Monitoring and Detection of Anomaly in Microservices Environments

Lauriana Patrícia Tavares Landim

Supervisors

Professor Eurico Ribeiro Lopes

Professor Luís Miguel Santos Silva Barata

Dissertation submitted to the School of Technology of the Polytechnic Institute of Castelo Branco in order to fulfill the requirements for obtaining the degree of Master's in Software Development and Interactive Systems, held under the scientific guidance of Coordinating Professor, Dr. Eurico Ribeiro Lopes, and the Coordinating Professor, Luís Barata of the Polytechnic Institute of Castelo Branco.

March 2023

Composição do júri

Presidente do júri

Doutora, Ângela Cristina Marques Oliveira

Vogais

Doutor, Mário Marques Freire

Professor Catedrático do Departamento de Informática da UBI

Doutor, Alexandre José Pereira Duro da Fonte

Professor Adjunto do IPCB

Doutor, Eurico Ribeiro Lopes

Professor Coordenador do IPCB

Dedicatory

To my parents, who have always been my support and helped me become who I am today.

Acknowledgments

First of all, I thank God for all my achievements and for never letting me lose faith, even in the most difficult moments.

I also thank my supervisors, professor Eurico Lopes and professor Luís Barata, for their constant support since the beginning, for their motivation and patience during all the stages of this project. Your encouragement helped me to improve more and more.

I thank my lovely boyfriend, for his love and for accompanying me on this journey, standing always by my side in the achievements and in the moments of frustrations.

I also thank my families who, even though they are far away, always support me in every way.

And for all the other people not mentioned who indirectly contributed to this thesis, I thank you sincerely.

Summary

Microservices architectures have become increasingly popular in recent years because of their scalability and agility. However, the distributed nature of this architecture also introduces some challenges, especially in terms of monitoring and detecting anomalies. Anomaly detection is the process of identifying anomalous events or patterns in data that do not conform to expected behavior. In microservices environments, this eventually becomes very important, since the number of services tends to grow increasingly, making the interaction between them complex.

Because it is recent, there are still few studies on the best approaches to detecting anomalies in microservices. This thesis investigates how well PyOD library algorithms can detect anomalous behavior in a microservices dataset. PyOD is an open-source Python toolbox for performing scalable outlier detection on multivariate data. Some benefits of PyOD are that it is scalable, includes several algorithms, and can detect anomalies in multivariate data. We also review among the PyOD, KNN and HBOS algorithms, which one performs better at detecting anomalies.

To evaluate the approach, we used TraceRCA dataset to detect anomalies such as application bugs, CPU exhausted, and network jam. This dataset contains logs from a real microservices system. The preliminary results show that the HBOS algorithm performs better than kNN, with Recall and F1-Score of 83% and 91%, respectively, while for kNN these metrics were 80% and 89%, respectively.

Keywords

Microservices, monitoring, anomaly detection, PyOD, outliers algorithms.

Abstract

A arquitetura de microserviços têm-se tornado cada vez mais popular nos últimos anos, devido à sua escalabilidade e agilidade. Contudo, a natureza distribuída desta arquitetura também introduz alguns desafios, especialmente em termos de monitorização e deteção de anomalias. A deteção de anomalias é o processo de identificação de eventos ou padrões anómalos em dados que não estão em conformidade com o comportamento esperado. Em ambientes de microserviços, isto acaba por se tornar muito importante, uma vez que o número de serviços tende a crescer cada vez mais, tornando a interação entre eles complexa.

Por ser recente, existem ainda poucos estudos sobre as melhores abordagens para a deteção de anomalias em microserviços. Esta tese investiga até que ponto podem os algoritmos da biblioteca PyOD detetar comportamentos anómalos num conjunto de dados de microserviços. PyOD é uma biblioteca Python *open source* para efetuar a deteção de anomalias em dados multivariados. Também analisamos entre os algoritmos PyOD, KNN e HBOS, qual deles tem um melhor desempenho na deteção de anomalias.

Para avaliar a abordagem, utilizamos o conjunto de dados TraceRCA para detetar anomalias tais como bugs de aplicação, esgotamento do CPU, e interferência na rede. Este conjunto de dados contém registos de um sistema de microserviços real. Os resultados preliminares mostram que o algoritmo HBOS tem melhor desempenho do que o kNN, com Recall e F1-Score de 83% e 91%, respetivamente, enquanto que para o kNN estas métricas foram de 80% e 89%, respetivamente.

Palavras chave

Microserviços, monitorização, deteção de anomalias, PyOD, algoritmos outliers;

Index

List of Figures

List of tables

List of abbreviations and acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
CAPSI	Conferência –Associação Portuguesa de Sistemas de Informação
CPU	Central Process Unit
CRF	Conditional Random Field
CSV	Comma-Separated Values
ELK	Elasticsearch Logstash Kibana
FN	False Negative
FP	False Positive
GCE	Google Cloud Engine
HBOS	Histogram-Based Outlier Score
HDFS	Hadoop Distributed File System
HDS	HTTP Dynamic Streaming
HLS	HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
IMAP	Internet Message Access Protocol
IT	Information Technology
KNN	k Nearest Neighbors
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
OCI	Open Container Initiative
PyOD	Python Outlier Detection
gRPC	Google Remote Procedure Call
REST	Representational State Transfer
RPC	Remote Procedure Call
RTMP	Real-Time Messaging Protocol
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TN	True Negative
TP	True Positive
VM	Virtual Machine

1. Introduction

The work carried out in this thesis was based on the positions of several authors and researchers, who have done investigations and have some expertise on the topic of the work. Of all the authors referred to, Sam Newman, Chris Richardson, and Marcello Cinque were the primary references for the work.

1.1. Characterization of the Context of the Problem

Microservices architecture has been gaining popularity, so much so that large enterprises have adopted this new architecture by migrating to microservices from a monolithic application or developing microservices-based applications from scratch. The motivations for this change rely on the various benefits and advantages of implementing this architectural style.

Instead of building a single giant, monolithic application, the idea is to split an application into a set of smaller and interconnected services (Richardson, 2016). Meanwhile, as the number of microservices increases, the interactions between them tend to increase significantly, causing difficulties in monitoring the system, and its evolution. It requires methods and tools to assist in monitoring the development of this type of architecture (Apolinário and França, 2021). The ultimate goal of monitoring is to correctly detect erroneous conditions before any business impact, while providing sufficient information for the problem to be remedied (Ohlsson, 2018). Accordingly, to (Cao et al., 2019), although there are many anomaly types in microservice, these can be classified as Request Exception, Runtime Exception, Timeout Exception and Other Exceptions. Other Exceptions include Security exceptions and Version Exceptions. Within the Runtime Exception category, there are more anomalies than in the other categories.

Research to detect anomalies in microservices have already been conducted, primarily using techniques such as log analysis and monitoring, trace comparison, statistical methods, and clustering-based techniques. However, there are still few studies that explore the use of algorithms for anomaly detection using dedicated tools.

1.2. Problem Formulation

One of the main challenges faced in microservices environments is fault monitoring due to the diversity of services. In other words, this means that by having more services in an application, there are more points of failure to investigate. As stated by (Hasselbring, 2016), since services can fail at any time, it is crucial to detect the failures quickly and, if possible, automatically restore services.

This thesis uses PyOD, an open-source Python toolbox for performing scalable outlier detection on multivariate data (Zhao et al., 2019). PyOD provides access to more than 40 detection algorithms, including probabilistic and proximity-based models. Through its wide range of algorithms, we choose k Nearest Neighbors (kNN) and

Histogram-based Outlier Score (HBOS) to predict anomalies in our dataset and compare the results.

1.3. Research Question

I. How well can the PyOD library algorithms detect anomalous behavior in a microservices dataset?
II. Among the PyOD, KNN and HBOS algorithms, which one performs better in detecting anomalies?

1.4. Goals

The primary objective of this thesis is to monitor and detect anomalies in microservice environments, demonstrating to what extent existing solutions assist in problem-solving associated with the adoption of this architecture. To achieve this, the following goals will be pursued:

1. Define and collect metrics about microservices in a given context.
2. Identify and prepare the dataset.
3. Detect anomalies in microservice using PyOD Anomaly Detection Algorithms.
4. Compare the result of the metrics obtained for different algorithm types.
5. Implement an alert system prototype when an anomaly is detected. (Optional)
6. Implement a root cause analysis solution. (Optional)

1.5. Structure

This thesis is structured as follows:

Chapter 2 presents an overview of concepts and technologies that are fundamental to this work, as well as related work on the topic of this thesis. This includes reviewing the literature on microservices, monitoring, anomaly detection, and virtualization technologies.

Chapter 3 explains the proposed approach and the entire process involving data preparation and solution implementation.

Chapter 4 presents the method used and the dataset. It also specifies the metrics for validating the approach and describes the obtained results for each of the algorithms presented in the chapter 3.

Chapter 5 describes the metrics used to validate the model and also analyzes and discuss the results obtained.

Chapter 6 concludes the thesis with a summary of the work carried out and the research gains. It also describes some aspects considered important for future work.

1.6. Timeline of activities

The development of this work was subdivided into a number of stages carried out over approximately one year. The timeline below illustrates stages activities during the entire process, from start to completion.

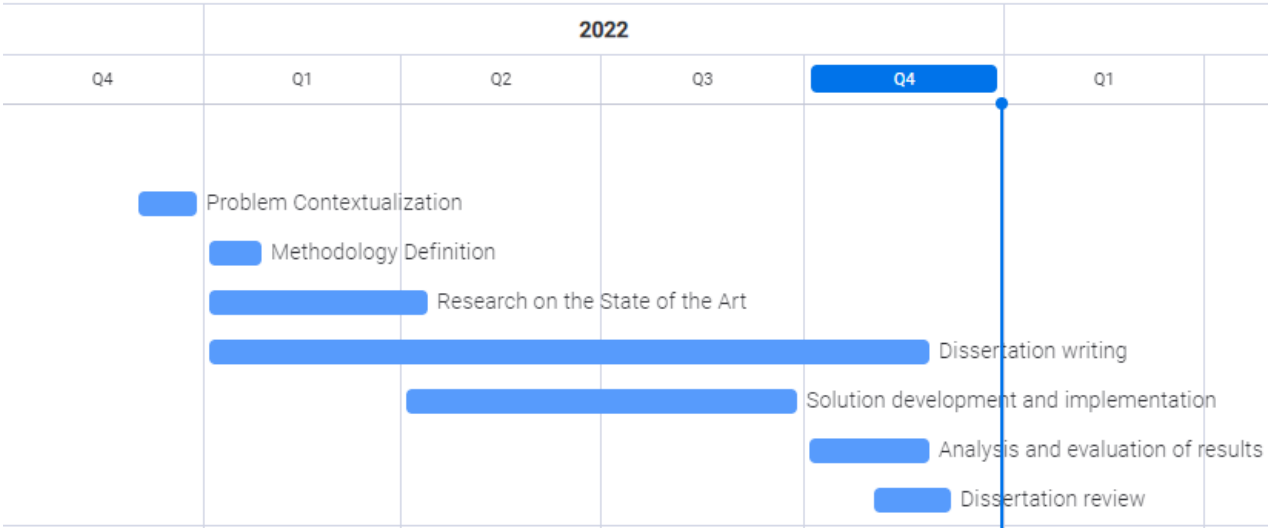

Figure 1 - Timeline of activities execution.

