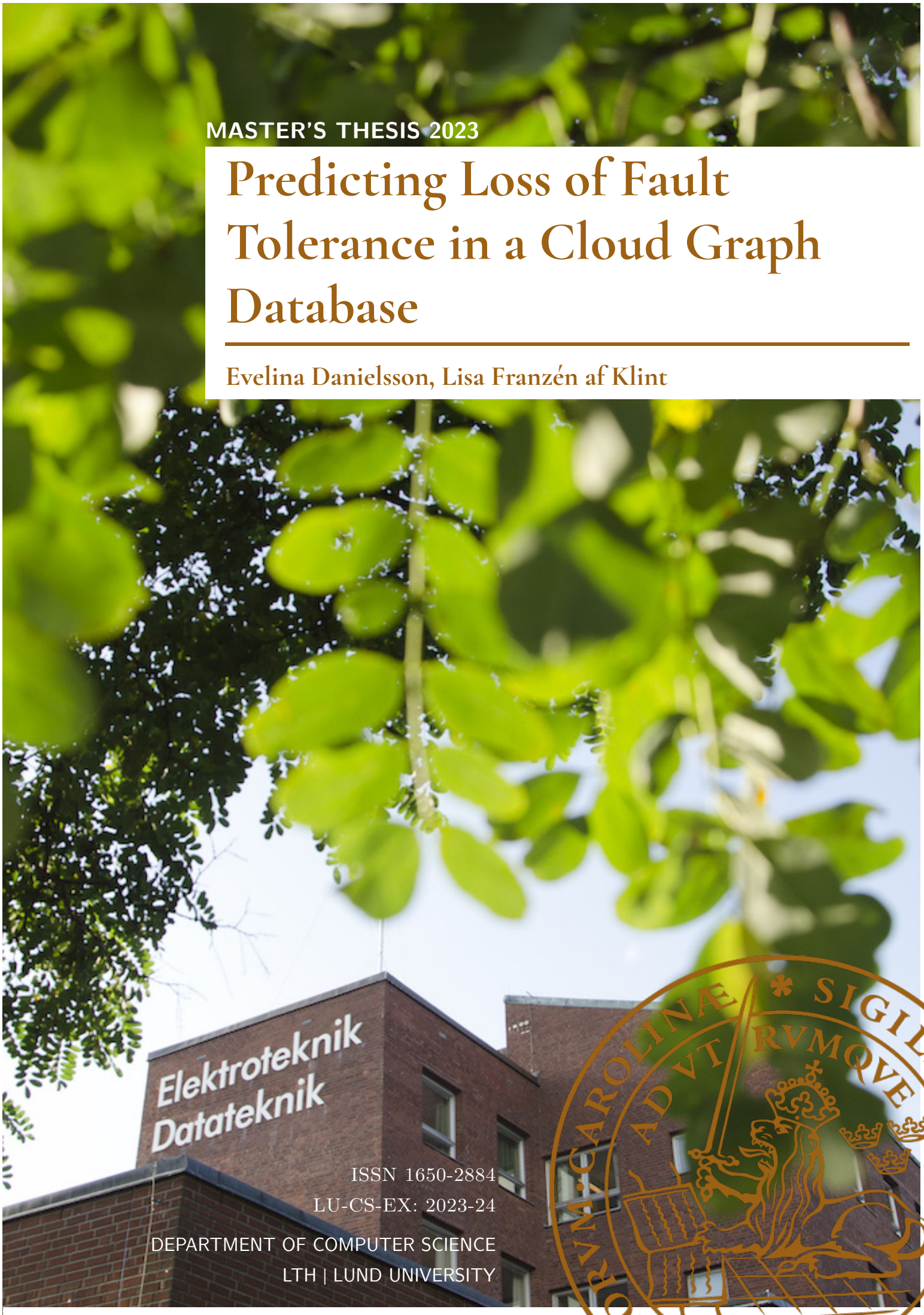


MASTER'S THESIS 2023

Predicting Loss of Fault Tolerance in a Cloud Graph Database

Evelina Danielsson, Lisa Franzén af Klint

ISSN 1650-2884
LU-CS-EX: 2023-24

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-24

Predicting Loss of Fault Tolerance in a Cloud Graph Database

Förutsäga förlust av feltolerans i en molntjänst

Evelina Danielsson, Lisa Franzén af Klint

Predicting Loss of Fault Tolerance in a Cloud Graph Database

Evelina Danielsson
muh15eda@student.lu.se

Lisa Franzén af Klint
li6285fr-s@student.lu.se

June 27, 2023

Master's thesis work carried out at Neo4j.

Supervisors: Olof Englund, olof.englund@neotechnology.com
Johan Eker, johan.eker@ericsson.com

Examiner: Markus Borg, markus.borg@cs.lth.se

Abstract

With the development of cloud computing it becomes increasingly popular with applications which are hosted on the cloud and used over the internet. In order to keep the system operational and prevent loss of data in case of failure, many systems adapt fault tolerance. Fault tolerance is defined as a system's ability to continue operating without loss of functionality when one or more of its components fail. Therefore, it becomes essential to be able to detect and predict when a system is at risk of losing fault tolerance. Every anomalous behaviour in a system is a potential cause to an incident that can lead to the system losing this quality. By detecting anomalies that can contribute to fault intolerance, this can be prevented.

In this thesis the authors have researched the area of conducting efficient anomaly detection using metrics that monitor a cloud database service in production. The authors used two different approaches to determine which metrics could be of interest. On the chosen metrics, two different machine learning models were implemented and evaluated on its success of identifying anomalies correlating to fault intolerance.

The authors found that the error of the connection rate to HAProxy was the only metric that exhibited a reasonable correlation, and neither model proved effective in predicting when a database would be at risk of becoming fault intolerant. The models used were the statistical method HBOS and a neural network LSTM-based autoencoder, where HBOS performed marginally better.

Keywords: Fault tolerance, anomaly detection, metrics, machine learning

Acknowledgements

We would like to thank our supervisor at LTH, Johan Eker, for all valuable input and discussions during this project. Additionally, we want to thank Markus Borg for agreeing to be our examiner of this thesis.

A big thank you to the wonderful people at Neo4j, especially Olof Englund, we would have been truly lost without you.

Contents

Chapter 1
Introduction

1.1 Background

Cloud computing has with its on-demand and easy access via the internet grown in popularity among businesses over the past years. The advancement in cloud technology has in turn paved the way for SaaS, *software as a service*, a delivery model where the application is hosted on the cloud and used over the internet. Today, SaaS has become the most common service hosted on the public cloud and the dominant software delivery model. [29]

One of the companies that has adapted this model, is the graph database provider Neo4j. In November 2019, Neo4j Inc. announced the first fully managed graph Database as a Service, *Neo4j Aura*. This new system requires a complex infrastructure consisting of almost 50 micro-services taking care of everything from incoming web requests to monitoring of the database. As customers rely on the database staying operational it becomes essential to be able to predict when the databases are at risk of failure.

