BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2023 Bc. Adam Ludes

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

ANOMALY AND THREAT DETECTION IN AUDIT LOGS USING MACHINE LEARNING

DETEKCE ANOMÁLIÍ A ÚTOKŮ V AUDIT LOGU POMOCÍ UMĚLÉ INTELIGENCE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. Adam Ludes

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. Adrián Tomašov

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Adam Ludes ***ID:*** 211802

Year of study: 2 ***Academic year:*** 2022/23

TITLE OF THESIS:

Anomaly and threat detection in audit logs using machine learning

INSTRUCTION:

The thesis focuses on anomaly and threat detection in audit logs from container orchestration platforms. The goal is to detect undesirable requests leading to a denial of service or private data leakage. The semestral part of the thesis investigates the given problem and gathers necessary data. The data are inspected and then examined with basic statistical tests. The results highlight the best preprocessing methods and machine learning algorithms used in the analysis. The diploma thesis implements highlighted preprocessing methods with machine learning models and compares them against state-of-the-art solutions. The last part is to deploy implemented models into a production infrastructure.

RECOMMENDED LITERATURE:

[1] CHALAPATHY, Raghavendra; CHAWLA, Sanjay. Deep learning for anomaly detection: A survey. arXiv preprint arXiv:1901.03407, 2019.

[2] AHMAD, Subutai, et al. Unsupervised real-time anomaly detection for streaming data. Neurocomputing, 2017, 262: 134-147.

Date of project specification: 6.2.2023 ***Deadline for submission:*** 19.5.2023

Supervisor: Ing. Adrián Tomašov

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

The thesis explores cloud-native architecture, anomaly detection techniques, machine learning, and data analysis to develop an anomaly detection model for audit logs from the Red Hat OpenShift Container Platform. Statistical methods and time series analysis for anomaly detection are introduced, while machine learning models and preprocessing techniques are implemented and evaluated. The results demonstrate limitations in traditional models for handling anomalies in deeply nested data, while the NLP model shows robust performance. This research provides valuable insights and is a reference for researchers and practitioners in cloud-native architecture, anomaly detection, machine learning, and data analysis.

KEYWORDS

Anomaly Detection, Cloud-native, Data Analysis, Kubernetes, Machine Learning, OpenShift

ABSTRAKT

Tato práce představuje softwarové architektury založené na cloudu, techniky detekce anomálií, strojové učení a analýzu dat za účelem vytvoření modelu pro detekci anomálií v audit lozích z Red Hat OpenShift Container Platform. Jsou představeny statistické metody a analýza časových řad pro detekci anomálií, zatímco jsou implementovány a hodnoceny modely strojového učení a techniky předzpracování dat. Výsledky ukazují omezení tradičních modelů při zpracování anomálií v hluboce vnořených datech, zatímco model zpracovávající přirozený jazyk prokazuje robustní výkon. Tato práce poskytuje cenné poznatky a může být použita jako reference pro výzkum i praxi v oblasti softwarových architektur založených na cloudu, detekce anomálií, strojového učení a analýzy dat.

KLÍČOVÁ SLOVA

Analýza dat, Cloud-native, Detekce anomálií, Kubernetes, OpenShift, Strojové učení

Rozšířený abstrakt

Tato diplomová práce se podrobně zabývá cloudovou architekturou, technikami detekce anomálií, strojovým učením a analýzou dat. Jejím cílem je vytvořit model detekce anomálií pomocí strojového učení na audit lozích z kontejnerové platformy Red Hat OpenShift.

První kapitola nabízí podrobný výklad cloud-nativní architektury. Je v ní vysvětlen historický vývoj, který vedl k jejímu vzniku od virtuálních strojů až ke kontejnerům. Následně představuje klíčový vývojový přístup DevOps společně s mikroslužbami, na kterých je cloud-nativní architektura postavena. Dále je představen princip orchestrace kontejnerů, který je v cloud-nativní archtektuře klíčový. Na závěr jsou představeny dvě platformy sloužící k orchestraci kontejnerů – Kubernetes a Red Hat OpenShift Container Platform, jejíž audit logy jsou zkoumány v této práci.

Ve druhé kapitole byla vysvětlena problematika detekce anomálií a hrozeb, k čemu slouží a jaké základní metody lze použít, jako je metoda mezikvartilového rozpětí, Grubbsův test a modely Gaussových směsí, a také analýza časových řad pro identifikaci anomálií v sekvenčních datech.

Třetí kapitola se komplexně věnuje strojovému učení. Dělí strojové učení na základní druhy – s učitelem, bez učitele, jejich kombinaci a zpětnovazební učení. Dále zahrnuje metriky hodnocení výsledků dosažených modely strojového učení jako je přesnost, výtěžnost, F1 skóre a křivka provozní charakteristiky přijímače a nákladové funkce. Zmiňuje nejpoužívanější metody strojového učení používané k detekci anomálií jako algoritmus k-nejbližších sousedů, local outlier factor, DBSCAN, metoda podpůrných vektorů a Isolation Forest. Zabývá se také neuronovými sítěmi, jejich architekturou, vrstvami a zaměřuje se na autoenkodéry, které lze používat k detekci anomálií a Generative Pre-trained Transformer (GPT) sloužící ke zpracování přirozeného jazyka. Představuje také princip tokenizace textu a architekturu tzv. “transformerů”, které jsou základní stavební bloky GPT modelu.

Následuje kapitola věnovaná analýze dat, která byla poskytnuta kolegy v Red Hatu ze dvou interních OpenShift clusterů a tudíž jsou považována za citlivá a nemohou být publikována. Tato kapitola používá nástroje jako *pandas*, *ydata-profiling*, *matplotlib*, *seaborn* a *Scikit-learn* k čištění, komplexní analýze a vizualizaci těchto dat. Dále představuje princip transformace hluboce vnořených dat z původního formátu JSON do 2D “tabulkových” dat vhodných pro tradiční metody detekce anomálií včetně autoenkodérů. Následně je představen princip rekurzivní transformace těchto dat do textové formy podobné větám přirozených jazyků, která je vhodná pro modely GPT.

Pátá kapitola se věnuje implementaci již zmíněných metod předzpracování dat a

modelů za pomoci knihoven *PyTorch*, *Transformers*.

V poslední kapitole byly představeny výsledky těchto modelů, které byly měřeny na datech nepoužitých při procesu trénování. Do těchto dat bylo zamícháno 10 000 upravených logů, které sloužily k simulaci anomálie. Výsledný dataset obsahoval 203 297 prvků. Pro všechny modely byla představena matice záměn.

Výsledky odhalují omezení při předzpracování vnořených dat do 2D formátu, konkrétně při převodu textu na celočíselné značky. To vede k tomu, že tradiční modely nejsou schopny rozlišit mezi normálními daty a anomáliemi. Z tohoto důvodu nebyly pro tyto modely zhodnoceny zbývající metriky, neboť bylo předem jisté, že jejich výkon není dobrý, a tyto modely nejsou vhodné pro použití v reálných systémech.

Model GPT naopak vykazuje slibný výkon při detekci anomálií. Vykazuje robustnost díky efektivnímu využití konvencí v textu, což mu umožňuje zvládnout malé změny v často se měnících oblastech a zároveň zůstat citlivý na změny v relativně stabilních oblastech.

Vysoká míra výtěžnosti 99,33 % naznačuje, že model GPT dokáže identifikovat významnou většinu anomálií v souboru dat. Míra přesnosti 98,37 % navíc naznačuje, že model produkuje relativně nízký počet falešně pozitivních výsledků.

Kromě toho skóre AUC 0,84 pro křivku ROC naznačuje, že model GPT dokáže účinně rozlišovat mezi anomáliemi a normálními případy. V tomto ohledu však existuje prostor pro zlepšení.

Vzhledem k těmto faktorům lze model GPT považovat za slibný přístup k detekci anomálií, zejména při práci s textem nebo hluboce vnořenými a proměnlivými daty. Je slibný pro reálné aplikace při detekci anomálií v rámci prezentovaných datových souborů.

Tento dokument poskytuje cenné poznatky tím, že rozsáhle zkoumá cloudovou architekturu, techniky detekce anomálií, strojové učení a analýzu dat. Může sloužit jako komplexní reference pro výzkumné pracovníky, odborníky z praxe a nadšence a podporuje hlubší porozumění těmto oblastem.

Poznatky z této práce jsou nyní blíže zkoumány a v blízké době budou aplikovány v produkční infrastruktuře k detekci neoprávněných či chybných zásahů v OpenShiftových clusterech.

LUDES, Adam. *Anomaly and threat detection in audit logs using machine learning*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2023, 91 p. Master's Thesis. Advised by Ing. Adrián Tomašov

Author's Declaration

Author:	Bc. Adam Ludes
Author's ID:	211802
Paper type:	Master's Thesis
Academic year:	2022/23
Topic:	Anomaly and threat detection in audit logs using machine learning

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I want to express my heartfelt appreciation to all those who have aided and supported me in completing my thesis. First and foremost, I owe a debt of gratitude to my supervisor, Ing. Adrian Tomašov, who provided me with guidance, expertise, and unwavering support throughout my research journey. Their invaluable insights and constructive feedback were instrumental in shaping the direction of my work.

I am also grateful to my colleagues Mark Freer and Hilliary Lipsig, who recommended this topic, provided knowledge and expertise, and kindly helped me access the datasets I used. Without them, I could not have embarked on this journey.

My family and friends have been my constant pillars of support, providing me with encouragement, understanding, and patience. Their unwavering belief in my abilities has been a constant source of motivation and inspiration, and I am truly indebted to them.

Contents

List of Figures

List of Listings

Introduction

Computer systems and software have become essential in almost every industry as technology advances rapidly. The software industry has made remarkable progress in meeting the increasing demand for efficiency, scalability, resiliency, and security. New technologies like containers and container orchestration platforms have emerged to address these challenges, optimising hardware utilisation and enhancing the performance of distributed systems.

These advancements have led to innovative software architectures, such as microservices, which streamline development cycles by adopting DevOps practices. Additionally, infrastructure management concepts like Immutability, Declarative Configuration, and online self-healing have simplified the deployment and maintenance of software systems.

This thesis analyses audit logs from the OpenShift Container Platform, based on another popular container orchestration platform, Kubernetes, to comprehensively examine the logs using statistical tests. The primary objective is identifying the most relevant log features and determining appropriate preprocessing techniques. These findings are later utilised to develop multiple machine learning models that are assessed to select the best-performing one, which will be used on production systems to help engineers filter anomalies quickly and effectively.

By exploring the complexities of audit logs in containerised environments, this research aims to enhance our understanding of the OpenShift Container Platform and its underlying Kubernetes infrastructure. The findings and insights obtained from this investigation will contribute to developing effective machine-learning models and anomaly detection techniques in container-based systems.

1 Cloud-native architecture

The term "cloud-native" is commonly used in marketing, but its meaning is often unclear. It generally refers to an architectural approach for developing and operating applications that fully utilise cloud computing principles like scalability, resilience, and flexibility. Cloud-native applications are designed to be deployed on distributed cloud platforms and infrastructures instead of traditional on-premise applications [1].

Cloud-native applications have a global reach, spreading their data across multiple data centres. This improves the reliability of the application, reduces delays, and ensures data integrity. As a result, these applications can easily handle a growing user base. Additionally, microservices enable scaling of only specific parts of the application, reducing resource requirements. This also facilitates seamless updates of individual parts of the application without causing significant disruptions, eliminating the need for restarting the entire product [1].

The robustness of the architecture is also a key feature of cloud-native applications. These applications do not break down due to any infrastructure hiccup and will continue to work immediately after the issue is resolved, making them self-healing [1].

Cloud-native applications are possible thanks to automated development stages, leveraging DevOps practices, Continuous Integration, Continuous Delivery, microservices, and Container Orchestration Platforms [2].

1.1 Virtualization tools

Previously, server admins required developers to use identical tool versions on the same system for all applications. This reduced the workload of maintaining multiple versions of the same tool on one system. However, this meant that a new server was needed every time a different tool version was required.

Furthermore, assigning specific resources to individual applications was not feasible, as all applications shared the same resource pool. As a result, if one application used most of the resources, it would negatively impact the performance of the other applications [3].

More servers can be deployed to solve these problems, requiring unnecessary hardware and underutilising physical machines. Additionally, the cost of spinning up another server can be too high, hindering development. This is why virtualisation was created as an alternative solution [3].

Virtualisation overcomes this issue since it allows separate environments to run side by side on the same physical machine. A software abstraction layer separates

these environments and should not interfere with each other, provided the virtualisation software has no bugs. With this approach, the data centre footprint is significantly reduced, which reduces costs and brings other benefits, such as faster server provisioning [4].

1.1.1 Virtual machines

Virtual Machines (VMs) themselves date to the 1950s but did not start to get much use until the 1990s when notable mentions like SoftPC, Virtual PC, and VMWare Workstation (system virtual machines) were released. Later, in the mid-2000s, hardware assists were implemented in processors, which made virtual machines more powerful, and with that came projects like Linux *Kernel-based Virtual Machine* (KVM) and VirtualBox [4].

System Virtual Machines enable full virtualization. Even the hardware is simulated, and any guest *Operating System* (OS) can be installed on the VM. Any number of virtual machines can be present on a single physical machine. Since they contain their own OS, they can each have different applications installed with different versions of dependencies [5].

However, with full virtualisation come some drawbacks. Since every VM has its own OS, there is some resource overhead that is necessary for the VM to function, reducing the resources the applications can utilise.

Process Virtual Machines solve a different problem than the one described earlier and, as such, are not the focus of this thesis. They run on top of the hardware and operating system. Their primary purpose is to enable portability, allowing programs to run on any system on which the process virtual machine can be installed. For example, Java programs are compiled into Java byte code, which is interpreted or compiled *just-in-time* (JIT) by the Java Runtime Environment [5].

1.1.2 Containers

Containers are examples of operating system-level virtualisation: these solutions package applications with all their dependencies, but they do not contain any OS. Instead, they use features of the host OS to separate the application from the host without the need to simulate the hardware and any guest OS. These features in the case of the Linux kernel consist mainly of namespaces and cgroups [6].

Namespaces abstract global system resources and isolate resources between running applications. Linux Control Groups (cgroups) enable limiting the use of system hardware, such as core count, memory limits, disk **io!**, etc. [7]. This helps solve the noisy neighbour problem, which occurs when an application takes the majority of

available resources and disrupts the operation of other applications on a given system.

These features ultimately isolate the processes inside the container and limit the number of system resources they have access to, allowing them to run in any environment, be it a local development instance or a production instance running in the cloud.

This dramatically simplifies the development process since applications retain full functionality in all environments throughout software production and deployment.

Containers vs. Virtual Machines

In a way, containers are similar to virtual machines since they house everything necessary for applications to run, but there are some differences.

Virtual machines leverage a hypervisor to virtualise physical hardware, and a guest operating system needs to be installed on each machine, inflating the images to several gigabytes. The guest OS also uses resources even when idle, increasing resource overhead and reducing total performance.

Containers, conversely, contain only the application and its dependencies. This is due to them utilising the host operating system's features to separate the applications from the host without the need to virtualise. Since containers do not use a guest OS, an idle container does not use any resources, and container sizes are measured in megabytes [3].

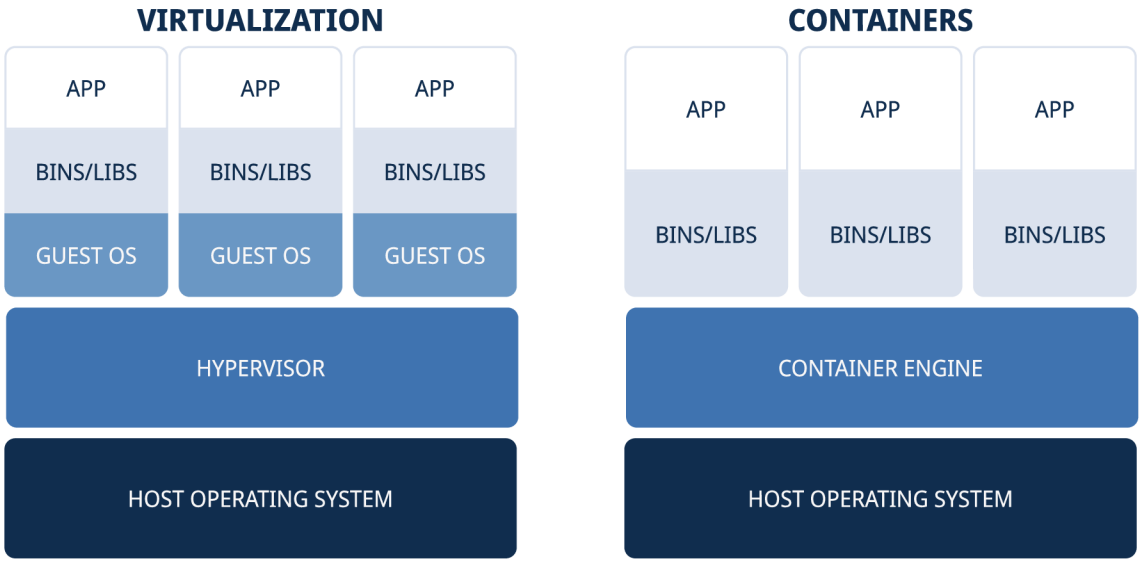

Fig. 1.1: Virtualized deployment vs. Containerized deployment. Adapted from [8].

The absence of hardware and guest OS virtualisation makes containers considerably faster, portable, and more lightweight, allowing more containers to be present on a system than virtual machines [3].

However, containers are not a replacement for virtual machines. Instead, they are best used together. For example, running a container on a different operating system than for which it was created is not trivial.

Virtual machines improve infrastructure by allowing system administrators to expand the number of systems from the available servers. In contrast, containers improve development by enabling DevOps practices and, together with the microservices architectural style, increase portability, development speed, and product resiliency while utilising resources better, increasing computation density.

1.2 DevOps

> *DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production while ensuring high quality.* [9]

DevOps is a software development approach prioritising collaboration and communication between development and operations teams. This methodology emphasises automation, continuous integration, continuous delivery, and rapid iteration to improve software development and deployment speed, quality, and efficiency.

The DevOps philosophy promotes the idea that development and operations teams should work closely throughout the entire software development lifecycle. This includes planning, development, testing, deployment, and maintenance. DevOps aims to eliminate traditional bottlenecks and delays by breaking down silos between teams and improving collaboration and communication.

1.2.1 Continuous Integration, Continuous Delivery/Deployment

*Continuous Integration and Continuous Delivery/Deployment (*CI/CD) is a collection of tools and practices that assist software development teams in building, testing, and deploying software with greater speed and dependability.

Continuous Integration (CI) involves regularly integrating code changes into a shared repository and automatically building and testing the code to ensure it works correctly. CI aims to catch and fix issues early in the development cycle, reducing the risk of errors and bugs that can cause delays and downtime [11].

Continuous Delivery (CD) automates the deployment process of software to production environments. Code changes are automatically built, tested, and validated

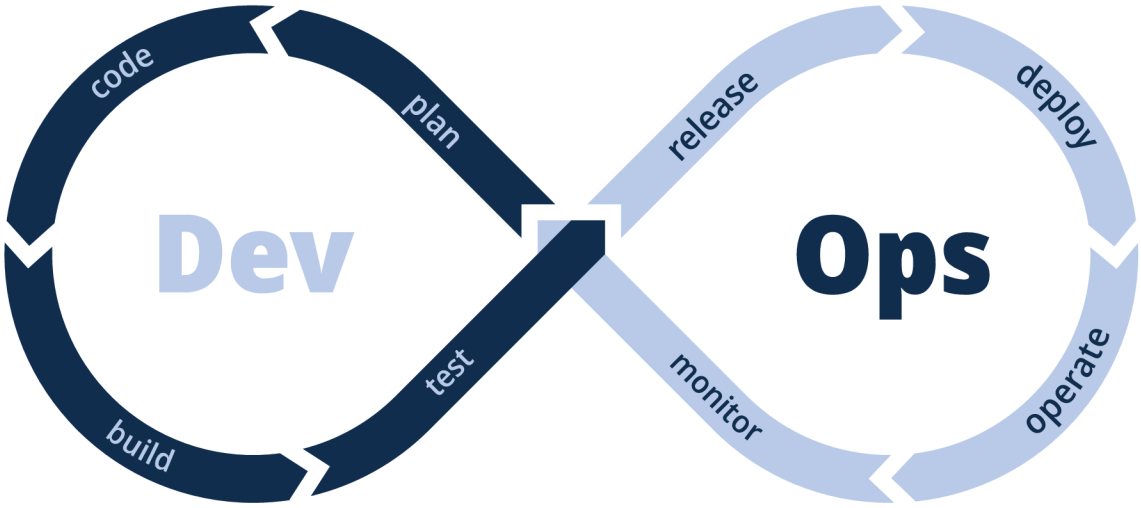

Fig. 1.2: DevOps Toolchain. Adapted from [10].

through a continuous delivery pipeline and then deployed to a staging or production environment. This enables teams to release new features and updates more rapidly and confidently while minimising the chances of errors and downtime [11].

Continuous Deployment (CD) is an automated process where code changes are directly deployed to production environments without manual intervention. This method requires a strong focus on testing and validation to ensure safety and stability [11].

Implementing CI/CD involves several tools and technologies, such as source code management systems, build automation tools, testing frameworks, containerisation platforms like Docker and Kubernetes, and deployment automation tools like Ansible and Chef. Adopting CI/CD practices can help organisations enhance the speed, quality, and reliability of their software development and deployment while reducing errors and downtime and accelerating innovation and time to market

1.2.2 Microservices

Microservices is a software development method that involves dividing large, complex applications into smaller, independent services that can be tested, developed, and deployed more efficiently and easily. This approach helps streamline the development process.