The project was divided into seven phases in total. The first phase focused on contextualizing the problem, defining the object of the thesis. The second phase consisted in defining the methodology and the path to be followed in order to carry out the work. In the third phase, a state-of-the-art survey was started, identifying the literature related to the topic of this work and using it as a basis. In parallel, the writing of the dissertation was started, as progress was being made. In the development and implementation phase of the solution the approach to be used is defined, including the dataset, the libraries, and the method for anomaly detection. After the solution was implemented, the results obtained for the approach used were analyzed and evaluated. Finally, the thesis and the project as a whole were revised.

1.7. List of Publications

This thesis was partially based on the work presented at the following conference, in which it was proposed the PyOD toolkit for anomaly detection in microservices:

Landim, L., Barata, L., & Lopes, E. (2022). A Simple approach to detect anomalies in microservices-based systems using PyOD. 22ª Conferência Da Associação Portuguesa de Sistemas de Informação, 10.

2. Literature Review and Related Work

In this chapter, we cover the main concepts and technologies related to microservices. We first discussed the microservices architecture and its characteristics, then we presented monitoring and anomaly detection and their application in microservices. Finally, we presented virtualization technologies and performed a comprehensive analysis of existing articles and research papers on this topic.

2.1. Microservices Architecture

As more businesses seek to reduce development costs and increase agility, microservices architecture is becoming increasingly popular. (Fowler and Lewis, 2014), define microservice architecture as an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. According to (IBM Cloud Education, 2021), microservices are widely used and benefit many industries worldwide. These benefits include faster delivery, improved scalability, more flexibility in technology choice, and greater autonomy and will be described further on. Twitter, Netflix, Apple, and eBay are examples of organizations from various domains that take advantage of these benefits to adopt microservices-based approaches.

Instead of a single monolithic unit, applications built using microservices are made up of autonomous, loosely coupled services (Bruce and Pereira, 2019); in other words, they are loosely dependent on each other. An example of this architectural style is shown in figure 2. As pointed out by (Newman, 2015), when services are loosely coupled, a change to one service should not require a modification of another. The whole point of a microservice is to make a change to one service and deploy it without needing to change any other part of the system. Although some definitions given by the authors vary a bit, the terms, independent, small and autonomous are always evident. The concept of microservices is also often accompanied by the definition of the Single Responsibility Principle, coined by Robert C. Martin. Furthermore, according to (Newman, 2015) the principle states "Gather together those things that change for the same reason and separate those things that change for different reasons.", reinforcing the idea of autonomous and independent services.

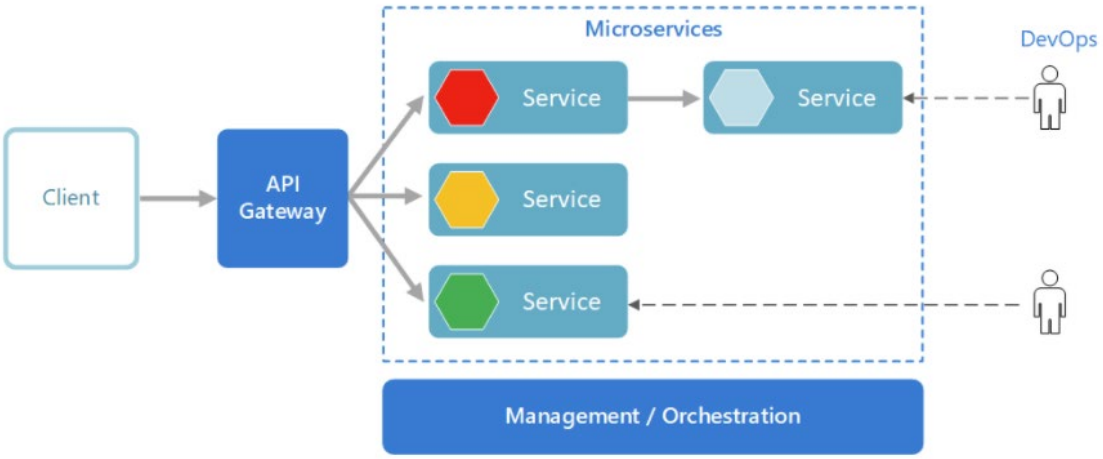

Figure 2 - Microservices Architecture Style, Source (Price et al., 2021)

2.1.1. Common Microservice Architecture Characteristics

Although different authors have presented definitions for Microservices Architecture, which vary somewhat from one another, there are characteristics that they all advocate. Based on the insights of (Newman, 2015), (Bruce & Pereira, 2019), and (Fowler & Lewis, 2014) we can conclude that these characteristics could be summarized in the following:

- **Focused on Doing One Thing Well** – the services are independent and focused on doing one thing well, following the principle of Single Responsibility Principle, which says:

"Gather together those things that change for the same reason, and separate those things that change for different reasons." (Newman, 2015)

- **Autonomy** – services are developed around business resources and each service is a separate entity and operates and changes independently of the others.
- **Resilience and Isolation of Failures**- applications need to be designed to tolerate service failure. (Bruce and Pereira, 2019) state that microservices are a natural mechanism for isolating failure: application or infrastructure failure may only affect part of the system if deployed independently.
- **Decentralized Data Management** - Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems (Fowler and Lewis, 2014).
- **Decentralized Governance** - Each service can choose the best combination of technology for the use cases. This also allows for a different form of standardization, where developers create useful tools to be used by others when solving problems similar to the ones they are facing.

2.1.2. Communication in Microservices Systems

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol such as HTTP, Advanced Message Queuing Protocol (AMQP), or a binary protocol like TCP, depending on the nature of each service (Anil, 2021). In an architecture where one of the main objectives is the development of loosely coupled services independent of each other, communication between these services is fundamental. To solve problems related to communication in distributed system, some standards have been developed. The most common communication styles according to (Petrasch, 2017) are: messaging, domain-specific protocol, and Remote Procedure Call (RPC). Anyway, the communication strategy sometimes varies from case to case. In table 1, it is possible to see the comparison between the three communication styles.

Table 1 - Communication Styles in Microservices, source (Richardson, 2021)

Mechanisms	Solution	Example of technologies
Messaging	Use asynchronous messaging for inter-service communication. Services communicating by exchanging messages over messaging channels.	• Apache Kafka • RabbitMQ
Remote Procedure Call (RPC)	Use RPC for inter-service communication. The client uses a request/reply-based protocol to make requests to a service.	• REST • gRPC • Apache Thrift
Domain-specific protocol	Use a domain-specific protocol for inter-service communication.	• Email protocols such as SMTP and IMAP • Media streaming protocols such as RTMP, HLS, and HDS

2.1.3. Benefits of Microservice Architecture

In monolithic applications, fixing bugs or making new developments becomes a complex task as the system grow. This process is even more complicated for companies that need as much agility as possible to keep up with the growth of the business. The microservices architecture has many benefits that overcome the most common problems of monolithic applications. According to (Jamshidi et al., 2018), three of the most important ones are faster delivery, improved scalability, and greater autonomy. Fast delivery is associated with ease of deployment, also considered a benefit of

microservices by (Newman, 2015). Still, according to the same author, with microservices, a change can be made to a single service and deployed independently of the rest of the system. This will allow the code to be deployed faster, and if a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve. It also means that the new functionality can be brought to customers faster.

Improved scalability is achieved through small services that enable to scale just the services that need scaling, allowing to run other parts of the system on smaller and less powerful hardware, minimizing cost (Newman, 2015). Another benefit of microservices is technological heterogeneity since, with this architecture, there is more flexibility in choosing the technological stack for a given type of problem. Eventually, microservices are designed to be resilient, which means improved service availability and an uninterrupted user experience (Daya et al., 2015).

The chart in figure 3 illustrates the main advantages of microservices according to a study by (Viggiato et al., 2018). The results show that on a scale from one (very important) to four (not important), clearly: Independent deploy (45.7, very important) is considered the most prominent advantage in a microservices architecture. However, for the other three advantages, we can see that more than 50% of respondents chose scores one or two, indicating that these characteristics are indeed relevant when working with microservices.

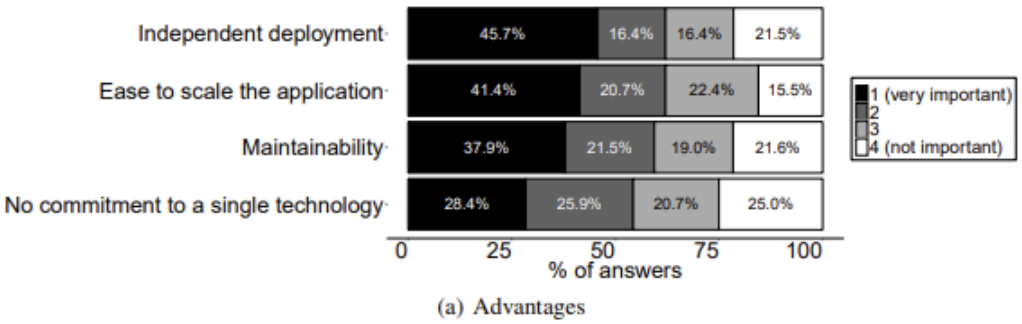

Figure 3 - Advantage of Microservices – Source (Viggiato et al., 2018)

2.1.4. Challenges of Microservice Architecture

Since Microservices architecture is still new, there are many open questions and possibilities for optimization in this field. Although there is already some progress in the research on the topic, the same studies have proven that some aspects and challenges of this architecture are still unexplored. According to research conducted by (Ghofrani & Lübke, 2018), the distributed nature of microservices architecture is one of the main challenges in developing and debugging systems based on this architectural style.

Another biggest challenge of this architecture is dealing with communication between services caused by the distribution of the system's various components. This

is because, unlike a monolithic application, all services in a Microservice architecture are deployed separately and need to communicate with each other. This can lead to issues such as latency, data consistency, and network failures. Another challenge, as stated by (Jamshidi et al., 2018), is managing and monitoring resources. With more services, more components are needed to test and debug, making it harder to identify and fix problems quickly. Finally, Microservice architecture requires a lot of DevOps to keep everything running smoothly. This includes monitoring, logging, and managing all the services' performance and security.

Despite the challenges, microservice architecture is still a great way to create complex systems and applications as long as these challenges remain paramount. One can create a robust and secure system with the right approach and proper planning.

2.2. Microservices and Monolithic Applications

Monolithic applications have been the traditional approach to software development for years, but they are now being replaced by a more flexible and efficient model known as microservices.