To understand what this entails, it is important to understand the distinction between fault, failure and error. A fault can be described as an unusual hardware or software condition. There are many causes that can result in a fault, for example network overflow, processor or memory damage, as well as mistakes made in the software specification or implementation. A fault can further be considered active if it produces an error in the system, and inactive if not. This means that if we have a fault it is not certain it will lead to an error in the system, the fault may be out of the boundaries of the system functionality. If an error is produced as a result of an active fault, it will cause a deviation from the expected result in the system which in turn results in a failure. [47]

To reduce the risk of failure it has become common for systems to adapt fault tolerance. Fault tolerance is defined as a system's ability to continue operating without loss of function-

ality when one or more of its components fail. In a fault tolerant system the use of backup components ensures that in case of component failure, the backup component can take its place and therefore endure no loss of service. Component backup can come in many different forms, the most common are: *multiple hardware systems*, where for example two databases can be located on two different servers; *multiple instances of software* where traffic can be rerouted to another instance if one instance encounters an error or goes offline; and finally *backup sources of power*, such as generators. [17]

Every anomalous behaviour in a system is a potential cause to an incident that can lead to a failure and compromise the quality of the service. [39]. Due to the complexity of large-scale applications in cloud environments, these systems often experience a high percentage of operational failures. The application constantly becomes subjected to regular changes as a result of sporadic operations, such as on-demand scaling, upgrade, migration and reconfiguration. In order to detect anomalies systems are often monitored by metrics.

To oversee the state of the system, metrics can be used to monitor different processes such as memory usage and CPU usage. When monitoring the system based on metrics, the issue becomes that the system operators have to track multiple monitoring metrics and therefore receive too much monitoring information. This results in many false warnings and alerts which distracts the system operators from critical abnormal situations in the system. In the worst case, this has caused system operators to disable system monitoring. The excessive amount of metric information also complicates as well as delays the process of detecting anomalies, making the operator unaware of an issue at critical times in the system. Therefore, it becomes necessary to select a subset of monitoring metrics that is relevant for a distinct monitoring requirement. Today, this is done mainly manually by system operators, based on their knowledge of the domain. It is not only time consuming, but also becoming more and more difficult as the number of resource type cloud environments are ever growing as well as the variety of associated metrics. [18] It therefore becomes essential to be able to efficiently identify relevant metrics that can be used to detect anomalies in a system.

1.2 Project Objective

The aim of this thesis is to examine whether anomaly detection using monitoring metrics can be employed to alert when a database cluster is predicted to lose fault tolerance. The objective of this thesis is to collect metric data, identify relevant metrics in regard to fault intolerance as well as exploring, evaluating, and comparing different anomaly detection models. The evaluation will be conducted with regard to evaluation metrics such as precision, recall and F1-score.

1.3 Research Questions

From the problems clearly formulated we can establish motivated research questions. The data that is going to be used is metric data collected from the monitored system, Neo4j AuraDB. Therefore, our research questions aim to address both the challenges of data selection, as well as the anomaly detection problem. The research questions are summarised as follows:

- **RQ1:** Can we identify metrics that correlate with the event of a database becoming fault intolerant.
- **RQ2:** Based on the metrics identified in RQ1, can we determine, through anomaly detection, when a database is at risk of becoming fault intolerant?

1.3.1 Our Contribution

This thesis addresses the gap in knowledge regarding how anomaly detection with high-dimensional big data can be applied in a complex distributed system. Specifically, it evaluates diverse approaches to evaluating metrics as well as various models for anomaly detection. Finally, it proposes a novel method to generate a HBOS model that incorporates temporal information.

1.4 Individual Contributions

This thesis was done with a high amount of collaboration between the authors. While many steps were implemented together, Evelina had more responsibility of data collection and the LSTM based autoencoder, while Lisa took more responsibility of data preprocessing, SMA ,and EMA. HBOS and evaluation of the model was done in collaboration between the authors. Additionally, the authors contributed equally in writing the thesis.

1.5 Thesis Outline

This thesis will begin with an introduction where the background of the problem is presented along with the project's objectives. Chapter 2 will define cloud computing along with containerisation and how such an application can be managed with Kubernetes. This information is necessary to understand the system in which the experiments of this thesis were conducted. Chapter 2 continues to explain the system and how fault tolerance is reflected in it. The thesis moves on to theory regarding anomaly detection and AI in Chapter 3. The theoretical background to anomaly detection is described along with the models that will be used in the next part of the thesis. Chapter 4 aspires to disclose the progress of the thesis including data collection, data preprocessing, metric selection, applying anomaly detection, and how the results were evaluated. The results of the experiments are found in Chapter 5 and is followed by a discussion of said result in Chapter 6. The thesis is concluded with the conclusions that could be drawn in Chapter 7.

1.6 Related Work

Anomaly detection has been used for decades across various domains, and cloud computing is no exception for this type of application. There has been a lot of research done regarding anomaly detection techniques in cloud computing systems. Moreover, the complexity of cloud systems as well as its immense monitoring tools results in high-dimensional big data. Therefore, being able to apply anomaly detection on high-dimensional data is becoming crucial. In this section we will present related work on both evaluating anomaly techniques and on how to handle high-dimensional data.

Many studies have been made in order to compare different anomaly detection algorithms. The authors in [23] conducted a comparative evaluation of 10 different algorithms for anomaly detection. The algorithms were evaluated both in regard to speed and accuracy. The algorithms analysed could be grouped into clustering based algorithms, nearest-neighbour algorithms and Histogram-based Outlier Score (HBOS). In HBOS an univariate histogram is constructed displaying the frequency distribution of each feature. This method will be implemented and used in this thesis, and will therefore be described in detail in Section 3.3.3. The authors found HBOS to be up to 5 times faster than clustering based algorithms and up to 7 times faster than nearest-neighbour algorithms. Additionally, they concluded that HBOS performed as reliable as state-of-the-art algorithms when detecting global anomalies, i.e. anomalies that fall outside the normal range for the entirety of the data set, but unsatisfactory on anomalies that diverge from the normal range of the surrounding data set.

In addition, the authors in [16] conducted a survey of existing anomaly detection techniques. It divides the techniques into different categories based on the underlying approach of the technique and identifies each category's advantages and disadvantages. The categories analysed are defined as: classification based, nearest neighbour, clustering based, statistical, information theoretic and spectral. In this thesis the techniques implemented belong to classification based anomaly detection and statistical anomaly detection, and can be read in further detail in Sections 3.3.1 3.3.3.

In [48] the authors outline classical methods of anomaly detection such as clustering, isolation forest or using statistical anomaly detection with gaussian distribution. In these methods the anomalies are identified based on the value of a single data point, the previous points are not taken into account. Therefore, these algorithms are often unsuited for detecting anomalies in time-series data. Instead, they propose the use of LSTM-Based autoencoders. Autoencoders are made up of two modules: encoder and decoder. While the encoder learns the underlying features the decoder recreate the original data based on the features. In a LSTM-based autoencoder, the autoencoder is built on LSTM layers. This theory is further explained in this thesis in Section 3.3.2.