In a microservices architecture, every service has a unique purpose or task and interacts with other services through clear *Application Programming Interfaces* (APIs). Typically, each service is created and launched independently, providing more flexibility and agility in software development.

Monoliths

Traditionally, software applications were built as a collection of many built-in, interconnected modules that could not function independently. These are called monolithic applications or monoliths.

Most programming languages are designed to build particular executable applications (monoliths) that rely on resource sharing within the same machine. They result in a technology lock-in: developers must continue using the same language and framework throughout the entire application [12].

These applications prove challenging to use on distributed systems since they are difficult to maintain and suffer from *"dependency hell"*, where different library versions can cause misbehaviour of the application or a crash. Any change in a module requires restarting the whole application. The application's scalability is very limited since in cases where only a few modules are strained by the traffic, the only way to scale is to deploy more instances of the whole application and load balancing those [12].

Microservices

Microservices implement as little functionality as necessary, making their code base small and reducing the risk of bugs. However, since all the building blocks are separated into different microservices, communication needs to be established between them. There are several solutions to this problem. Microservices can communicate using traditional request/response messages, notifications without a response, or a publish/subscribe model [13].

They can be deployed side by side with older versions, and other services that depend on them can be gradually modified to move between versions. Due to that, the end product will never require a complete restart since all the microservices are restarted as needed, resulting in very short product downtime and easier maintenance. They are easily containerised and scaled as needed, and apart from the technology used to make microservices communicate (protocols, data, etc.), they bring no additional lock-in [12].

Some key characteristics of microservices include:

Decentralized Microservices are designed to be independent of each other, with each service responsible for its own data and functionality.

Scalable Services can be scaled up or down as needed without affecting other parts of the application.

Resilient Services are designed to be fault-tolerant, with redundant components and automatic failover to ensure continuous availability.

Composable Services can be combined and reused in different ways to create new applications and functionality.

Polyglot Services can be developed in different programming languages and technologies based on the specific requirements of each service.

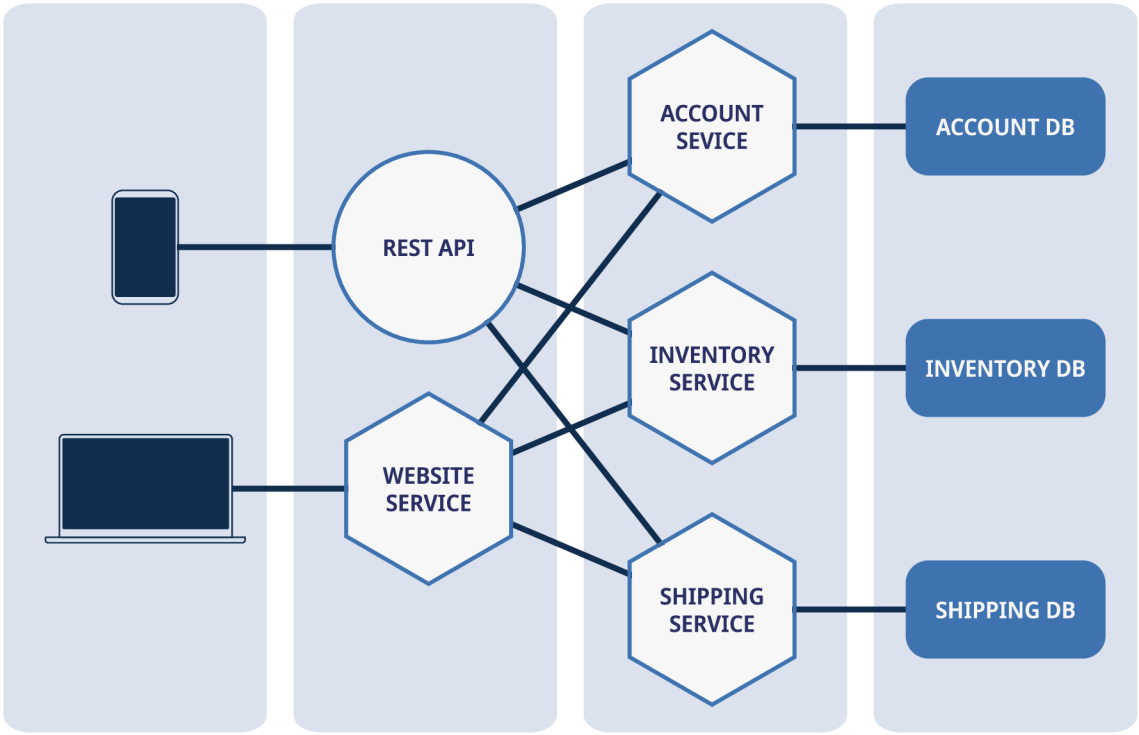

Fig. 1.3: Microservice Architecture diagram. Adapted from [14].

1.3 Container orchestration

Container orchestration automates container deployment, management, scaling, and networking. The orchestrator usually provides features to limit container resources, load balancing, autoscaling, scheduling, health monitoring, and fault tolerance [15].

Container orchestration platforms are used to deploy containerised applications in cloud environments. These platforms are either self-deployed to bare metal servers, using *Infrastructure as a service* (IaaS) to build the platform on virtual machines, or using ready-made distributed clusters – *Platform as a service* (PaaS).

- *Resource limit controls* ensure the given container does not exceed the maximum allowed memory and CPU usage [15]. These constraints are used to make scheduling decisions. They can help prevent a container from consuming too much memory due to memory leaks and ensure that all the containers have the necessary resources, combating the noisy neighbour problem.

- *Scheduling* controls the number of containers placed on the desired cluster nodes. Project maintainers can specify the desired number of containers and their node affinity, and the scheduler will try to fulfil the desire (when possible).
- The *load balancer* distributes the load between container instances, usually using round-robin scheduling, but other load balancers can also be used.
- *Health checks* ensure containers are not in a faulty state using simple readiness and liveliness requests.
- Using the results of health checks and resource limits, *fault tolerance* can be achieved by destroying faulty containers and containers using too many resources. After a container is destroyed, a new one is recreated to take its place to continue regular operation.
- *Autoscaling* automatically adds more container instances if the load is too great. A resource threshold usually defines this; a new instance is created if current instances exceed the threshold.

The leading container orchestration platforms include Kubernetes, Red Hat OpenShift Container Platform, Docker Swarm, Google Kubernetes Engine, Amazon Elastic Container Service, Azure Kubernetes Service, Marathon, and Centurion [16].

1.4 Kubernetes

Kubernetes is an open-source container orchestration platform initially developed at Google and introduced to the public in 2014. It has become the standard for deploying and managing cloud-native applications at nearly every public cloud [17]. Kubernetes is nowadays a proven infrastructure for distributed systems at every scale, be it a couple of single board computers like the Raspberry Pi or a global scale cluster spanning different continents [18].

Kubernetes provides a fast-paced, distributed, and scalable platform. When paired with already mentioned DevOps practices, such as CI/CD and microservices, the applications deployed in Kubernetes become reliable, scalable, and maintain availability even during software rollouts and maintenance [17].

1.4.1 Core Kubernetes Concepts

The core concepts that allow Kubernetes to be as reliable and fast-paced are [17]:

- Immutability
- Declarative configuration
- Online self-healing systems.

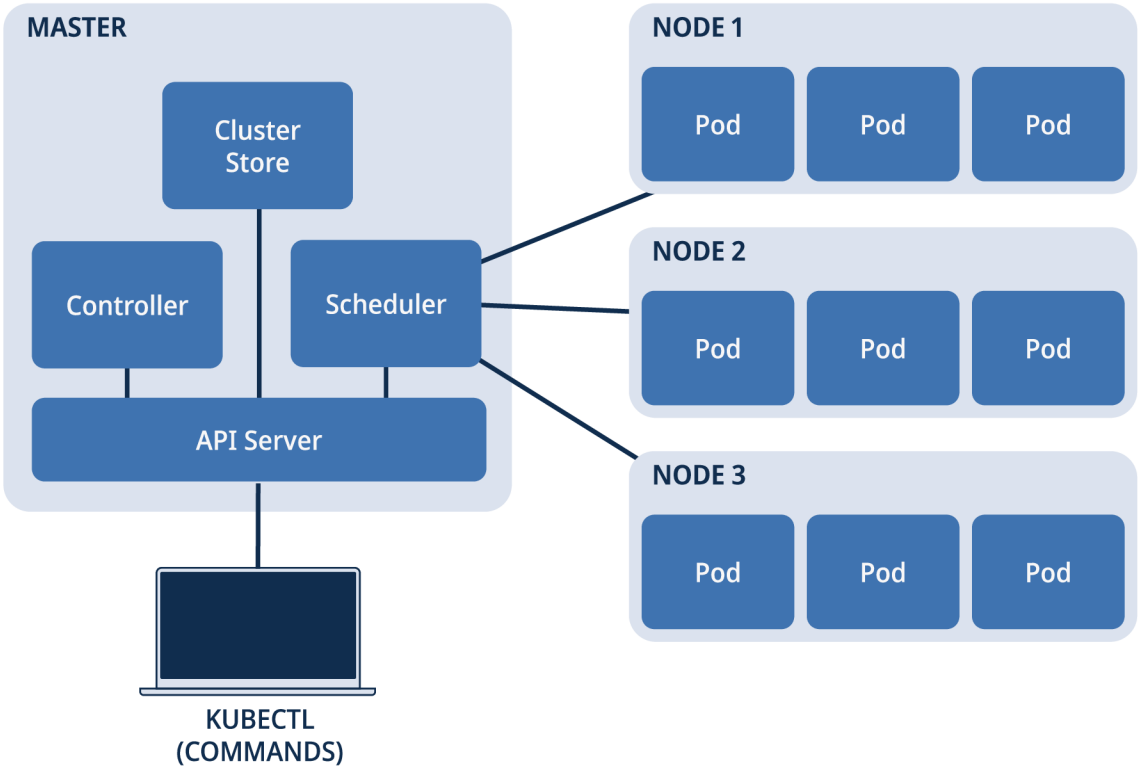

Fig. 1.4: Kubernetes Architecture diagram. Adapted from [19].

Immutability

Immutable infrastructure describes the property of an object in a system that does not change via any user input. Compared to traditional mutable infrastructure (servers, computers, software systems, etc.), which applies changes through incremental updates, immutable systems completely replace the running image with a new one in a single operation [17]. In the case of containers, the differences between mutable and immutable principles of operation can be described with the following example of a software update:

- The traditional mutable update would consist of logging into a container, running a command to update a new version of the software inside the container, and restarting it [17].
- On the other hand, an update that adheres to the immutable principles would require building a new container image with the new software version already in it, pushing it to a container registry, pulling the new image from the registry on the cluster, destroying, and redeploying the container [17].

At first glance, the second approach requires more work. However, the advantage of such an approach is that all images contain the same software and dependencies. The differences between images are easily spotted and can be traced and fixed sooner

in case of an error. These two approaches can also be considered an example of the Pet vs Cattle analogy [20].

Immutability would pose a problem when data must be saved and kept persistent between container images. This issue can be solved by using volumes. Volumes are file systems mounted inside the container independent of the container lifecycle and can be reused between image versions to store data and configuration [21].

Declarative configuration

Compared to an imperative configuration, where a sequence of commands achieves the end state of the system, declarative configuration describes the desired state, and Kubernetes ensures that this desire is reflected in the actual state of the system, if possible. For example, instead of a sequence of three commands to start three containers, the declarative configuration would specify three replicas of the container. Kubernetes would perform the required steps to achieve that state [17].

This configuration allows one to ensure that the desired state is met continually. In contrast, the imperative configuration could break where the desired state would not be understood without executing the command sequence. This feature can be further used in conjunction with traditional software development tools such as source control, code review, and testing to create what can be referred to as *Infrastructure as code* (IaC). [22]

Online self-healing systems

Since Kubernetes uses declarative configuration, it continuously ensures that the system's current state matches the desired state, even when disturbed. Instead of setting up alerts and having a human react to them, Kubernetes will repair the state to the configured one. So, for example, if the configuration specifies the desired state as having three replicas of a service and an administrator mistakenly deletes or creates one more, or one replica crashes, Kubernetes will destroy the additional replica or create a new one as needed to match the configured state [17].

1.4.2 Kubernetes components and objects

A Kubernetes cluster consists of a set of worker machines called nodes. Nodes host the application workload managed by the cluster's control plane. The control plane is a container orchestration layer that exposes API and Interfaces to define, deploy, and manage the lifecycle of containers inside the cluster. It usually runs across multiple nodes to provide fault tolerance and high availability [19].

Nodes

In Kubernetes, a node is a worker machine in a cluster that runs containerised applications. It comprises a kubelet, container runtime, kube-proxy, and an operating system. Nodes execute assigned tasks and workloads while communicating with the control plane. A Kubernetes cluster can function with a single node; however, a typical deployment would consist of multiple nodes.

Namespaces

Namespaces allow groups of resources to be isolated within the same cluster. They are intended to be used in environments where many users are spread across multiple teams or projects. Namespaces provide a scope for naming objects; objects must be uniquely named only within a namespace but not cluster-wide [23].

Pods

A pod is a collection of application containers and volumes running in the same execution environment. They are the smallest deployable unit in Kubernetes, meaning that all containers within a pod will always run on the same machine. These applications share the same IP address and port space, have the same hostname and are expected to be closely related but not as much to be part of the same container [17].

For example, creating a pod consisting of a volume where data is stored, a container serving web requests accessing the data in the volume, and a container periodically fetching new data to put save in the volume would make sense. These two containers are closely related, but the server has a higher priority, while the updater can run with a "best-effort" quality of service [17].

Services

Service objects provide an abstract way of exposing an application to make it accessible outside the cluster. They define a set of pods and a policy on accessing them [24].

Replica Sets

Replica sets are configuration objects that describe how many copies of pods are supposed to be created and managed. They can be configured to set the minimum and maximum number of replicas to be managed and a metric to decide when a new replica should be created, such as the CPU utilisation threshold [17].

Ingress

Ingress is an API object that acts as a gateway or entry point to expose *Hypertext Transfer Protocol* (HTTP) and *Hypertext Transfer Protocol Secure* (HTTPS) routes to services within the cluster. It provides external access to services by defining rules for routing and load balancing incoming traffic, terminates *Transport Layer Security* (TLS), etc. [25].

Deployments

Deployments are Kubernetes objects that manage the release of new versions. After a configuration change, deployments can gracefully move applications to the next version according to the configuration and use health checks to ensure that the new version of the application is running correctly. If the rollout fails, deployments allow rolling back versions [26].

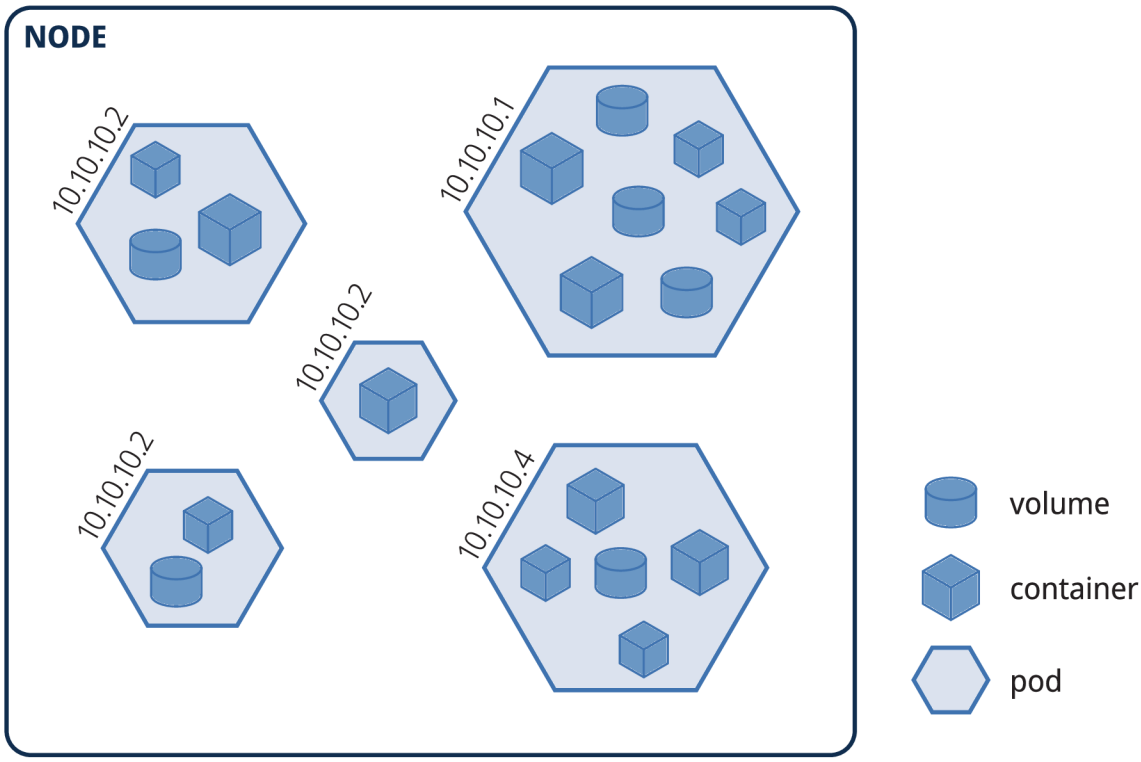

Fig. 1.5: Kubernetes node overview. Adapted from [27].

1.5 Red Hat OpenShift Container Platform

> *For all that Kubernetes can do to orchestrate containers, users still need to integrate other components like networking, ingress, load balancing, storage, monitoring, logging, multi-cluster management, continuous integration and continuous delivery (CI/CD), and more to accelerate the development and delivery of containerised applications at scale. Red Hat OpenShift offers these components with Kubernetes at its core.* [28]

Red Hat OpenShift Container Platform (RHOCP) is an open-source container orchestration platform created by Red Hat. It is an enterprise Kubernetes-based application platform with many features added on top. Since it is an enterprise platform, it comes with additional security policies. RHOCP also separates the control plane from the workload by default, resulting in a few sets of nodes – master nodes dedicated to control plane components, infrastructure nodes for maintenance and routing purposes, and worker nodes running the entirety of user workload.

Some notable differences from Kubernetes include:

Ease of use OpenShift offers a user-friendly interface and simplified workflows that facilitate the management and deployment of containerised applications. In contrast, Kubernetes demands more manual configuration and administration.

Security and compliance Enterprises can easily manage and secure their containerised applications with OpenShift's integrated security features and compliance controls. While Kubernetes offers some security features, they require manual configuration and management.

Multi-cloud support OpenShift can be deployed in different environments, including on-premises, public, or hybrid cloud configurations. It provides a consistent set of APIs and tools across all environments. In contrast, Kubernetes can be deployed in any environment, but the tools necessary to enable the various deployments must be installed separately [29].

Application services OpenShift provides a wide range of application services, such as databases, messaging, and logging, which can be easily integrated into containerised applications.

Red Hat technology OpenShift integrates Red Hat technology, such as components from *Red Hat Enterprise Linux* (RHEL). Unlike Kubernetes, which can be installed on any Linux distribution, RHOCP uses mostly *Red Hat Enterprise Linux CoreOS* (RHCOS), an immutable container-oriented operating system [29].

Figure 1.6 represents a typical Red Hat OpenShift Container Platform architecture. This architecture ensures the availability, scalability, and efficient management of containerised workloads in the Red Hat OpenShift Container Platform.

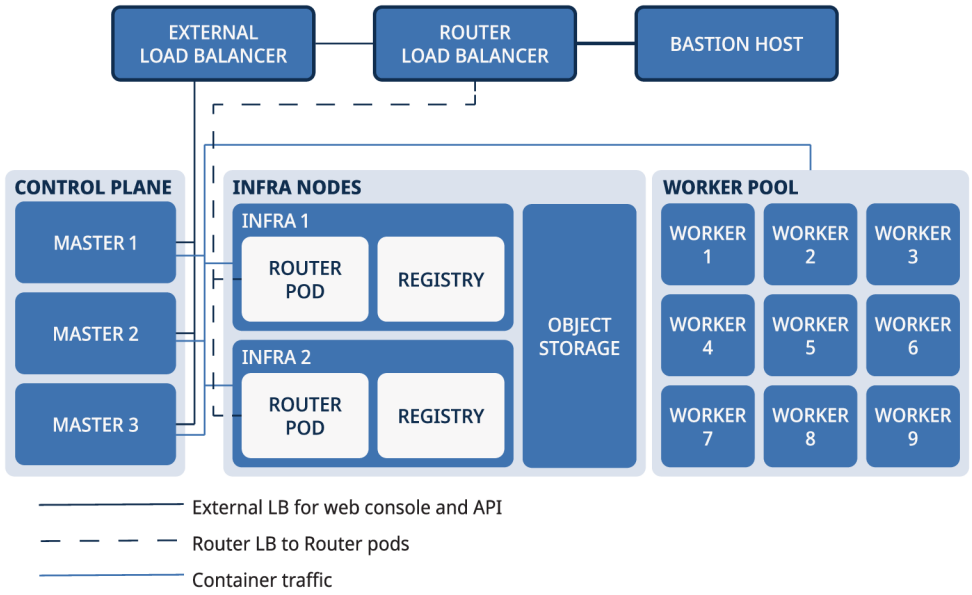

Fig. 1.6: RHOCP Architecture diagram. Adapted from [30].

Master Nodes OpenShift deploys multiple master nodes for high availability and fault tolerance. These nodes host the control plane components, including the API server, scheduler, and controller manager. They work together to manage and orchestrate the cluster [29].

Worker Node Worker nodes are where the actual application workloads run. They host containers and execute tasks assigned by the control plane [29].

Infra Nodes Infra nodes, or infrastructure nodes, are specialised worker nodes dedicated to running router pods. These pods handle the external network traffic and provide routing and load-balancing capabilities to route requests to the appropriate services within the cluster [29].