In a monolithic architecture, all functionality is encapsulated into one single application, so its modules cannot be executed independently. This type of architecture is tightly coupled, and all the logic for handling a request runs in a single process (Ponce et al., 2019). While this approach works well in certain situations, it can lead to problems when the system grows in size and complexity. Over time, it can be challenging to know where a change needs to be made because the codebase is so large. Despite a drive for clear, modular monolithic codebases, arbitrary in-process boundaries often break down (Newman, 2015). Consequently, as per (Richardson, 2016), it makes fixing bugs and implementing new features correctly more difficult and time consuming. Microservices, on the other hand, are built as multiple, independent services that can be changed, updated, and deployed independently. This makes them more flexible and adaptable to changing business needs. In addition, microservices are more scalable than monolithic applications, as individual services can be added or removed as needed. These capabilities are detailed in figure 4. Unlike monolithic applications, microservices are more fault-tolerant and reliable. If a service fails, other services will still continue to function, reducing the risk of downtime. However, the trade-off is that it can be more difficult to debug since, the individual services must be debugged separately.

Ultimately, the decision of which approach to use depends on the project's requirements. If in need of something that is faster and easier to implement, a microservices architecture may be the right choice. However, if scalability is not required and a more straightforward and easier-to-maintain approach is needed, a monolithic application may be a better fit.

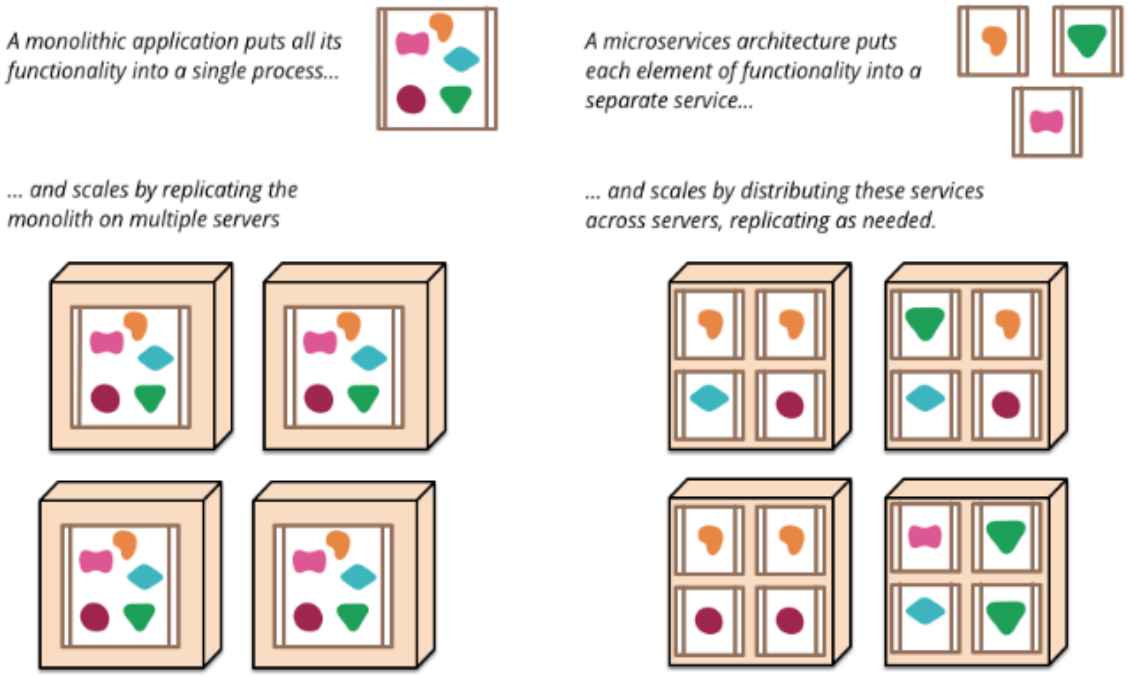

Figure 4 - Monolithic architecture and Microservices architecture (Fowler and Lewis, 2014).

2.3. Monitoring

Microservices monitoring is essential for any enterprise that relies on this technology to keep its applications running. Through service monitoring, it is possible to identify potential issues before they become significant problems. It is a core reliability engineering practice, that aims to improve service continuity and reduce downtime by gathering and analyzing various data sources, such as metrics, event logs and traces, on to the execution of a given system (Cinque et al., 2019). Monitoring can provide an early warning system of something going wrong that triggers development teams to follow up (Hasselbring, 2016). This helps ensure that the system is running as expected and without any glitches.

2.3.1. Log Analysis

Gathering feedback about computer systems states is a daunting task. To this aim, it is a common practice to have programs report on their internal state, for instance, through journals and logfiles, that can be analyzed by system administrators (Bertero et al., 2017). Logs are human-readable text files that report sequences of text entries ranging from regular to error events. A typical log entry contains a timestamp, the identifier of the source of the event, a severity (e.g., debug, warning, error), and a free text message (Cinque et al., 2013). In microservices log analysis is a valuable tool for monitoring and maintaining the performance of the systems. By logging, an application can easily provide information about which events occurred. These can be errors, but also events like the registration of a new user, which are mostly interesting for statistics (Wolff, 2016). Popular tools for log analysis include: Splunk ("Splunk | Turn Data Into Doing," 2021), Logstash ("Logstash," 2021) and Graylog ("Graylog," 2021).

2.3.2. Monitoring Tools

As microservice applications' size and complexity grow, the number and diversity of infrastructure resources (for example, virtual machines, containers, services, messages, thread pools, and logs) that must be continuously monitored and managed at runtime also increase (Jamshidi et al., 2018). While microservices monitoring can be a complex process, several tools and resources are available for monitoring, including open source solutions, cloud-based services, and commercial monitoring solutions. The following is a description of some of these existing monitoring solutions.

ELK Stack

ELK is an acronym for several open source tools: Elasticsearch, Logstash, and Kibana. Elasticsearch is the engine of the Elastic Stack, which provides analytics and search functionalities. Logstash is responsible for collecting, aggregating, and storing data to be used by Elasticsearch. Kibana provides the user interface and insights into data previously collected and analyzed by Elasticsearch (Amoany, 2021).

Prometheus

Prometheus is an open-source monitoring and alerting toolkit that is often used in combination with Grafana to visualize the collected metrics. (AWS, 2021).

Graphite

Graphite can store numerical data and is optimized for processing time series data. Such data frequently occur during monitoring. The data can be analyzed in a web application. Graphite stores the data in its own database. After some time, the data are automatically deleted. It accepts monitoring values in a straightforward format via a socket interface (Wolff, 2016).

Zipkin

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data (Zipkin, 2021).

Grafana

Grafana is an open-source software for monitoring and analyzing. One of its major characteristics is that it supports many different data sources, from popular CloudWatch, Elasticsearch, Graphite, and influxDB, to OpenStack Gnocchi or Google Calendar (Yang and Huang, 2019).

2.4. Anomaly Detection

In microservices-based systems, anomaly detection is crucial because, as they grow larger, these types of systems sometimes become more prone to failure. For Chandola et al. (2009), anomaly detection refers to the problem of finding patterns in data that do not conform to the expected behavior. These nonconforming patterns are often

referred to as anomalies, outliers, discordant observations, exceptions, aberrations, surprises, peculiarities, or contaminants in different application domains. Anomalies in a system can be defined as deviations from normal behavior. Anomalies often indicate that a component is failing or is about to fail and should hence be detected as soon as possible (Forsberg, 2019).

2.4.1. Anomaly Types

The data produced by microservices is not the same. Consequently, the nature of the anomalies manifests itself in several ways. An anomaly can be categorized in the following ways (Ahmed et al., 2016).

Point anomaly

When a particular data instance deviates from the usual pattern of the dataset, it can be considered a point anomaly. For a realistic example, if a persons' normal car fuel usage is five liters per day but if it becomes fifty liters in any random day, then it is a point anomaly (Ahmed et al., 2016). This thesis focuses on these types of anomalies.

Contextual anomaly

When a data instance behaves anomalously in a particular context, it is termed a contextual or conditional anomaly; for example, expenditure on a credit card during a festive period, e.g., Christmas or New Year, is usually higher than during the rest of the year. Although it can be high, it may not be anomalous as high expenses are contextually normal in nature. On the other hand, an equally high expenditure during a non-festive month could be deemed a contextual anomaly (Ahmed et al., 2016).

Collective anomaly

When a collection of similar data instances behaves anomalously concerning the entire dataset, the group of data instances is termed a collective anomaly (Ahmed et al., 2016). As an illustrative example given by (Chandola et al., 2009), we have the following sequence of actions on a web server:

… http-web, buffer-overflow, http-web, http-web, smtp-mail, ftp, http-web, ssh, smtp-mail, http-web, **ssh, buffer-overflow, ftp**, http-web, ftp, smtp-mail, http-web …

The highlighted sequence of events (buffer-overflow, ssh, ftp) corresponds to a typical Web-based attack by a remote machine followed by copying of data from the host computer to a remote destination via ftp. It should be noted that this collection of events is an anomaly, but the individual events are not anomalies when they occur in other locations in the sequence (Chandola et al., 2009).

Figure 5 illustrates these three types of anomalies in a sensor network. A point anomaly occurs in the period from 20 to 40, where one instance of data is abnormal relative to the others. A contextual anomaly occurs at time period 43, but it would not be considered an anomaly if it had occurred at times t1, t2, t3 or t4. Finally, a collective

anomaly occurs between t2 and t3 and, although the data instances have the same pattern, compared to the whole dataset, they are an anomaly.

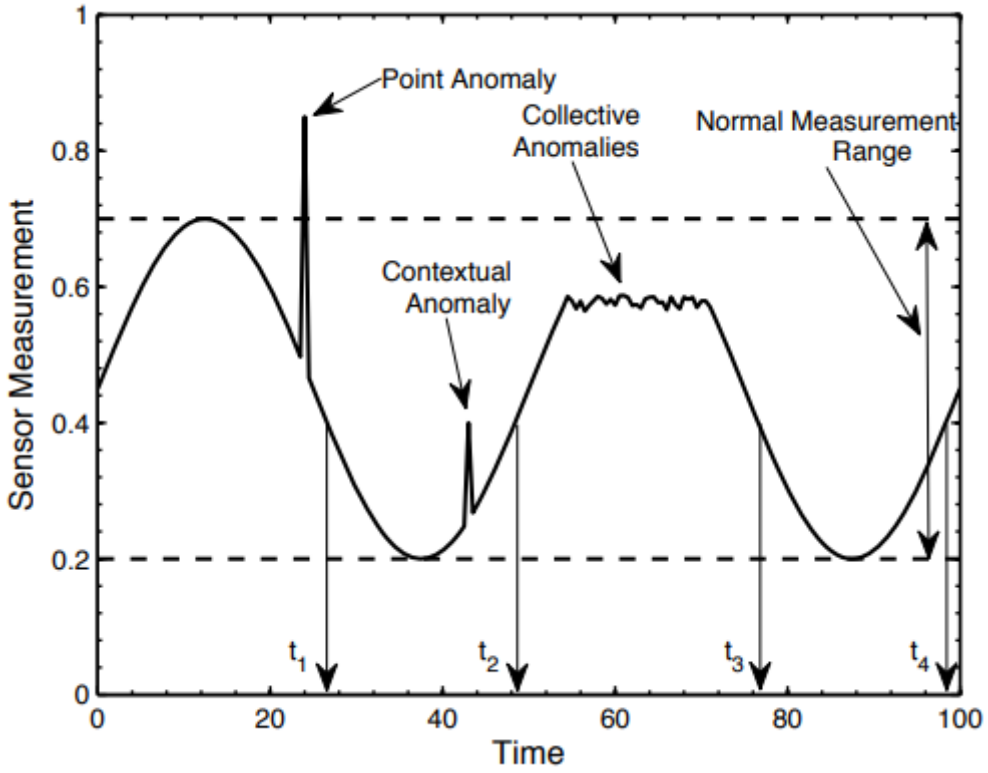

Figure 5 - Point, Contextual and Collective Anomalies (O'Reilly et al., 2014).