Another paper that employed this approach is the work by the authors in [44], where they constructed an LSTM-based autoencoder to detect anomalies in network data. By training their model only on normal data, they could model normal data. In case of an anomaly, the reconstruction error, the difference between the input of the encoder and output of the decoder, would be greater. What is unique in this paper was that the authors found that if

the normal and anomalous data was too similar the difference in reconstruction error could be too small and result in miss-classifications. In order to improve the models detection rate, they trained the data from the encoder on the one-class SVM algorithm to improve the anomaly classification. They found that their proposed model efficiently could detect anomalies presented in the data.

The authors in [21] argue that continuous monitoring of a cloud system leads to an overwhelming volume of data, and together with a large number of performance metrics, results in an extremely complex data model. They further state that because of this, the selection of appropriate performance metrics is crucial for the effectiveness of anomaly detection systems. The unique challenges brought by both high dimensionality and big data problems are identified and discussed in [52]. The authors define and discuss the limitations of both traditional approaches as well as the state-of-the-art when working with high-dimensional and big data.

Traditionally, euclidean distance is used in anomaly detection methods in order to estimate the similarity within the data set as it is considered the most common distance metric. However, this is based on low-dimensional data sets. In [49] the authors discuss how similarity calculated with euclidean distance would perform if used for high-dimensional data sets. The authors found that the Euclidean distance between two similar data observations and the distance between two dissimilar data observations in multiple dimensions can be almost equal. Therefore, the authors concluded that euclidean distance was unsuited to find similarity in high-dimensional data.

Different state-of-the-art methods for feature selection are addressed in [54]. The methods suggested are divided into filter methods, wrapper methods and embedded methods. The filter method is based on ranking the features according to some criteria in order to determine the most important features, this is done independently of the target variable. Examples of filter methods that handles the features independently include gain ratio [45] and chi-square [37] while minimum redundancy maximum relevance (mRMR) [43] is used as an example where the interactions among features are taken into account. While, filter methods perform well on smaller sets of features it does not scale well when the number of features exceed tens of thousands. The authors additionally propose wrapper and embedded methods when working on large feature sizes, i.e. exceeding tens of thousands. They concluded that feature selection implemented with *support vector machine* (SVM) works well on high-dimensional classification problems. Hermes and Buhmann [28] found that with applied SVM they could reduce features without significant loss in classification accuracy.

Further, the authors in [18] propose a method for improving anomaly detection in cloud systems using log and metric correlation analysis. The authors argue that existing anomaly detection methods based solely on metrics are not sufficient for detecting complex issues in cloud systems, and that analysing the correlation between logs and metrics can provide additional insight into system behaviour. The proposed method involves selecting relevant metrics and logs, computing their correlation through regression analysis, and using machine learning techniques to identify anomalies based on the relevant metrics and logs. The authors suggest that their method could be used to improve cloud system operations and re-

duce downtime.

In the final paper studied [21], the authors argues that with the use of mutual information (MI), a method to measure the importance and redundancy of performance metrics, an incremental search algorithm is employed for selecting the most relevant metrics, and principal component analysis (PCA) is used to reduce the metric dimension. The authors also propose a semi-supervised decision tree classifier to detect anomalies by exploring the most important metrics and labelled data. The author found that their proposed methods can reduce metric dimensions and identify anomalies effectively.

Chapter 2
Cloud Systems

This chapter will start by presenting all related theory of cloud computing and how Kubernetes help manage containerised applications. After this, graph databases will be explained before moving on to the system on which the experiments are conducted. The system will be explained in detail, both on a high and on a low level. Further, fault tolerance and common causes that lead to loss of fault tolerance will be defined and explained. Finally, the characteristics of the actual data from the system and how it can be collected will be disclosed.

2.1 Cloud Computing

The term *cloud computing* comes from its ability to access information remotely in the cloud or a virtual space. It enables the users to store information such as files and applications on remote servers and access it regardless of geographical position. The term refers to both the services delivered through the Internet as well as the hardware and system software in the data centres that enable these services. [20] Examples of these services include email, backup, and data retrieval. In contrast, when talking about a *cloud*, it is only the data centre's hardware and software that we are referring to [8].

The advancement in cloud technology has in turn paved the way for *software as a service* (SaaS), a delivery model where the application software is hosted on the cloud. The user can access and integrate with the software through a web browser, desktop client or API. The service can be hosted on its own infrastructure or, the more common option, with a cloud service provider such as amazon web services, google cloud or microsoft azure. Today, SaaS has become the most common service hosted on the public cloud and the dominant software delivery model. [29, 1]

The benefits and advantages that SaaS offers to the customers are many, one of the most important being the low-to-non management overhead. It is the SaaS service provider that is responsible for managing and maintaining the service to meet the performance, availability, and data protection standards specified in the service level agreement (SLA). SaaS applications are also protected from data loss as the application data is stored in the cloud, meaning the data will not be lost if the user's device crashes or breaks. [29].

SaaS has several categories in turn, one of them being *Database-as-a-Service (DBaaS)*. This cloud computing service offers a cloud database system, saving the user from purchasing and setting up their own hardware, installing the database software or manage the database themselves.[2] As a SaaS service the provider has the same managing and maintaining responsibility, DBaaS is quickly gaining popularity and is being adapted by companies. [4]

A SaaS solution can adopt different cloud architectures called single-tenant and multi-tenant. In a single-tenant cloud architecture a single instance of the software and its supporting infrastructure provide for a single customer. This means that each customer has its own instance of the software, no sharing of software or resources happen between customers. In a multi-tenant cloud architecture the instance of software and supporting infrastructure is instead used by several customers. Here, the customers share the software, although each customer's data is isolated and remains hidden from the other customers. [13] Each architecture type comes with different benefits and drawbacks. The biggest advantage of single-tenant is data security as each customer's data is isolated in different instances, in case of a data breach only the breached instance would be affected. The largest disadvantage, that also is the largest advantage for multi-tenant, is resource usage. In single-tenant resources often become underutilised as they can only be accessed by one customer while in multi-tenant the resources can be better utilised as they are accessible by several customers.[13, 22]

2.1.1 Containerisation

Before, when applications ran on physical servers there was no way to share resources or define resource boundaries with other servers. This caused resource allocation issues, for example if several applications were running on a physical server, there was no way to divide the resources between them. Therefore, it could result in one application taking up most of the available resources causing the other applications to underperform. To solve this problem, containerisation was introduced. Containerisation is the technique of packing an application including its dependencies into a container giving it the following characteristics: *lightweight*, *isolated*, *portable* and enables *resource efficiency*. Each container has its own file system, network interfaces, and resources, this allows applications to run independently of each other. In this way containers provide process-level isolation. Additionally, containers are highly portable as it allows applications to run consistently across different environments, for example development machines, testing environments, and production servers. Containers are considered lightweight when compared to virtual machines (VM), which is a software emulation of a physical computer system. While VMs run complete and separated Operation Systems (OS) instances, containers instead share the system's OS kernel, eliminating the need for duplicating an entire OS for each container. Finally, containers can increase resource efficiency. By sharing the system's resources, such as CPU, memory, and disk space, its resource utilisation is optimised. [33]