Registry and Object Storage OpenShift utilises a registry to store and manage container images used by applications. It enables efficient image distribution and deployment. Object storage refers to a storage system that stores various files and objects required by applications [29].

External Load Balance An external load balancer sits outside the OpenShift cluster and distributes incoming traffic across worker or infra nodes.

Router Load Balancer The router load balancer is an internal component within the OpenShift cluster. It distributes incoming external traffic to the appropriate router pods on infra nodes [29].

Bastion Host A bastion host, sometimes called a jump host, is a secure gateway that provides access to the OpenShift cluster for administration and management purposes. It is a controlled entry point for authorised users to interact with the cluster.

2 Anomaly Detection

This chapter provides an overview of anomaly detection, covering the principles, methods, and applications of detecting anomalies in different data types. It briefly introduces statistical and time series analysis approaches commonly used in the field and mentions the fundamental techniques for threat detection. The content aims to give readers the basic knowledge needed to understand and apply effective anomaly detection strategies in various domains, providing a solid foundation for further exploration of the topic.

Anomaly detection is a method used in data analysis and machine learning to pinpoint data points or patterns significantly different from the norm. This process helps identify unexpected or unusual events or behaviour in a dataset, which could indicate a malfunction, cyberattack, or other unusual activity. Various anomaly detection methods include statistical analysis, machine learning algorithms, and neural networks [31].

Identifying possible security threats to a system or network is called threat detection. These threats can come from cybercriminals, hackers, insiders, or malware. Techniques like monitoring network traffic, user activity, and system behaviour are used to detect potential security breaches. Intrusion detection systems, log analysis, and behavioural analytics are commonly used for threat detection.

To maintain the security and integrity of computer systems, networks, and data, utilising both anomaly and threat detection techniques is essential. These techniques enable organisations to quickly and efficiently detect and respond to potential security incidents, ultimately preventing harm to their assets and reputation.

2.1 Techniques for Anomaly and Threat Detection

This section provides a brief overview of techniques used for detecting anomalies and threats. While threat detection techniques are not the main focus of this work, we briefly introduce them alongside anomaly detection techniques, as they are relevant to the topic of this thesis. It should also be noted that utilising multiple techniques would enhance the accuracy and efficiency of the results. The techniques listed here briefly introduce their fundamental concepts, which can give readers a high-level understanding before further exploring their applications.

Anomaly detection techniques:

- *Statistical analysis* can identify outliers or anomalies in a dataset, such as standard deviation, quartiles and regression analysis.
- *Time series analysis* can identify anomalies in a time-based dataset, such as detecting a sudden spike in network traffic or an unexpected drop in website

visits.

- *Machine learning algorithms* can be trained on standard data patterns to identify deviations or anomalies in new data, such as decision trees, random forests, and neural networks. While machine learning algorithms may incorporate statistical methods as part of their training, they can also capture more complex data patterns that may not be detectable using simple statistical techniques.

Threat detection techniques:

- *Signature-based detection* uses a database of known malware signatures or patterns to identify and block threats, such as antivirus software and intrusion detection systems [32].
- *Behaviour-based detection* looks for unusual behaviour patterns in the system or network activity, such as unexpected file transfers or attempts to access unauthorised resources [33].
- *Heuristic analysis* uses a set of rules or algorithms to identify potentially suspicious behaviour, such as monitoring for changes to critical system files or unusual login attempts [34].

In the following sections, we aim to provide a concise yet comprehensive overview of both statistical and time series analyses. We will then delve deeper into machine learning in the subsequent chapter. We aim to deliver a solid foundation in these fundamental concepts to understand better the key principles underpinning data science and artificial intelligence.

2.2 Statistical techniques

Statistical techniques are widely used in anomaly detection to identify deviations in data distributions and variable relationships. They aim to detect data points that significantly deviate from the norm, representing potential anomalies. From simple methods based on normality assumptions to advanced approaches for complex data distributions, these techniques offer diverse options for detecting outliers without complex models or extensive training. This section outlines the key statistical techniques, discussing their strengths, weaknesses, and applicability in various contexts.

2.2.1 Interquartile Range method (IQR)

The Interquartile Range (IQR) method is another simple statistical technique for detecting anomalies, especially in non-Gaussian data distributions. It is based on the concept of quartiles, where the data is divided into four equal parts. The IQR is calculated as the difference between the first quartile (Q1) and the third quartile

(Q3). Anomalies are data points that fall below Q1 - 1.5IQR or above Q3 + 1.5IQR. This method is less sensitive to extreme values than the Z-score method and can be more robust in certain situations. However, it may still struggle with complex or multi-modal data distributions [31].

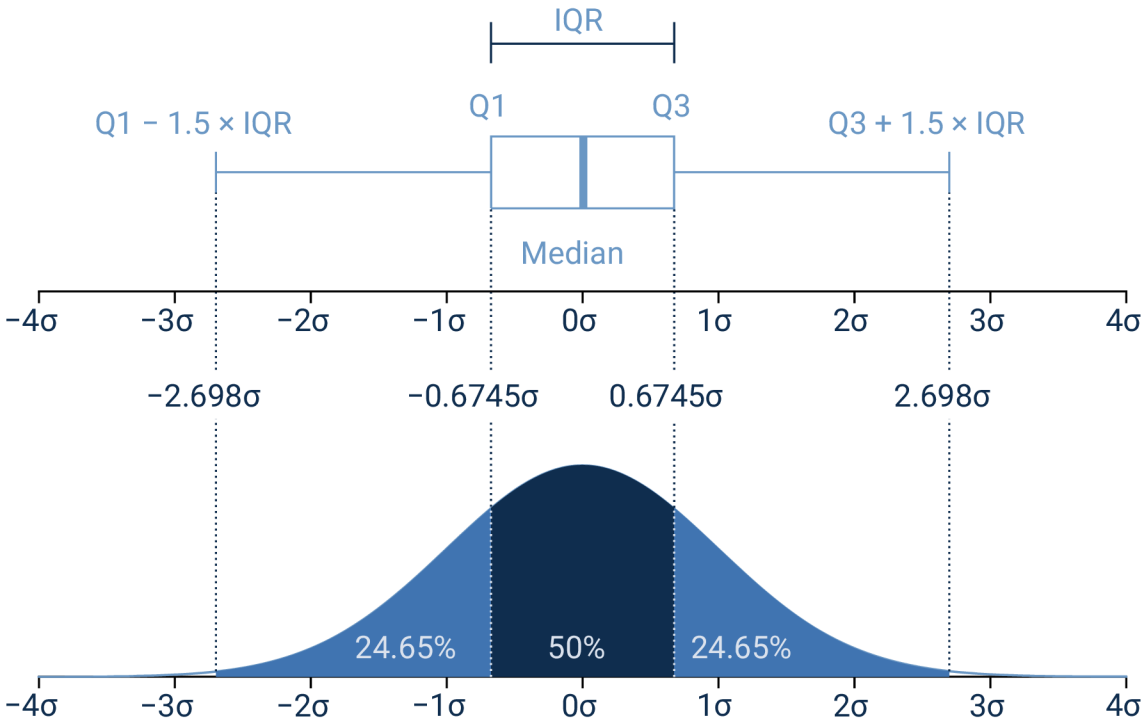

Fig. 2.1: Interquartile Range diagram. Adapted from [35].

2.2.2 Grubbs' test

Grubbs' test is a statistical method used to detect a single outlier in a univariate dataset, assuming it follows a Gaussian distribution. The test calculates the standardised value of the data point with the most significant absolute deviation from the mean. It compares it to a critical value derived from the t-distribution. The data point is an outlier if the standardised value exceeds the critical value. Grubbs' test is helpful for detecting a single outlier but may not be suitable for detecting multiple outliers or handling data with non-Gaussian distributions [31].

2.2.3 Gaussian Mixture Models (GMM)

Gaussian Mixture Models (GMM) are a more advanced statistical technique for anomaly detection, capable of modelling complex data distributions by representing them as a combination of multiple Gaussian distributions. GMMs use an

expectation-maximization (EM) algorithm to estimate the parameters of these Gaussian distributions iteratively. Anomalies can be detected by calculating the likelihood of each data point belonging to any of the Gaussian distributions, and those with a low likelihood are considered outliers. GMMs are more flexible than the Z-score, IQR, and Grubbs' test methods, as they can handle multi-modal data and account for various data shapes. However, they require more computational resources and may be sensitive to the initial parameter settings [31].

GMM will be used as the primary statistical technique for anomaly detection.

2.3 Time series analysis

Time series analysis is a statistical method used to study and understand data collected over time. This technique involves analysing and modelling the patterns, trends, and relationships within a time-based dataset to predict future outcomes or detect abnormalities. It is beneficial for data with a temporal component, as it can reveal trends and patterns that may not be evident with other statistical methods [36].

Time series analysis is popular in several fields, such as finance, economics, and environmental science. Its purpose is to analyse and predict patterns and trends over time. Time series analysis is applied to many areas, including the analysis of stock prices, forecasting of economic indicators, prediction weather patterns, and evaluating climate changes.

Detecting anomalies is a critical part of analysing time series data. It helps to identify patterns, trends, or events that are unexpected or unusual in a dataset ordered by time. There are three types of anomalies: point, contextual, and collective. Point anomalies are data points that deviate significantly from the norm, whereas contextual anomalies are data points that seem unusual within a specific context, like a particular time of day or season. Collective anomalies involve a group of data points that display abnormal behaviour, even though individual data points may not be anomalous [36].

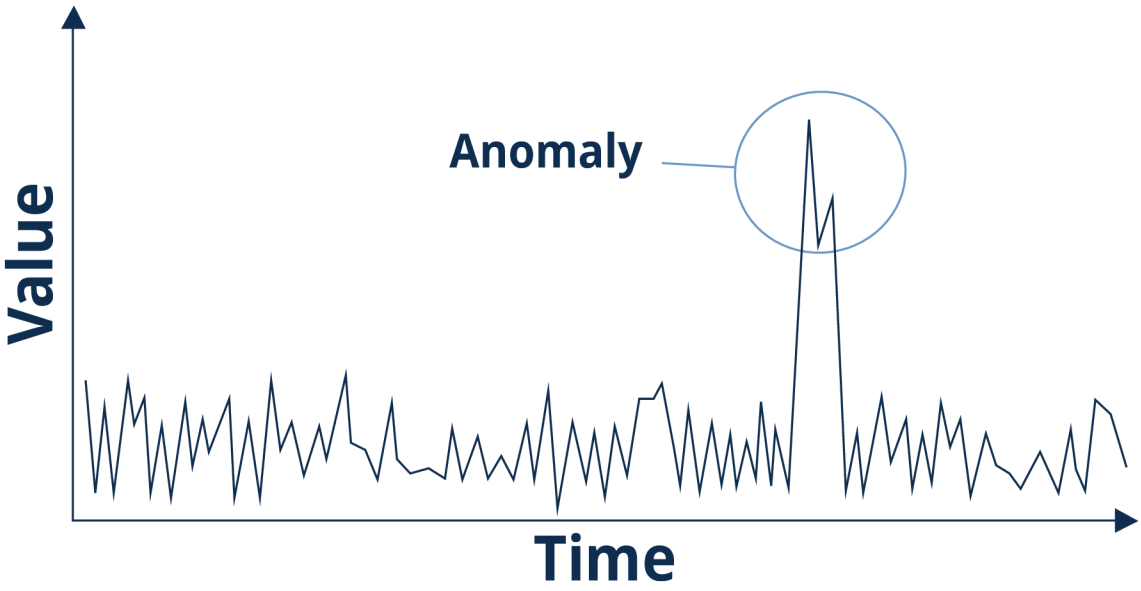

Fig. 2.2: Time series anomaly diagram.

3 Machine Learning

Machine learning is a branch of artificial intelligence that develops algorithms and statistical models to allow computer systems to learn and improve from their experiences. In simpler terms, machine learning algorithms can automatically learn patterns and relationships in data without explicit programming. To achieve this, we expose the algorithm to a vast dataset and train it to identify patterns or make predictions based on that data [37].

Machine learning can be categorised into four main types: supervised, unsupervised, semi-supervised, and reinforcement learning.

- *Supervised learning* is a process of training a model with a labelled dataset. In simpler terms, every example in the dataset is associated with a label or target value. The main objective is to create a mapping between input variables (features) and the output variable (label) so that the model can make precise predictions on new, unseen data. Examples of supervised learning include image classification, speech recognition, and sentiment analysis [37].
- *Unsupervised learning* involves training a model without labelled data, which means there are no assigned target values for each example. Its objective is comprehending the data's fundamental structure, like patterns, clusters, or anomalies. Unsupervised learning techniques include clustering, anomaly detection, and dimensionality reduction [37].
- *Semi-supervised learning* leverages a small amount of labelled data for supervision while using unlabeled data to capture underlying patterns. This approach is practical when labelled data is limited or costly, allowing for efficient utilisation of available resources. [37].
- *Reinforcement learning* teaches an agent to make decisions that result in the highest possible reward. The agent learns by interacting with the environment and receiving positive or negative feedback. Some examples of reinforcement learning applications include game playing, robotics, and autonomous driving [37].

Our primary focus is detecting anomalies in multivariate time series, which refers to a collection of time series data where multiple variables are measured simultaneously over a period of time. We will compare commonly used techniques, such as clustering, density-based methods, and one-class classifiers. Our evaluation will be based on their ability to detect anomalies and their computational efficiency accurately.

The thesis will investigate using *Natural Language Processing* (NLP) to detect anomalies. Along with traditional methods, we will use a fine-tuned *Generative Pre-trained Transformer* (GPT) language model to determine the likelihood of a

sentence anomalous. In our scenario, this approach is beneficial as the data primarily comprises text in a nested *JavaScript Object Notation* (JSON) format with a highly irregular structure, which deviates from the typical tabular data that machine learning algorithms or models might expect.

Due to the complexity of the data and the lack of labelling, unsupervised learning methods will be utilised.

We aim to assess the efficiency of the NLP-based anomaly detection approach by comparing its performance with traditional methods on the benchmark dataset. We will also explore the effects of different factors, such as the training dataset's size, the language model's complexity, and the choice of pre-processing techniques. Our findings will have practical implications, and we will investigate potential real-world uses of NLP-based anomaly detection.

3.1 Performance Evaluation Metrics

When using machine learning to detect anomalies and identify new patterns in data, it is crucial to have evaluation metrics in place. These metrics help assess the performance of various algorithms and models.

Precision, recall, F1 score, and area under the receiver operating characteristic (ROC) curve are this field's most commonly used evaluation metrics. Precision measures the number of true positives compared to all predicted positive instances, while recall measures the number of true positives compared to all actual positive instances. Accuracy is the ratio of correctly predicted instances (both positive and negative) to the total number of instances. The F1 score is a balanced performance measure considering precision and recall. The area under the ROC curve (AUC) is a metric that evaluates how well the algorithm distinguishes between normal and anomalous instances [38].

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.2}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3.3}$$

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

$$\text{AUC} = \int_{-\infty}^{\infty} \text{Recall}(f), \text{d}(\text{FPR}(f)) \tag{3.5}$$

where $\text{FPR} = \frac{\text{FP}}{\text{FP}+\text{TN}}$.

Other metrics that may be used to evaluate anomaly and novelty detection include accuracy, mean squared error (MSE), negative log-likelihood (NLL), and root mean squared error (RMSE).

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.6}$$

$$\text{NLL} = -\frac{1}{n}\sum_{i=1}^{n}\log(p(y_i|\mathbf{x}_i)) \tag{3.7}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.8}$$

Please note that in the above equations:

- y_i represents the true value for the i-th sample.
- $\hat{y}_i$ represents the predicted value for the i-th sample.
- n represents the total number of samples.
- TP stands for True Positives (correctly predicted positive samples).
- TN stands for True Negatives (correctly predicted negative samples).
- FP stands for False Positives (negative samples incorrectly predicted as positive).
- FN stands for False Negatives (positive samples incorrectly predicted as negative).
- TPR(f) represents the True Positive Rate (also known as Sensitivity or Recall) at a given threshold f.
- FPR(f) represents the False Positive Rate at a given threshold f.
- The integration in the AUC equation represents the integration over the entire range of thresholds.
- $p(y_i|\mathbf{x}_i)$ represents the conditional probability of the true value y_i given the input features $\mathbf{x}_i$.

3.2 Common Anomaly Detection Techniques

In this segment, we will delve into a handful of machine learning-based anomaly detection techniques that have gained significant traction in the industry. These techniques include clustering, density-based methods, and one-class classifiers designed to identify and handle anomalies effectively [39].

3.2.1 Distance-based techniques

Distance-based techniques are crucial in anomaly detection, focusing on the proximity between data points to uncover potential outliers. These methods identify

anomalies with significantly different characteristics than their neighbours by measuring the similarity or dissimilarity between data points. Distance-based techniques often rely on simple metrics and require minimal assumptions about the data distribution, making them versatile and applicable in various situations.

K-Nearest Neighbours

K-Nearest Neighbours (k-NN) is a non-parametric machine learning algorithm used for classification and regression tasks. k-NN works on finding the K closest data points in the training set to a new, unlabeled data point and assigning a label or value based on most K-nearest neighbours [39].

The k-NN algorithm determines the most frequent class label among the K-nearest neighbours for classification tasks and assigns it to the new data point. Regarding regression tasks, the k-NN algorithm calculates the mean of the K-nearest neighbours and uses it as the predicted value for the new data point [39].

The k-NN algorithm has a significant benefit in its simplicity and ease of interpretation. It does not presume anything about the data's distribution, and the results are straightforward. Nevertheless, k-NN has some constraints, such as its sensitivity to the K value, the curse of dimensionality, and computational complexity. These limitations can impact its ability to detect anomalies effectively.

When detecting anomalies, k-NN can serve as a one-class classifier that identifies data points that do not match the learned model. However, the accuracy of anomaly detection heavily depends on the choice of K and distance metric. As a result, it is crucial to carefully evaluate and adjust hyperparameters to achieve the best performance from k-NN in anomaly detection tasks.

Local Outlier Factor

The *Local Outlier Factor* (LOF) is a well-known algorithm that detects datasets anomalies by measuring a data point's local density compared to its neighbours. This unsupervised machine learning algorithm is widely used for this purpose [39].

To start the LOF algorithm, we select a data point from the dataset and identify its k-nearest neighbours using a distance metric. Next, we calculate the chosen data point's *Local Reachability Density* (LRD) by finding the inverse of the average distance of its k-nearest neighbours. This LRD value helps us determine how isolated and densely populated the data point is compared to its neighbours.

To determine if a data point is an outlier, we calculate its LOF score by dividing the LRD of the data point by the average LRD of its k-nearest neighbours. If the resulting score is greater than 1, the data point is an outlier compared to its neighbours and the rest of the dataset [39].

The LOF algorithm is a helpful tool that can identify outliers or anomalies by analysing all data points in a dataset. It is beneficial for detecting anomalies within dense regions of the dataset and distinguishing them from anomalies in sparse regions.

While the LOF algorithm is valuable, it has certain limitations to keep in mind. One limitation is that selecting hyperparameters k and distance metrics can significantly affect the algorithm's performance. Additionally, managing large datasets can be difficult due to the computational complexity of the LOF algorithm.

3.2.2 Clustering-based techniques

Clustering-based techniques are a popular choice for anomaly detection, utilising unsupervised learning algorithms to group similar data points and uncover hidden patterns. By identifying clusters within the data, these methods distinguish anomalies as points that do not fit well into any cluster or are distant from cluster centroids.

Clustering-based techniques offer the advantage of adaptability to various data structures and the ability to discover irregularly shaped clusters without prior knowledge of the number or distribution of anomalies.

k-Means clustering

K-Means clustering is an unsupervised learning method that groups similar data points into k-distinct clusters. It works by assigning each data point to the nearest cluster centroid and updating the centroid positions based on the mean of the points within each cluster [39].

Anomalies are identified as distant points from their assigned cluster centroids. However, K-Means assumes that clusters are spherical and equally sized, which may not always hold. The algorithm is also sensitive to the initial centroid placement and the choice of k. Nonetheless, K-Means is easy to use and computationally efficient [39].

Density-Based Spatial Clustering of Applications

DBSCAN is a clustering algorithm that detects clusters of arbitrary shapes and identifies noise points that are considered anomalies. It groups closely packed data points based on a distance metric and density threshold and treats points that do not belong to any cluster as noise [39].

Unlike k-Means clustering, DBSCAN does not require specifying the number of clusters beforehand and can automatically discover them based on the input data's

density distribution. This makes it suitable for detecting complex and irregularly shaped data anomalies. However, DBSCAN may face difficulties when the data contains clusters with varying densities, and it is sensitive to the choice of distance metric and density threshold parameters [39].

3.2.3 Supervised and semi-supervised techniques

Supervised and semi-supervised techniques are potent anomaly detection approaches that rely on labelled data to build predictive models. While effective when labels are available, they may not be suitable when obtaining accurate labels is challenging or infeasible. This thesis will focus on unsupervised techniques, as the data at hand makes labelling difficult. Thus, this section briefly introduces supervised and semi-supervised techniques but excludes them from further exploration in this study.

Support Vector Machines

Support Vector Machines (SVM) are typically used for classification and regression tasks. In anomaly detection, SVM can be applied as a binary classification problem, where data points are labelled normal or anomalous. SVM aims to find the optimal hyperplane that separates the two classes, maximising the margin between the closest points of each class, called support vectors [39].

One-Class SVM, a variant of the standard SVM, is designed explicitly anomaly detection. It works by finding the smallest hyperplane that encloses most data points, treating points outside this hyperplane as anomalies. SVM is effective for high-dimensional data and can handle non-linear relationships using kernel functions. However, it may not scale well with large datasets and is sensitive to the choice of hyperparameters, such as the cost parameter and kernel functions [39].