2.4.2. Anomaly Detection in Microservices

Though the focus of this thesis is on anomaly detection in microservices, anomaly detection has been widely applied in countless application domains such as medical and public health, fraud detection, intrusion detection, industrial damage, image processing, sensor networks, robots behavior, and astronomical data, as stated by Ahmed et al. (2016). The application of anomaly detection in the metrics emitted by microservices like CPU and memory utilization, error rate and response time, improves its reliability because these metrics provide important information about the components' performance, thus allowing to act in time in case some abnormal behavior is identified. To do this, it is necessary to know the normal state of the microservice. As per (Düllmann, 2017), this means that the existing data (e.g., response times) is used to learn the normal behavior of an application. To decide whether the current situation is an anomaly or not, thresholds can be set. This can happen either manually by setting the threshold to a fixed value or can be determined automatically using the normal behavior as an indicator. Figure 6 illustrates the normal behavior of the data and then what is considered abnormal, comprising an anomaly.

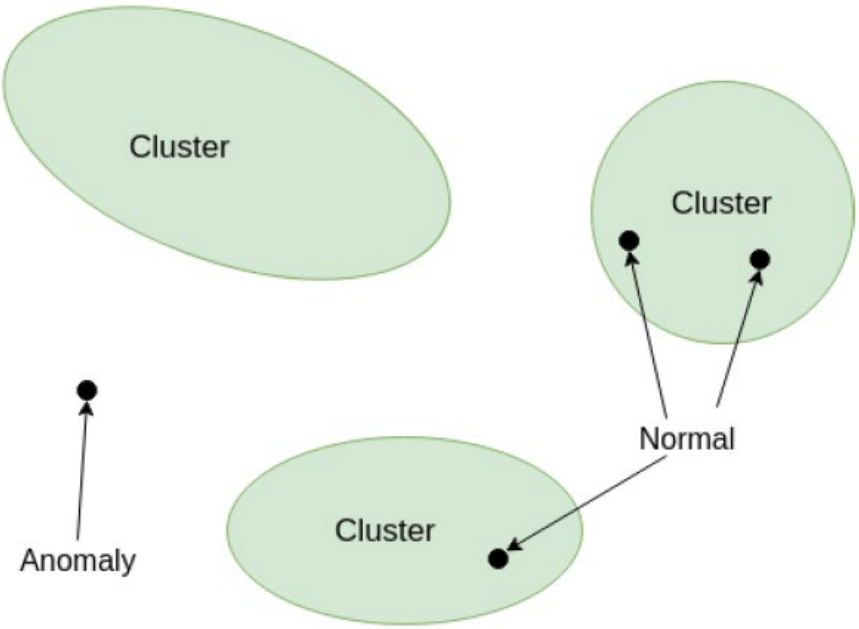

Figure 6 - Clusters that represents the normal behavior. Data points outside the clusters are considered anomalies (Forsberg, 2019).

2.4.3. Anomaly Detection Solutions

While this thesis has a focus on anomaly detection using PyOD, there are other methods based on other approaches, some even using different technologies. Below are other alternatives, some of them in Python.

Tensor-based Outlier Detection (PyTOD) it is a tensor-based outlier detection system that abstracts Outlier Detection (OD) applications into tensor operations for efficient GPU acceleration. TOD leverages both the software and hardware optimizations in modern deep learning frameworks to enable efficient and scalable OD computations on distributed multi GPU clusters (Zhao et al., 2022).

Scikit-learn Novelty and Outlier Detection, the scikit-learn project provides a set of machine learning tools that can be used both for novelty and outlier detection. It supports some popular algorithms like LOF, Isolation Forest, and One-class SVM (scikit-learn, 2022).

Python Streaming Anomaly Detection (PySAD) is an open-source python framework for anomaly detection on streaming multivariate data. PySAD provides methods for online/sequential anomaly detection, for example, anomaly detection on streaming data, where model updates itself as a new instance arrives (Yilmaz, 2022).

RapidMiner Anomaly Detection Extension, the Anomaly Detection Extension for RapidMiner comprises the unsupervised anomaly detection algorithms, assigning individual anomaly scores to data rows of sample sets. It allows finding data, which are significantly different from normal, without the need for the data to be labeled (Amer and Goldstein, 2012).

2.5. Virtualization

The ability to abstract a physical server to a virtual machine has become a common trend in computing. This abstraction is commonly referred to as virtualization. (Golden, 2011), defines virtualization as an approach to pooling and sharing technology resources to simplify management and increase asset use so that IT resources can more readily meet business demand. Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs) (IBM, 2021a). As depicted in figure 7, this means that businesses can run multiple operating systems on the same physical server, reducing the need for multiple physical machines. Virtualization also allows for the quick deployment of new services or applications, as well as the ability to quickly scale up or down depending on demand.

In compute virtualization, there is a virtualization layer between the hardware and the virtual machine. The virtualization layer is also called a hypervisor (Jain and Choudhary, 2016). As stated by (Chiueh, 2020), most of virtualization technologies present similar operating environments to the end user; however, they tend to vary widely in their levels of abstraction they operate at and the underlying architecture.

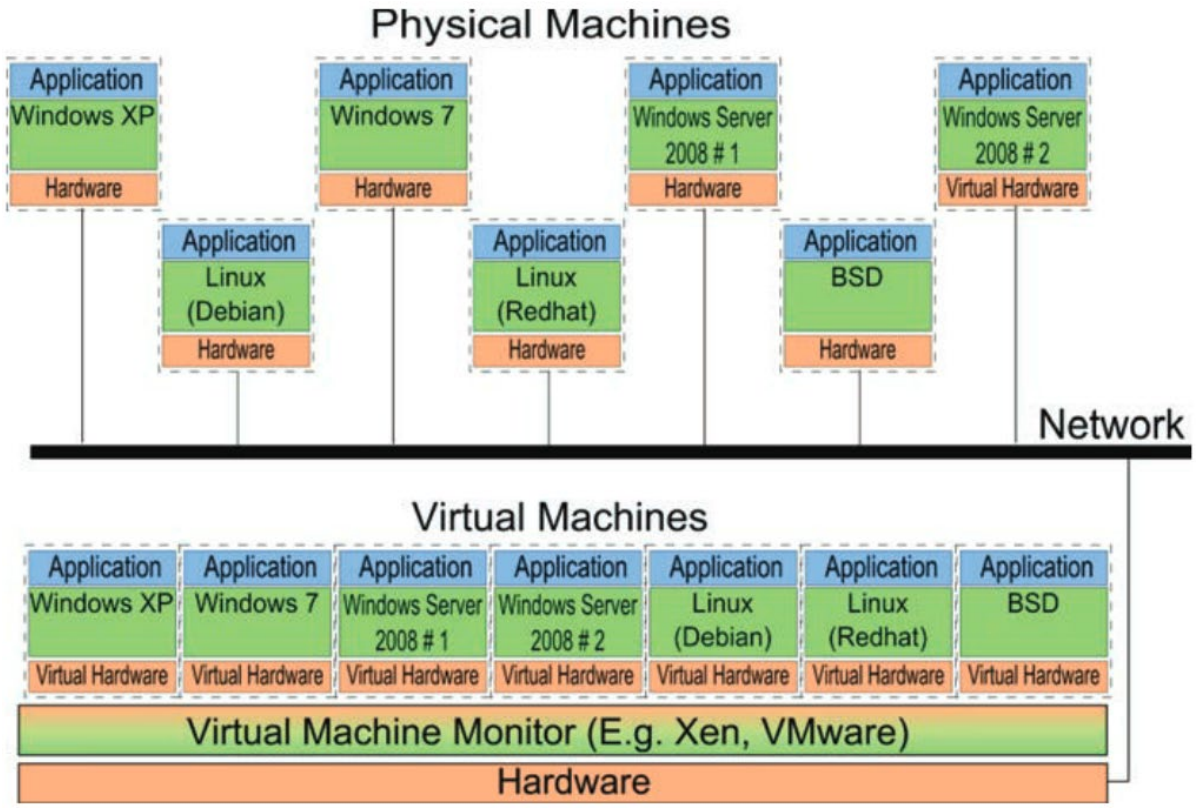

Figure 7 - Seven physical systems (top) and a virtualized equivalent implementation (bottom), source (Pearce et al., 2013)

2.5.1. Virtualization Types

Although there are several types of virtualizations today, some are more popular due to the frequency with which it is used. They are as follows:

Server Virtualization

Basically, server virtualization enables the use of multiple virtual servers (virtual machine) on a physical server by using virtualization software. Here, each virtual server behaves as a physical device running its own operating systems (Jain and Choudhary, 2016).

Network virtualization

It is a method of combining the available resources in a network by splitting up the available bandwidth into channels. Each channel can be assigned to a particular server or device in real-time (Pawar and Bhelotkar, 2011).

Storage virtualization

Is the process of abstracting logical storage from physical storage. The physical storage resources (such as disk drives) are aggregated into storage pools, from which the logical storage is created and presented to the application environment. (Golden, 2011).

Application virtualization

Refers to a separation of program execution from program display; in other words, a program like Microsoft Word executes on a server located in the data center, but the graphical output is sent to a remote client device. The end-user sees the full visual display of the program and is able to interact with it via keyboard and mouse (Golden, 2011).

Desktop Virtualization

Unlike application virtualization, where one or more applications are displayed or streamed from a central server, in desktop virtualization, a user's entire PC executes on a central server, with the graphical display output to a client device. (Golden, 2011). The advantage is that it makes it easier to deploy updates, control access, and ensure compliance.

2.5.2. Containers

Containers are a type of virtualization that has been very important for modern computing. According to (Burns et al., 2016), containers encapsulate the application environment, abstracting away many details of machines and operating systems from the application developer and the deployment infrastructure. This typical feature allows the applications to be easily and quickly deployed on any device, regardless of the underlying operating system. Containers are a similar but more lightweight virtualization concept; they're less resource and time-consuming, thus they've been

suggested as a solution for more interoperable application packaging in the cloud (Pahl, 2015).