Containers provide a great way to bundle as well as run the application. It further makes a distributed system more scalable, portable and resilient, as it allows the application to be packed once and be run consistently across the different components of the system. However, the containers need to be managed in order to ensure there is no downtime. Downtime refers to a system or service becoming unavailable or not functioning as expected. For example, in the case of one of the containers going down, another would need to start. Here is where *Kubernetes* comes into play; in what way will be explained in the following section. [26, 32, 33]

Kubernetes

Kubernetes is an orchestration system to help manage containerised applications. Kubernetes provides a framework for running distributed systems with resilience. It takes care of essential tasks such as scaling and failover for the application. Failover refers to the process of transferring the workload and functionality to a secondary system if the primary system experiences failure. When Kubernetes is deployed, it results in a *Kubernetes cluster*.

Kubernetes Cluster

A Kubernetes cluster is a set of nodes that are used to run containerised applications. Each node is a physical or virtual machine, together operating as part of one system. A cluster must consist of one master node and at least one worker node to be operational. The master node is responsible for managing the state of the cluster, this includes scheduling workloads, scaling applications, implementing updates and maintaining the desired state of the cluster. The worker nodes host the *pods* which run the containers that constitute the application. The pods are run based on the instructions received from the master node. These instructions

include how many containers a pod should run, where to run them and how to configure them. [32, 25]

2.2 Graph Databases

A graph database stores and organises information as nodes and relationships, rather than as tables or documents. The different architectures are illustrated in figure 2.1. The nodes can hold any number of properties and are connected to each other through relationships. The relationships are directed, meaning it has a defined start and end node, and must contain a type. Additionally, the nodes also may contain properties. The foremost advantage in using a graph database is the ability to quickly traverse the data through its relations. In existing relational databases the relationships are often navigated with expensive JOIN operations or cross-look-ups, making it both slow and complicated. Moreover, as it is easy to add new nodes and make new relationships, the number of relationships from a node does not affect performance, it allows the data to be stored without being restricted to a predefined model.[42] [7]

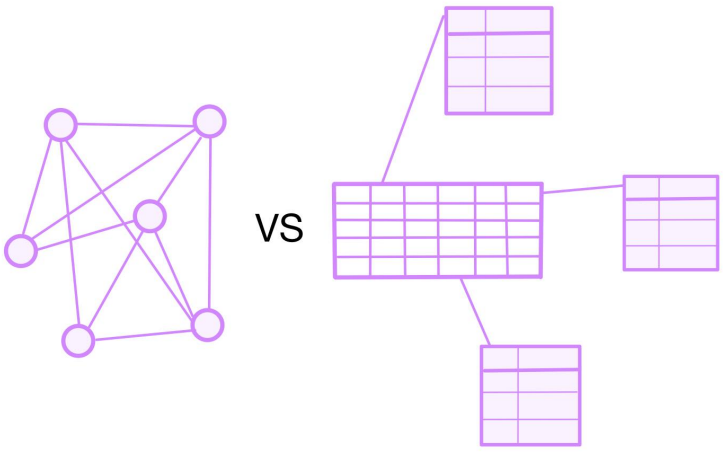


Figure 2.1: Graph database versus a relational database

2.3 The AuraDB System

The system studied in this thesis is the Neo4j AuraDB, a fully managed native DBaaS launched by Neo4j, Inc. in 2019.[41]. It is available to customers with three different subscription plans: *AuraDB Free*, *AuraDB Professional* and *AuraDB Enterprise*. Each plan offers different levels of functionality and support. AuraDB Free has a limited instance size and is restricted to one graph database for the customer. This plan is recommended for smaller development projects or for experimenting. AuraDB Professional and AuraDB Enterprise both offer unlimited database instances with an allowed instance size of 64GB for Professional and 384GB for Enterprise. The professional plan uses a multi-tenant architecture while Enterprise uses

single-tenant. In this thesis we will focus on data collected from databases from the Neo4j AuraDB Professional plan, with Google Cloud Platform (GCP) as a cloud provider.

The AuraDB platform consists of a Kubernetes cluster with corresponding orchestras, which themselves are a Kubernetes cluster, see Figure 2.2 for a graphical illustration. The brain cluster consists of mostly stateless applications that mainly translate web requests into changes in the system. When a new database is to be instantiated it is placed in an orchestra depending on its plan and/or region. For example, there could be a specific orchestration reserved for databases belonging to professional plan, placed in Sweden.

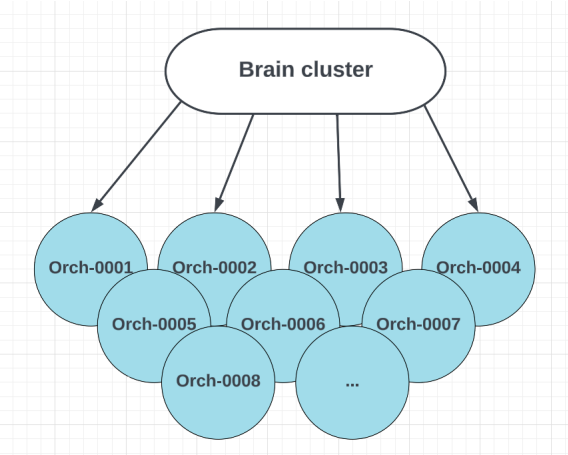


Figure 2.2: Brain cluster with orchestration cluster

Orchestra

As mentioned, each orchestra is in itself a kubernetes cluster. Figure 2.3 shows an illstrative example of an orchestra and its nodes. Each orchestra contains a number of nodes or otherwise called *hosts*. In this thesis we will refer to them as hosts. These hosts are restricted to the specified orchestra, i.e. an orchestras cannot use hosts that belong to other orchestras. Every host in itself can contain several pods, which run the containers that make up the application. A pod can enclose several containers with different functions. The containers for AuraDB can for example contain a database core, labelled *db_x* in the figure, or a component that conducts health-checking. It is worth noting that a pod can only contain one database core, it is not possible for several databases to belong to the same pod. As illustrated with the arrow *A* in 2.3 the pods are able to move between the hosts in the orchestra. A database cluster consists of three database cores, located in three different pods. The three pods must be distributed across three different hosts. An example of how a database cluster needs to be distributed between the hosts can be viewed in 2.3, where the database cores labelled *db_1* belong to the same database cluster. As an orchestra can enclose many hosts, there can be several database clusters belonging to an orchestra. Database clusters will be discussed in detail in the next section.

