 2023. [Online]. Available: <https://microservices.io/>
- [17] "What is a REST API?" Accessed: Feb. 12, 2023. [Online]. Available: <https://www.redhat.com/en/topics/api/what-is-a-rest-api>
- [18] "What does an API gateway do?" Accessed: Feb. 12, 2023. [Online]. Available: <https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do>
- [19] V. Mokhor, S. Honchar, and A. Onyskova, "Cybersecurity Risk Assessment of Information Systems of Critical Infrastructure Objects," *2020 IEEE International Conference on Problems of Infocommunications Science and Technology, PIC S and T 2020 - Proceedings*, pp. 19–22, Oct. 2021, doi: 10.1109/PICST51311.2020.9467957.
- [20] L. Maglaras, I. Kantzavelou, and M. A. Ferrag, "Digital Transformation and Cybersecurity of Critical Infrastructures," *Applied Sciences 2021, Vol. 11, Page 8357*, vol. 11, no. 18, p. 8357, Sep. 2021, doi: 10.3390/APP11188357.
- [21] "Artificial Intelligence Cybersecurity Challenges — ENISA." Accessed: Oct. 12, 2023. [Online]. Available: <https://www.enisa.europa.eu/publications/artificial-intelligence-cybersecurity-challenges>
- [22] A. Sampaio, "Improving Systematic Mapping Reviews," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, pp. 1–8, Nov. 2015, doi: 10.1145/2830719.2830732.
- [23] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giarretta, S. T. Larsen, and S. Dustdar, "Microservices: Migration of a Mission Critical System," *IEEE Trans Serv Comput*, vol. 14, no. 5, pp. 1464–1477, 2021, doi: 10.1109/TSC.2018.2889087.
- [24] E. Solberg, *The transition from monolithic architecture to microservice architecture A case study of a large Scandinavian financial institution*. Oslo, 2022.
- [25] A. Furda, L. Van Den Berg, G. Reid, G. Camera, and M. Pinasco, "Developing a Microservices Integration Layer for Next-Generation Rail Operations Centers," *IEEE Softw*, vol. 39, no. 5, pp. 9–16, 2022, doi: 10.1109/MS.2022.3179030.
- [26] R. P. Pontarolli, J. A. Bigheti, L. B. R. de Sá, and E. P. Godoy, "Towards security mechanisms for an industrial microservice-oriented architecture," *2021 14th IEEE*

International Conference on Industry Applications, INDUSCON 2021 - Proceedings, pp. 679–685, Aug. 2021, doi: 10.1109/INDUSCON51756.2021.9529415.

- [27] B. Yang, F. Zhang, and S. U. Khan, “An Encryption-as-a-service Architecture on Cloud Native Platform,” *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, vol. 2021-Janua, 2021, doi: 10.1109/ICCCN52240.2021.9522248.
- [28] “Kubernetes.” Accessed: Feb. 13, 2023. [Online]. Available: <https://kubernetes.io/>
- [29] “HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer.” Accessed: Feb. 13, 2023. [Online]. Available: <https://www.haproxy.org/>
- [30] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, and A. Zomaya, “AI-enabled Secure Microservices in Edge Computing: Opportunities and Challenges,” *IEEE Trans Serv Comput*, 2022, doi: 10.1109/TSC.2022.3155447.
- [31] M. O. Pahl and M. Loipfinger, “Machine learning as a reusable microservice,” *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*, pp. 1–7, Jul. 2018, doi: 10.1109/NOMS.2018.8406165.
- [32] H. Bangui, B. Rossi, and B. Buhnova, “A Conceptual Antifragile Microservice Framework for Reshaping Critical Infrastructures,” *Proceedings - 2022 IEEE International Conference on Software Maintenance and Evolution, ICSME 2022*, pp. 364–368, 2022, doi: 10.1109/ICSME55016.2022.00040.
- [33] N. N. Taleb, *Antifragile : things that gain from disorder*, 1st ed. Random House, 2012.
- [34] “Gretel.ai — Incorporate generative AI into your data.” Accessed: Feb. 24, 2023. [Online]. Available: <https://gretel.ai/>
- [35] S. Jacob, Y. Qiao, and B. Lee, “Detecting Cyber Security Attacks against a Microservices Application using Distributed Tracing,” 2021, doi: 10.5220/0010308905880595.
- [36] S. Jacob, Y. Qiao, Y. Ye, and B. Lee, “Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks,” *Comput Secur*, vol. 118, p. 102728, Jul. 2022, doi: 10.1016/J.COSE.2022.102728.
- [37] J. Chen, H. Huang, and H. Chen, “Informer: Irregular traffic detection for containerized microservices RPC in the real world,” *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019*, pp. 389–394, Nov. 2019, doi: 10.1145/3318216.3363375.
- [38] I. Macedo, S. Wanous, N. Oliveira, O. Sousa, and I. Praça, “A tool to support the investigation and visualization of cyber and/or physical incidents,” *Advances in Intelligent Systems and Computing*, vol. 1368 AISC, pp. 130–140, Dec. 2021, doi: 10.48550/arxiv.2112.01103.

- [39] "What is Distributed Tracing? How it Works & Use Cases | Datadog." Accessed: Feb. 13, 2023. [Online]. Available: <https://www.datadoghq.com/knowledge-center/distributed-tracing/>
- [40] "Managed Open-Source Elasticsearch and OpenSearch Search and Log Analytics – Amazon OpenSearch Service – Amazon Web Services." Accessed: Oct. 01, 2023. [Online]. Available: <https://aws.amazon.com/opensearch-service/>
- [41] "OpenSearch." Accessed: Oct. 01, 2023. [Online]. Available: <https://opensearch.org/>
- [42] "Preprocess logs for anomaly detection in Amazon OpenSearch | AWS Big Data Blog." Accessed: Oct. 01, 2023. [Online]. Available: <https://aws.amazon.com/blogs/big-data/preprocess-logs-for-anomaly-detection-in-amazon-opensearch/>
- [43] "Dataflow | Google Cloud." Accessed: Oct. 01, 2023. [Online]. Available: <https://cloud.google.com/dataflow>
- [44] "Anomaly detection using streaming analytics & AI | Google Cloud Blog." Accessed: Oct. 01, 2023. [Online]. Available: <https://cloud.google.com/blog/products/data-analytics/anomaly-detection-using-streaming-analytics-and-ai>
- [45] "What is Elastic Machine Learning? | Machine Learning in the Elastic Stack [8.10] | Elastic." Accessed: Sep. 29, 2023. [Online]. Available: <https://www.elastic.co/guide/en/machine-learning/8.10/machine-learning-intro.html>
- [46] "Circuit Breaker pattern - Azure Architecture Center | Microsoft Learn." Accessed: Oct. 01, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>
- [47] "Circuit Breaker Pattern." Accessed: Oct. 01, 2023. [Online]. Available: <https://blog.soumendrak.com/circuit-breaker-design-pattern-997c3521c1c4>
- [48] "Implementing Resiliency Patterns in Microservices | AppMaster." Accessed: Oct. 01, 2023. [Online]. Available: <https://appmaster.io/blog/microservices-architecture-resiliency-patterns>
- [49] "Rate Limiting pattern - Azure Architecture Center | Microsoft Learn." Accessed: Oct. 01, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting-pattern>
- [50] "Health Check." Accessed: Oct. 01, 2023. [Online]. Available: <https://microservices.io/patterns/observability/health-check-api.html>
- [51] "Fallback pattern · Microservices Architecture." Accessed: Oct. 01, 2023. [Online]. Available: <https://badia-kharroubi.gitbooks.io/microservices-architecture/content/patterns/communication-patterns/fallback-pattern.html>