Decision trees

Decision trees can be used for both classification and regression tasks. They work by recursively splitting the input data based on feature values, creating a tree-like structure representing the relationships between features and the target variable.

Decision trees can classify data points based on their feature values. They can handle non-linear relationships and are easy to interpret. However, decision trees may be prone to overfitting. In addition, obtaining accurate labels for supervised anomaly detection can be challenging, as anomalies are typically rare and hard to characterise [39].

3.2.4 Tree-based ensemble methods

Tree-based ensemble methods combine multiple decision tree learners to enhance anomaly detection performance. Among these techniques, Isolation Forest is designed explicitly for unsupervised anomaly detection and offers unique advantages like handling high-dimensional data and computational efficiency. In this study, we will focus on Isolation Forests for detecting anomalies.

Isolation Forest

Isolation Forest is based on the concept of isolation. A tree-based algorithm works by partitioning the dataset into subsets and identifying anomalies as data points isolated in fewer partitions.

The Isolation Forest algorithm randomly selects a feature and a value for that feature and then splits the data into two partitions based on the selected value. The process is repeated recursively for each partition until all data points are isolated. Data points isolated in fewer partitions are considered anomalies, while those isolated in many are considered normal [40].

The Isolation Forest algorithm has several advantages, including its ability to handle high-dimensional datasets and its computational efficiency. It also requires minimal domain knowledge, making it applicable to many datasets [40].

A significant drawback is that it may not perform well on datasets with many anomalies or anomalies close to standard data points. Additionally, the algorithm's performance can be sensitive to the choice of hyperparameters, such as the number of trees and the maximum depth of the trees [40].

3.3 Neural networks

Neural networks are computational models inspired by the structure and function of the human brain. They consist of interconnected nodes or neurons organised into layers, which work together to learn patterns and representations from input data. Artificial neurons are the fundamental building blocks of neural networks, designed to mimic biological neurons' behaviour. Each artificial neuron receives input, processes it, and passes the result to other neurons in the network [37].

A typical neural network has three types of layers: the input layer, one or more hidden layers, and the output layer. The input layer receives the input data, while the output layer provides the final prediction or result. The hidden layers, which lie between the input and output layers, transform the input data into meaningful representations that help the network make accurate predictions [37].

Each connection between neurons has a weight associated with it, representing the strength of the connection. During training, these weights are adjusted to minimise the difference between the network's predictions and target values (i.e., the error). This is achieved through backpropagation, which involves computing the error gradient concerning each weight and updating the weights accordingly [37].

There are several types of neural networks, each with its unique architecture and properties. Some of the most common types include:

- *Feedforward Neural Networks* (FNN): The simplest form of neural networks, where information flows in a single direction from the input layer to the output layer without any loops or cycles. FNNs are helpful for various tasks, such as classification and regression [41].
- *Convolutional Neural Networks* (CNN): These networks are designed to handle grid-like data, such as images or multidimensional arrays. CNNs use convolutional layers to scan the input data and learn local features, making them particularly effective for tasks like image recognition, object detection, and natural language processing [41].
- *Recurrent Neural Networks* (RNN): RNNs are designed to handle sequential data, such as time series or natural language text. Unlike FNNs, RNNs have connections that form loops, allowing them to maintain a hidden state that can capture information from previous time steps. RNNs can model temporal dependencies and perform tasks like language translation, speech recognition, and time-series prediction [41].
- *Long Short-Term Memory* (LSTM) Networks: LSTMs are a particular type of RNN designed to overcome vanishing gradients, a common problem in training deep RNNs. LSTMs can effectively model long-range dependencies in sequential data, making them suitable for tasks like machine translation, text generation, and time-series analysis [41].

3.3.1 Neurons

In artificial neural networks, a perceptron (also called an artificial neuron or node) is a simplified computational model inspired by its biological counterpart.

An artificial neuron receives input from other neurons or input data, processes the input by applying a weighted sum and an activation function, and passes the output to other neurons in the network. The weights associated with the connections between neurons are adjusted during the learning process to minimise the prediction error and improve the network's performance [37].

Artificial neurons are the basic building blocks of neural network architectures, enabling them to learn and represent complex patterns and relationships in the input

data [37].

3.3.2 Layers of neurons

Neural networks use layers to group artificial neurons or nodes that perform specific operations on input data. The purpose of layering is to allow the network to learn hierarchical representations of the input data, which helps it to capture complex patterns and relationships.

$$\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{3.9}$$

Equation 3.9 represents a fully connected (dense) layer, where each neuron is connected to every input in the input vector. In this equation:

- x represents the input vector to the dense layer.
- W denotes the weight matrix, where each row represents the weights associated with a neuron in the layer.
- b represents the bias vector, which is added element-wise to the weighted sum.
- σ represents the activation function applied element-wise to the weighted sum, producing the output vector y of the dense layer.

A neural network can effectively transform and process the input data to produce accurate predictions or classifications by combining multiple layers with different architectures and functionalities. As the network processes the input data, the layers' arrangement helps it learn more complex and abstract features, making it better at generalising to new, unseen data.

- *Input layer*: The input layer receives the input data and passes it to the subsequent layers. Each neuron in the input layer represents a single feature or dimension of the input data. The dimensionality of the input data determines the number of neurons in the input layer.
- *Hidden layers*: Hidden layers lie between the input and output layers and transform the input data into meaningful representations that help the network make accurate predictions. The neurons in the hidden layers apply a weighted sum of their inputs, followed by an activation function (e.g., ReLU, sigmoid, or tanh) to introduce non-linearity. A neural network can have multiple hidden layers, and the number of neurons in each layer can vary depending on the complexity of the problem and the network architecture.
- *Output layer*: The output layer provides the final prediction or result from the network. The number of neurons in the output layer depends on the task the network is designed to perform. For example, in a binary classification problem, the output layer may have a single neuron with a sigmoid activation function to represent the probability of a data point belonging to the positive

class. For multi-class classification, the output layer typically has as many neurons as there are classes, with a softmax activation function to produce class probabilities.

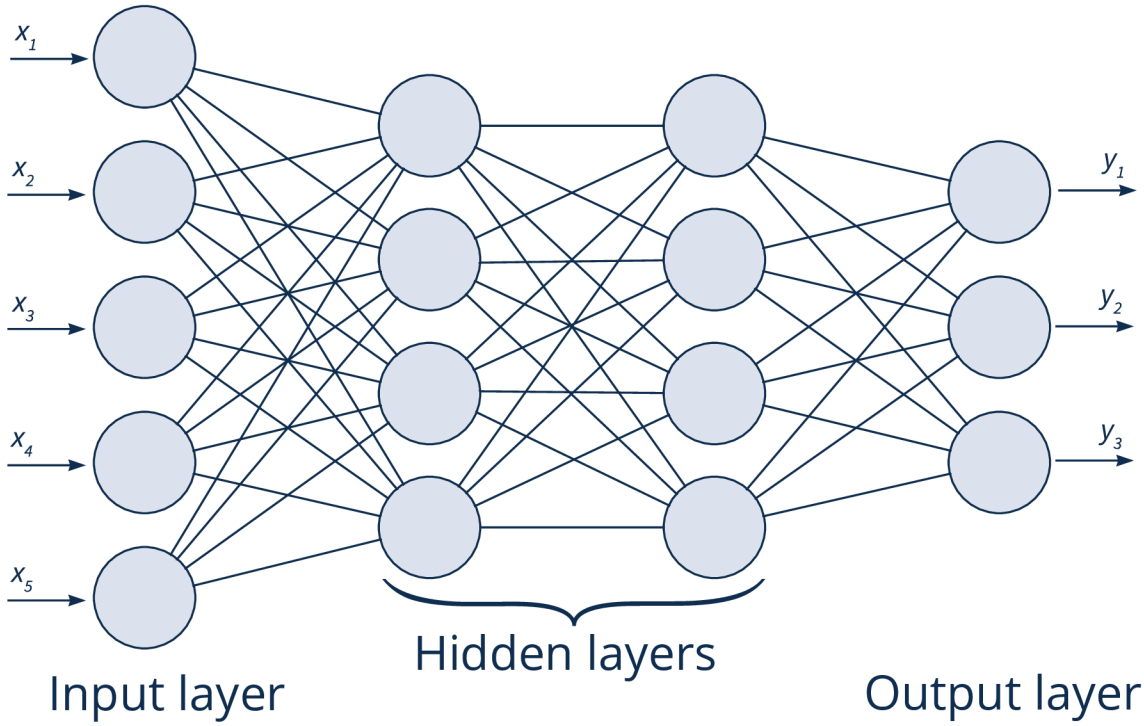

Fig. 3.1: Layers of a neural network.

In addition to these primary layer types, more specialised layer types can exist depending on the network architecture and the problem being solved:

- *Convolutional layers*: Used in convolutional neural networks (CNNs) for grid-like data (e.g., images), convolutional layers apply a series of filters or kernels to the input data to learn local features and patterns. Convolutional layers effectively capture spatial information and are commonly used in tasks like image recognition and natural language processing [42].
- *Recurrent layers*: Used in recurrent neural networks (RNNs) for sequential data (e.g., time series or text), recurrent layers maintain a hidden state that can capture information from previous time steps. This allows RNNs to model temporal dependencies in the data. Examples of recurrent layers include simple RNN layers, Long Short-Term Memory (LSTM) layers, and Gated Recurrent Unit (GRU) layers [42].
- *Transformer layers*, central to the groundbreaking Transformer architecture, efficiently model long-range dependencies in natural language processing. These

layers combine Multi-Head Self-Attention and Position-wise Feed-Forward Networks. Later in this chapter, we will explore their applications in anomaly detection [42].

- *Fully connected layers*: Also known as dense or linear layers, fully connected layers combine the learned features from previous layers and produce the final prediction. In a fully connected layer, each neuron is connected to every neuron in the previous and subsequent layers [42].

3.3.3 Autoencoders

Autoencoders are artificial neural networks used to learn efficient codings of unlabeled data. They are trained on a set of standard data points and then used to reconstruct the same data points. Anomalies are data points with a high reconstruction error or cannot be reconstructed accurately.

The autoencoder algorithm consists of an encoder and a decoder. The encoder compresses the input data into a low-dimensional representation, and the decoder reconstructs the original data from the compressed representation. During training, the autoencoder learns to minimise the difference between the input and reconstructed data [43].

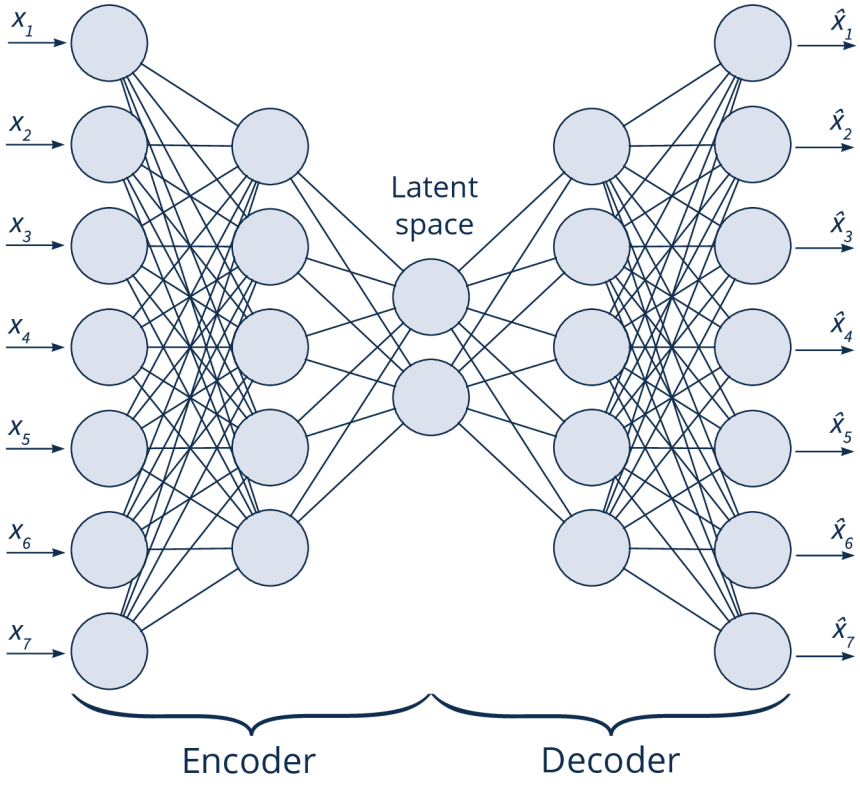

Fig. 3.2: Autoencoder diagram.

One of the main advantages of the autoencoder algorithm is its ability to handle complex data structures and nonlinear relationships between variables. It can also be used for unsupervised anomaly detection, making it useful for datasets with few labelled anomalies [43].

As previously mentioned, the limitation of unsupervised learning is that it may not perform well on datasets with many anomalies or anomalies significantly different from the usual data points. Additionally, the choice of hyperparameters, such as the number of hidden layers and neurons, can significantly impact the algorithm's performance.

3.3.4 Generative Pre-trained Transformer (GPT)

Generative Pre-trained Transformer (GPT) is a type of neural network architecture used for natural language processing tasks, such as language modelling, text generation, and sentiment analysis. It was developed by OpenAI and is based on the Transformer architecture, a type of neural network architecture that uses attention mechanisms to process data sequences [44].

GPT is a pre-trained language model trained on massive amounts of text data, such as Wikipedia, books, and news articles. It learns the statistical properties of language, such as the distribution of words and the relationships between words, and can generate coherent and grammatically correct sentences.

The GPT architecture consists of a multi-layered Transformer decoder network that generates text by predicting the next word in a sequence based on the previous words. The network is pre-trained on a large corpus of text data and then fine-tuned on specific tasks, such as sentiment analysis or text classification [44].

One of the main advantages of GPT is its ability to generate high-quality text that is coherent and contextually relevant. It can also be fine-tuned for specific language tasks, making it a powerful tool for natural language processing applications [44].

However, the GPT architecture also has some drawbacks. One is that it requires massive amounts of data for pre-training, which can be computationally expensive and time-consuming. Additionally, the generated text can sometimes be biased or inaccurate, depending on the quality and representativeness of the training data.

Our use case involves fine-tuning a GPT-2 model using the provided dataset. However, instead of generating new data, we will use the model to determine the likelihood of a given sentence occurring. This will help us detect any anomalies in the dataset. Before proceeding, let us review the core concepts that make GPT work.

Tokenisation

Tokenisation is a fundamental step in natural language processing that involves segmenting a text into smaller units called tokens. Depending on the specific application, these tokens are typically words but can also be characters, subwords, or other meaningful units. The primary purpose of tokenisation is to convert unstructured text data into a structured format that can be analysed computationally. This technique is frequently used as a preprocessing step in NLP to enable further analysis or modelling of the text data. By breaking down the text into smaller, more manageable pieces, tokenisation facilitates a range of NLP tasks, such as part-of-speech tagging, sentiment analysis, and text classification [45].

Transformers

Transformers are a type of neural network architecture that has emerged as one of the most successful approaches to *Natural Language Processing* (NLP). Unlike traditional RNNs, transformers process input text in parallel, breaking it down into smaller tokens embedded into high-dimensional vectors. These vectors are then passed through multiple layers utilising a self-attention mechanism to focus on relevant input parts. This allows transformers to capture long-range dependencies and effectively model the structure of natural language [46].

Transformers have achieved state-of-the-art NLP benchmarks, including GLUE and SuperGLUE. They are highly versatile and can be fine-tuned for various NLP applications, such as language modelling, machine translation, and text classification. In addition, they can handle both sequential and non-sequential inputs, making them a powerful tool for analysing a wide range of natural language data [46].

Figure 3.3 shows the architecture of a single transformer layer (where Nx is the ID the layer) which has two sub-layers – a multi-head self-attention mechanism, and a position-wise fully connected feed-forward network [47].

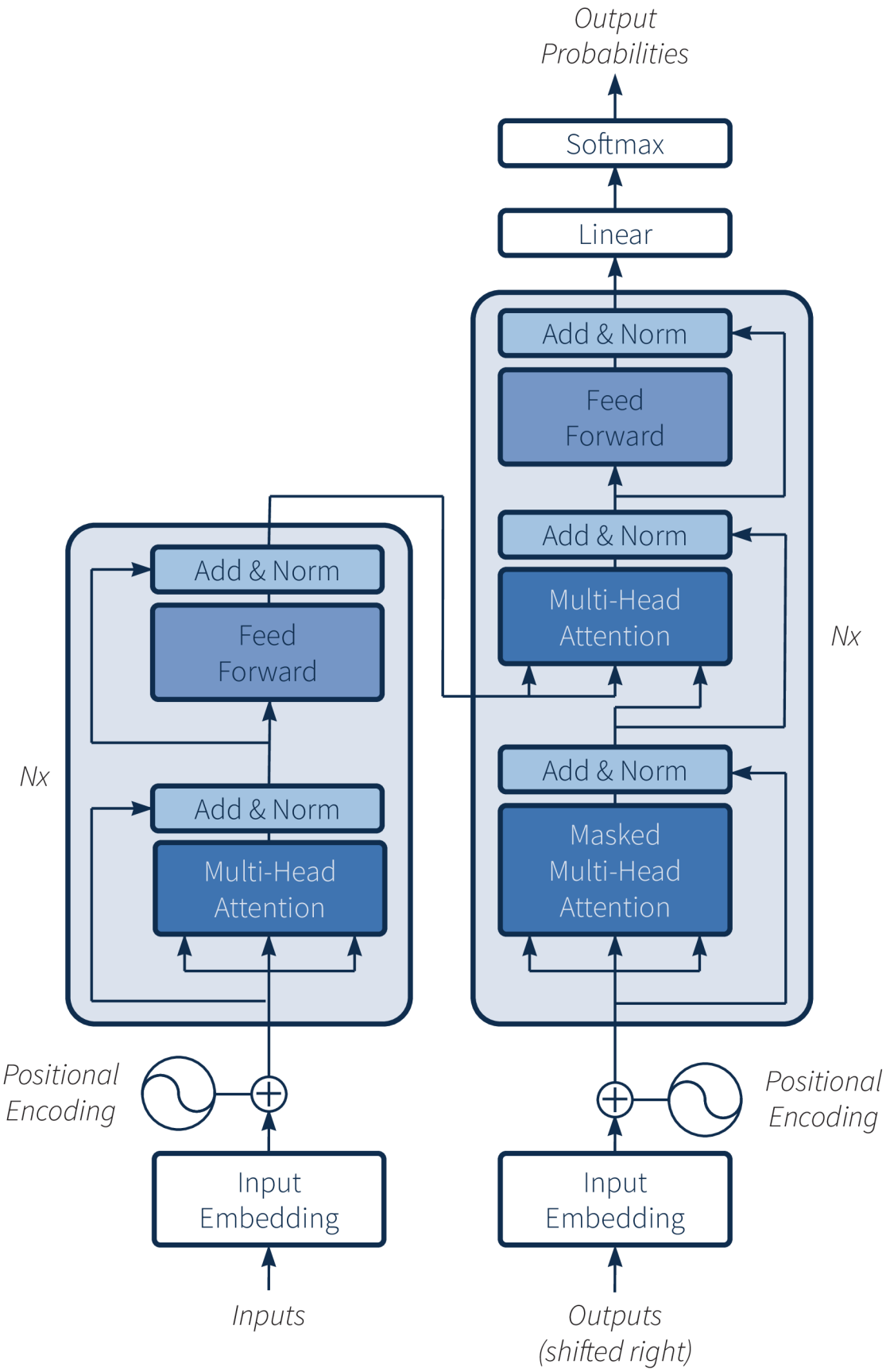

Fig. 3.3: Transformer architecture. Adapted from [47]

4 Data Analysis

In this chapter, we conduct a thorough analysis of the data collected for the purpose of anomaly detection in machine learning. We examine the data for any patterns, trends, or anomalies that could impact the performance of the anomaly detection models. In the process, we may encounter various challenges and issues, such as missing or inconsistent data, noisy data, imbalanced data distribution, or outliers. We propose and apply suitable solutions to address these problems and improve the quality and usefulness of the data. By conducting a comprehensive analysis of the data, we aim to obtain a deeper understanding of the data characteristics and enable the development of effective and accurate anomaly detection models.

4.1 Data Collection

Our colleagues at Red Hat generously provided the data used in this study. The dataset was obtained from two internal OpenShift Container Platform clusters, contains three months' worth of audit logs and underwent a pre-processing stage, which involved removing certain elements to ensure data privacy and confidentiality.

While the dataset is not entirely raw, it is deemed suitable for exploratory data analysis and machine learning to determine the most suitable method that will be used to train on the raw data. The pre-processed data has been carefully reviewed and deemed appropriate for this study.

It is important to note that the dataset contains sensitive information and will be treated with the utmost care and confidentiality throughout the study. Although the dataset does not contain personally identifiable or customer information, it is still deemed sensitive and will be kept private.

4.2 Analysis and Preprocessing Tools

Practical data analysis and exploration are critical to developing accurate and robust machine learning models. Data analysis and preprocessing tools provide a means to gain insight into the data, identify patterns, and prepare it for machine learning algorithms.

This section overviews this study's data analysis and preprocessing tools. These tools include various Python libraries, such as pandas, NumPy, and Matplotlib, widely used for data manipulation, analysis, and visualisation.

4.2.1 pandas

`pandas` is a popular Python data manipulation and analysis library. It provides a powerful and flexible toolset for working with structured data, such as tables or spreadsheets, allowing users to clean, transform, and analyse data efficiently [48].

Pandas is built on top of NumPy, another popular Python library for numerical computing, and offers additional functionalities for working with time series and missing data. The library provides data structures for handling one-dimensional (Series) and two-dimensional (DataFrame) data, with a wide range of built-in functions for data manipulation, such as filtering, grouping, pivoting, merging, and reshaping [49].