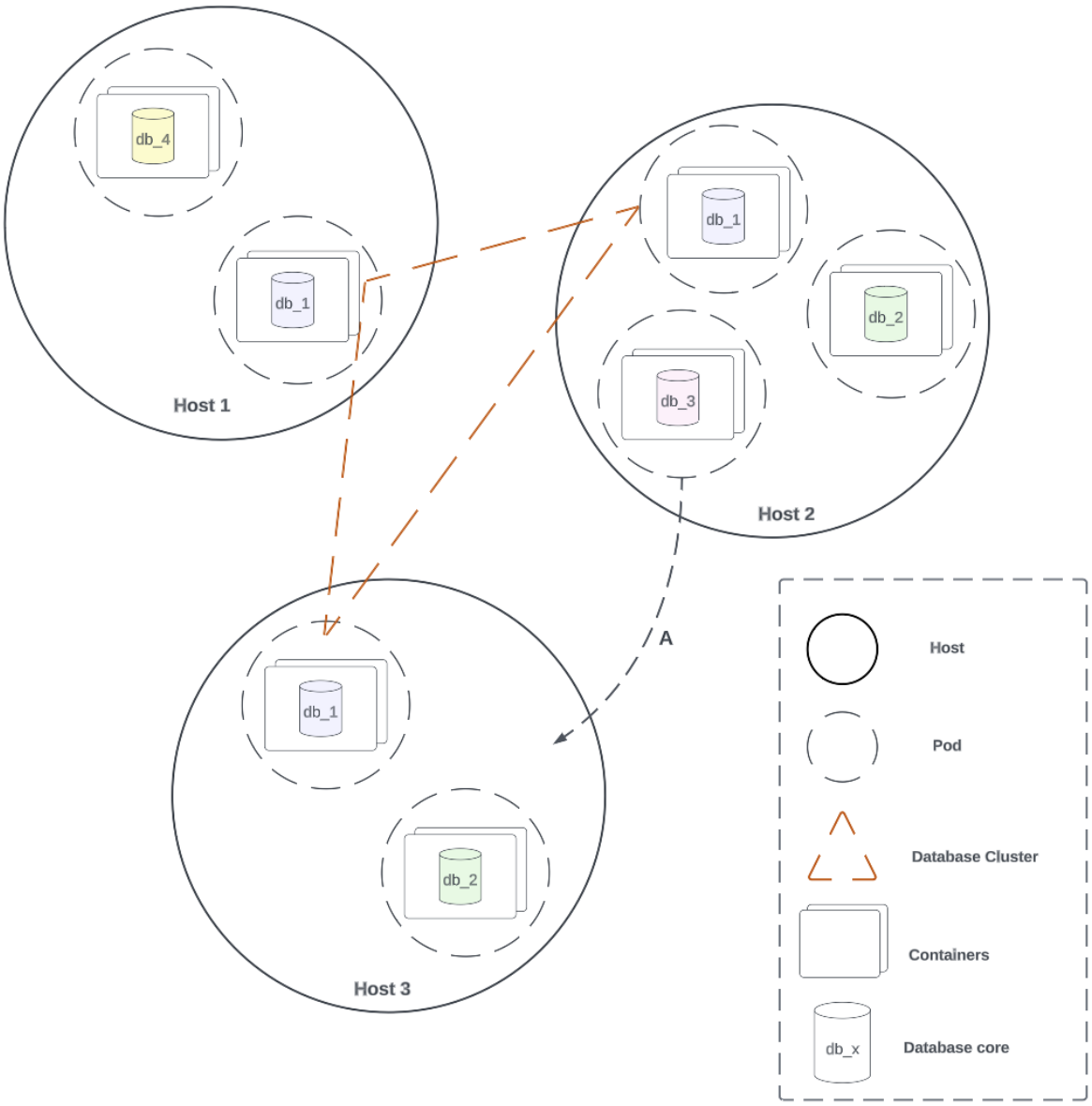


Figure 2.3: System overview of an orchestra containing hosts, with encloses pods with containers belonging to a database cluster.

2.3.1 Database Cluster

As mentioned, three database cores together form a database cluster. Since the cores are located on different pods it becomes important to establish rules regarding the collaboration between the pods. For this system, the consensus algorithm *raft* is used. A consensus algorithm is used in distributed systems to obtain agreement among the different components on a shared state or decision. Raft states that all operations must go through a designated *leader*, i.e. only the leader can accept requests sent from the client. For our database cluster it means that one of the pods is the appointed *leader* with the other two being read replicas of the leader, they are called *followers*. The leader's role is to keep an overview over the clus-

ter's well-being and to communicate to its followers. When a request is sent to the leader it follows the outline as stated in the Figure 2.4. The leader receives the request and appends it to its transaction log, a log that records all changes made to the data in the cluster. The log entry is replicated and sent to its followers, which in turn sends a confirmation to the leader that it was received. In this way the replicas, or followers, are kept up to date with the changes made by the client.

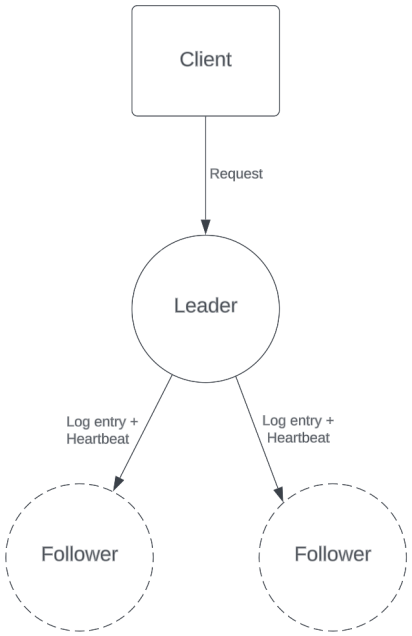


Figure 2.4: Visualisation of how the leader communicates to its followers when receiving a request from the client.

The leader monitors the state of the database cluster by periodically sending out what is called *heartbeat* to its followers. This serves as a way of informing its followers that it is still alive and actively managing the cluster. Additionally, it serves as a way to discover if any member of the cluster has become unavailable. When receiving a heartbeat message the followers send a confirmation back to the leader. If the number of confirmations does not match the number of followers, the leader knows that one of its followers has become unavailable. The followers has a defined *election timeout* and if it does not receive a heartbeat within this time frame the follower assumes that the leader has failed and starts an election process. This election timeout is randomised to prevent multiple followers from starting elections simultaneously. While there may only be one leader at a time, every pod in a database has the capacity of becoming the leader. So in the case of an election process, one of the former followers becomes the new leader. [3]

The advantages of using database clusters instead of a single-instance database are primarily *data safety* and *high availability*.

- **Data safety:** If on a single machine, the disk becomes corrupted or fails, the access to the data would be lost. However, if the same happened to a cluster you can simply communicate to another machine and your data can be recovered.
- **High availability:** By distributing the database across multiple pods, a cluster can continue functioning even if one or more pods fail, it does not lose availability. If a pod goes down, the cluster automatically redirects the workload to the remaining pods, ensuring uninterrupted service. This is further discussed and visualised i 2.3.2. Additionally it enables online upgrades without losing availability, called rolling upgrade and is further explained below.