- [52] “Bulkhead pattern - Azure Architecture Center | Microsoft Learn.” Accessed: Oct. 01, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead>
- [53] “Safety-Critical Validation”.
- [54] “The Concept of Domain-Driven Design Explained | by Sara Miteva | Microtica | Medium.” Accessed: Oct. 11, 2023. [Online]. Available: <https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f>
- [55] “What is the Domain Model in Domain Driven Design? | Culttt.” Accessed: Sep. 23, 2023. [Online]. Available: <https://culttt.com/2014/11/12/domain-model-domain-driven-design/>
- [56] “Secure by Design.” Accessed: Oct. 12, 2023. [Online]. Available: <https://aptori.dev/learn/secure-by-design-and-default>
- [57] “The C4 model for visualising software architecture.” Accessed: Sep. 16, 2023. [Online]. Available: <https://c4model.com/>
- [58] P. Kruchten, “Architectural Blueprints-The ‘4+1’ View Model of Software Architecture,” *IEEE Softw*, vol. 12, no. 6, pp. 42–50, 1995.
- [59] R. C. Martin, “Design Principles and Design Patterns,” 2000, Accessed: Sep. 18, 2023. [Online]. Available: www.objectmentor.com
- [60] “Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST).” Accessed: Sep. 18, 2023. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- [61] “Docker: Accelerated Container Application Development.” Accessed: Sep. 17, 2023. [Online]. Available: <https://www.docker.com/>
- [62] “What is Docker? ”In Simple English” | by Yann Mulonda | Medium.” Accessed: Sep. 17, 2023. [Online]. Available: <https://yannmjl.medium.com/what-is-docker-in-simple-english-a24e8136b90b>
- [63] “The Onion Architecture : part 1 | Programming with Palermo.” Accessed: Sep. 18, 2023. [Online]. Available: <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>
- [64] “Onion Architecture In ASP.NET Core 6 Web API.” Accessed: Sep. 18, 2023. [Online]. Available: <https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-6-web-api/#>

- [65] "P of EAA: Service Layer." Accessed: Sep. 19, 2023. [Online]. Available: <https://martinfowler.com/eaacatalog/serviceLayer.html#>
- [66] "P of EAA: Data Mapper." Accessed: Sep. 19, 2023. [Online]. Available: <https://martinfowler.com/eaacatalog/dataMapper.html>
- [67] "Strategy Design Pattern." Accessed: Sep. 19, 2023. [Online]. Available: https://sourcemaking.com/design_patterns/strategy
- [68] "InversionOfControl." Accessed: Sep. 19, 2023. [Online]. Available: <https://martinfowler.com/bliki/InversionOfControl.html>
- [69] "Dependency injection - .NET | Microsoft Learn." Accessed: Sep. 19, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>
- [70] "Designing the infrastructure persistence layer - .NET | Microsoft Learn." Accessed: Sep. 19, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern>
- [71] "Service Registry Design Pattern in Microservices Explained | by Soma | Javarevisited | Medium." Accessed: Sep. 24, 2023. [Online]. Available: <https://medium.com/javarevisited/service-registry-design-pattern-in-microservices-explained-a796494c608e#>
- [72]. ".NET | Build. Test. Deploy." Accessed: Sep. 24, 2023. [Online]. Available: <https://dotnet.microsoft.com/en-us/>
- [73] "Introducing .NET 5 - .NET Blog." Accessed: Sep. 24, 2023. [Online]. Available: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>
- [74]. ".NET Framework is dead -- long live .NET 5." Accessed: Sep. 24, 2023. [Online]. Available: <https://betanews.com/2019/05/07/future-of-dotnet/>
- [75]. ".NET (and .NET Core) - introduction and overview - .NET | Microsoft Learn." Accessed: Sep. 24, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/core/introduction>
- [76] "A tour of C# - Overview - C# | Microsoft Learn." Accessed: Sep. 24, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>
- [77] "Serilog — simple .NET logging with fully-structured events." Accessed: Sep. 24, 2023. [Online]. Available: <https://serilog.net/>
- [78] "serilog-contrib/serilog-sinks-elasticsearch: A Serilog sink that writes events to Elasticsearch." Accessed: Sep. 24, 2023. [Online]. Available: <https://github.com/serilog-contrib/serilog-sinks-elasticsearch>

- [79] “Overview | Elastic Common Schema (ECS) Reference [8.10] | Elastic.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/guide/en/ecs/current/ecs-reference.html>
- [80] “Get Started | ECS Logging .NET Reference | Elastic.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/guide/en/ecs-logging/dotnet/current/setup.html>
- [81] “Overview of Entity Framework Core - EF Core | Microsoft Learn.” Accessed: Sep. 24, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/ef/core/>
- [82] “Steeltoe - Home.” Accessed: Sep. 24, 2023. [Online]. Available: <https://steeltoe.io/>
- [83] “domaindrivendev/Swashbuckle.AspNetCore: Swagger tools for documenting API’s built on ASP.NET Core.” Accessed: Sep. 24, 2023. [Online]. Available: <https://github.com/domaindrivendev/Swashbuckle.AspNetCore>
- [84] “Application Performance Monitoring (APM) with Elastic Observability | Elastic.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/observability/application-performance-monitoring>
- [85] “ASP.NET Core | APM .NET Agent Reference [1.x] | Elastic.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/guide/en/apm/agent/dotnet/current/setup-asp-net-core.html>
- [86] “Big Picture — Ocelot 1.0.0 documentation.” Accessed: Sep. 24, 2023. [Online]. Available: <https://ocelot.readthedocs.io/en/latest/introduction/bigpicture.html>
- [87] “What is Service Discovery? Definition and Related FAQs | Avi Networks.” Accessed: Sep. 24, 2023. [Online]. Available: <https://avinetworks.com/glossary/service-discovery/#>
- [88] “Client-side service discovery pattern.” Accessed: Sep. 24, 2023. [Online]. Available: <https://microservices.io/patterns/client-side-discovery.html>
- [89] “Server-side service discovery pattern.” Accessed: Sep. 24, 2023. [Online]. Available: <https://microservices.io/patterns/server-side-discovery.html>
- [90] “Service Discovery in a Microservices Architecture - NGINX.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>
- [91] “Consul by HashiCorp.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.consul.io/>
- [92] “Microsoft Data Platform | Microsoft.” Accessed: Sep. 24, 2023. [Online]. Available: <https://www.microsoft.com/en-us/sql-server>