Pandas is widely used in data analysis and machine learning applications, making it an essential tool for any data scientist or machine learning practitioner. Its ability to easily handle large datasets and provide efficient data cleaning and preparation methods has made it a preferred tool among the data science community. The library also offers excellent visualisation capabilities, allowing users to create charts and graphs to understand the data better. Overall, Pandas is a powerful and versatile library that has revolutionised data analysis in Python and continues to play a significant role in developing machine learning models [48].

4.2.2 ydata-profiling

`ydata-profiling` is a Python library used for exploratory data analysis of structured datasets, built on top of the popular Pandas library. This library provides an efficient way to quickly generate a comprehensive dataset report, including statistics, visualisations, and other useful information. The report generated by ydata-profiling provides valuable insights into the data, including data distribution, missing values, the correlation between features, and various other data quality metrics. This library is highly customisable, allowing users to control the level of detail in the report and the type of visualisations used. Ydata-profiling is a time-efficient way to generate insights into the data that can help guide the machine learning modelling process [50].

4.2.3 matplotlib

`matplotlib` is a Python library widely used for data visualisation and plotting. It provides various tools for creating high-quality and customisable plots, charts, and graphs for data analysis and presentation. Matplotlib is designed to work seamlessly with other Python libraries, such as NumPy and Pandas, making it an essential tool for data scientists and machine learning practitioners. The library provides

various plotting functions, including scatter plots, line plots, histograms, and bar charts. Matplotlib also offers extensive customisation options, allowing users to adjust every plot aspect, including colour scheme, font, size, and style. This library is highly flexible and can be used for various visualisation tasks, from simple data exploration to complex data presentations. Overall, Matplotlib is an essential tool for data visualisation in Python, providing a powerful and versatile set of functions for generating high-quality plots and charts [51].

4.2.4 seaborn

`seaborn` is a Python library built on Matplotlib, providing a high-level interface for visually appealing statistical graphics. This library is designed to simplify the creation of complex visualisations and make it easier for data scientists and machine learning practitioners to explore and understand their data. Seaborn offers a wide range of plotting functions, including scatter plots, line plots, histograms, and heat maps, optimised for visualising statistical relationships in data [52].

Seaborn is highly customisable and provides extensive control over plot aesthetics, such as colour palettes, font size, and plot style. It also offers advanced features for exploring data distributions, such as kernel density estimation and probability density functions. Seaborn is particularly useful for data scientists and machine learning practitioners interested in exploring relationships between variables, such as correlation or regression analysis.

In addition to its visualisation capabilities, Seaborn offers convenient tools for working with multi-dimensional data, such as FacetGrid and PairGrid, allowing users to visualise relationships across multiple variables. Seaborn's intuitive interface and extensive documentation make it popular among data scientists and machine learning practitioners [52].

4.2.5 Scikit-learn

`Scikit-learn`, also known as sklearn, is a popular machine-learning library in Python. It provides many supervised and unsupervised learning algorithms and tools for data preprocessing, model selection, and evaluation. Sklearn is built on top of other scientific computing libraries in Python, such as NumPy, SciPy, and matplotlib, and is designed to be easy to use and integrate with other data analysis and visualisation tools. Its intuitive API and extensive documentation make it popular for beginners and machine learning experts. Sklearn offers a variety of algorithms for tasks such as classification, regression, clustering, and dimensionality reduction, making it a powerful tool for solving a wide range of problems [53].

The Sklearn library was utilised in this study's pre-processing and machine-learning stages. Sklearn's data preprocessing tools were used during pre-processing to prepare the data for model training, including a label encoder and feature scaling. In the machine learning stage, Sklearn's algorithms for unsupervised learning were utilised to build and evaluate machine learning models.

4.3 Tabular Data Preparation and Exploration

This section focuses on preparing the data in a tabular format optimised for machine learning models. We explore the data using statistical methods and correlation analysis and conduct data preprocessing tasks to improve data quality and relevance for machine learning algorithms.

The tabular data format was adopted for all machine learning models employed in this study except the GPT model. The GPT model utilises the same dataset but employs a different data format in the form of sentences. Specifically, the data was transformed from its original format to sentences, where each sentence corresponds to a single observation in the dataset. The data preparation and exploration phase involved various techniques to ensure the suitability of the dataset for the machine learning models employed in this study.

4.3.1 Data Cleanup

The data collected for this study was in the form of newline-delimited JSON files, where each log entry was represented as a JSON object on a separate line in the log file. However, to utilise the data with the pandas library and machine learning algorithms, it was necessary first to flatten the deeply nested data structure of the JSON files.

During the data preparation process, the first issue encountered was invalid lines in the original data that contained unterminated JSON data. These lines were likely caused by excessively long lines or incorrect redaction. To address this issue, a solution was devised to read the data line by line, parsing the JSON as it was encountered. Any line that was unable to be parsed was subsequently dropped.

Following the successful resolution of the invalid lines issue, the data was flattened using the pandas library, and the preparation process continued without any further issues.

4.3.2 Data Exploration

The collected logs for this study were in the form of deeply nested JSON structures, which presented challenges during the data preparation process. While some log fields were required, the optional fields often contained valuable information, such as the objects being modified, the associated security policy, and the nature of the operation[1]. The log data also depended on the type of request made, such that an update operation would require different attributes than a delete operation. Consequently, the flattened data initially contained more than 1,000 columns, with many of the columns predominantly empty.

The issue was addressed by attempting to remove empty attributes by retaining columns with consistently non-empty values throughout the entire dataset. However, this resulted in only ten columns, indicating the loss of valuable data. Subsequently, all columns that contained more non-empty values than empty ones across the entire dataset were retained, resulting in 20 columns that still did not contain important information.

Upon realising the limitations of these initial approaches, the dataset was split into subsets based on the *verb* attribute, which corresponds to the Kubernetes verb associated with the request made. To retain as much data as possible, columns that contained less than 10% non-empty values were removed from each subset. The resulting list of columns was combined from all subsets, resulting in a current filter of 55 columns across the dataset. This approach allowed for a more efficient representation of the log data while retaining some information.

4.3.3 Data Correlation

The exploration and analysis of data correlations are essential for identifying meaningful relationships and attributes within a dataset. One common approach is utilising correlation matrices, which visually represent the relationship between each column in the dataset. The ydata-profiling library includes correlation matrices in its report, but this approach is limited to only columns containing numerical data, thus omitting a significant amount of information.

To address this limitation, we employed the pandas library to convert text columns into a categorical representation, replacing text with numbered indices. We then used the categorical dataset to calculate the correlation matrix in pandas. The resulting correlation matrix was further visualised using the seaborn and matplotlib libraries, which generated correlation matrices in graphical form.

[1]The Kubernetes API reference is beyond the scope of this thesis; see the official documentation for details: `https://kubernetes.io/docs/reference/config-api/apiserver-audit.v1/`

4.3.4 Correlation Matrix

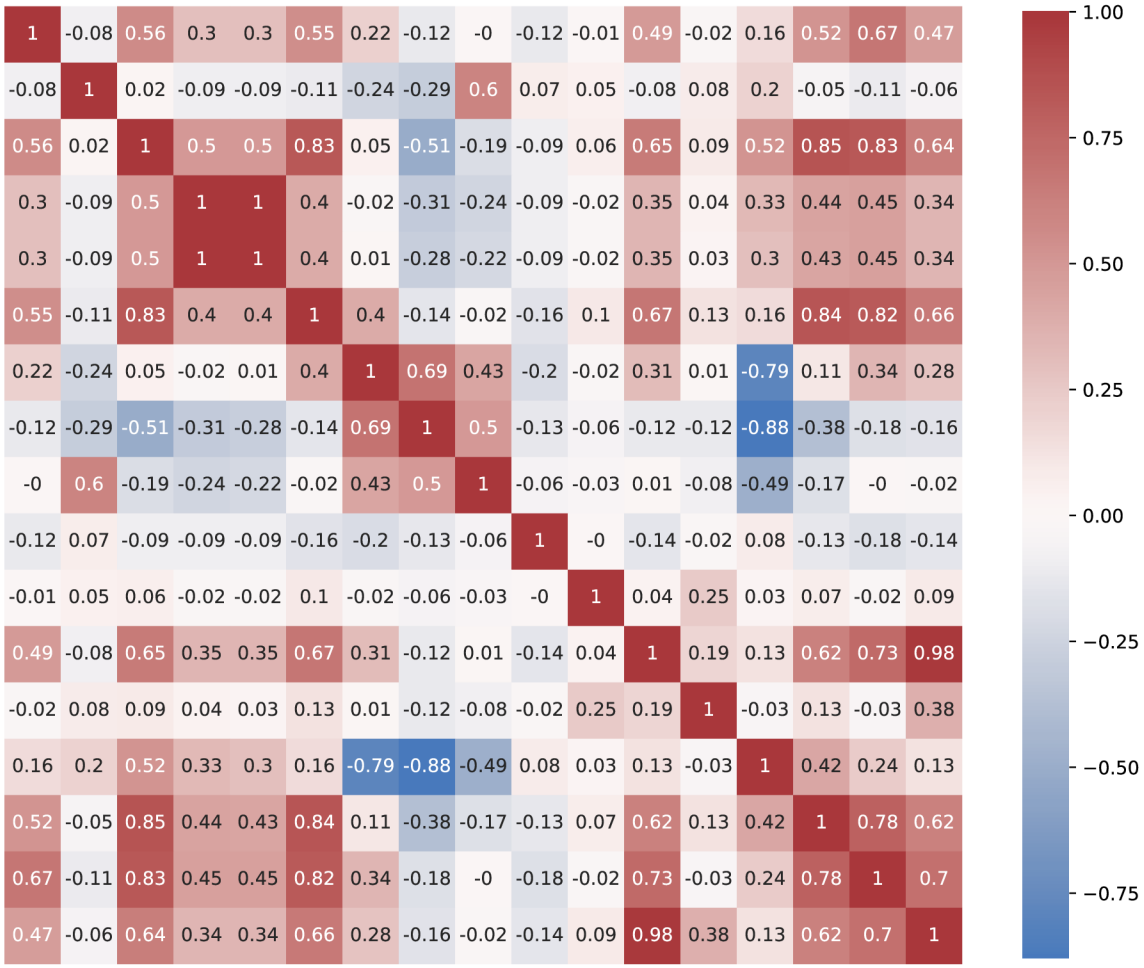

Fig. 4.1: Correlation matrix.

The resulting correlation matrix demonstrates the correlations between attributes of a table. A value of 1 indicates a direct proportional relationship between the given attributes, while a value of -1 indicates an inverse proportional relationship. A value of 0 suggests a lack of correlation between the given attributes.

Identifiable clusters of correlating attributes in the correlation matrix may indicate duplicate features within the dataset. These results can later be used when training the machine learning model, where these duplications could be filtered out with a minimal loss in detection performance.

Additional correlation matrices of the pruned dataset and verb subsets mentioned in Section 4.3.2 can be found in Appendix B.

4.3.5 Preprocessing Methods

In the case of time-series data, the timestamp information is critical in understanding the patterns and trends that emerge over time. However, the raw timestamp information cannot be directly used as input to machine learning algorithms, as they require numerical data. Instead, the difference between consecutive logs is calculated, resulting in a numerical value representing the elapsed time between two events. This time difference can be used as an input feature for the machine learning models.

It is necessary to note that the models used in this study do not hold context over time; they do not consider any temporal relationships or dependencies between the data points. Instead, the models learn patterns and anomalies based solely on the features of each item. While this approach may not capture the complex relationships within the time-series data, it provides a simple and effective way to identify anomalous behaviour.

Label encoder is a common technique for converting categorical data into numerical data. This approach is beneficial when dealing with string data, enabling machine learning algorithms to work with this type of data. The label encoder assigns a numerical value to each unique string within the dataset, creating a one-to-one mapping between the original and numerical data. This process preserves the underlying relationships between the string values while converting them into a format that the machine learning algorithms can use.

Finally, the min-max scaler is a data preprocessing technique that scales the dataset's features to values between 0 and 1. This normalisation step can help ensure that all features have a similar range and for improving the performance of some machine learning algorithms. By scaling the data, the models can learn from the data more effectively, resulting in more accurate predictions. Together, the label encoder and min-max scaler enable the machine learning models to analyse the time-series data and quickly identify anomalies effectively.

Learning long-term trends and dependencies within the data would be possible using other methods. For example, recurrent neural networks (RNNs) can model the temporal relationships between the data points and capture long-term dependencies within the time-series data.

This approach comes with a trade-off regarding computational complexity and model interpretability. RNNs are complex models that require significant computational resources to train, and their internal workings can be difficult to interpret. Furthermore, the choice of the model architecture and hyperparameters can significantly impact the model's performance, making it challenging to determine the best approach for a given dataset.

In addition to utilising recurrent neural networks (RNNs) to model the temporal relationships between data points, another approach for identifying patterns and anomalies in time-series data is to analyse the frequency of logs occurring in a given time frame.

By aggregating the logs into time intervals (e.g., hourly or daily) and analysing the frequency of logs within each interval, it is possible to identify patterns or anomalies that may not be immediately apparent from the raw time series data. For example, a sudden increase or decrease in the frequency of logs within a given interval may indicate anomalous behaviour that warrants further investigation.

4.4 Sentence Generation from Nested Data

The recursive conversion of deeply nested JSON logs into sentences is an important data preparation step that facilitates using such data in natural language processing applications, including fine-tuning GPT models.

The nested structure of JSON logs can make it difficult to extract relevant information for language processing tasks. The recursive transformation process involves traversing through each layer of the JSON hierarchy and recursively converting it into a sentence, where each sentence represents a separate log entry. This process eliminates unnecessary brackets and quotes, simplifies the data structure, and ensures that each log entry is represented as a separate sentence in the resulting data.

This transformation is beneficial for fine-tuning GPT models on log data. GPT models are trained to generate text based on the patterns and structures present in the input data. By converting the JSON logs into sentences, we provide the GPT model with a meaningful sequence of sentences that accurately represents the underlying structure and content of the log data. This approach ensures that the GPT model can learn to generate text that accurately reflects the patterns and relationships present in the original data.

The recursive conversion of deeply nested JSON logs into sentences is a crucial data preparation step for fine-tuning GPT models on log data. This transformation simplifies the data structure, eliminates unnecessary brackets and quotes, and provides the GPT model with a meaningful sequence of sentences that accurately represents the underlying patterns and structures in the log data.

5 Implementation

This chapter will explain the technical aspects of the machine learning models utilised in this thesis. We will cover the code snippets for developing and training these models, including the libraries utilised, the preprocessing steps to prepare the data, and the specific model architectures implemented.

This detailed analysis will provide readers with a comprehensive understanding of the technical intricacies of building and training machine learning models and how these models can be applied to solve real-world problems.

5.1 Tools Used

In this section, we want to introduce the tools we used to create our machine-learning model. We utilised PyTorch and Transformers in addition to the libraries already introduced. These tools are widely used in both the research and industry communities. We selected these tools because of their powerful capabilities, versatility, and ease of use. These tools enabled us to create and evaluate our model efficiently and effectively.

5.1.1 PyTorch

PyTorch is a machine learning framework that has gained popularity in academic and industry circles. It is known for being flexible, efficient, and easy to use. PyTorch is a free, open-source software library created by Facebook's AI research group. It allows developers to build and train machine learning models using a high-level programming interface.

One of PyTorch's key features is its dynamic computational graph, making model construction more flexible and efficient than other frameworks such as TensorFlow. This feature enables developers to construct neural networks during model execution, making experimenting with new architectures easier and iterating on model designs quickly [54].

PyTorch also offers a variety of modules and classes for building and training neural networks, such as layers, activation functions, loss functions, and optimisers. These modules can be combined to create complex neural network architectures, and developers can efficiently train the entire model [54].

5.1.2 Transformers

The Transformers library, developed by Hugging Face, is an open-source software library offering comprehensive tools for building and training natural language pro-

cessing (NLP) models [55].

It is built on top of PyTorch and includes state-of-the-art implementations of popular transformer-based architectures like BERT, GPT-2, and RoBERTa. Due to its ease of use, flexibility, and powerful capabilities, Transformers has become popular for NLP practitioners.

The library offers a range of modules and classes that enable users to create and train transformer-based models. These include pre-processing tools, model architectures, and fine-tuning scripts.

One of the critical features of Transformers is its ability to pre-train models on large-scale datasets, which can be fine-tuned on specific tasks like sentiment analysis, text classification, and question answering to achieve state-of-the-art performance on a wide range of downstream NLP tasks. In addition, Transformers offers a range of pre-trained models [55].

5.2 Tabular Preprocessing

This section will discuss the snippets of code used to pre-process data for tabular anomaly detection models. Pre-processing is a crucial step in machine learning, as it involves transforming raw data into a format that is suitable for use by the model. In the context of tabular anomaly detection, pre-processing may involve handling missing values, scaling or normalising features, or performing other transformations to the data.

As mentioned in the previous chapter, the data we are working with is in the form of deeply nested JSON objects, some of which are incomplete.

Listing 5.1 defines a function called `flatten_data` that takes a file as input and performs operations to convert JSON data into a tabular format using Pandas. It reads the file and loads each line of JSON data, skipping lines with errors. It then converts the data to a DataFrame using `pd.json_normalize` and converts the "requestReceivedTimestamp" column to a datetime format. The flattened DataFrame is then saved as a CSV file, and memory resources are freed.

Next, the data is pruned using a function called `drop_columns` shown in Listing 5.2. It helps to remove unnecessary columns and calculate the time difference between events using Pandas. The code also reads a CSV file, sorts the DataFrame by `"requestReceivedTimestamp"`, removes unwanted columns, and saves the pruned DataFrame as a CSV file. Additionally, the code frees up memory resources after execution.

Afterwards, the dataset was shuffled to distribute the two clusters' data uniformly. Subsequently, it was split into the training and testing sets, with the training set containing 80% of the entire dataset.

```python
def flatten_data(file: str):
    json_lines = []
    with open(file) as read_file:
        for line in read_file:
            try:
                json_line = json.loads(line.rstrip())
            except JSONDecodeError:
                continue
            json_lines.append(json_line)
    df = pd.json_normalize(json_lines, sep=".")
    df["requestReceivedTimestamp"] = pd.to_datetime(
        df["requestReceivedTimestamp"]
    )
    """data is saved and memory is freed."""
```

Listing 5.1: JSON Flattening.

```python
def drop_columns(file: str):
    df = pd.read_csv(file, low_memory=False)
    df.sort_values(by="requestReceivedTimestamp", inplace=True)
    df["requestReceivedDelta"] = (
        df["requestReceivedTimestamp"]
        .diff()
        .apply(lambda x: x / np.timedelta64(1, "ns"))
        .fillna(0)
        .astype("int64")
    )
    """Same as the above but for the 'stageDelta' column"""
    df1 = df[["requestReceivedDelta", "stageDelta"] + COLUMNS[2:]]
    """data is saved and memory is freed."""
```

Listing 5.2: Data Pruning.

The Function `label_scale` shown in Listing C.1 does two things: it encodes labels and scales the input data using Scikit-learn. It divides the input files into training and testing lists, fills missing values with zeros, and applies label encoding and scaling to each file. The processed data is saved as a CSV file, and memory is freed. The label encoding classes are also saved as NumPy objects, and the scaling objects are saved as Pickle objects so they can be loaded and used later.

After undergoing pre-processing, the data has been meticulously refined and is now fully prepared to be utilised in machine learning techniques.

5.3 Sentence Preprocessing

In this section, we will introduce a recursive technique utilised to convert JSON data into a sentence-like representation suitable for use in natural language processing. By recursively converting the nested structure of the JSON data into a sentence-like representation, we can effectively leverage the power of natural language processing

techniques to analyse and learn from the data to calculate later the probability of a data point being an anomaly.

This technique is based on two functions – `_dict_str` and `_list_str`, which function very similarly and only offer small differences based on the datatype passed to the function, which is why only `_dict_str` will be shown in Listing 5.3.

```
def _dict_str(dic: dict) -> str:
    result = ""
    for key, value in dic.items():
        if type(value) == dict:
            if result == "":
                result = f"{key} {_dict_str(value)}"
            else:
                result += f" {key} {_dict_str(value)}"
        elif type(value) == list:
            if result == "":
                result = f"{key} {_list_str(value)}"
            else:
                result += f" {key} {_list_str(value)}"
        elif value is None:
            pass
        else:
            if result == "":
                result = f"{key} {value}"
            else:
                result += f" {key} {value}"
    return result
```

Listing 5.3: Recursive string conversion.

These two functions recursively traverse the data to form a string that resembles a sentence while still containing all the information. Based on the data type of each value in the key-value pair of the passed dictionary, it is either converted to a string directly or recursed more deeply. The `_list_str` function is virtually identical, only looping through the items of a list instead of key-value pairs of a dictionary.

No extra pre-processing is necessary, and the result can be directly passed to the model.

5.4 Scikit-learn models

Scikit-learn provides various machine-learning methods, including those evaluated in this thesis. In this section, we will briefly show how to use them.

In Listing 5.4 the training dataset is imported into a pandas DataFrame using `pd.read_csv()`, and the isolation forest model is trained using the `fit()` method. After training, the model generates predictions for the test dataset using the `predict()` method. The results variable is a NumPy array that matches the length of the

```
train_df = pd.read_csv(path)
iso_forest = sklearn.ensemble.IsolationForest(random_state=0).fit(train_df)

test_df = pd.read_csv(path)
result = iso_forest.predict(test_df)
```

Listing 5.4: IsolationForest training and predictions.

test dataset. It contains values of 1 for normal results and -1 for anomalies. The `random_state` parameter is used to control the pseudo-randomness of the model to achieve consistent results across multiple runs.