Just like hypervisor-based virtualization, container is also a virtualization technique. In hypervisor-based virtualization as per (Jha et al., 2018), each VM has its own operating system irrespective of the host machine running on a hypervisor, whereas in container-based virtualization, the container utilizes the services provided by the host operating system using container engine. The difference between these two types of virtualizations is illustrated in figure 8, showing that they differ in their management of guest operating system components.

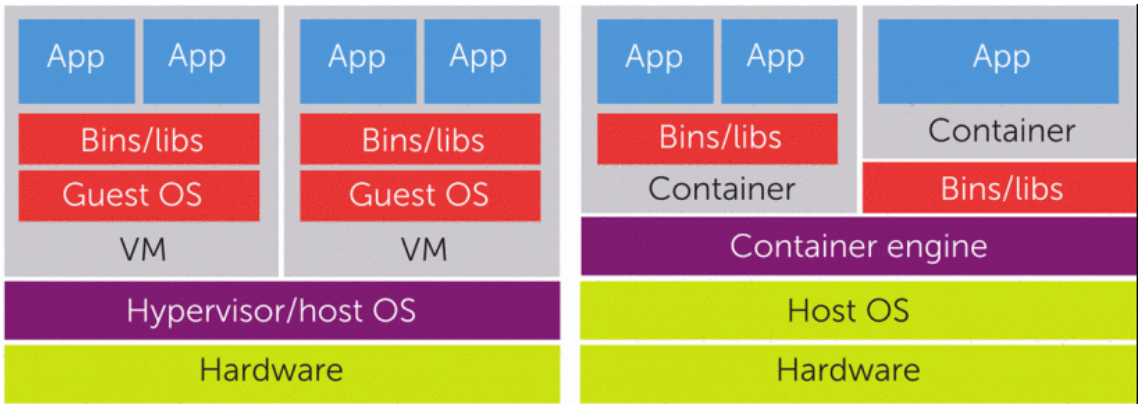

Figure 8 - Virtualization architecture with hypervisor and container, (Pahl, 2015)

Xen (Smyth, 2012), VMware (IBM, 2021b) and Kernel-based Virtual Machine (KVM) (Chirammal et al., 2016) are some examples of hypervisor-based virtualization, while Docker (Potdar et al., 2020) and LXC (Jha et al., 2018) represent container-based virtualization.

2.5.3. Docker

Docker is a container platform used to deploy applications in a virtual environment. It's a popular tool due to its ability to deploy applications quickly and easily across different platforms. As stated by (Ahmed and Pierre, 2018), with Docker applications are packaged in the form of images which contain a part of a file system with the required libraries, executables, configuration files, etc. It also appends an additional layer of deployment engine over a container environment where the applications are executed and virtualized (Potdar et al., 2020). The Docker platform essentially consists of a few components, namely: Docker Containers, Docker Client-Server, Docker Images, e Docker Registries. The following is a description of each of these components.

Docker Containers

Docker Containers are created by Docker image. To run the application in a confined way, every kit required for the application is to be held by the container. The container

images can be created based on the service requirement for the application or software (Potdar et al., 2020).

Docker Client-Server

Docker technology mainly refers to client-server architecture. The client communicates to the Docker daemon, which acts as a server that is present within the host machine. Figure 9 shows this communication with client and also with Registry. The daemon works as three major processes running, building, and distributing containers. Both the Docker container and daemon can be placed in a single machine (Potdar et al., 2020).

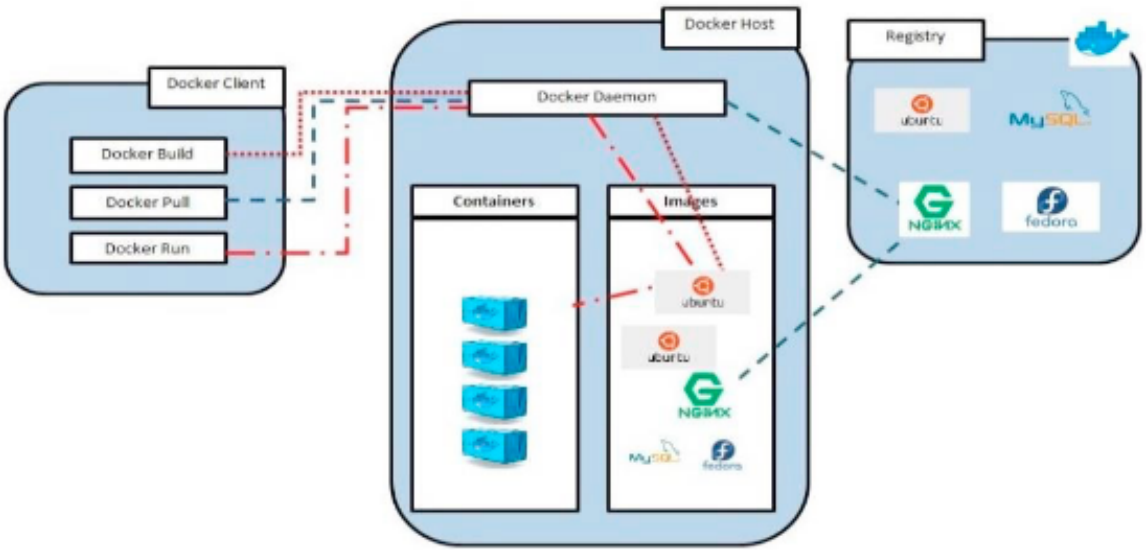

Figure 9 - Architecture of Docker Container, (Potdar et al., 2020)

Docker Images

Docker images are composed of multiple layers stacked upon one another: every layer may add, remove, or overwrite files present in the layers below itself. This enables developers to build new images very easily by simply specializing in pre-existing images (Ahmed and Pierre, 2018).

Docker Registries

Docker images are placed in docker registries. It works correspondingly to source code repositories where images can be pushed or pulled from a single source. There are two types of registries, public and private. Docker Hub is called a public registry where everyone can pull available images and push their own images without creating an image from scratch. Images can be distributed to a particular area (public or private) by using docker hub feature (Rad et al., 2017).

2.5.4. Kubernetes

According to (Burns et al., 2019) Kubernetes is a platform for creating, deploying, and managing distributed applications. These applications come in many different shapes and sizes, but ultimately, they are all comprised of one or more programs that run on individual machines.

From an architectural point of view, Kubernetes introduces the pod concept, a group of one or more containers (e.g. Docker, or any OCI compliant container system) with shared storage and network (Medel et al., 2018). Each pod contains one or more containers, which are isolated from each other. Kubernetes also has several features that allow monitoring resource usage, deploying applications across multiple clusters, and even automating the deployment of new services. Some of the organizations that support Kubernetes in their technologies include:

- Google (for Google Cloud Engine, GCE),
- Microsoft (for Microsoft Azure),
- VMware,
- IBM (for SoftLayer e OpenStack), and
- Red Hat (OpenStack Distribution).

2.5.4.1. Kubernetes Architecture

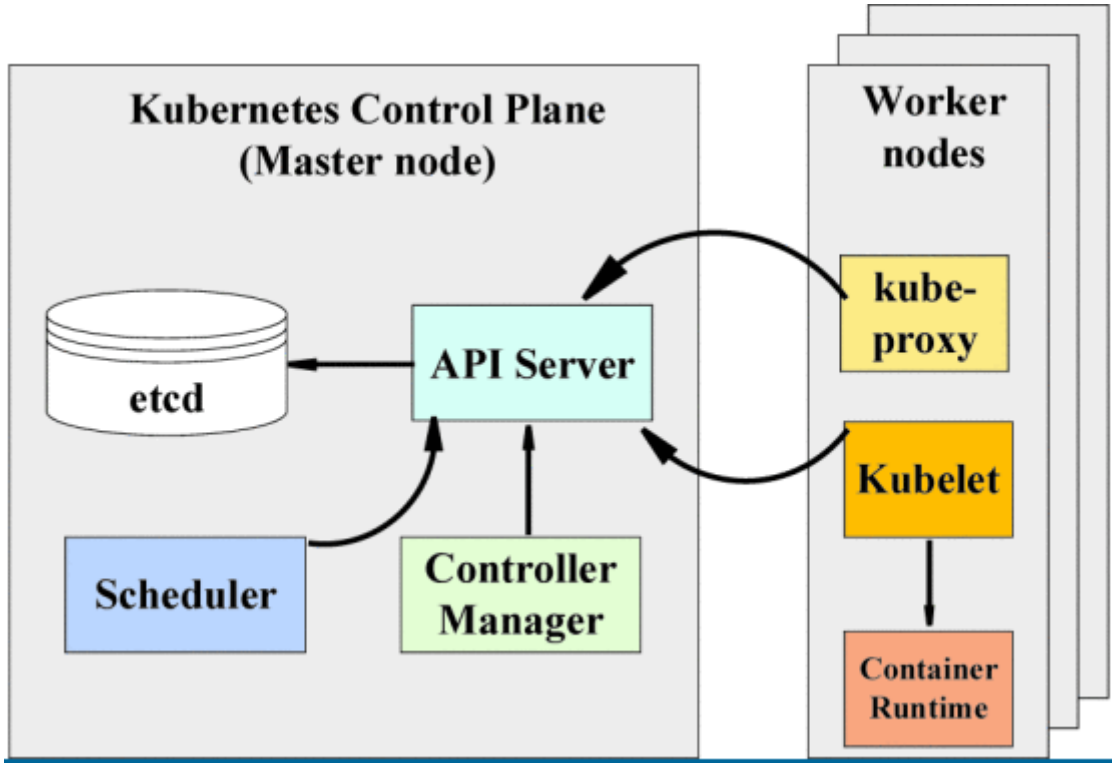

Figure 10 - Kubernetes architecture, source (Botez et al., 2020)

Kubernetes can also be considered a cluster system. Each cluster comprises nodes that are subdivided into two categories: the master and the worker nodes. Figure 10

illustrates this architecture. As per (Botez et al., 2020), the master has multiple components: (1) kube-apiserver: it provides communication between the other components; (2) kube-scheduler: is a component that assigns newly created resources to worker nodes; (3) etcd: is a data store which contains cluster configuration; and (4) kube-controller-manager, responsible for monitoring worker nodes, replication and others. Kubernetes worker nodes are responsible running the application containers . The worker node components are: kubelet - that handles containers and communicates with the API server, kube-proxy which performs connection forwarding between application components, and container runtime which is the software that runs the containers (Botez et al., 2020).

2.6. Related Work

In studying the literature, it was possible to verify that there are already some studies investigating anomaly detection in microservices. A brief comparison of these existing works is presented in Table 2. Many of them use statistical methods or methods based on log analysis. The analysis of these studies motivated the use of other approaches that rely on using a set of algorithms from an existing library.

(Meng et al., 2021) proposed an anomaly detection approach for microservices applications by comparing traces. They employed Bench4Q as a dataset to demonstrate the use of the proposed technique, and to validate the approach, they used a microservices-based application called Social Network (Yashchenko, 2016).

(Cao et al., 2019) used the Conditional Random Field (CRF) based anomaly detection method and proved through experiments that the method can accurately find the faults in microservice system, and the accuracy and the recall rate are relatively high.