- [93] "Definition of database server | PCMag." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.pcmag.com/encyclopedia/term/database-server>
- [94] "Editions and supported features of SQL Server 2022 - SQL Server | Microsoft Learn." Accessed: Sep. 24, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/sql/sql-server/editions-and-components-of-sql-server-2022?view=sql-server-ver16&preserve-view=true>
- [95] "PostgreSQL: The world's most advanced open source database." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.postgresql.org/>
- [96] "Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/elasticsearch/>
- [97] "Kibana: Explore, Visualize, Discover Data | Elastic." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/kibana>
- [98] "Metricbeat: Lightweight Shipper for Metrics | Elastic." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/beats/metricbeat#system>
- [99] "Heartbeat overview | Heartbeat Reference [8.10] | Elastic." Accessed: Sep. 24, 2023. [Online]. Available: <https://www.elastic.co/guide/en/beats/heartbeat/current/heartbeat-overview.html>
- [100] "Initialize Discovery Client | Steeltoe." Accessed: Sep. 25, 2023. [Online]. Available: <https://docs.steeltoe.io/api/v3/discovery/initialize-discovery-client.html>
- [101] "HashiCorp Consul | Steeltoe." Accessed: Sep. 25, 2023. [Online]. Available: <https://docs.steeltoe.io/api/v3/discovery/hashicorp-consul.html>
- [102] "Discovering Services | Steeltoe." Accessed: Sep. 25, 2023. [Online]. Available: <https://docs.steeltoe.io/api/v3/discovery/discovering-services.html>
- [103] Reading Craze, "Hypothesis Formulation in Research - Reading Craze." Accessed: Feb. 19, 2023. [Online]. Available: <http://readingcraze.com/index.php/hypothesis-formulation-research/>
- [104] Leading Agile, "GQM Approach: Agile Metrics - LeadingAgile." Accessed: Feb. 19, 2023. [Online]. Available: <https://www.leadingagile.com/2017/05/agile-metrics-gqm-approach/>
- [105] M.-D. Cojocaru, A. Uta, and A.-M. Oprescu, "Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications".
- [106] T. Schirgi, "Architectural Quality Attributes for the Microservices of CaRE," 2021.

- [107] S. Li *et al.*, “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review,” *Inf Softw Technol*, vol. 131, p. 106449, Mar. 2021, doi: 10.1016/J.INFSOF.2020.106449.
- [108] “What is Testing Pyramid?” Accessed: Oct. 10, 2023. [Online]. Available: <https://www.headspin.io/blog/the-testing-pyramid-simplified-for-one-and-all>
- [109] “What Is Unit Testing? | SmartBear.” Accessed: Oct. 10, 2023. [Online]. Available: <https://smartbear.com/learn/automated-testing/what-is-unit-testing/>
- [110] D. Huizinga and A. Kolawa, “Best Practices for Testing and Code Review,” *Automated Defect Prevention*, pp. 250–270, 2007.
- [111] “Integration Testing: A Detailed Guide | BrowserStack.” Accessed: Oct. 11, 2023. [Online]. Available: <https://www.browserstack.com/guide/integration-testing>
- [112] “What is End To End Testing? | BrowserStack.” Accessed: Oct. 10, 2023. [Online]. Available: <https://www.browserstack.com/guide/end-to-end-testing>
- [113] “Unit testing fundamentals - Visual Studio (Windows) | Microsoft Learn.” Accessed: Oct. 10, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019>
- [114] “Mocking Framework for Unit Testing - Telerik JustMock.” Accessed: Oct. 11, 2023. [Online]. Available: <https://www.telerik.com/products/mocking/unit-testing.aspx>
- [115] “Home > xUnit.net.” Accessed: Oct. 11, 2023. [Online]. Available: <https://xunit.net/>
- [116] “devlooped/moq: The most popular and friendly mocking framework for .NET.” Accessed: Oct. 11, 2023. [Online]. Available: <https://github.com/devlooped/moq>
- [117] “Home - AutoFixture.” Accessed: Oct. 11, 2023. [Online]. Available: <https://autofixture.github.io/>
- [118] “Fluent Assertions - Fluent Assertions.” Accessed: Oct. 11, 2023. [Online]. Available: <https://fluentassertions.com/>
- [119] “Shared Context between Tests > xUnit.net.” Accessed: Oct. 12, 2023. [Online]. Available: <https://xunit.net/docs/shared-context>
- [120] “What Is CI/CD and How Does It Work? | Synopsys.” Accessed: Oct. 12, 2023. [Online]. Available: <https://www.synopsys.com/glossary/what-is-cicd.html>
- [121] “Why CI/CD pipelines can’t replace local software development operations: Managing expectations vs. reality - Cloudomation.” Accessed: Oct. 12, 2023. [Online]. Available: <https://cloudomation.com/en/cloudomation-blog/ci-cd-pipelines-local-development-operations/>