Similarly, Local Outlier Factor and Gaussian Mixture models are fitted and used to generate predictions in Listing 5.5.

```
    lof = sklearn.neighbors.LocalOutlierFactor(
        n_neighbors=350, novelty=True
        ).fit(train_df)
    gmm = sklearn.mixture.GaussianMixture(
        n_components=8, random_state=0
        ).fit(train_df)
```

Listing 5.5: LOF and GMM training.

This snippet only shows the training of these models as predictions are made in the same way as in the case of Isolation forests.

The `n_neighbors` parameter is used to set the number of closest neighbours used to calculate the local density of each data point, and the `novelty` parameter is used to allow the model to respond to previously unseen data. The `n_components` parameter corresponds to the number of mixture components present in the datasets.

These numbers were chosen arbitrarily for a first look at the results, and a grid search algorithm was meant to determine the best-performing ones later. However, we soon realised a fatal pre-processing flaw that will become apparent in the next chapter, so they were not further explored.

5.5 Autoencoders

Much of the time dedicated to this thesis was focused on Autoencoders. They were explored before the already mentioned sklearn models and exhibited the same problems. This section will briefly show the implementation before we move on to the best-performing solution, Natural Language Processing.

The development was based on the widely used Pytorch Template Project[1] which enables the developer to focus on the things that matter the most, such as writing

[1] `https://github.com/victoresque/pytorch-template`

data loaders and designing the models. At the same time, the actual training process is already implemented and highly configurable.

Autoencoders were selected for anomaly detection because they can utilise the GPU to vastly improve their performance. The plan was to calculate the loss for a particular data point and determine whether it was anomalous based on a predetermined threshold. This threshold was supposed to be selected as an average of many artificially prepared anomalous log results.

The `KubeDataLoader` class, shown in Listing C.2 is a custom data loader for loading data from CSV files in PyTorch. The class inherits from BaseDataLoader, a PyTorch built-in class for data loading, and overrides its `__init__` method.

In the `__init__` method, the class constructor takes input parameters such as the directory containing the data, the batch size, whether to shuffle the data, the validation split ratio, the number of workers for loading the data, and the training and test CSV file names. The constructor then calls the parent class constructor with the dataset loaded using the `load_datasets` method.

Depending on the training flag, the `load_datasets` method reads the training or test CSV file. The CSV file is loaded using Pandas and then converted to a list of lists. Each sublist is then converted to a PyTorch FloatTensor. Finally, the list of tensors is zipped into a list of tuples, where each tuple contains a pair of tensors representing an input and output pair.

```
class DeepMinMaxedAutoencoder(nn.Module):
    def __init__(self, dims=[56, 256, 128, 64], use_bias=True):
        super(DeepMinMaxedAutoencoder, self).__init__()
        enc_layers = []
        dec_layers = []
        for i in range(len(dims) - 1):
            enc_layers.append(nn.Linear(dims[i], dims[i + 1], bias=use_bias))
            enc_layers.append(nn.ReLU())
        for i in reversed(range(1, len(dims))):
            dec_layers.append(nn.Linear(dims[i], dims[i - 1], bias=use_bias))
            dec_layers.append(nn.ReLU())
        self.encoder = nn.Sequential(*enc_layers)
        self.decoder = nn.Sequential(*dec_layers)

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Listing 5.6: DeepMinMaxedAutoencoder Class.

The `DeepMinMaxedAutoencoder` class shown in Listing 5.6 is a PyTorch module for building a deep autoencoder neural network. In the `__init__` method, the constructor takes a list of dimensions representing the input and output sizes of each layer in the network and a flag indicating whether to use bias terms in the

linear layers. The constructor then builds the encoder and decoder networks using PyTorch's `nn.Linear` and `nn.Tanh` layers.

The encoder is defined as a PyTorch `nn.Sequential` module that applies linear transformations and tanh activations to the input data. The decoder is defined similarly but in reverse order, with the output layer having the same size as the input layer.

In the forward method, the input data x is passed through the encoder to obtain the compressed representation `encoded` and then passed through the decoder to obtain the reconstructed output `decoded`. The reconstructed output is returned as the output of the forward pass.

5.6 Natural Language Processing

The model is based on `distilgpt2` [56]. This model is the smallest version of the Generative Pre-trained Transformer 2 (GPT-2). Its original use case is to generate text, but we will fine-tune it in this thesis on the prepared sentenceized data and use it for anomaly detection.

Later we can calculate the loss on a given sentence and determine whether it is anomalous based on a threshold that we will determine by preparing a dataset full of anomalous data and another one with good data. The decision boundary will be selected based on the distributions of both datasets' results.

This incredibly flexible approach allows us to use the entire dataset without removing features. It is way more robust and capable of handling small changes very well.

The model was fine-tuned using a ready-made script in the `transformers` library repository[2]. This script is well written and commented, and allows training or fine-tuning language models using any available HugginFace datasets or custom ones as well. Listing 5.7 shows an example command of how the script can be used. The fine-tuning was done on a *Red Hat OpenShift Data Science* (RHODS) cluster with an NVIDIA Tesla T4 GPU attached. The process lasted two days, resulting in a high-performance model.

[2] `https://github.com/huggingface/transformers/blob/main/examples/pytorch/language-modeling/run_clm.py`

```
python run_clm.py \
    --model_name_or_path distilgpt2 \
    --train_file path_to_train_file \
    --validation_file path_to_validation_file \
    --per_device_train_batch_size 4 \
    --per_device_eval_batch_size 4 \
    --do_train \
    --do_eval \
    --save_total_limit 10 \
    --output_dir /tmp/test-clm
```

Listing 5.7: GPT-2 Fine-tuning command.

In this command, the following arguments are used:

–**model_name_or_path:** Specifies the model to be used for training or fine-tuning.

–**train_file:** Specifies the path to the training file containing the data used to train the language model.

–**validation_file:** Specifies the path to the validation file containing the data used for model evaluation during training to monitor its performance.

–**per_device_train_batch_size:** Sets the batch size for training data, determining the number of training samples processed together in each iteration. Here, it is set to 4.

–**per_device_eval_batch_size:** Sets the batch size for evaluation data, determining the number of evaluation samples processed together during model evaluation. It is also set to 4.

–**do_train:** Indicates that the training process should be executed, triggering the training loop to train the language model using the specified training file.

–**do_eval:** Specifies that model evaluation should be performed, triggering the evaluation process using the validation file to assess the model's performance.

–**save_total_limit:** Defines the maximum number of checkpoints or saved models to keep during training. It is set to 10, meaning that the 10 most recent checkpoints will be saved.

–**output_dir:** Specifies the directory where the trained model and associated files will be saved. In this case, it is set to "/tmp/test-clm".

To use the trained model, the following shown in Listing 5.8 is used to process a sentenceized log and return the probability of this sentence occurring.

The score function takes a string and returns a probability-like score indicating how well it matches the pre-trained neural network model. It tokenises the input sentence, calculates the loss using the pre-trained model, and returns the exponential of the negative loss value as the output score. The tokens are split into chunks for strings that exceed the maximum length defined in the model. The loss is calculated

for each chunk and added together, producing an average after exponentiating.

```
def score(sentence: str) -> np.float32:
    ids = (
        tokenizer(
            tokenizer.bos_token + sentence + tokenizer.eos_token, return_tensors="
    pt"
        )
        .to(device)
        .input_ids[0]
    )
    loss = np.float32(0.0)

    with torch.no_grad():
        if len(ids) > (tokenizer.model_max_length - 2):
            max_len = tokenizer.model_max_length - 2
            chunks = torch.split(ids, max_len)
            for ch in chunks:
                ch_loss: np.float32 = (
                    model(input_ids=ch, labels=ch).loss.cpu().detach().numpy()
                )
                loss = np.nansum([loss, ch_loss])
            loss = loss / len(chunks)
        else:
            loss = model(input_ids=ids, labels=ids).loss.cpu().detach().numpy()
    if loss == np.nan:
        loss = np.float32(0.0)
    return np.exp(-loss)
```

Listing 5.8: GPT-2 score function.

6 Results

In this chapter, we will share the outcomes of our chosen machine-learning models on a dataset previously unseen by these models during their training phase. We assessed the models' ability to detect anomalies in the data by introducing tampered data points, where their values were replaced with randomly generated ones. This was to replicate real-world scenarios where anomalies can occur due to errors, malfunctions, or malicious intent. We added 10,000 tampered data points to the dataset, which was then mixed among roughly 193,000 natural logs, resulting in a total of 203,297 data points to be assessed.

To ensure the reliability and robustness of the results, we carefully picked evaluation metrics commonly used in the machine learning field, including accuracy, precision, recall, and F1-score. Using these metrics, we accurately and effectively evaluated the models' performance in detecting anomalies.

Through this evaluation and analysis, we hope to contribute to the growing research on anomaly detection in machine learning and provide valuable insights into the use of these models for real-world applications. Overall, the results we present in this chapter demonstrate the effectiveness of our chosen models and their potential to be applied for detecting anomalies in deeply nested data.

The tampered and untampered data were first evaluated separately, and to select the decision boundary, the intersection of their distributions was selected, as shown in the figure below. The selected boundary point is highlighted in blue around the score of 0.42.

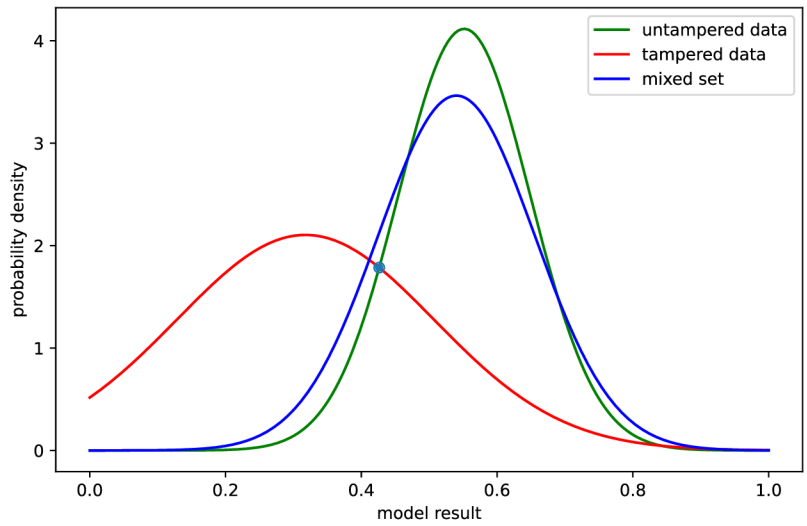

Fig. 6.1: GPT model result distributions.

Model	True		False	
	Positives	**Negatives**	**Positives**	**Negatives**
IsolationForest	130,957	4,202	5,798	62,339
LocalOutlierFactor	17,061	9,128	872	176,235
GaussianMixture	0	10,000	0	193,297
Autoencoder	0	10,000	0	193,297
GPT-2	191,839	6,990	3,182	1,286

Tab. 6.1: Confusion matrices for tested models.

The presented table reveals a significant issue with the machine learning models trained on the tabular representation of the converted data. The problem was identified as a significant limitation of encoding string data into integer labels. Specifically, the label encoder generates the same integer value for each known string while incrementing its dictionary for previously unseen strings.

6.1 Preprocessing limitations

Due to the nature of the data, each log contains slightly different information. This is the main limitation of the presented approach. The differences between the numbers are tiny, so most, if not every previously unseen log, will get classified as an anomaly because there is simply no way to decide if an item is an anomaly or just new data that is otherwise fine.

Large and small changes within the same attribute will receive the same treatment – just an incrementation of the dictionary by one. Combined with the MinMax scaler, all of these new or anomalous instances will exceed the scale present during training, making almost the entire testing dataset seemingly anomalous.

This behaviour is unacceptable as it requires engineers to manually verify the vast majority of logs flagged as anomalous, rendering the approach impractical for real-world applications. It is questionable whether the real-world performance would even amount to at least the numbers in the above table. As such, it is no better than having engineers verify every log in the first place.

6.2 GPT model performance

On the other hand, the GPT model does not exhibit this behaviour rather than relying on data of a rigid format. It learns the conventions present in the text we show it during training. This results in a significantly more robust performance. It is resistant to small changes in frequently changing areas while being sensitive to changes in areas that do not change as much.

For this reason, additional performance metrics were calculated for this model:

Metric	Value (%)
Recall	99.33
Precision	98.37
Accuracy	97.80
F1 Score	98.85

Tab. 6.2: GPT-2 Performance Metrics.

In anomaly detection, it is crucial to balance precision and recall. Precision is the proportion of correctly identified anomalies out of all instances labelled as anomalies. At the same time, recall measures the proportion of correctly identified anomalies out of all actual anomalies in the dataset. The GPT model has a high recall rate of 99.33%, indicating its ability to detect anomalies effectively. However, the precision rate of 98.37% suggests that some instances identified as anomalies may be false positives.

It is crucial to achieve a precision-recall balance depending on the specific requirements and priorities of the anomaly detection task. If missing anomalies is costly, prioritising recall becomes crucial. Conversely, emphasising precision may be more critical if false positives have severe consequences. This work sought a balance between precision and recall, which is well demonstrated by the similarity of both scores.

The *Receiver Operating Characteristic* (ROC) curve provides further insights into the model's performance. It illustrates the trade-off between the true positive rate (sensitivity) and the false positive rate (1-specificity) at various decision thresholds.

The ROC curve for the GPT model, shown in Figure 6.2, indicates an *Area Under the Curve* (AUC) score of 0.84. A higher AUC score generally indicates a better ability to discriminate between the two classes. In this case, the GPT model demonstrates a reasonably good level of discrimination, although there is still room for improvement. It implies the model can effectively rank instances and assign higher anomaly scores to true anomalies than normal ones. However, some

overlap or ambiguity may be near the decision boundary, leading to false positives or negatives.

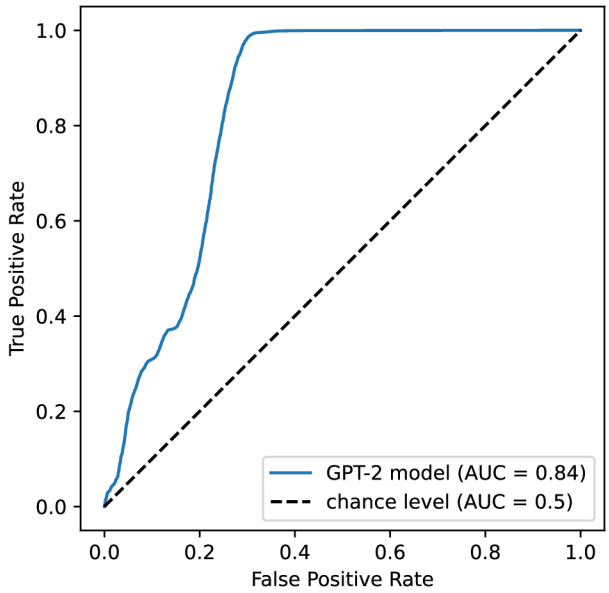

Fig. 6.2: Receiver Operating Characteristic curve for the GPT-2 model.

Conclusion

The thesis extensively explored several significant research areas, introducing cloud-native architecture, anomaly detection techniques, machine learning, and data analysis. The goal was to develop an anomaly detection model using machine learning on a data set of audit logs from Red Hat OpenShift Container Platform.

The initial chapters focused on cloud-native architecture, explaining the concept and examining virtualisation tools such as virtual machines and containers. We also discussed the DevOps approach, continuous integration, and microservices, highlighting their relevance in modern software development practices. Additionally, container orchestration, specifically the widely adopted Kubernetes framework, was thoroughly investigated, encompassing core concepts, components, and objects. Finally, Red Hat OpenShift Container Platform, an enterprise-grade solution for container orchestration, was introduced.

We then focused on anomaly detection, particularly emphasising various statistical techniques. These included the Interquartile Range method, Grubbs' test, and Gaussian Mixture Models, all playing crucial roles in detecting anomalies in diverse datasets. Moreover, time series analysis techniques were explored to capture temporal patterns and identify anomalies in sequential data.

To broaden the scope of understanding, we extensively investigated the domain of machine learning. This involved comprehensively examining performance evaluation metrics commonly employed in assessing model performance. We also explored various anomaly detection techniques, encompassing distance-based and clustering-based approaches, supervised and semi-supervised methods, and tree-based ensemble models. Additionally, we explored the domain of neural networks, covering the intricacies of neurons, layered architectures, autoencoders, and the state-of-the-art Generative Pre-trained Transformer (GPT) model.

The data analysis chapter analysed the provided dataset using various tools and techniques. Data collection, cleanup, exploration, and visualisation were performed to gain valuable insights. Additionally, innovative methods were employed to generate interpretable sentence-like representations from nested data. The findings and insights obtained from this analysis serve as a foundation for subsequent modelling and decision-making processes.

Finally, we addressed the implementation aspect of the research, incorporating essential tools such as PyTorch and Transformers. The chapter encompassed detailed discussions on tabular and sentence preprocessing techniques and the utilisation of Scikit-learn models, autoencoders, and natural language processing for practical data analysis and anomaly detection.

The last chapter presents the outcomes of the selected machine learning models

on a dataset previously unseen by the models, evaluating their anomaly detection performance. Tampered data points were introduced in-between unmodified data to simulate real-world scenarios, resulting in 203,297 data points for assessment. The evaluation employed standard metrics such as *accuracy*, *precision*, *recall*, and *F1-score*. The results reveal limitations in preprocessing nested data into a 2D format, specifically during the conversion of strings to integer labels. This results in the traditional tabular models being unable to differentiate between normal data and anomalies.

The GPT model demonstrates promising performance in anomaly detection. It exhibits robustness by effectively leveraging the conventions in the text, allowing it to handle small changes in frequently changing areas while remaining sensitive to changes in relatively stable areas.

The high recall rate of 99.33% suggests that the GPT model can identify a significant majority of anomalies in the dataset. Additionally, the precision rate of 98.37% indicates that the model produces a relatively low number of false positives.

Furthermore, the AUC score of 0.84 for the ROC curve suggests that the GPT model can effectively discriminate between anomalies and normal instances. However, there is room for improvement in this aspect.

Considering these factors, the GPT model can be regarded as a promising approach for anomaly detection, mainly when dealing with text or deeply nested and variable data. It holds promise for real-world applications in detecting anomalies within the presented datasets.

This document has contributed valuable insights to the academic community by extensively exploring these diverse research areas. The findings presented herein serve as a comprehensive reference, offering researchers, practitioners, and enthusiasts a deeper understanding of cloud-native architecture, anomaly detection techniques, machine learning, and data analysis.

The knowledge gained from this study is currently being reviewed, and a solution based on it will soon be deployed in production RHOCP clusters.

Bibliography

1. GANNON, Dennis; BARGA, Roger; SUNDARESAN, Neel. Cloud-native applications. *IEEE Cloud Computing*. 2017, vol. 4, no. 5, pp. 16–21.
2. RED HAT. *Understanding cloud-native applications*. 2022-05. Available also from: `https://www.redhat.com/en/topics/cloud-native-apps`.
3. THE KUBERNETES AUTHORS. *Kubernetes – Overview*. 2022-11. Available also from: `https://kubernetes.io/docs/concepts/overview`.
4. CAMPBELL, Sean; JERONIMO, Michael. An introduction to virtualization. *Published in "Applied Virtualization", Intel*. 2006, pp. 1–15.
5. CHIUEH, Susanta Nanda Tzi-cker; BROOK, Stony. A survey on virtualization technologies. *Rpe Report*. 2005, vol. 142.
6. IBM CLOUD EDUCATION. *Containers*. 2021-06. Available also from: `https://www.ibm.com/cloud/learn/containers`.
7. KOVÁCS, Ákos. Comparison of different Linux containers. In: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. 2017, pp. 47–51.
8. VM FARMS INC. *The complete guide to containers vs VMs for DevOps*. 2019. Available also from: `https://www.stack.io/blog/containers-vs-vms`.
9. BASS, Len; WEBER, Ingo; ZHU, Liming. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.
10. WIKIMEDIA COMMONS. *File:Devops-toolchain.svg — Wikimedia Commons, the free media repository*. 2020. Available also from: `https://commons.wikimedia.org/w/index.php?title=File:Devops-toolchain.svg&oldid=504012285`.
11. SHAHIN, Mojtaba; BABAR, Muhammad Ali; ZHU, Liming. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*. 2017, vol. 5, pp. 3909–3943.
12. DRAGONI, Nicola; GIALLORENZO, Saverio; LAFUENTE, Alberto Lluch; MAZZARA, Manuel; MONTESI, Fabrizio; MUSTAFIN, Ruslan; SAFINA, Larisa. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*. 2017, pp. 195–216.
13. RICHARDSON, Chris. *Pattern: Messaging*. 2022. Available also from: `https://microservices.io/patterns/communication-style/messaging.html`.
14. RICHARDSON, Chris. *Pattern: Microservice Architecture*. 2022. Available also from: `https://microservices.io/patterns/microservices.html`.

15. CASALICCHIO, Emiliano. Container orchestration: a survey. *Systems Modeling: Methodologies and Tools*. 2019, pp. 221–235.

16. BENTALEB, Ouafa; BELLOUM, Adam SZ; SEBAA, Abderrazak; EL-MAOUHAB, Aouaouche. Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*. 2022, vol. 78, no. 1, pp. 1144–1181.

17. BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey; EVENSON, Lachlan. *Kubernetes: up and running*. "O'Reilly Media, Inc.", 2022.

18. THE KUBERNETES AUTHORS. *Running in multiple zones*. 2022-05. Available also from: `https://kubernetes.io/docs/setup/best-practices/multiple-zones/`.

19. THE KUBERNETES AUTHORS. *Kubernetes Components*. 2022-10. Available also from: `https://kubernetes.io/docs/concepts/overview/components/`.

20. MENCHACA, Joaquín. *DevOps Concepts: Pets vs Cattle*. 2018-05. Available also from: `https://joachim8675309.medium.com/devops-concepts-pets-vs-cattle-2380b5aab313`.

21. THE KUBERNETES AUTHORS. *Volumes*. 2022-11. Available also from: `https://kubernetes.io/docs/concepts/storage/volumes/`.

22. MORRIS, Kief. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

23. THE KUBERNETES AUTHORS. *Namespaces*. 2022-11. Available also from: `https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/`.

24. THE KUBERNETES AUTHORS. *Services*. 2021-12. Available also from: `https://kubernetes.io/docs/concepts/services-networking/service/`.

25. THE KUBERNETES AUTHORS. *Ingress*. 2022-10. Available also from: `https://kubernetes.io/docs/concepts/services-networking/ingress/`.

26. THE KUBERNETES AUTHORS. *Deployments*. 2022-08. Available also from: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment`.

27. THE KUBERNETES AUTHORS. *Viewing Pods and Nodes*. 2022. Available also from: `https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/`.

28. RED HAT. *Red Hat OpenShift vs. Kubernetes*. 2022-10. Available also from: `https://www.redhat.com/en/technologies/cloud-computing/openshift/red-hat-openshift-kubernetes`.

29. RED HAT. *An overview of the architecture for OpenShift Container Platform*. 2022. Available also from: https://access.redhat.com/documentation/en-us/openshift_container_platform/4.11/html/architecture/index.

30. CURRY, Marc. *Viewing Pods and Nodes*. 2017. Available also from: https://cloud.redhat.com/blog/openshift-container-platform-reference-architecture-implementation-guides.

31. CHANDOLA, Varun; BANERJEE, Arindam; KUMAR, Vipin. Anomaly detection: A survey. *ACM computing surveys (CSUR)*. 2009, vol. 41, no. 3, pp. 1–58.

32. AL-ASLI, Mohammed; GHALEB, Taher Ahmed. Review of Signature-based Techniques in Antivirus Products. In: *2019 International Conference on Computer and Information Sciences (ICCIS)*. 2019, pp. 1–6. Available from DOI: 10.1109/ICCISci.2019.8716381.

33. JUNEJO, Khurum Nazir; GOH, Jonathan. Behaviour-based attack detection and classification in cyber physical systems using machine learning. In: *Proceedings of the 2nd ACM international workshop on cyber-physical system security*. 2016, pp. 34–43.

34. BAZRAFSHAN, Zahra; HASHEMI, Hashem; FARD, Seyed Mehdi Hazrati; HAMZEH, Ali. A survey on heuristic malware detection techniques. In: *The 5th Conference on Information and Knowledge Technology*. 2013, pp. 113–120.

35. WIKIMEDIA COMMONS. *File:Boxplot vs PDF.svg — Wikimedia Commons, the free media repository*. 2012. Available also from: https://commons.wikimedia.org/wiki/File:Boxplot_vs_PDF.svg.

36. WU, Hu-Sheng. A survey of research on anomaly detection for time series. In: *2016 13th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. 2016, pp. 426–431. Available from DOI: 10.1109/ICCWAMTIP.2016.8079887.

37. GÉRON, Aurélien. Hands-on machine learning with scikit-learn and tensorflow: Concepts. *Tools, and Techniques to build intelligent systems*. 2017.

38. MEIRA, Jorge; ANDRADE, Rui; PRAÇA, Isabel; CARNEIRO, João; BOLÓN-CANEDO, Verónica; ALONSO-BETANZOS, Amparo; MARREIROS, Goreti. Performance evaluation of unsupervised techniques in cyber-attack anomaly detection. *Journal of Ambient Intelligence and Humanized Computing*. 2020, vol. 11, pp. 4477–4489.

39. CHANDOLA, Varun; BANERJEE, Arindam; KUMAR, Vipin. Anomaly Detection: A Survey. *ACM Comput. Surv.* 2009, vol. 41, no. 3. ISSN 0360-0300. Available from DOI: 10.1145/1541880.1541882.

40. LIU, Fei Tony; TING, Kai Ming; ZHOU, Zhi-Hua. Isolation-Based Anomaly Detection. *ACM Trans. Knowl. Discov. Data.* 2012, vol. 6, no. 1. ISSN 1556-4681. Available from DOI: 10.1145/2133360.2133363.

41. SCHMIDHUBER, Jürgen. Deep learning in neural networks: An overview. *Neural networks.* 2015, vol. 61, pp. 85–117.

42. THE LINUX FOUNDATION. *Torch.nn.* Available also from: https://pytorch.org/docs/stable/nn.html.

43. YUAN, Lun-Pin; LIU, Peng; ZHU, Sencun. Recomposition vs. Prediction: A Novel Anomaly Detection for Discrete Events Based On Autoencoder. *CoRR.* 2020, vol. abs/2012.13972. Available from arXiv: 2012.13972.

44. RADFORD, Alec; WU, Jeffrey; CHILD, Rewon; LUAN, David; AMODEI, Dario; SUTSKEVER, Ilya, et al. Language models are unsupervised multitask learners. *OpenAI blog.* 2019, vol. 1, no. 8, p. 9.

45. MIELKE, Sabrina J; ALYAFEAI, Zaid; SALESKY, Elizabeth; RAFFEL, Colin; DEY, Manan; GALLÉ, Matthias; RAJA, Arun; SI, Chenglei; LEE, Wilson Y; SAGOT, Benoît, et al. Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP. *arXiv preprint arXiv:2112.10508.* 2021.

46. GILLIOZ, Anthony; CASAS, Jacky; MUGELLINI, Elena; ABOU KHALED, Omar. Overview of the Transformer-based Models for NLP Tasks. In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS).* 2020, pp. 179–183.

47. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N; KAISER, Łukasz; POLOSUKHIN, Illia. Attention is All you Need. In: GUYON, I.; LUXBURG, U. Von; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems.* Curran Associates, Inc., 2017, vol. 30. Available also from: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

48. BLOICE, Marcus D; HOLZINGER, Andreas. A tutorial on machine learning and data science tools with python. *Machine Learning for Health Informatics.* 2016, pp. 435–480.

49. THE PANDAS DEVELOPMENT TEAM. *pandas-dev/pandas: Pandas*. Zenodo, 2023. Version v2.0.1. Available from DOI: 10.5281/zenodo.7857418.

50. BRUGMAN, Simon. *ydata-profiling: Exploratory Data Analysis for Python*. 2019. Available also from: https://github.com/ydataai/ydata-profiling.

51. HUNTER, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*. 2007, vol. 9, no. 3, pp. 90–95. Available from DOI: 10.1109/MCSE.2007.55.

52. WASKOM, Michael L. seaborn: statistical data visualization. *Journal of Open Source Software*. 2021, vol. 6, no. 60, p. 3021. Available from DOI: 10.21105/joss.03021.

53. PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

54. PASZKE, Adam; GROSS, Sam; MASSA, Francisco; LERER, Adam; BRADBURY, James; CHANAN, Gregory; KILLEEN, Trevor; LIN, Zeming; GIMELSHEIN, Natalia; ANTIGA, Luca; DESMAISON, Alban; KÖPF, Andreas; YANG, Edward Z.; DEVITO, Zach; RAISON, Martin; TEJANI, Alykhan; CHILAMKURTHY, Sasank; STEINER, Benoit; FANG, Lu; BAI, Junjie; CHINTALA, Soumith. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR*. 2019, vol. abs/1912.01703. Available from arXiv: 1912.01703.

55. WOLF, Thomas; DEBUT, Lysandre; SANH, Victor; CHAUMOND, Julien; DELANGUE, Clement; MOI, Anthony; CISTAC, Pierric; RAULT, Tim; LOUF, Rémi; FUNTOWICZ, Morgan; DAVISON, Joe; SHLEIFER, Sam; PLATEN, Patrick von; MA, Clara; JERNITE, Yacine; PLU, Julien; XU, Canwen; SCAO, Teven Le; GUGGER, Sylvain; DRAME, Mariama; LHOEST, Quentin; RUSH, Alexander M. Transformers: State-of-the-Art Natural Language Processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, 2020, pp. 38–45. Available also from: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

56. SANH, Victor; DEBUT, Lysandre; CHAUMOND, Julien; WOLF, Thomas. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In: *NeurIPS EMC2̂ Workshop*. 2019.

Symbols and abbreviations

API	Application Programming Interface
AUC	Area Under the Curve
CD	Continuous Delivery
CI	Continuous Integration
CNN	Convolutional Neural Networks
FNN	Feedforward Neural Networks
GPT	Generative Pre-trained Transformer
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a service
IaC	Infrastructure as code
JIT	just-in-time
JSON	JavaScript Object Notation
k-NN	K-Nearest Neighbours
KVM	Kernel-based Virtual Machine
LOF	Local Outlier Factor
LRD	Local Reachability Density
LSTM	Long Short-Term Memory
NLP	Natural Language Processing
OS	Operating System
PaaS	Platform as a service
RHCOS	Red Hat Enterprise Linux CoreOS
RHEL	Red Hat Enterprise Linux
RHOCP	Red Hat OpenShift Container Platform

RHODS	Red Hat OpenShift Data Science
RNN	Recurrent Neural Networks
ROC	Receiver Operating Characteristic
SVM	Support Vector Machines
TLS	Transport Layer Security
VM	Virtual Machine

List of appendices

A Attached media

Attached to this document is an archive containing the source code. No data or trained models could be included for the reasons mentioned in 4.1