Rolling Upgrade

A rolling upgrade of a cluster refers to the process of upgrading its members, one at a time. In our case, the members refers to each of the pods of the database cluster. One of the causes that can result in a rolling upgrade, can for example be that the customer wishes to resize the database. The upgrade follows several steps, as illustrated in Figure 2.5. First two new pods are added to the cluster, named *pod 4* and *pod 5* in the figure. One by one the pods are switched out. As can be seen in the second step of the figure, *pod 1* has been exchanged with *pod 6*. As the cluster still had enough active pods when *pod 1* was taken offline, it remained in operation. The same process is repeated and *pod 2* is exchanged with *pod 4* and *pod 3* is exchanged with *pod 5*. Finally pod 5 is selected to be the new leader and the database has executed a rolling upgrade. This approach ensures that the database cluster remains operational all through the upgrade, meaning it has no downtime.

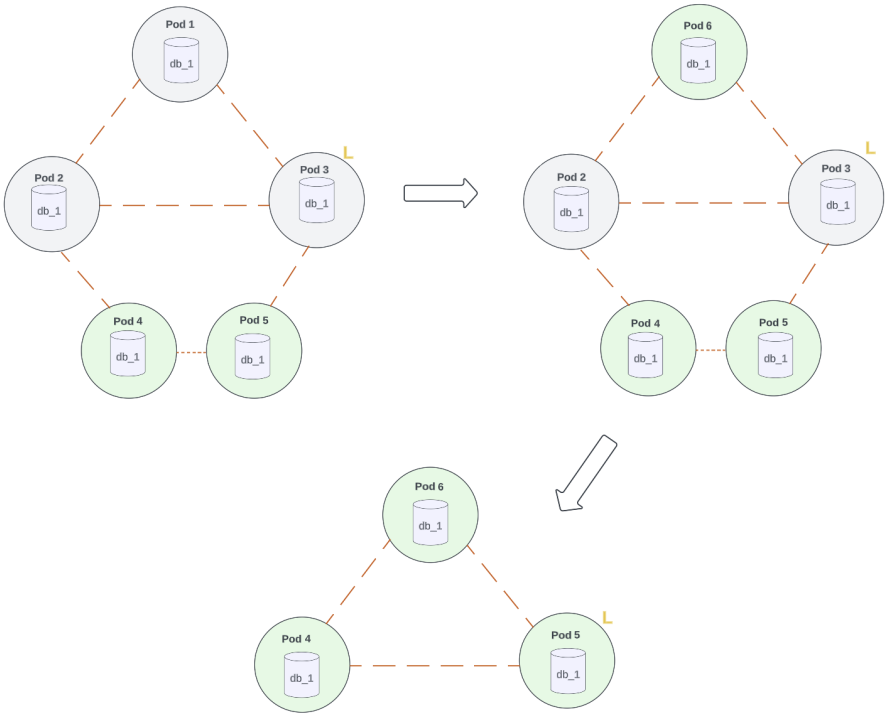


Figure 2.5: Rolling upgrade of a database cluster

2.3.2 Database Fault Intolerance

As mentioned a database consists of three pods that together form a cluster. If all pods are operating as normal the database is healthy and is considered to be fault tolerant. However, if one pod was to become unavailable the other pods can no longer communicate with it. The database is now considered to be fault intolerant. This means that if another pod in the same database cluster was to go down, the database would no longer be operational. If this occurs, the database is said to be unavailable; it would no longer be accessible or operational, preventing the client from interacting with it. The process of a database going from healthy to unavailable is illustrated in Figure 2.6.

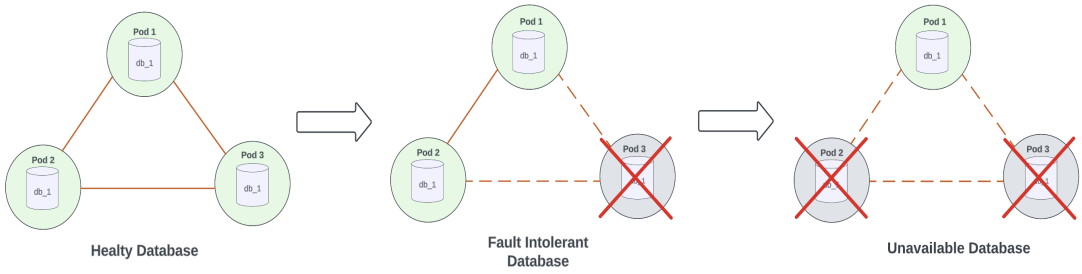


Figure 2.6: Process of database losing fault tolerance. Illustrates a database going from healthy to fault intolerant to unavailable

2.3.3 Causes of Fault Intolerance

There are many causes for a database to become fault intolerant. By inspecting issue-reports made by Neo4j regarding their AuraDB system we could determine and categorise some of their more frequent causes. As our experiments were conducted on professional databases, the reviewed issues were limited to databases belonging to this plan.

- *Customer behaviour*: If the customer overloads their database with a heavy incoming write process it can result in an out of memory (OOM) error and cause the pod to crash.
- *Resizing*: When trying to resize the database it got stuck.
- *Rolling upgrade*: Cluster becoming unstable due to a heavy workload which results in issues when data is replicated across the cluster. As a symptom, the rolling upgrade becomes stuck.
- *Election*: A follower gets stuck in a loop and repeatedly calls re-election.

2.4 System Monitoring

The data used in this thesis project is queried from Datadog, a system Neo4j uses to monitor its metrics. Datadog gets the data from Prometheus, a systems monitoring framework that

collects and stores metrics as time series data. [53] Time series data can be regarded as a collection of observations, obtained through repeated measurements over time. Sampling metrics within a defined time frame results in what can be regarded as time series data. [27]

2.4.1 System Data

The metric data that is being used in this project is collected through the Datadog API. Metric data from different levels are collected; from the application itself up to system level. When querying for data, there are a lot of options to consider. The metric in question can be modified to query data from different subgroups. For example, if we want to query for data about container memory usage from a specific orchestra we can specify that in the query. The metric data originating from that orchestra is grouped by a grouping function. The grouping function used, either returns the average, max, min, or sum of the data given different arguments to group on. As an example, the container memory usage from a specific orchestra can be grouped on the database id, this will be referred to as *dbid* in this thesis. This give us several time series, as the specific orchestra contains a number of databases, and each database is returning its own time series. There can be additional time series that are returned when the queried data is not connected to a specific database. For instance, there can be memory usage that is connected to other services running on the same orchestra. [38]

Given that the system studied in this thesis is cloud native and consists of several microservices, there is a lot of metric data that can be extracted from it. In addition, the microservices themselves can have many processes running within them; the number varying from time to time. As described above, a database consists of three cores that each holds a copy of the data. However, there are occasions where the number of cores deviates from this number, for example during a rolling upgrade. As each core is located on a separate pod, this process leads to the creation and deletion of pods in the distributed system. The dynamic behaviour of the system is important to take into consideration when collecting the metric data. Different processes related to a database can spin up and down. While the system is running it will create new pods with new names which are located on different hosts, while old pods are removed from the cluster and thus from the hosts. In order to be able to consider different levels in the distributed system, from the physical unit up to the application itself, we chose to query data in a way that makes it possible to capture this dynamic behaviour.