- [122] “Continuous Integration & Deployment | element61.” Accessed: Oct. 12, 2023. [Online]. Available: <https://www.element61.be/en/competence/continuous-integration-deployment>
- [123] “What is CI/CD? (Differences, Benefits, Tools, Fundamentals) | BrowserStack.” Accessed: Oct. 12, 2023. [Online]. Available: <https://www.browserstack.com/guide/what-is-ci-cd>
- [124] “Code Quality, Security & Static Analysis Tool with SonarQube | Sonar.” Accessed: Oct. 12, 2023. [Online]. Available: <https://www.sonarsource.com/products/sonarqube/>
- [125] “Jenkins.” Accessed: Oct. 12, 2023. [Online]. Available: <https://www.jenkins.io/>
- [126] “Continuous Integration and Delivery - CircleCI.” Accessed: Oct. 12, 2023. [Online]. Available: <https://circleci.com/>
- [127] “Azure DevOps Services | Microsoft Azure.” Accessed: Oct. 12, 2023. [Online]. Available: <https://azure.microsoft.com/en-us/products/devops>
- [128] “Cloud Build serverless CI/CD platform | Google Cloud.” Accessed: Oct. 12, 2023. [Online]. Available: <https://cloud.google.com/build>
- [129] “Cloud Run: Container to production in seconds | Google Cloud.” Accessed: Oct. 12, 2023. [Online]. Available: <https://cloud.google.com/run>
- [130] “tsenart/vegeta: HTTP load testing tool and library. It’s over 9000!” Accessed: Oct. 01, 2023. [Online]. Available: <https://github.com/tsenart/vegeta>
- [131] “App-vNext/Polly: Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner. From version 6.0.1, Polly targets .NET Standard 1.1 and 2.0+.” Accessed: Oct. 01, 2023. [Online]. Available: <https://github.com/App-vNext/Polly>
- [132] “Quality of Service — Ocelot 1.0.0 documentation.” Accessed: Oct. 01, 2023. [Online]. Available: <https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html?highlight=polly#quality-of-service>
- [133] “Swarm mode overview | Docker Docs.” Accessed: Oct. 01, 2023. [Online]. Available: <https://docs.docker.com/engine/swarm/>
- [134] “Snyk | Developer security | Develop fast. Stay secure. | Snyk.” Accessed: Sep. 30, 2023. [Online]. Available: <https://snyk.io/>
- [135] “CWE - Common Weakness Enumeration.” Accessed: Sep. 30, 2023. [Online]. Available: <https://cwe.mitre.org/index.html>

- [136] "CVE - CVE." Accessed: Sep. 30, 2023. [Online]. Available: <https://cve.mitre.org/index.html>
- [137] "Common Vulnerability Scoring System SIG." Accessed: Sep. 30, 2023. [Online]. Available: <https://www.first.org/cvss/>
- [138] "Snyk Vulnerability Database | Snyk." Accessed: Sep. 30, 2023. [Online]. Available: <https://security.snyk.io/>
- [139] "JSON Web Token Introduction - jwt.io." Accessed: Sep. 30, 2023. [Online]. Available: <https://jwt.io/introduction>
- [140] P. A. Koen, H. M. J. Bertels, and E. J. Kleinschmidt, "Managing the front end of innovation-part II: Results from a three-year study," *Research Technology Management*, vol. 57, no. 3, pp. 25–35, May 2014, doi: 10.5437/08956308X5703199.
- [141] P. Koen *et al.*, "Providing Clarity and A Common Language to the 'Fuzzy Front End,'" <http://dx.doi.org/10.1080/08956308.2001.11671418>, vol. 44, no. 2, pp. 46–55, 2001, doi: 10.1080/08956308.2001.11671418.
- [142] S. Nicola, E. P. Ferreira, and J. J. P. Ferreira, "A NOVEL FRAMEWORK FOR MODELING VALUE FOR THE CUSTOMER, AN ESSAY ON NEGOTIATION," <https://doi.org/10.1142/S0219622012500162>, vol. 11, no. 3, pp. 661–703, Jul. 2012, doi: 10.1142/S0219622012500162.
- [143] A. Graf and P. Maas, "Customer value from a customer perspective – a comprehensive review," *Service Value als Werttreiber*, pp. 59–87, 2014, doi: 10.1007/978-3-658-02140-5_3.
- [144] A. Osterwalder, Y. Pigneur, G. Bernarda, and A. Smith, *Value Proposition Design: How to Create Products and Services Customers Want*, 1st ed. Wiley, 2014.
- [145] J. Borza, "FAST Diagrams: The Foundation for Creating Effective Function Models General Dynamics Land Systems," 2011.

Annex A

A.1 Value analysis

This chapter presents the value analysis of this project. For that, the New Concept Development Model is used, as well as the designation of the value to the client, the perceived value, and the value proposition. Finally, a functional analysis is made.

A.1.1 New Concept Development Model

The New Concept Development Model is divided into three parts, those being the (1) engine, the (2) key elements, and (3) environmental factors, which influence the previous two [140]. The engine is the center of the model, focusing on the vision, strategy, culture, resources, teams, and collaboration [141]. There are five key elements, those being (1) opportunity identification, (2) opportunity analysis, (3) idea generation, (4) idea selection, and (5) concept definition, and these are analyzed in the following sections [140]. The environmental factors are considered external factors (e.g., competition, marketplace trends, technologies) that influence the other two parts of the model.

The key elements and how these are related are presented in **Error! Reference source not found.** [141].

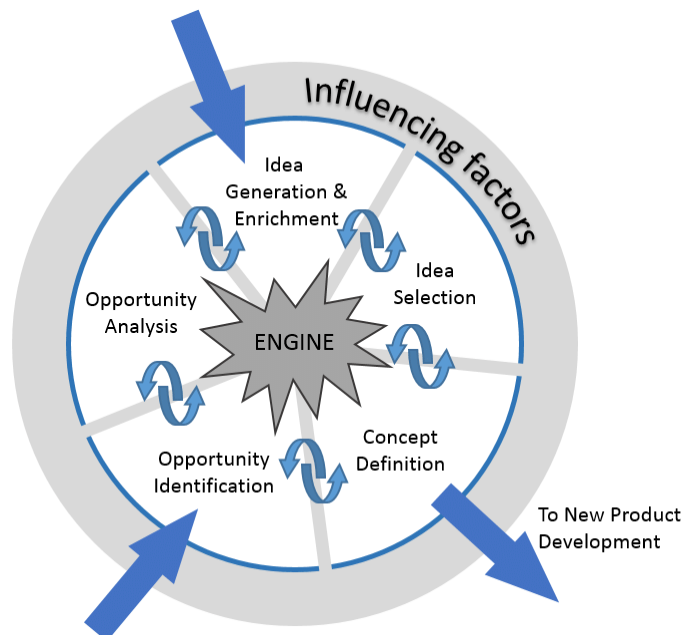


Figure 74 - The New Concept Development model

Looking at the key elements bidirectional arrows between them can be observed. This means that there doesn't necessarily exist a linear path to go through. There are also two incoming arrows, pointing at the two distinct elements – Opportunity Identification and Idea Generation and Enrichment –, meaning that these two are the possible starting points for the analysis. Lastly, the outgoing arrow leaving the element of Concept Definition to the exterior is the last step of the process.

A.1.1.1 Opportunity identification

Opportunity identification consists of the analysis of opportunities and threats inherent to the business. The main contributors are factors such as cultural tendencies, resources, and the economy, among others [141].

The opportunity arises from the need to understand how the microservice architecture can respond in the context of critical systems to the ever-growing rising challenges of cybersecurity, specifically in how AI (i.e., machine learning) may assist, and having these into account, in how it affects systems operational resilience. It is known that when systems are hit or flooded with attacks, such as DDoS, even if resilient and with a great number of resources (i.e., computational resources) these will most likely be impacted in their performance and availability even if it is a small impact.