(Barakat, 2017) used the Kieker framework for performance monitoring and analysis of a microservices-based application and determined that some parameters about service performance can be visualized.

(Sun et al., 2021) use the MonitorRank and FSB-RWRANK algorithms to automatically develop service dependency graphs and thereby monitor microservice indicators and detect possible anomalies.

(Li et al., 2021a) detected abnormal traces in microservices using a trace analysis-based approach called TraceRCA. They used the Train-Ticket benchmark to perform the experiments and deployed with Kubernetes.

(Wang et al., 2020) proposed an automatic anomaly diagnosis approach for microservices based applications with statistics. They built baselines using Call Trees, and the results showed that the method could accurately locate microservices causing anomalies.

Table 2 - Comparison of Related Work on Anomaly Detection

Works	Technique Used	Dataset	Indicators	Tools	Recommendations
(Meng et al., 2021)	Trace Comparison	Social Network Bench4Q	Anomalies in the structure of traces Response time anomalies	Zipkin and Jaeger for collection of traces. Docker Container and Kubernetes for service deploy.	They recommend an execution monitoring-based approach.
(Cao et al., 2019)	Conditional Random Field	Sock-shop	Authorization failure IO exception Remote connection timeout Operation exception Short circuit exception Data source connection exception Request parameter incorrect Semantic exception	Prometheus for monitoring Kubernetes for deploy	The experimental results show that the proposed method has greater precision than Markov's hidden model.
(Sun et al., 2021)	FSB-RWRank	Sock-shop	CPU utilization Memory utilization Latency Disk utilization	Kafta for collection of indicators HDFS for storing indicators Doris database for reporting visualization. Prometheus for collecting KPIs from microservices.	***
(Barakat, 2017)	Monitoring and analysis through Kieker Framework	Existing application, developed in Spring Framework	Count of calls to the method and components Minimum, average, maximum and total execution time of components	Kieker framework to monitor the performance of microservices Kieker Analysis for application analysis.	Extend the analysis with ExplorViz.
(Li et al., 2021a)	Trace analysis through TraceRCA	Train-Ticket	Application bugs CPU exhausted Network jam	Kubernetes for deployment	Extend for detection of structural anomalies
(Wang et al., 2020)	Statistical Methods and Traces Comparison	Train-Ticket	CPU hog Network congestion	Zipkin for collection of traces Docker for containers deploy	***

The entire literature review process conducted showed that microservices do indeed have various advantages. However, the surveys also point to some challenges, especially when it comes to monitoring services and detecting possible failures. For monitoring, there are already some tools, but in terms of anomaly detection, it was found that the studies are still very vague.

Also, the characteristics of autonomous and independent services have made microservices architecture often being associated with containers. The reason being that they are easy to deploy, scale and move with minimal impact, they meet the requirements of microservices.

Using this literature review as a basis, it is intended in the following chapters to address the method and tools to be used for these works.

3. Method

This chapter describes the approach used for our anomaly detection solution. We cover the library used, as well as the anomaly detection algorithms. We also discuss the source of the data, the entire process of preparing it and implementing the solution and finally, the metrics used to evaluate the model.

3.1. PyOD

There are many attempts to solve the anomaly detection problem. The more widely applicable approaches are unsupervised algorithms, as they do not need labeled training data meeting the requirements of practical systems (Amer & Goldstein, 2012). PyOD is an open-source Python toolbox for performing scalable outlier detection on multivariate data (Zhao, Nasrullah, & Li, 2019). Some benefits of PyOD are that it is scalable, includes several algorithms, and can detect anomalies in multivariate data. The toolkit consists mainly of 3 major groups, specifically: 1 - Individual anomaly detection algorithms, which includes more than 40 algorithms of varying types; 2 - Combined anomaly detection frameworks; and 3 - A set of utility functions. As per the authors, since 2017, PyOD has been successfully used in various academic research and commercial products, such as (Zhao et al., 2021) and (Zhao, Nasrullah, Hryniewicki, et al., 2019).

3.2. Dataset

To train our model with PyOD, it was used TraceRCA dataset (Li et al., 2021b), a study data, from another research using the trace analysis approach. This dataset contains two sets of data: A) and B). For this work, we used set B, which contains traces of a large Internet service provider's production microservice system. In addition, set B includes a file with the faults injected into the application. Although our dataset is unsupervised, this fault file will be used only to evaluate the solution and not as part of training the model.

According to (Chandola et al., 2009), aspects such as Nature of Input Data, Type of Anomaly, Data Labels and Output of Anomaly Detection must be considered before addressing an anomaly detection problem. Regarding these aspects, we characterize our dataset, as represented in Table 3, based on Simple meta-features. Meta-features describe properties of the data which are predictive of the performance of machine learning algorithms trained on them (Rivolli et al., 2019). The following are the primary meta-features employed.

Table 3 - Dataset Characteristics

Meta-features	Value
Number of examples (instances)	450000
Number of attributes (features)	5
Number of missing values of features	0
Number of binary attributes	1
Number of numeric attributes	2
Number of categorical attributes	2

The dataset file is in csv format and its 5 features include "timestamp", "latency", "succ", "source" and "target". These features are described in Table 4, along with their respective data types. Although the logs exceed 1,000,000 records, for testing purposes, random five-day records were used, corresponding to a total of 450.000. It should also be noted that the faults injected are of type CPU exhaustion, memory exhaustion, host network error, container network error, and database failures.

Table 4 - Description of the dataset features

Feature	Description	Type	Possible/accepted values
trace_id	Id of each single trace	String	
timestamp	Precise indication of date and time when a request was made	Number	
latency	The time in seconds it took for the request to be completed.	Number	From 0 to 28700
Succ	Indicates whether the request was successful or not	Boolean	True or False
source	Indicates the source where the request came from	String	docker_001, docker_002, docker_003, docker_004, docker_005, docker_006, docker_007, docker_008
target	Indicates the target of each request	String	docker_002, docker_003, docker_004, docker_005, docker_006, docker_007, docker_008, db_003, db_007, db_009, os_021, os_022, docker_001

3.3. Algorithms

The choice of algorithm to detect anomalies depends on a few factors. These factors, as mentioned in the previous section are based on the characteristics of the dataset. Given the characteristics of the dataset, we developed our model with the following two algorithms:

kth Nearest Neighbor (kNN)

The k-nearest neighbors algorithm, or kNN, is one of the simplest machine learning algorithms. Usually, k is a small, odd number - sometimes only one. The larger k is, the more accurate the classification will be, but the longer it takes to perform the classification (KNN, 2019). According to (Chandola et al., 2009), in this algorithm, the anomaly score of a data instance is defined as its distance to its kth nearest neighbor in a given data set.

The basic parameters for this algorithm according to PyOD documentation are as follows:

contamination : float in (0., 0.5), optional (default=0.1). The amount of contamination of the data set, in other words, the proportion of outliers in the data set. Used when fitting to define the threshold on the decision function.

n_neighbors : int, optional (default = 5). Number of neighbors to use by default for k neighbors queries.

method : string, optional (default='largest'). Valid values are 'largest', 'mean', “median”.

- “largest”: use the distance to the kth neighbor as the outlier score.
- “mean”: use the average of all k neighbors as the outlier score.
- “median”: use the median of the distance to k neighbors as the outlier score

metric: string or callable, default=”minkowski”. Metric to use for distance computation. Default is “minkowski”, which results in the standard Euclidean distance when p = 2. The valid values for metrics are those of the two libraries:

- **scikit-learn**: “cityblock”, “cosine”, “euclidean”, “l1”, “l2”, “manhattan”.
- **scipy.spatial.distance**: “braycurtis”, “canberra”, “chebyshev”, “correlation”, “dice”, “hamming”, “jaccard”, “kulsinski”, “mahalanobis”, “matching”, “minkowski”, “rogerstanimoto”, “russellrao”, “seuclidean”, “sokalmichener”, “sokalsneath”, “sqeuclidean”, “yule”.

Because our dataset has only two classes, outliers and inliers, for our solution, we had to change the model in order to set K = 3. The Contamination parameter we assume the to be 20%, since our sample is considerably small with respect to the total data. For the other metrics the default value was kept.

Histogram-based Outlier Score (HBOS

According to Goldstein & Dengel (2012), although this algorithm is only a combination of univariate methods unable to model dependencies between features, its fast computation is charming for large data sets. This is perfectly fitting for detecting anomalies in microservices that usually produce large numbers of logs and have many parameters to consider. Additionally, for every single feature (dimension), a univariate histogram is constructed first. If the feature comprises categorical data, a simple counting of the values of each category is performed, and the relative frequency (height of the histogram) is computed. For numerical features, two different methods can be used: (1) Static bin-width histograms or (2) dynamic bin-width histograms.

As per PyOD documentation, this algorithm comprises the following parameters:

n_bins: integer or string, optional (default=10). The number of bins. "auto" uses the birge-rozenblac method for automatic selection of the optimal number of bins for each feature.

alpha: float in (0, 1), optional (default=0.1). The regularizer for preventing overflow.

tol: float in (0, 1), optional (default=0.5). The parameter to decide the flexibility while dealing the samples falling outside the bins.

contamination: float in (0., 0.5), optional (default=0.1). The amount of contamination of the data set.

With the exception of Contamination, which in the same way as the KNN algorithm was assigned a value of 20%, the default value was maintained for all other parameters.

3.4. Implementation

A few steps were taken to detect anomalies in the dataset using PyOD. First, the dataset was cleaned and the features, “latency” and “target”, that we consider to be more predictive, were prepared and converted to a format that the algorithms could understand. This means that, since the algorithms used in the solution require numerical input data, the categorical attribute "target" was encoded into a numerical attribute. This process is referred to as Label Encoding and was done using the Preprocessing module of the scikit-learn library. Similarly, but using a different strategy, the "succ" attribute was transformed to represent numeric values. Therefore, False became 0 and True became 1.

To better understand the data, these features were plotted on a scatter plot. From figure 11 it can be seen that the anomalies are concentrated further to the right of the graph corresponding to requests that have greater than or equal to 30000 milliseconds of latency and that occurred on docker_004.