Chapter 3
Anomaly Detection and AI

This section will include all relevant theory regarding anomaly detection. We will define what an anomaly is and describe different types of anomalies. Further, the concept of labelled data will be explained as well as how it affects what anomaly detection models can be used. Thereafter, some theory about AI is given which is followed by a description of the anomaly detection methods used in this thesis. Finally, this chapter shows how anomaly detection can be evaluated.

3.1 Defining an Anomaly

An anomaly can be defined as a pattern or an observations that diverges from an established normal behaviour. The anomalies represent instances that deviate significantly from the majority of the data or established patterns, standing out as unusual or abnormal occurrences. [16] This is illustrated in Figure 3.1 where *N_1* and *N_2* illustrate two normal regions as they encapsulate most data points. Points that deviate sufficiently from the normal regions, such as points *a_1*, *a_2* and points within the region *A_3* are anomalies. An anomaly can for example manifest as an unexpectedly high request response time or/and a reduced request throughput. Anomaly causes can be broadly divided into two categories: internal and external. Internal causes refers to both software and hardware bottlenecks, such as bugs in the application code or resource capacity saturation. External causes include resource contention by services belonging to other cloud subscribers that are competing for resources. [40]

Anomalies are generally categorised into three types: *point anomalies*, *contextual anomalies* and *collective anomalies*. A point anomaly is an individual data point that falls outside the normal range for the entirety of the data set. This anomaly is considered the simplest type of anomaly. Examples of point anomalies are extreme values in a time-series, and data points deviating from a cluster of other data points. Contextual anomalies do not necessarily deviate from the entire data set, but diverge from the normal range of the surrounding data set. This anomaly is most often explored in time-series data. Finally, collective anomalies refers to when a collection of data points deviates when compared to the entire data set. The individual data points may not themselves be considered outliers, however occurring together creates an anomaly. Collective anomalies are often investigated when handling sequence data. [16] Certain anomaly detection models can be superior in identifying one type of anomaly but perform poorly on the others. It is therefore beneficial to identify the most common type of anomaly to choose the right model to implement. [6] If no information concerning the nature of the anomalies is known, it is recommended to use point anomalies detection methods. [10] In our thesis we search for all types of anomalies.

3.1.1 Challenges of Anomaly Detection

In anomaly detection the straightforward approach is to define an anomaly as any observation in the data which does not belong to the normal region, which represents the normal behaviour. Despite its simplicity, there are several factors that can be challenging in anomaly detection. [16]

- **Defining normal region:** When the normal region is being defined it is very difficult to encompass every possible normal behaviour. Additionally, the boundary concerning what is considered normal and anomalous behaviour can often be imprecise. As a result, an anomalous observation that is situated close to the boundary can in fact be normal, and vice-versa.

- **Dynamic normal behaviour:** What is normal behaviour may not always be constant but keep evolving. Thus the current defined normal behaviour may not be adequately representative in the future.

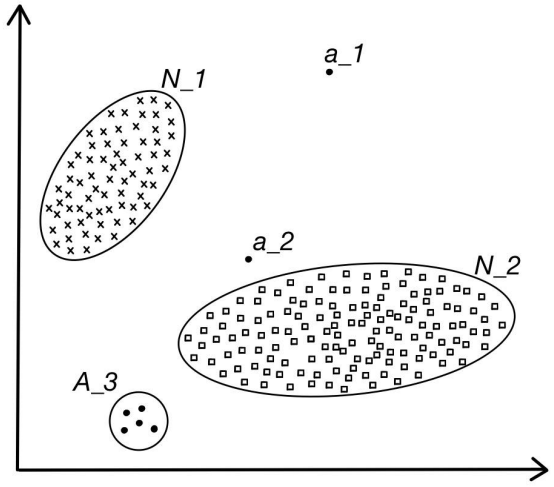


Figure 3.1: Anomalies in a 2-dimensional data set where *N_1* and *N_2* are normal regions, *a_1* and *a_2* and the region *A_3* are point anomalies

- **Different application domains:** What is considered an anomaly may differ for different application domains. In a medical domain a seemingly small deviation in for example body temperature might be classified as an anomaly, while in a stock market domain similar deviations would be considered normal behaviour. Therefore, applying anomaly detection techniques developed for one domain to another is not a straightforward process, as the criteria for identifying anomalies differ significantly.

- **Availability of labelled data:** The model's access to available labelled data for training and validation is often a significant issue.

- **Data noise:** Distinguishing and removing noise from the data is often challenging as it can bear similarities to actual anomalies, making it difficult to differentiate between the two.

- **Accurate classification:** Failure to detect an anomaly and having it go unnoticed can have unpredictable impacts on the system. On the other hand, mistakenly identifying an anomaly, also called a false positive, causes a disruption on regular operations in order to investigate the anomaly. It is therefore important to have a high accuracy in detecting anomalies without it leading to too many false positives.

As a result of the above challenges the general problem of anomaly detection is not easy to solve. Existing anomaly detection techniques primarily solve a specific formulation of the problem, based on factors such as the nature of the data, availability of labelled data, as well as the types of anomalies to be detected. Therefore, it is important to determine the characteristics of the domain and data in order to elect a suitable anomaly detection technique. [16]

3.2 Data Analysis and AI

Data analysis is the science of analysing raw data to produce information. Many of the techniques used for data analysis have been automated into mechanical processes and algorithms that convert raw data into information for human consumption. [19] When talking about data analysis for large amounts of complex data or big data, the term data science is often used. [11] In data science, techniques based on machine learning and artificial (AI), are frequently used to extract meaningful information and to predict future patterns and behaviours. Machine learning, in fact, is a form of AI, where the aim is for the machine to learn, adapt and improve from that data. An example of information that can be obtained by applying data science, is if a data set contains any type of anomalies. In this section, a brief background on data analysis and AI is given in regards to anomaly detection.

3.2.1 Statistical Analysis of Data

As mentioned previously, a key aspect in anomaly detection techniques is the nature of the input data available. The input data set generally consists of a collection of data points, which represents a specific example or measurement. In the context of time series data, the data points are observations collected at different points in time. The data set may consist of a single variable (univariate) or multiple variables (multivariate) recorded at regular intervals such as every second or minute. [16] Depending on this characteristic of the data set, it is analysed differently.