```
├── autoencoder/ ..... autoencoder source code
│   ├── base/ ..... abstract base classes
│   │   └── ...
│   ├── configs/ ..... configuration settings
│   │   └── autoencoder.json ..... autoencoder config
│   ├── data_loader/ ..... data loading
│   │   └── data_loaders.py
│   ├── logger/ ..... tensorboard visualization and logging
│   │   ├── logger.py
│   │   ├── logger_config.json
│   │   └── visualization.py
│   ├── model/ ..... models, losses, and metrics
│   │   ├── loss.py
│   │   ├── metric.py
│   │   └── model.py
│   ├── trainer/ ..... trainers
│   │   └── trainer.py
│   ├── utils/ ..... small utility functions
│   │   └── util.py
│   ├── test.py ..... main training script
│   ├── train.py ..... main testing script
│   ├── parse_config.py ..... handling config files and cli options
│   └── results.py ..... results visualization
├── data/ ..... data directory
├── gpt/ ..... gpt source code
│   ├── gpt.py ..... log evaluation script
│   └── results.py ..... results visualization
├── scripts/ ..... various scripts
│   ├── analysis.py ..... data analysis script
│   ├── preprocessing.py ..... preprocessing script
│   └── sklearn-methods.py ..... sklearn anomaly detection
├── thesis/ ..... source code for this document
│   └── ...
├── README.md ..... introduction to the project
├── requirements-dev.txt ..... development requirements
└── requirements.txt ..... execution requirements
```

B Correlation matrices

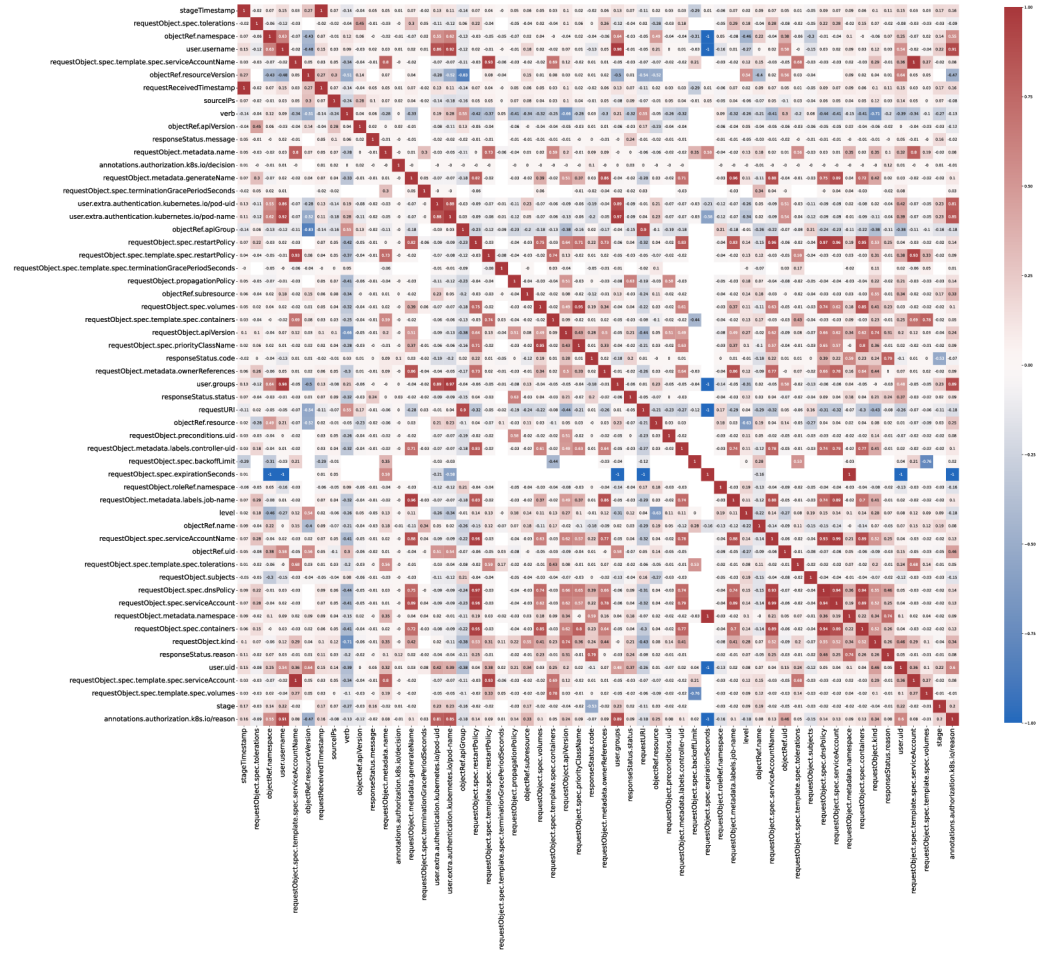

Fig. B.1: Correlation matrix of the pruned data.

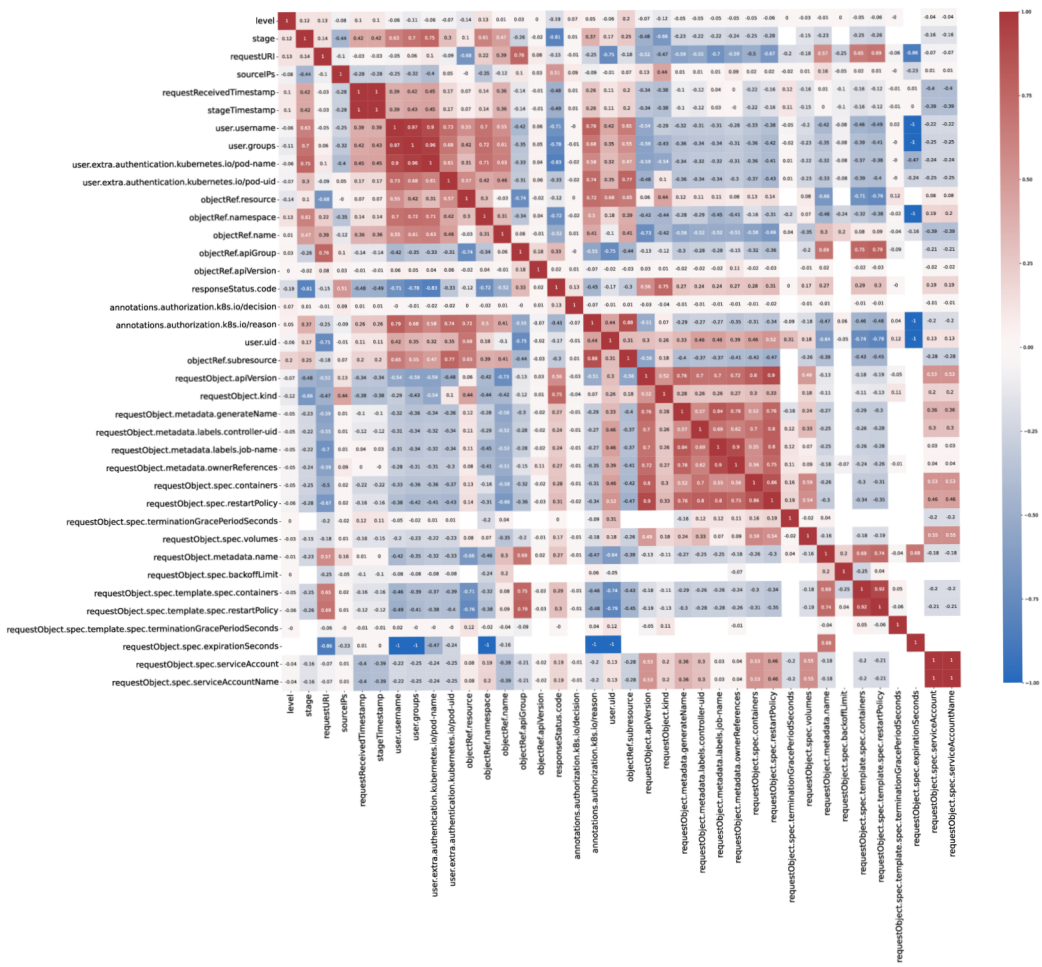

Fig. B.2: Correlation matrix of the subset with the *create* verb.

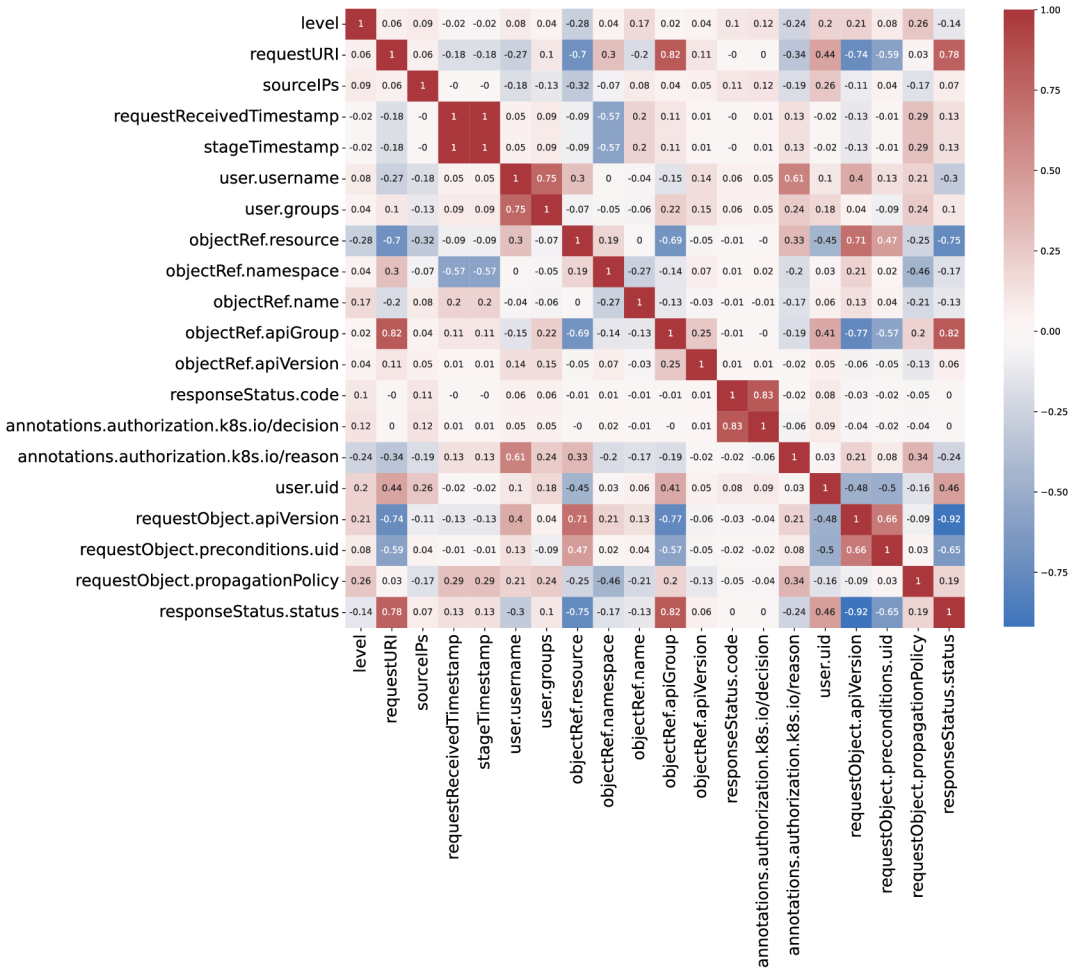

Fig. B.3: Correlation matrix of the subset with the *delete* verb.

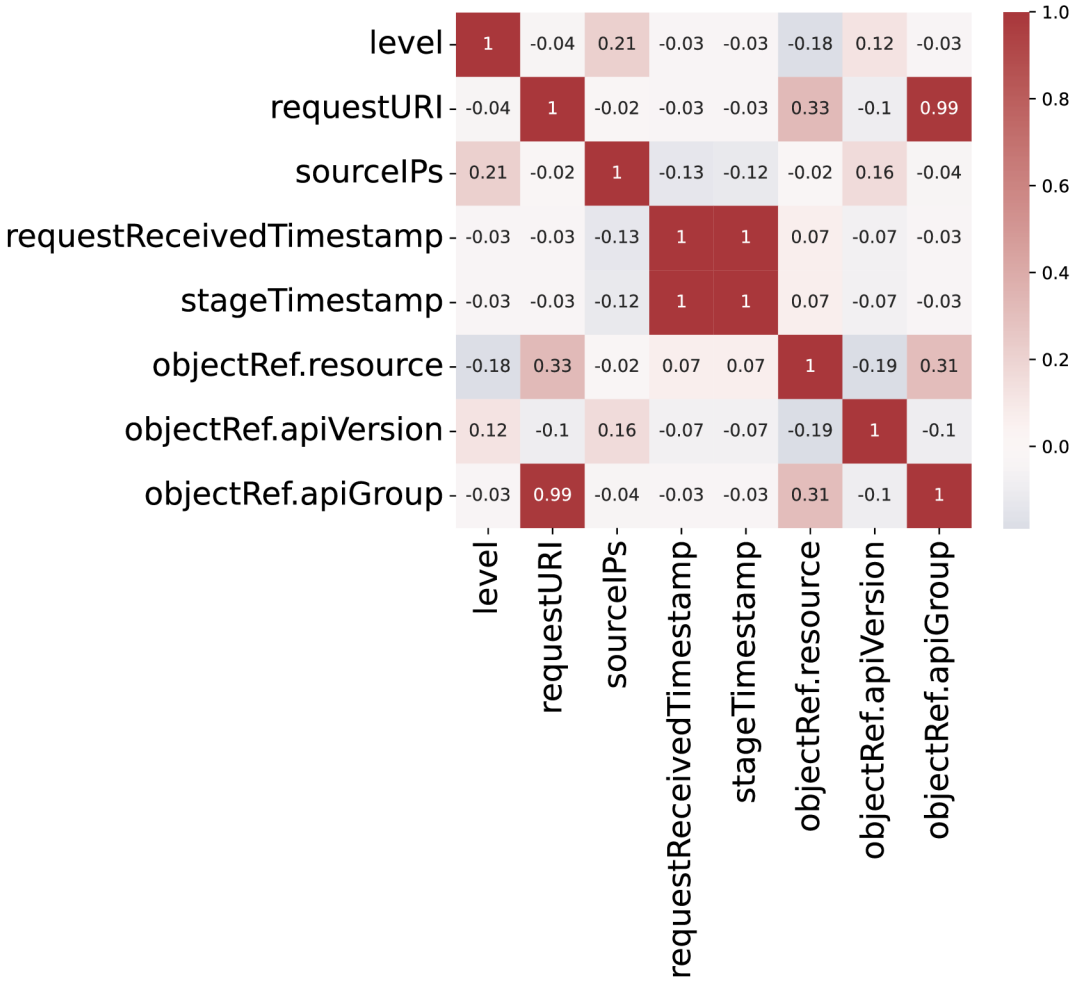

Fig. B.4: Correlation matrix of the subset with the *deletecollection* verb.

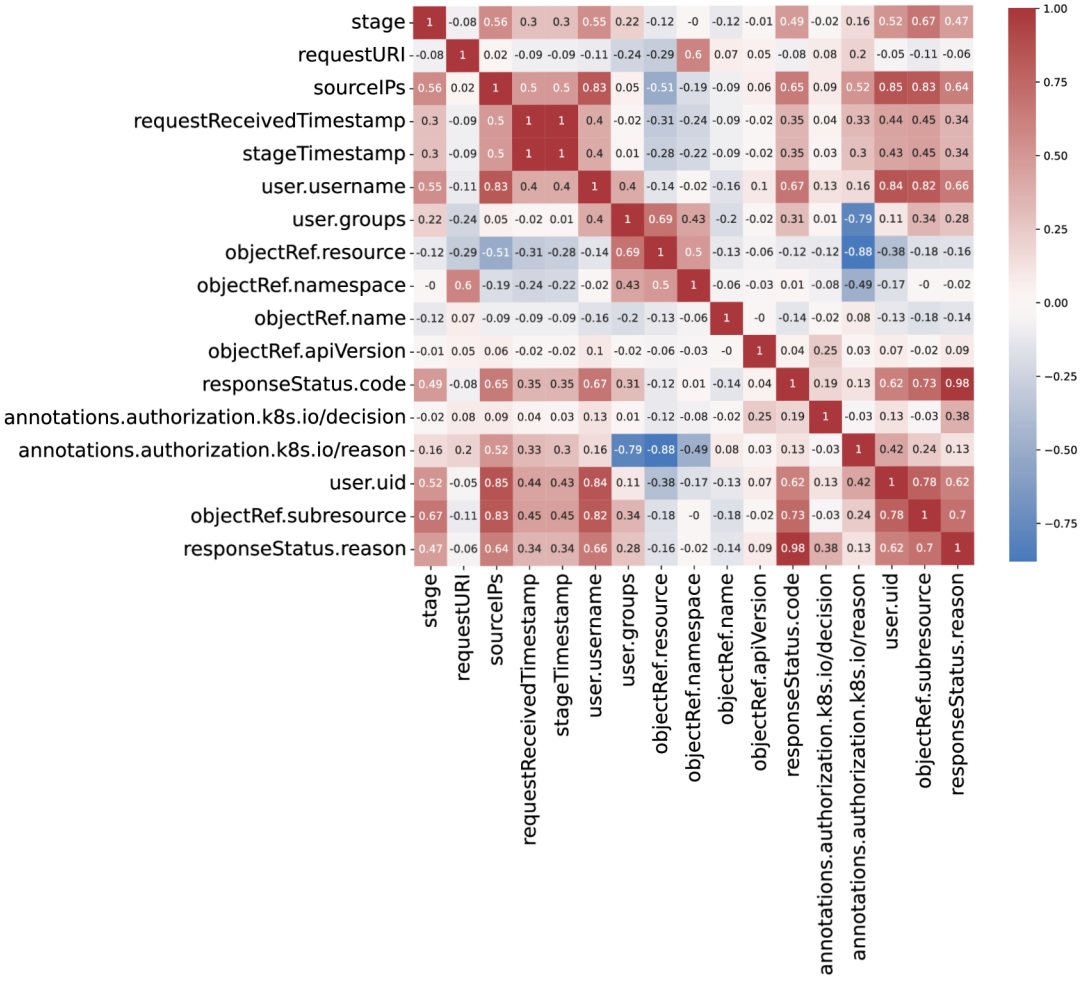

Fig. B.5: Correlation matrix of the subset with the *get* verb.

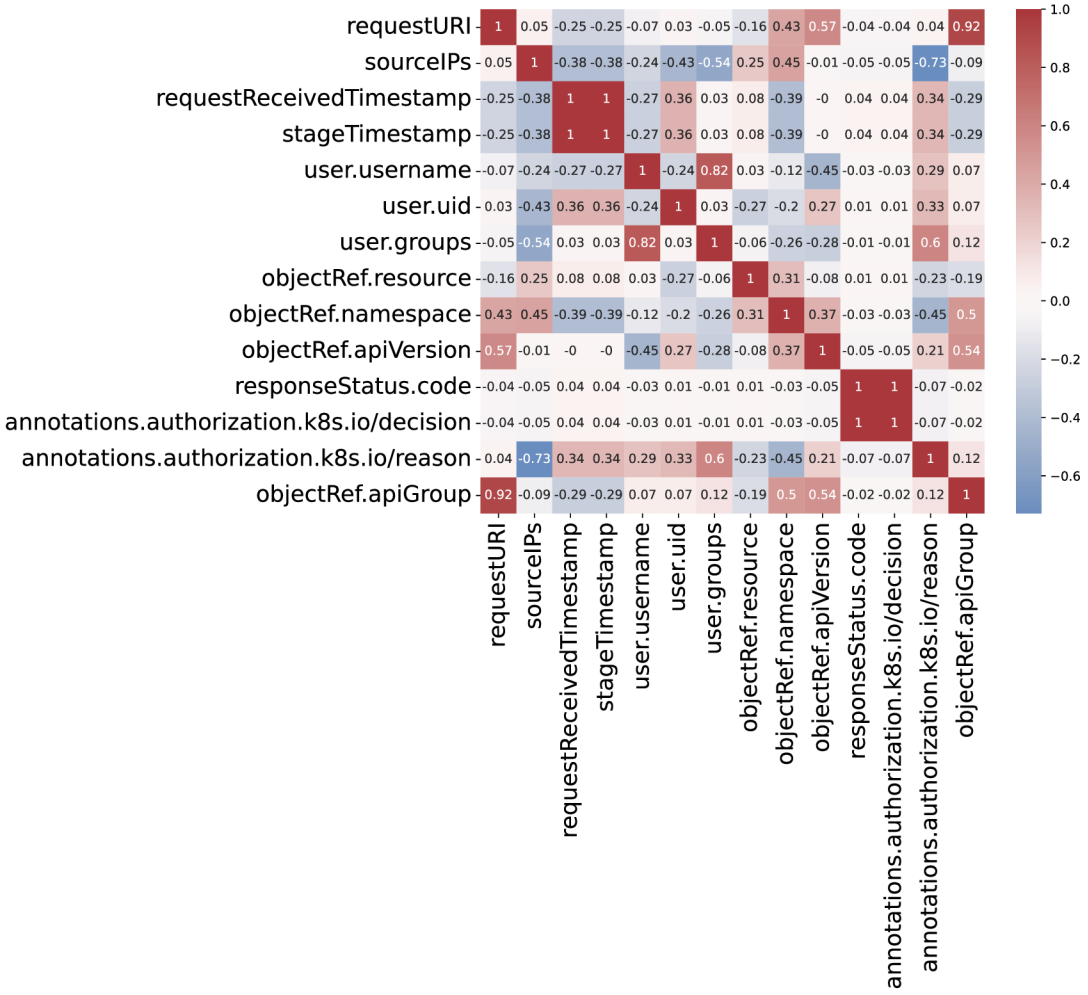

Fig. B.6: Correlation matrix of the subset with the *list* verb.

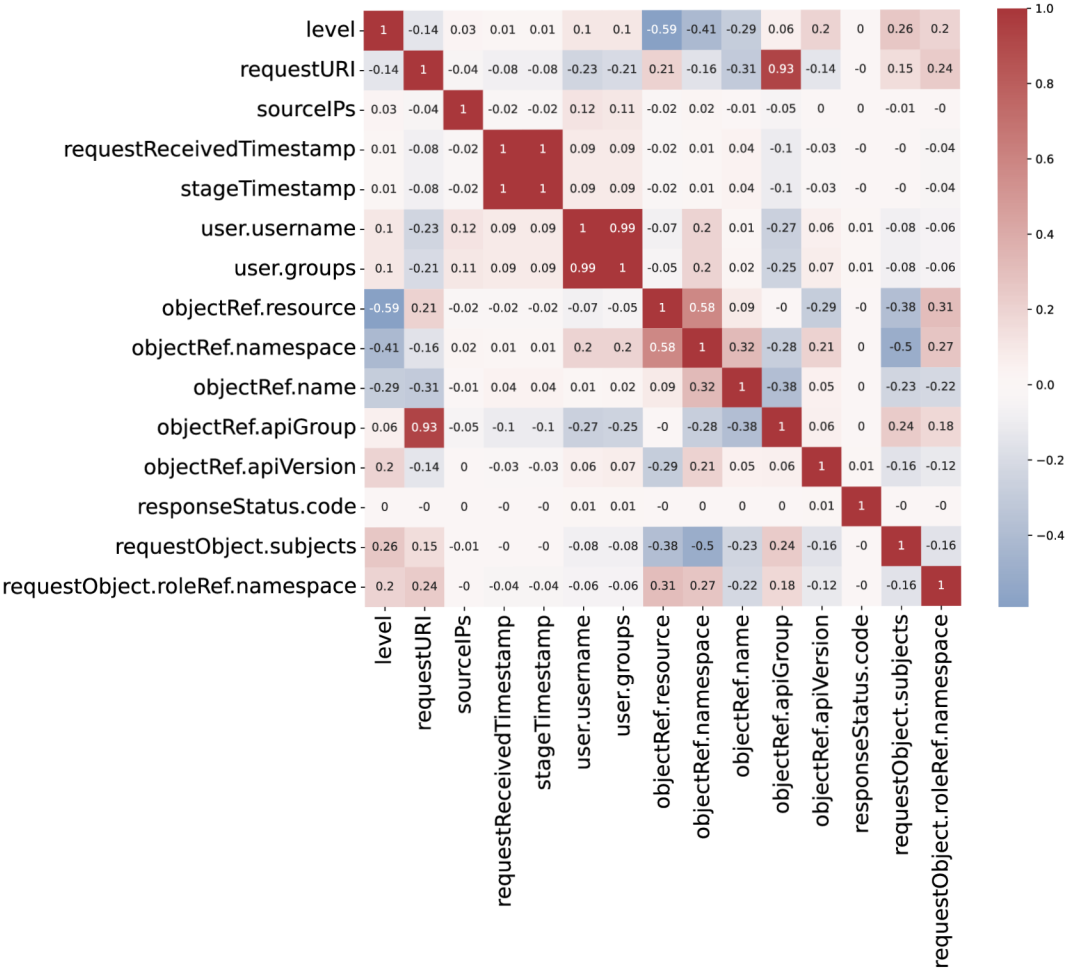

Fig. B.7: Correlation matrix of the subset with the *patch* verb.

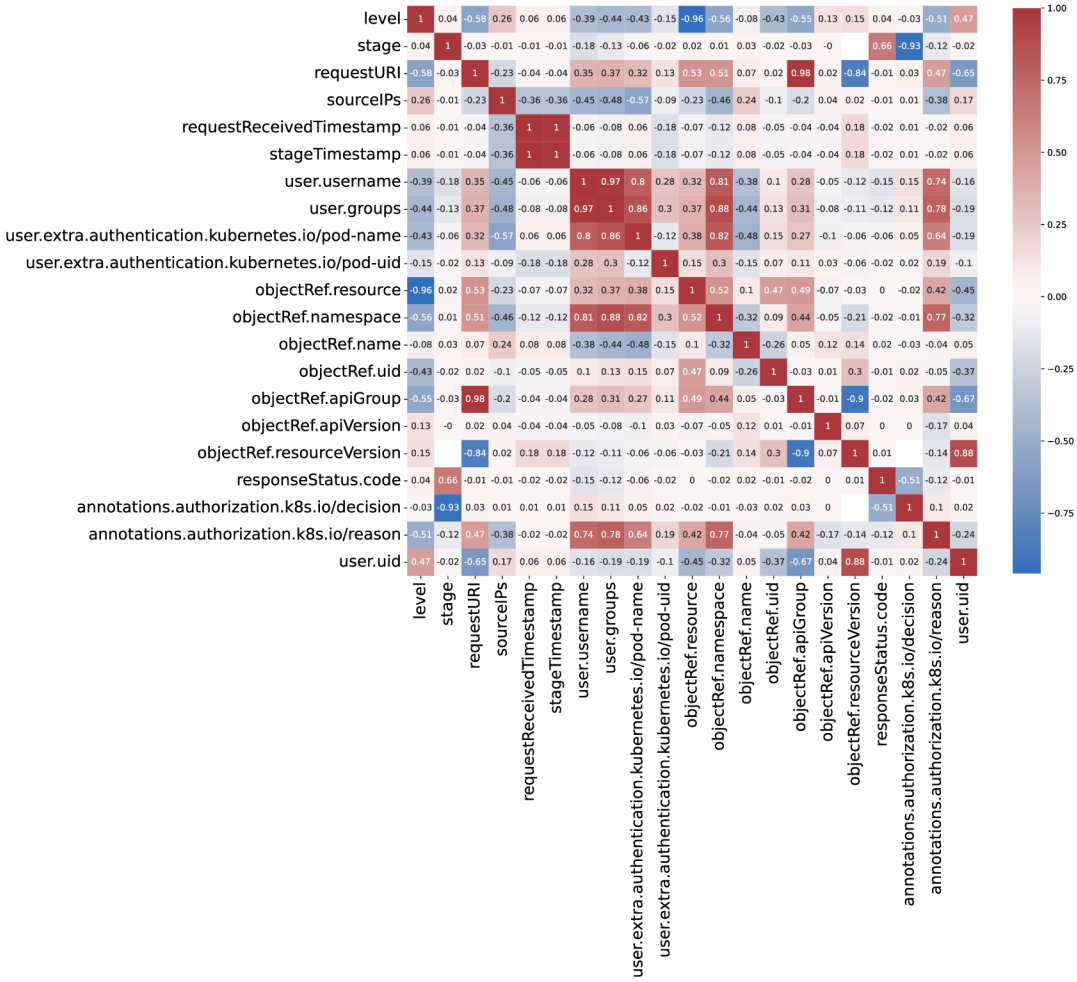

Fig. B.8: Correlation matrix of the subset with the *update* verb.

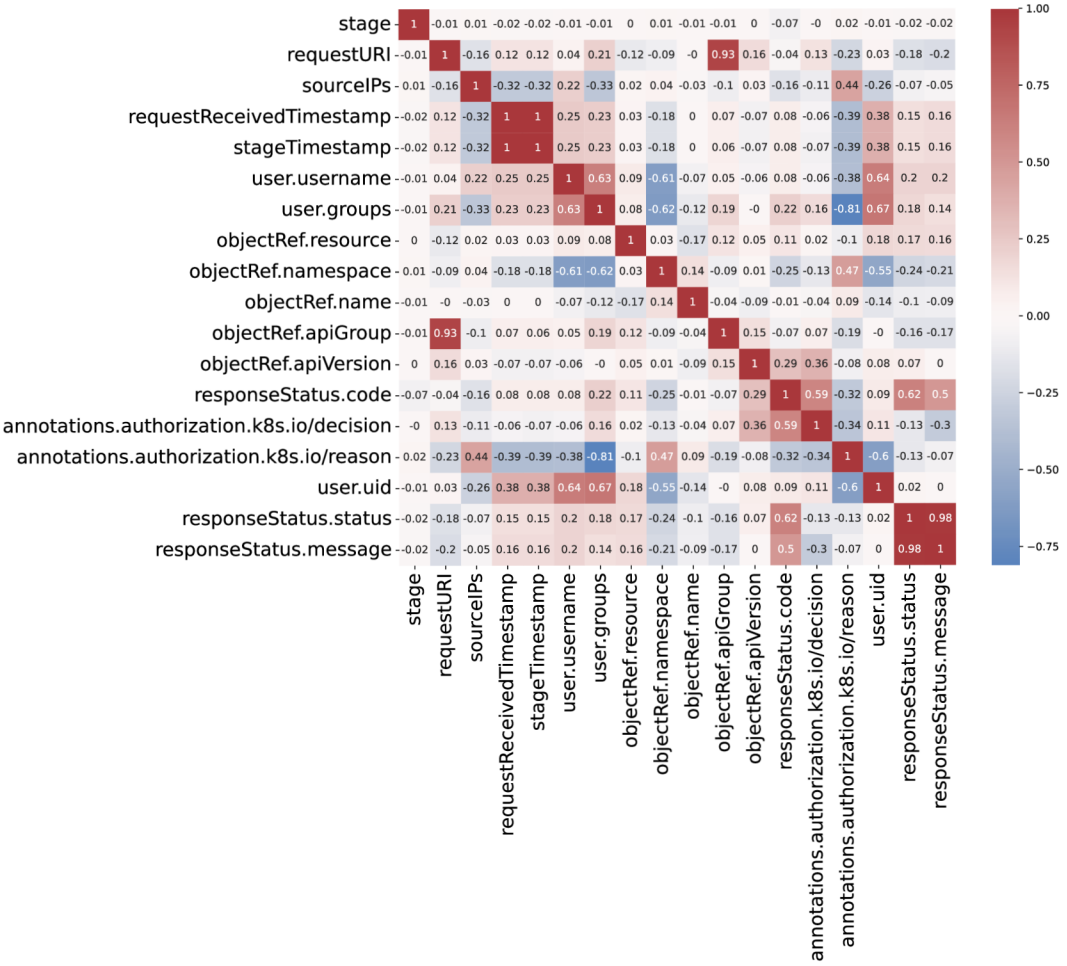

Fig. B.9: Correlation matrix of the subset with the *watch* verb.

C Source Code Listings