As presented in the Relevant studies and papers section, artificial intelligence can assist in the detection or prediction of possible malicious attacks on systems, and with these paired with a microservice architecture, knowing its underlying characteristics, such as scalability and resilience, it would be interesting in learning how these would respond accordingly for the systems to maintain their overall availability, performance, and operational resilience, or at least to reduce the impacts of the attacks.

A.1.1.2 Opportunity analysis

In opportunity analysis, the market and the probability of success of the identified opportunity should be studied. For this, planning and advantage assessment are made use of, depending on the business setting and its resources [141].

According to what was presented in section 2.1, current literature related to this subject is practically nonexistent, since these only cover or investigate some of the aspects this dissertation seeks to explore such as the detection of attacks using AI or the usage of security mechanisms (e.g., authentication and authorization) to strengthen systems security. None investigate the aspect of the operation resilience of the systems, in how these “bounce back” or respond to attacks or the combination of these aspects.

By understanding how all these aspects correlate to each other when combined, the results of this dissertation could offer an interesting solution or path for both industry and academia to pursue to elevate microservice architecture-based systems security and operational resilience.

A.1.1.3 Idea generation and enrichment

With an opportunity properly identified and analyzed, idea generation and enrichment can begin, turning the identified opportunity into concrete idea(s) [141].

The predefined idea is to design and implement a software prototype following the microservice architecture that aims to test the more relevant concepts for its adoption in the migration of critical systems. The relevant concepts would be the adoption of artificial intelligence for the detection of attacks and how these can be leveraged in maintaining the prototype's overall resilience, availability, and performance at optimal levels.

A.1.1.4 Idea selection

The investment in the selected idea(s) must be justified, as it is important to ensure that the idea(s) guarantee(s) a return [141].

The predefined idea was selected by default since it is the one believed to be more relevant in the context of this dissertation.

A.1.1.5 Concept definition

The concept definition presents the selected idea and its advantages [141].

To design and implement a software prototype that can prove the utility of the microservice architecture in the context of a critical system paired with artificial intelligence for the detection of attacks and how it can help maintain operational resilience.

A.1.2 Value

Value is defined according to the needs, criteria, interests, benefits, attitudes, and preferences, and it can vary according to distinct perceptions [142]. These variations among perceptions could be verified between the product developer (i.e., goods or services) and the client. This means that the value has a quantity of subjective nature.

In this dissertation's context, the value resides in understanding how the microservice architecture can position itself when facing cybersecurity challenges with artificial intelligence as a resource and how this impacts operational resilience. Should this position turn advantageous, it could help set a precedent in how microservices are secured and their operational resilience maintained or elevated to a higher standard.

A.1.2.1 Value to the client

Value to the client consists in the perception that the client has over a determined product or service [142]. This perception may also differ from client to client, according to the context and experience of each, meaning that value is influenced by a plethora of factors, as well as needs and preferences [143]. Value to the client can be considered positive if the client believes that he, or she, will incur more benefits than losses when acquiring a product or service.

When it comes to the dissertation subject, the client would be academia and the industry as these can look at the work performed in this dissertation as guidance (i.e., a path) for future research in the same or different concepts, in the case of academia, or the implementation in a real-world scenario, in the case of the industry.

A.1.2.2 Perceived value

Perceived value is related to the expected value by the client and the utility value, residing in the tradeoff between the initially developed expectations (i.e., before acquiring a product or service) and what is received (i.e., after acquiring a product or service) by the client [143]. It can also vary from client to client.

In this context, it is important to evaluate what are the benefits and sacrifices, for the client, in this study. In terms of benefits, it is expected that the microservice architecture-based critical systems will benefit from a higher level of security coming with the aid of an artificial intelligence that detects attacks and with this detection can help the system's defense, turning it more resilient. When it comes to sacrifices, these can result in a slight decrease in performance due to the addition of the artificial intelligence mechanism, and in higher costs of maintenance, although this last one can be negated if this dissertation proves its point, and in that sense, by turning systems more secure, could help avoid the costs that come with significant attacks.

A.1.2.3 Value proposition

Value proposition consists of how a business creates value, with a determined product or service, to a group of potential clients, one of the aspects being in how they differentiate or position themselves from the competition [144]. The value proposition must present the strengths of the product or service while bringing clarity to potential doubtfulness, assuring a better understanding from the client over the proposal.

This dissertation has the objective to understand what is the paper that microservices can perform in the context of critical systems, and the inherent characteristics of these, in terms of their operational resilience, when faced with cybersecurity challenges. To fulfill this objective, a software prototype will be designed and implemented. As previously mentioned, a higher level of security, aided by artificial intelligence in conjunction with the aspects of resilience and scalability from microservices, would turn systems much more robust and with a higher degree of operational resilience.

When it comes to the clients, which would be academia and the industry, it could lead to future research on improving these aspects, to look at different aspects or offer a vision of how it could be implemented in real-world scenarios.

A.1.3 Functional Analysis (FAST)

The creation of a Functional Analysis System Technique (FAST) diagram is important to better define the objectives of a project or to lead to a clearer path to a solution, specifying its actions, as well as their logical associations, and questioning the “Why?”, “How?”, “Who?”, and “When?” [145]. The diagram should present the objective of the project and how it will be achieved, making use of two limit functions, the higher order function (i.e., How) and the assumed function (i.e., Why?), that are connected through an intermediate set of functions whose responsibility is to explain and map the course of the process.

Error! Reference source not found. presents the dissertation's FAST diagram.