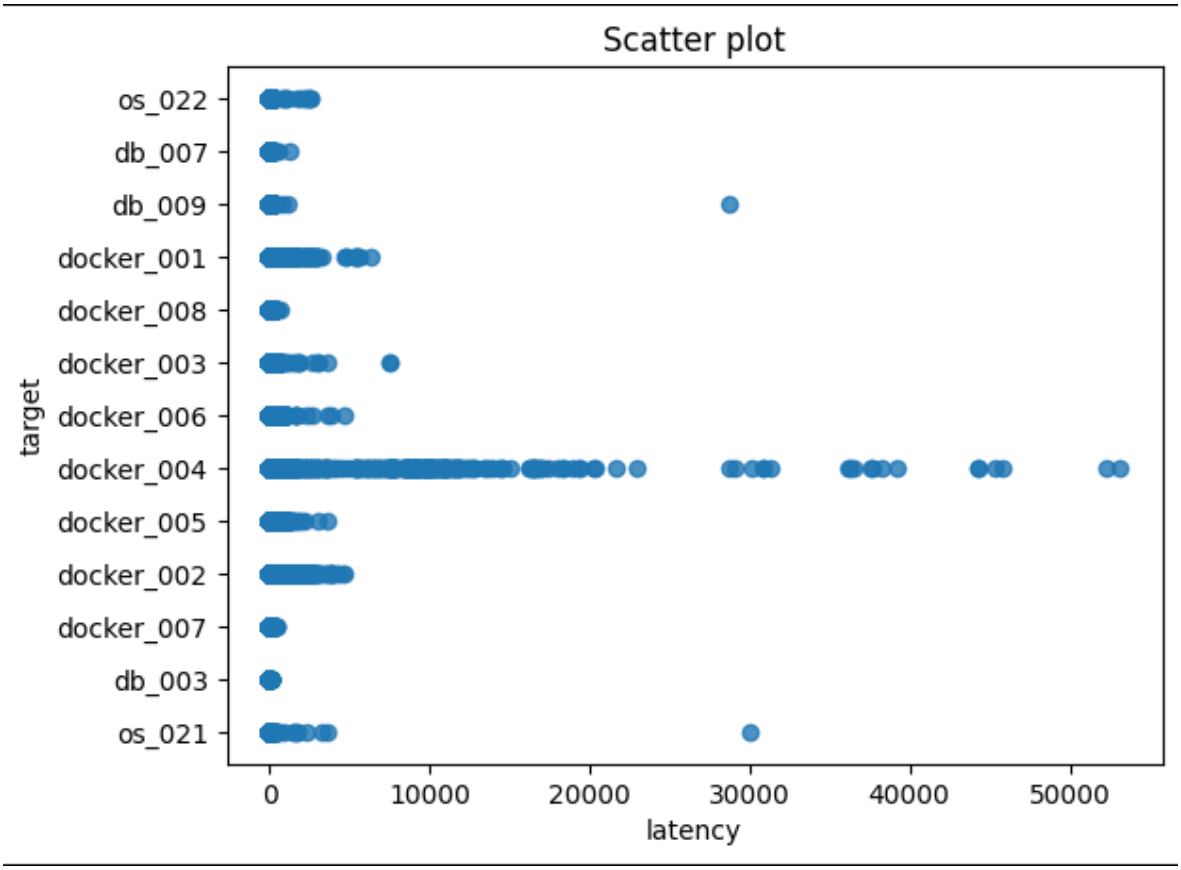

Figure 11 - Distribution of latency and target in a scatter plot.

Then we predict anomalies using the chosen algorithms mentioned above and PyOD methods, fit() and predict() to obtain the total number of outliers found in each one. The fit() method will fit the model to the input training instances while predict() method, predict if a particular sample is an outlier or not using the fitted detector. After that, a function was explicitly developed for calculating the confusion matrix. Based on the values predicted by the algorithm, this function's function is to calculate True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN). For that, considering our dataset, the validation dataset (fault list) and the confusion matrix concepts, we established that to be:

True Positive – the ground truth is positive, and the test predicts a positive. This means that, the observation is really an outlier, and the test has detected this accurately.

True Negative – the ground truth is negative, and the test predicts a negative. This means that, the observation is not an outlier, and the test accurately reports this as a non-outlier.

False Positive – the ground truth is negative; however, the test predicts a positive. This means that, the observation is not an outlier, but the test inaccurately reports this as an outlier.

False Negative – the ground truth is positive, but the test predicts a negative. This means that, the observation is really an outlier, but the test inaccurately reports this as not being an outlier.

The pseudocode for this function is represented in table 5.

Table 5 - Function to get Confusion Matrix

```
DEFINE FUNCTION getConfusionMatrix(faults, predicted):

    SET truePositive TO []
    SET trueNegative TO []
    SET falsePositive TO []
    SET falseNegative TO []
    SET predicted TO predicted.assign(date_time TO lambda x: df['timestamp'])

    FOR i, act IN faults.iterrows():

        SET minTime TO datetime.datetime.strptime(act.time_preliminary,'%Y-%m-%d %H:%M:%S+08:00')
        SET maxTime TO minTime + timedelta(minutes=5)

        truePositive.append(predicted.loc[(predicted.date_time >= minTime) & (predicted.date_time <= maxTime) & (predicted.outliers EQUALS 1) & (predicted.target EQUALS act.ground_truth)])

        trueNegative.append(predicted.loc[predicted.outliers EQUALS 0])

        falsePositive.append(predicted.loc[(predicted.date_time < minTime) | (predicted.date_time > maxTime) & (predicted.outliers EQUALS 1)])

        falseNegative.append(predicted.loc[(predicted.date_time < minTime) | (predicted.date_time > maxTime) & (predicted.outliers EQUALS 0)])

    ELSE:

        SET truePositive TO list(filter(lambda dfTP: not dfTP.empty, truePositive))
        SET trueNegative TO list(filter(lambda dfTN: not dfTN.empty, trueNegative))
        SET falsePositive TO list(filter(lambda dfFP: not dfFP.empty, falsePositive))
        SET falseNegative TO list(filter(lambda dfFN: not dfFN.empty, falseNegative))
```

RETURN

In summary, implementing the solution started with preparing the dataset and defining an anomaly detection method. Taking advantage of the PyOD toolkit, and based on the criteria for algorithm decision, we worked with kNN and HBOS algorithms. We also changed the “n-neighbors” and “Contamination” parameters in order to adapt them to our dataset. In developing the code, some of the library’s methods and utility functions were used to perform the prediction. The algorithms could detect some outliers in the overall data. However, the results alone do not guarantee the efficiency of the algorithm. For this reason, the solution also includes the application of some metrics to evaluate the model. The subsequent chapter discusses the results obtained based on these metrics.

4. Results Analysis

After all the development was done, it was necessary the usage of some metrics that could evaluate the solution. This chapter describes the metrics used to validate the model and also analyzes and discuss the results obtained.

4.1. Evaluation Metrics

To compare the performance of our algorithms, it was necessary the use of some metrics. From the literature review, it was possible to verify that recall and F1-score are among the most used metrics to evaluate the performance of algorithms. Thus, we determined the confusion matrix and calculated those metrics to find the best algorithm.

4.1.1. Confusion Matrix

The confusion matrix provides more knowledge about the performance of our model by providing information on correctly or incorrectly classified classes through which we can identify errors (Tanouz et al., 2021). Furthermore, the confusion matrix produces the TP, TN, FN and FP values that will be useful in the F1-Score and Recall calculations.

As mentioned in the Implementation section, a function was developed to return the confusion matrix. The obtained results for kNN and HBOS algorithms is represented in table 6 e 7 respectively. The results were then applied to the following metrics:

Precision: number of classified correct outputs, we can say exactness of model.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Recall: the measure of our model correctly identifying True Positives

$$Recall\ = \frac{TruePositives}{TruePositives\ +\ FalseNegatives}$$

F1 score: Average of Precision and Recall.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Table 6- kNN Confusion Matrix

		Actual	
		Positive	Negative
Predicted	Positive	TP 2457	FP 28
	Negative	FN 609	TN 446906

Table 7 - HBOS Confusion Matrix

		Actual	
		Positive	Negative
Predicted	Positive	TP 2460	FP 25
	Negative	FN 507	TN 446928

Substituting the confusion matrix values into the Accuracy, Recall, and F1-Score formulas yielded the results for the two algorithms. The results, as presented in table 8 show that HBOS performs slightly better than KNN, with Recall and F1-Score of 83% and 91%, respectively, while for kNN these metrics were 80% and 89%, respectively.

Table 8 - Performance of the algorithms

ALGORITHMS	RECALL	F1 - SCORE
kNN	80%	89%
HBOS	83%	91%

The recall tells us that among all the positive cases in the dataset, kNN correctly predicted 80% of the positive cases and HBOS 83%. On the other hand, F1-Score combines both precision and recall, which means that since recall is high, F1-socre is also high for both kNN and HBOS.

5. Conclusion

This thesis focused on the study and development of a microservice anomaly detection approach using PyOD. The PyOD library, which provides access to over 40 algorithms for supervised and unsupervised data, was used in TraceRCA dataset for anomaly detection. It has many advantages, one being the ability to work with multiple algorithms in a single anomaly detection tool. With PyOD it is possible to analyze and choose the most suitable algorithm for the dataset from the many available.

In the implemented solution, the kNN algorithm and HBOS were used. These two different algorithms, being kNN more focused on classification problems and HBOS on probabilistic ones, allowed us to compare and evaluate the results. The dataset was also analyzed and prepared in order to choose the best features. This analysis and preparation were one of the most challenging activities, because each algorithm has its own requirements and parameters. For the model to perform all the steps up to prediction, the dataset had to conform to these requirements. After the prediction process, with features "latency" and "target", metrics were calculated to see which algorithm performed best.

Overall, this work contributes significantly to the research community in an area that is still new - anomaly detection in microservices. This area enables researchers to make further comparisons with other algorithms based on the results obtained using this library. The main point of PyOD is that, regardless of the dataset's characteristics, it is possible to choose among the various algorithms available to see which one fits best. However, the fact that it does not act on applications, but rather on extracted datasets containing system logs could be a potential challenge. Another concern about PyOD models is that their performance tends to decrease when the dataset is small, making them better suited for large datasets.

For future works, we aim to expand the application of the algorithms to the entire dataset since usually, thousands of logs are produced per second in microservice systems. Furthermore, to achieve more precise results, we intend to evaluate other microservices dataset that has different characteristics and aggregate more metrics. We also intend to do further investigations with new developments aimed at identifying the root cause of the anomalies.

6. References

Ahmed, A., Pierre, G., 2018. Docker Container Deployment in Fog Computing Infrastructures, in: 2018 IEEE International Conference on Edge Computing (EDGE). Presented at the 2018 IEEE International Conference on Edge Computing (EDGE), pp. 1–8. https://doi.org/10.1109/EDGE.2018.00008

Ahmed, M., Naser Mahmood, A., Hu, J., 2016. A survey of network anomaly detection techniques. J. Netw. Comput. Appl. 60, 19–31. https://doi.org/10.1016/j.jnca.2015.11.016

Amer, M., Goldstein, M., 2012. Nearest-Neighbor and Clustering based Anomaly Detection Algorithms for RapidMiner.

Amoany, E., 2021. An introduction to monitoring using the ELK Stack [WWW Document]. Enable Sysadmin. URL https://www.redhat.com/sysadmin/what-is-elk-stack (accessed 12.29.21).

Anil, N., 2021. Comunicação em uma arquitetura de microsserviço [WWW Document]. URL https://docs.microsoft.com/pt-br/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (accessed 12.9.21).

Apolinário, D.R.F., França, B.B.N., 2021. A method for monitoring the coupling evolution of microservice-based architectures. J. Braz. Comput. Soc. 27, 17. https://doi.org/10.1186/s13173-021-00120-y

AWS, 2021. Implementing Microservices on AWS - AWS Whitepaper 34.