```python
def label_scale(files, group: str):
    le = preprocessing.LabelEncoder()
    scaler = preprocessing.MinMaxScaler()
    training_files = [f for f in files if "train" in Path(f).stem]
    testing_files = [f for f in files if f not in training_files]
    ordered_files = training_files + testing_files
    for file in ordered_files:
        df = pd.read_csv(file, low_memory=False)
        for idx, column in enumerate(df.columns):
            if str(df[column].dtype) == "object":
                df[column].fillna("0", inplace=True)
                df[column] = le.fit_transform(df[column])
            else:
                df[column].fillna(0, inplace=True)
        elif "train" in file:
            minmaxed = scaler.fit_transform(df)
        else:
            minmaxed = scaler.transform(df)
        minmax_df = pd.DataFrame(minmaxed, columns=df.columns)
        """data is saved and memory is freed."""
        if "train" in file:
            np.save(f"classes_{group}.npy", le.classes_)
            joblib.dump(scaler, f"scaler_{group}.gz")
```

Listing C.1: Labelling and scaling data.

```python
class KubeDataLoader(BaseDataLoader):
    def __init__(
        self,
        data_dir: str,
        batch_size: int,
        shuffle: bool = True,
        validation_split: float = 0.0,
        num_workers: int = 1,
        training: bool = True,
        train_file: str = "train.csv",
        test_file: str = "",
    ):
        self.data_dir = data_dir
        self.training = training
        self.train_file = train_file
        self.test_file = test_file
        self.dataset = self.load_datasets()
        super().__init__(
            self.dataset,
            batch_size,
            shuffle,
            validation_split,
            num_workers,
        )

    def load_datasets(self) -> List[Tuple[torch.FloatTensor, torch.FloatTensor]]:
        if self.training:
            lis = pd.read_csv(
                os.path.join(self.data_dir, self.train_file)
            ).values.tolist()
        else:
            lis = pd.read_csv(
                os.path.join(self.data_dir, self.test_file)
            ).values.tolist()

        tensors = [torch.FloatTensor(sublist) for sublist in lis]
        del lis

        result = list(zip(tensors, tensors))
        return result
```

Listing C.2: KubeDataLoader Class.