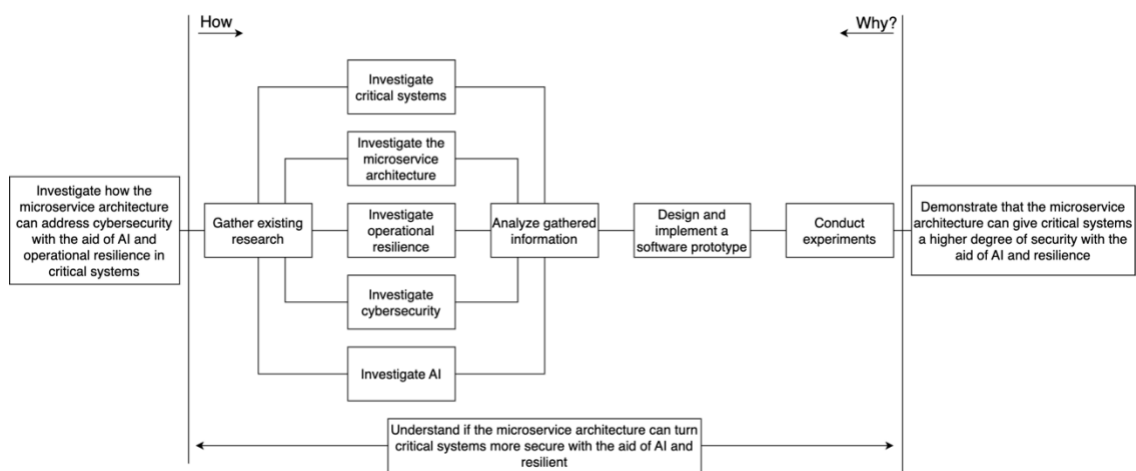


Figure 75 - FAST diagram for this dissertation

Analyzing the diagram above, the objective can be found at the bottom of the diagram “Understand if the microservice architecture can turn critical systems more secure with the aid of AI and resilient”. It is necessary to understand how that can be accomplished, considering research and investigation. All the functions will be conducted by the student. On the left side, the higher order function “Investigate how the microservice architecture can address cybersecurity with the aid of AI and operational resilience in critical systems”, followed by basic and secondary functions. Finally, the assumed function “Demonstrate that the microservice architecture can give critical systems a higher degree of security with the aid of AI and resilience”.

Annex B

B.1 Results per query on B-On and ScienceDirect

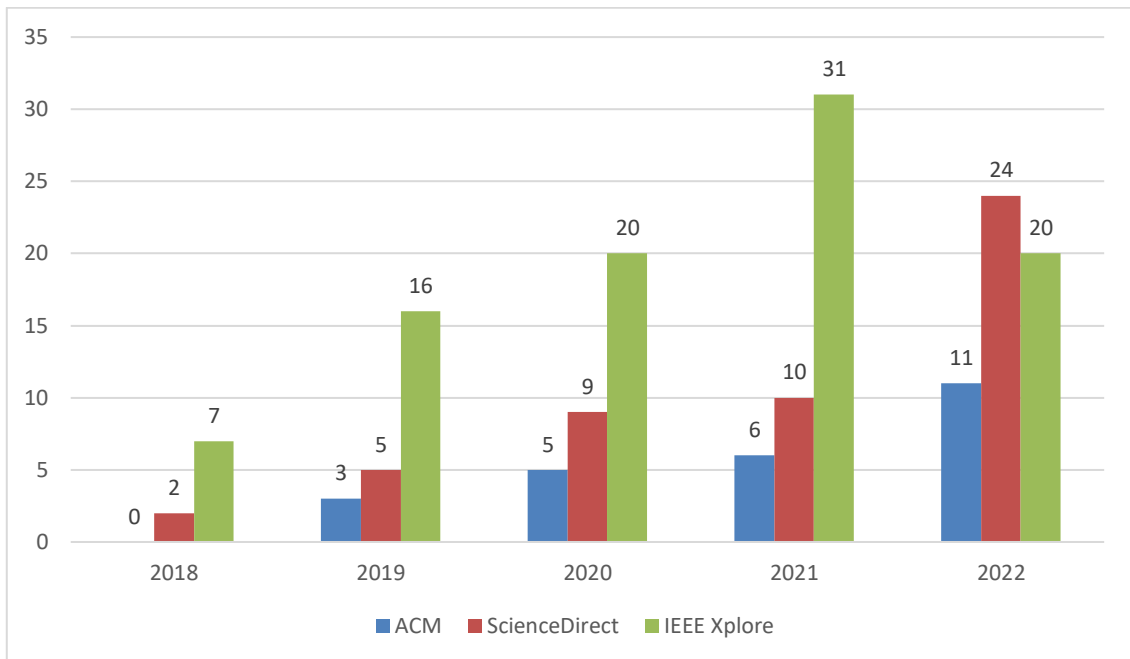


Figure 76 - Number of results presented by B-On and ScienceDirect using the query “microservices operational resilience OR microservices resilience”

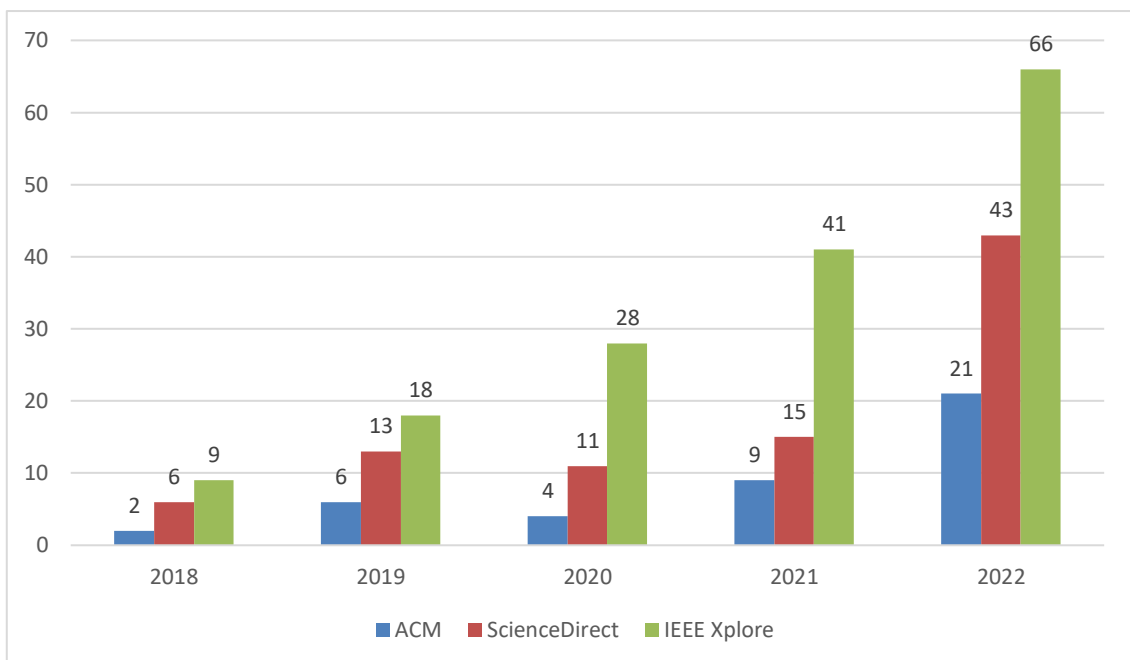


Figure 77 - Number of results presented by B-On and ScienceDirect using the query “microservices cybersecurity OR microservices cyber security”