Barakat, S., 2017. Monitoring and Analysis of Microservices Performance. J. Comput. Sci. Control Syst. 10, 19–22.

Bertero, C., Roy, M., Sauvanaud, C., Tredan, G., 2017. Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection, in: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). Presented at the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 351–360. https://doi.org/10.1109/ISSRE.2017.43

Botez, R., Iurian, C.-M., Ivanciu, I.-A., Dobrota, V., 2020. Deploying a Dockerized Application With Kubernetes on Google Cloud Platform, in: 2020 13th International Conference on Communications (COMM). Presented at the 2020 13th International Conference on Communications (COMM), pp. 471–476. https://doi.org/10.1109/COMM48946.2020.9142014

Bruce, M., Pereira, P., 2019. Microservices in Action. Shelter Island, NY.

Burns, B., Beda, J., Hightower, K., 2019. Kubernetes: Up and Running 277.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J., 2016. Borg, Omega, and Kubernetes. Commun. ACM 59, 50–57. https://doi.org/10.1145/2890784

Cao, W., Cao, Z., Zhang, X., 2019. Research on Microservice Anomaly Detection Technology Based on Conditional Random Field.

Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: A survey. ACM Comput. Surv. 41, 15:1-15:58. https://doi.org/10.1145/1541880.1541882

Chirammal, H.D., Mukhedkar, P., Vettathu, A., 2016. Mastering KVM virtualization: dive in to the cutting edge techniques of Linux KVM virtualization, and build the virtualization solutions your datacentre demands, First published. ed, Community experience distilled. Packt Publishing, Birmingham Mumbai.

Chiueh, S.N.T., 2020. A Survey on Virtualization Technologies 42.

Cinque, M., Cotroneo, D., Pecchia, A., 2013. Event Logs for the Analysis of Software Failures: A Rule-Based Approach. IEEE Trans. Softw. Eng. 39, 806–821. https://doi.org/10.1109/TSE.2012.67

Cinque, M., Della Corte, R., Pecchia, A., 2019. Advancing Monitoring in Microservices Systems, in: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Presented at the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 122–123. https://doi.org/10.1109/ISSREW.2019.00060

Daya, S., Duy, N.V., Eati, K., Ferreira, C.M., Glozic, D., Gucer, V., Gupta, M., Joshi, S., Lampkin, V., Martins, M., Narain, S., Vennam, R., 2015. Microservices: From Theory to Practice 170.

Düllmann, T.F., 2017. Performance Anomaly Detection in Microservice Architectures Under Continuous Change 89.

Forsberg, V., 2019. AUTOMATIC ANOMALY DETECTION AND ROOT CAUSE ANALYSIS FOR MICROSERVICE CLUSTERS 46.

Fowler, M., Lewis, J., 2014. Microservices a definition of this new architectural term. URL https://martinfowler.com/articles/microservices.html

Ghofrani, J., Lübke, D., 2018. Challenges of Microservices Architecture: A Survey on the State of the Practice 8.

Golden, B., 2011. Virtualization For Dummies, 3rd HP Special Edition 75.

Graylog [WWW Document], 2021. URL https://www.graylog.org/products/open-source (accessed 12.30.21).

Hasselbring, W., 2016. Microservices for Scalability: Keynote Talk Abstract, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE

’16. Association for Computing Machinery, New York, NY, USA, pp. 133–134. https://doi.org/10.1145/2851553.2858659

IBM, 2021a. virtualization-a-complete-guide [WWW Document]. URL https://www.ibm.com/cloud/learn/virtualization-a-complete-guide (accessed 12.30.21).

IBM, 2021b. vmware [WWW Document]. URL https://www.ibm.com/cloud/learn/vmware (accessed 12.6.22).

IBM Cloud Education, 2021. microservices [WWW Document]. URL https://www.ibm.com/cloud/learn/microservices (accessed 12.8.21).

Jain, N., Choudhary, S., 2016. Overview of virtualization in cloud computing, in: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). Presented at the 2016 Symposium on Colossal Data Analysis and Networking (CDAN), pp. 1–4. https://doi.org/10.1109/CDAN.2016.7570950

Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The Journey So Far and Challenges Ahead. IEEE Softw. 35, 24–35. https://doi.org/10.1109/MS.2018.2141039

Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Ranjan, R., 2018. A Holistic Evaluation of Docker Containers for Interfering Microservices, in: 2018 IEEE International Conference on Services Computing (SCC). Presented at the 2018 IEEE International Conference on Services Computing (SCC), pp. 33–40. https://doi.org/10.1109/SCC.2018.00012

Li, Z., Chen, J., Jiao, R., Zhao, N., Wang, Zhijun, Zhang, S., Wu, Y., Jiang, L., Yan, L., Wang, Zikai, Chen, Z., Zhang, W., Nie, X., Sui, K., Pei, D., 2021a. Practical Root Cause Localization for Microservice Systems via Trace Analysis, in: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). pp. 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521340

Li, Z., Chen, J., Jiao, R., Zhao, N., Wang, Zhijun, Zhang, S., Wu, Y., Jiang, L., Yan, L., Wang, Zikai, Chen, Z., Zhang, W., Nie, X., Sui, K., Pei, D., 2021b. Practical Root Cause Localization for Microservice Systems via Trace Analysis, in: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). Presented at the 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS), pp. 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521340

Logstash: Colete, analise, transforme logs [WWW Document], 2021. . Elastic. URL https://www.elastic.co/pt/logstash (accessed 12.30.21).

Medel, V., Tolosana-Calasanz, R., Bañares, J.Á., Arronategui, U., Rana, O.F., 2018. Characterising resource management performance in Kubernetes. Comput. Electr. Eng. 68, 286–297. https://doi.org/10.1016/j.compeleceng.2018.03.041

Meng, L., Ji, F., Sun, Y., Wang, T., 2021. Detecting anomalies in microservices with execution trace comparison. Future Gener. Comput. Syst. 116, 291–301. https://doi.org/10.1016/j.future.2020.10.040

Newman, S., 2015. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Sebastopol, USA.

Ohlsson, J., 2018. Anomaly Detection in Microservice Infrastructures.

O'Reilly, C., Gluhak, A., Imran, M.A., Rajasegarar, S., 2014. Anomaly Detection in Wireless Sensor Networks in a Non-Stationary Environment. IEEE Commun. Surv. Tutor. 16, 1413–1432. https://doi.org/10.1109/SURV.2013.112813.00168

Pahl, C., 2015. Containerization and the PaaS Cloud. IEEE Cloud Comput. 2, 24–31. https://doi.org/10.1109/MCC.2015.51

Pawar, U., Bhelotkar, M., 2011. Virtualization: a way towards dynamic IT, in: Proceedings of the International Conference & Workshop on Emerging Trends in Technology - ICWET '11. Presented at the the International Conference & Workshop, ACM Press, Mumbai, Maharashtra, India, p. 262. https://doi.org/10.1145/1980022.1980082

Pearce, M., Zeadally, S., Hunt, R., 2013. Virtualization: Issues, security threats, and solutions. ACM Comput. Surv. 45, 1–39. https://doi.org/10.1145/2431211.2431216

Petrasch, R., 2017. Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication, in: 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE). Presented at the 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 1–4. https://doi.org/10.1109/JCSSE.2017.8025912

Ponce, F., Márquez, G., Astudillo, H., 2019. Migrating from monolithic architecture to microservices: A Rapid Review, in: 2019 38th International Conference of the Chilean Computer Science Society (SCCC). Presented at the 2019 38th International Conference of the Chilean Computer Science Society (SCCC), pp. 1–7. https://doi.org/10.1109/SCCC49216.2019.8966423

Potdar, A.M., D g, N., Kengond, S., Mulla, M.M., 2020. Performance Evaluation of Docker Container and Virtual Machine. Procedia Comput. Sci., Third International Conference on Computing and Network Communications (CoCoNet'19) 171, 1419–1428. https://doi.org/10.1016/j.procs.2020.04.152

Price, E., Buck, A., Lee, D., Wray, S., 2021. Microservice architecture style - Azure Application Architecture Guide [WWW Document]. Microsoft Docs. URL

https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (accessed 12.7.21).

Rad, B.B., Bhatti, H.J., Ahmadi, M., 2017. An Introduction to Docker and Analysis of its Performance 9.

Richardson, C., 2021. Microservices Pattern: Microservice Architecture pattern [WWW Document]. microservices.io. URL http://microservices.io/patterns/microservices.html (accessed 12.7.21).

Richardson, C., 2016. From Design to Deployment. NGINX, Inc.

Rivolli, A., Garcia, L.P.F., Soares, C., Vanschoren, J., de Carvalho, A.C.P.L.F., 2019. Characterizing classification datasets: a study of meta-features for meta-learning.

scikit-learn, 2022. Novelty and Outlier Detection [WWW Document]. Scikit-Learn. URL https://scikit-learn/stable/modules/outlier_detection.html (accessed 1.2.23).

Smyth, N., 2012. Xen Virtualization Essentials - Virtuatopia [WWW Document]. URL https://www.virtuatopia.com/index.php?title=Xen_Virtualization_Essentials (accessed 12.31.22).

Splunk | Turn Data Into Doing [WWW Document], 2021. . Splunk. URL https://www.splunk.com (accessed 12.30.21).

Sun, Y., Zhao, L., Wang, Z., Cui, D., Yang, Y., Gao, Z., 2021. Fault Root Rank Algorithm Based on Random Walk Mechanism in Fault Knowledge Graph, in: 2021 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB). Presented at the 2021 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), pp. 1–6. https://doi.org/10.1109/BMSB53066.2021.9547194

Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E., 2018. Microservices in Practice: A Survey Study. ArXiv180804836 Cs.

Wang, T., Zhang, W., Xu, J., Gu, Z., 2020. Workflow-Aware Automatic Fault Diagnosis for Microservice-Based Applications With Statistics. IEEE Trans. Netw. Serv. Manag. 17, 2350–2363. https://doi.org/10.1109/TNSM.2020.3022028

Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G., 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. J. Syst. Softw. 182, 111061. https://doi.org/10.1016/j.jss.2021.111061

Wolff, E., 2016. Microservices: Flexible Software Architectures 410.

Yang, M., Huang, M., 2019. An Microservices-Based Openstack Monitoring Tool, in: 2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS). Presented at the 2019 IEEE 10th International Conference on Software

Engineering and Service Science (ICSESS), pp. 706–709. https://doi.org/10.1109/ICSESS47205.2019.9040740

Yashchenko, M., 2016. Social network.

Yilmaz, S.F., 2022. selimfirat/pysad.

Zhao, Y., Chen, G.H., Jia, Z., 2022. TOD: GPU-accelerated Outlier Detection via Tensor Operations. https://doi.org/10.48550/arXiv.2110.14007

Zhao, Y., Nasrullah, Z., Li, Z., 2019. PyOD: A Python Toolbox for Scalable Outlier Detection 7.

Zipkin, 2021. OpenZipkin · A distributed tracing system [WWW Document]. URL https://zipkin.io/ (accessed 12.29.21).