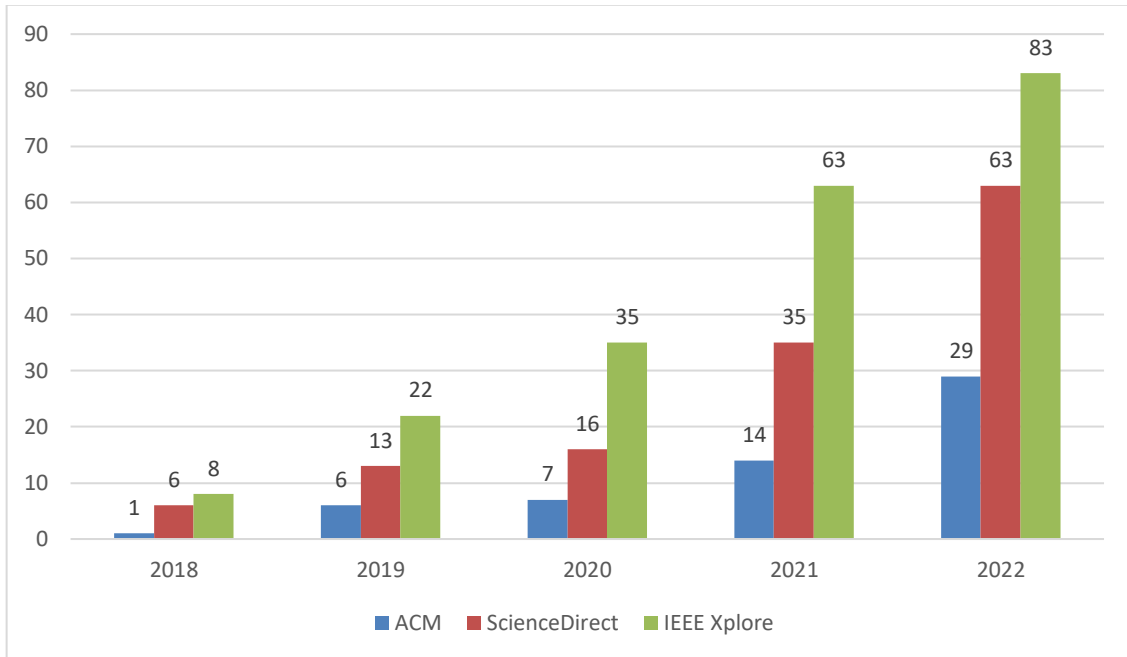


Figure 78 - Number of results presented by B-On and ScienceDirect using the query “microservices artificial intelligence”

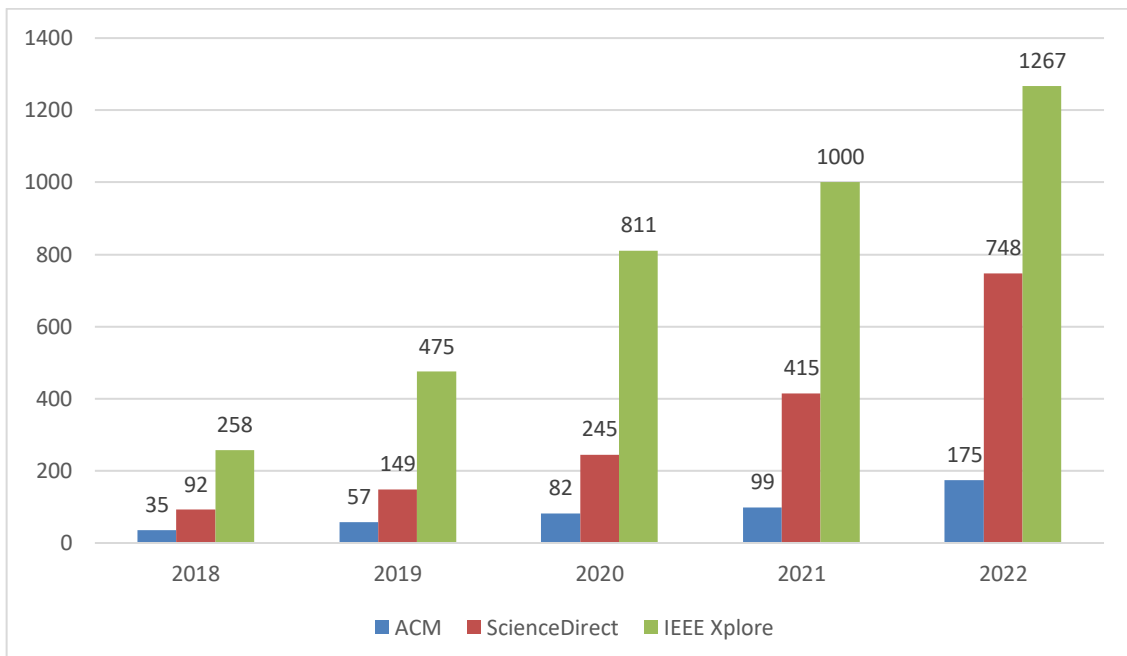


Figure 79 - Number of results presented by B-On and ScienceDirect using the query “artificial intelligence cybersecurity OR artificial intelligence cyber security”

B.2 Container level process views with more outcomes

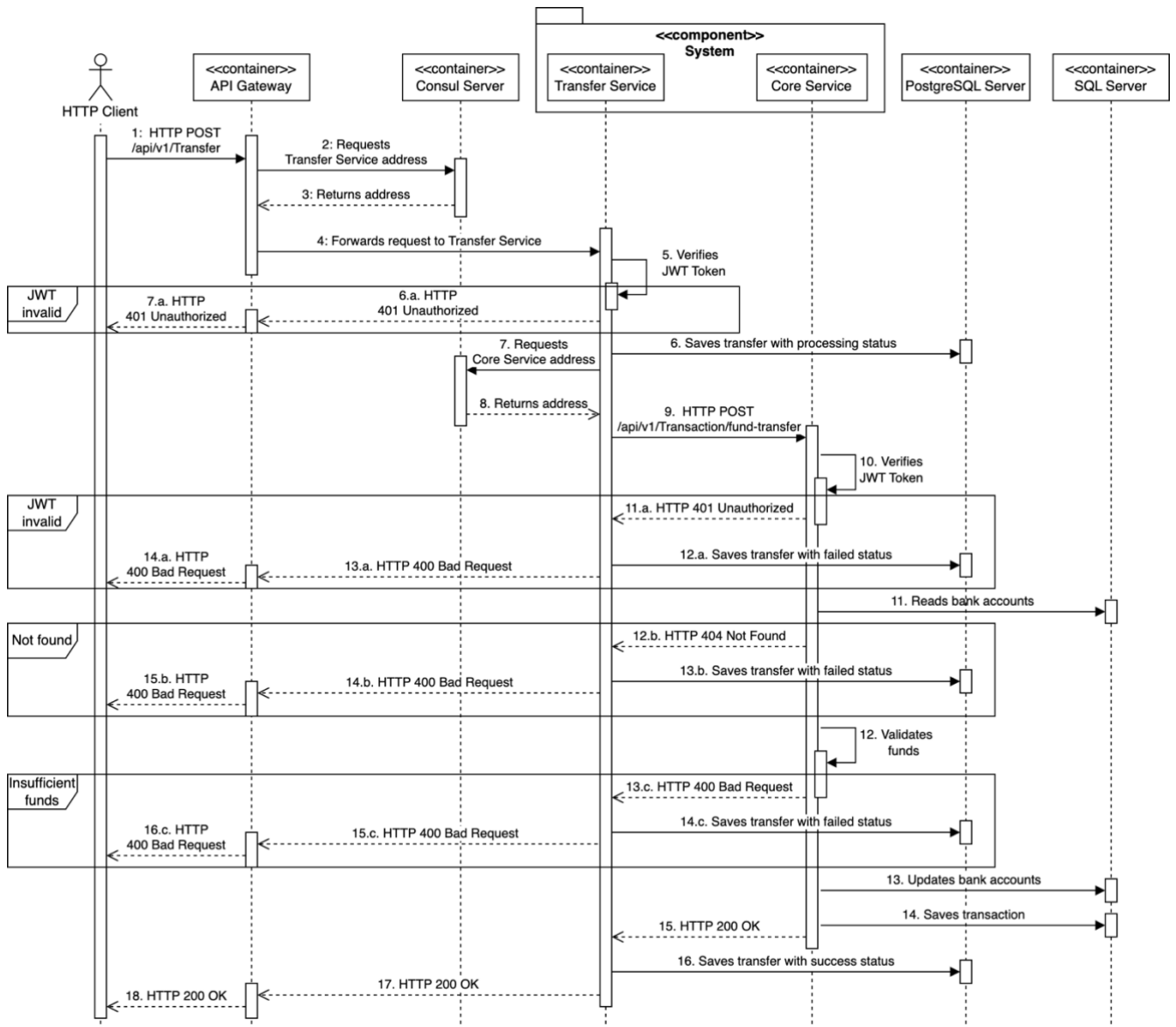


Figure 80 - Process view of the container level of the system regarding the execution of a fund transfer with all possible outcomes

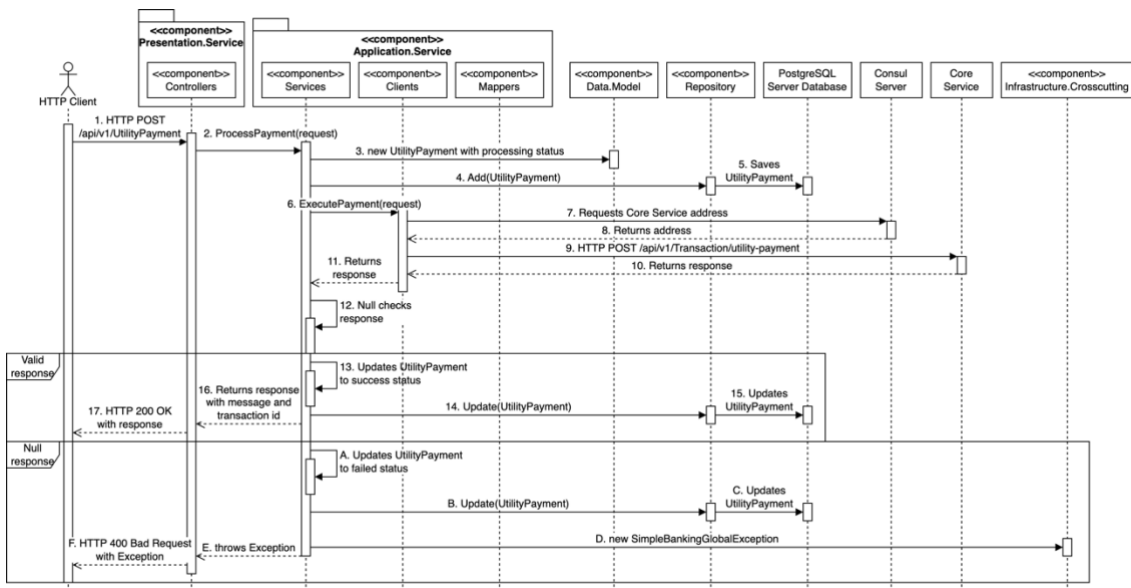


Figure 81 - Process view of the container level of the Payment Service container regarding payment execution with all paths

B.3 Cloud Run additional configuration options

General

Container port

8080

Requests will be sent to the container on this port. We recommend listening on \$PORT instead of this specific number.

Container command

Leave blank to use the entry point command defined in the container image.

Container arguments

Arguments passed to the entry point command.

Startup CPU boost

Start containers faster by allocating more CPU during startup time. [Learn more](#)

Capacity

Memory

512 MiB

Memory to allocate to each instance of this container.

CPU

1

Number of vCPUs allocated to each instance of this container.

Request timeout

300

seconds

Time within which a response must be returned (maximum 3600 seconds).


Maximum concurrent requests per instance

80

The maximum number of concurrent requests that can reach each instance. [What is concurrency?](#)

Figure 82 – Cloud Run container options regarding general and capacity options

Execution environment

The execution environment your container runs in. [Learn More](#) 

- Default**
Cloud Run will select a suitable execution environment for you.
- First generation**
Faster cold starts.
- Second generation**
Network file system support, full Linux compatibility, faster CPU and network performance.

Environment variables

[+ ADD VARIABLE](#)

Secrets

[ADD A SECRET REFERENCE](#)


Health checks


[+ ADD HEALTH CHECK](#)

Cloud SQL connections

[+ ADD CONNECTION](#)

Figure 83 – Cloud Run container options regarding execution environment, environment variables, secrets, health checks, and cloud SQL connections

Connect to other Google Cloud services like Google Cloud Storage or Google Cloud Firestore directly from your code. [Learn more](#) 

- Use HTTP/2 end-to-end
Use if your container is a gRPC streaming server or is able to directly handle requests in HTTP/2 cleartext. [Learn more](#) 

- Session affinity
Best effort to route requests from the same client to the same container instance.




- Connect to a VPC for outbound traffic
Allow your service to send traffic to a Virtual Private Cloud. [How to choose between VPC options?](#) 

Figure 84 – Cloud Run network options

Service account * 

Identity to be used by the created revision.

- Verify container deployment with Binary Authorization 
Clicking the checkbox will enable the Binary Authorization API.

Encryption

- Google-managed encryption key
No configuration required
- Customer-managed encryption key (CMEK)
Manage via [Google Cloud Key Management Service](#)

Figure 85 – Cloud Run security options