Univariate Analysis

The simplest analysis is called univariate, where the data being analysed only contains one variable. The main purpose of univariate analysis is often to describe and find patterns in the data. The focus is on understanding the characteristics and distribution of that specific variable, without considering its relation to other variables. Some of the most common univariate analyses include checking the standard deviation, the range, and the mean of a variable. Histogram, a frequency distribution graph, is often used to visualise univariate analysis. [50]

Bivariate Analysis

Bivariate analysis refers to the analysis that examines the relationship or association between two variables simultaneously. Contrary to univariate analysis, this analysis considers the interaction and dependencies between the variables in order to gain knowledge and insight into their relationship. The variables can both be dependent or independent to each other, but there must always be a Y-value for each X-value. This analysis is commonly executed by techniques such as scatter plots or correlation analysis. [50]

Multivariate Analysis

Finally, multivariate data refers to analysis of data sets that contain three or more variables. In other words, multivariate data sets consist of two or more variables that are measured on the same set of cases. For example, a multivariate data set might include a computer's weight, size, cpu, among other variables. This analysis examines how the variables interact and influence each other in order to gain insight of their relationship and patterns. In this analysis, various techniques and methods can be employed, for example cluster analysis or through descriptive statistics. [50]

3.2.2 Labelled and Unlabelled Data

When working with data in anomaly detection, data points can be assigned a label indicating whether they are classified as normal or anomalous. However, this can be an expensive task as it is often done manually as well as requires accurate and representative labelled data for all possible behaviours. Obtaining labels for anomalous data sets is particularly challenging due to the dynamic nature of anomalous behaviour, which may result in the emergence of new types of anomalies without corresponding labelled training data. Additionally, if the distribution of classifications in the labelled data set is skewed, it can pose challenges in building predictive models. This problem is referred to as the *imbalanced classification problem* and must be taken into consideration when deciding on what approach to take, since most machine learning algorithms assume balanced class distributions. [9] Based on the extent of available labelled data, anomaly detection techniques can operate in one of three following modes: *supervised*, *semi-supervised* and *unsupervised*. [16]

Supervised Anomaly Detection

In supervised anomaly detection the model assumes access to training sets labelled as both normal and anomalous. The typical approach is to create a predictive model for normal versus anomaly classes, where unseen data can be compared to determine which classification it should get. However, two major issues arise with this approach. Firstly, the number of anomalous occurrences is often much lower than normal occurrences in the training data, causing imbalanced class distributions of the data sets. Secondly, as mentioned above, it is often challenging to obtain accurate and representative labels for all anomalies. [16]

Semi-Supervised Anomaly Detection

Semi-supervised anomaly detection techniques assume that only normal training sets are labelled, making them more broadly applicable than supervised techniques. This detection technique is often implemented by building a model corresponding to normal behaviour and identifying anomalies based on deviations from the model. While more applicable, this technique has its own challenges. Firstly, it can be difficult to determine the threshold against which data is compared, especially when the training data contain labelled anomalies. Secondly, the success of this technique relies heavily on that the training data include representations of all possible normal data sets. Otherwise, normal data has a risk of being missclassified as an anomaly. Finally, if the anomalous data sets are similar to the normal ones, detecting these anomalies becomes challenging. [16]

Unsupervised Anomaly Detection

Unsupervised anomaly detection is a technique that does not require any labelled training data. Instead, it attempts to identify anomalies solely based on the characteristics of the data, without any prior knowledge of what constitutes normal or abnormal behaviour. While this technique is very applicable as it does not require labelled data, its success is based on the assumption that normal instances are far more frequent than anomalies in the test data. If this assumption is not true, this technique can produce a high rate of false alarms. Additionally, unsupervised techniques may not be effective in detecting rare anomalies, as they are often drowned out by the abundance of normal instances. [16]

3.2.3 Artificial Neural Networks

Artificial neural networks (ANN) are a category of machine learning techniques that simulate the learning mechanism observed in biological organisms. An ANN contains computational units referred to as neurons (also nodes), which can be connected to each other to form a network. Through this network, they can receive and send signals to other connected neurons, resembling the synapses in our brain. The connections between the nodes in an ANN are associated with weights which affects the function of the connected neurons. The ANN computes a function of the input values by propagating the computed values from each neuron to the subsequent ones, using the weights as intermediate parameters. Learning occurs by adjusting these weights. During the training process, input and target pairs are utilised as training data, representing the desired behaviour of the network. These pairs are used to update the weights. The training data is passed through the network, and the prediction error is calculated by comparing the output value with the target value for the given input. This prediction error serves as feedback to the network, providing information on the correctness of its weights. The calculated error is commonly known as the loss, and the function used to compute it is referred to as the loss function. By training the network, the objective is to alter the weights in order to modify the computed function, enabling more accurate predictions in future iterations of training. One widely used method to train neural networks is the *backpropagation algorithm*, which will be described further in Section 3.2.3. There are several different architectures of ANNs, some of which are described below.

Perceptron

The simplest form of a neural network is the perceptron, which consists of a single input layer and an output node. The basic architecture of the perceptron can be viewed in figure 3.2. Given a set of input values $\bar{X}$, the predicted value $\hat{y}$ is computed as shown in equation 3.2 The output of the perceptron $\hat{y}$, is given by applying an activation function to the sum of all inputs x_i multiplied by its weight w_i.

$$\hat{y} = f(\bar{W} \cdot \bar{X}) = f(\sum_{i=1}^{N} w_i x_i) \tag{3.1}$$

In many cases, predictions often contain an invariant component known as bias. In order to capture the bias, an additional bias variable b is included before applying the activation function. The updated equation, accounting for the bias, is as follows:

$$\hat{y} = f(\bar{W} \cdot \bar{X} + b) = f(\sum_{i=1}^{N} w_i x_i + b) \tag{3.2}$$

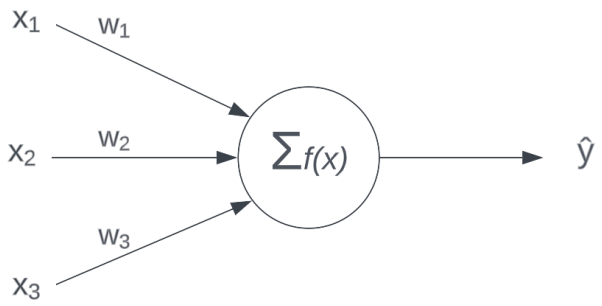


Figure 3.2: Illustration of perceptron with inputs x_1, x_2, x_3 with corresponding weights w_1, w_2, w_3. The weighted sum is applied to the activation function *f(x)* to obtain the output, $\hat{y}$.

The perceptron, despite consisting of two layers (input and computational), is considered a single-layer network since we only consider the computational layer while disregarding the input layer. The training process involves calculating the error value between the true value and the predicted value, denoted as $E(\bar{X}) = y - \hat{y}$. When the error is non-zero, the weights need to be updated in the negative direction of the error gradient. There are various methods to determine the magnitude of the weight update. The most common optimisation algorithm that does this is gradient descent, although it is often modified or extended in different ways. One such modification to the gradient descent is the Adam optimiser. By interpreting the perceptron as a simple computational unit, multiple units can be put together to create a more complex and powerful model. [5] This leads us to the next architecture where we describe multi-layer neural networks.

Feed Forward Neural Network

In a multi-layer network, there are multiple computational layers. The additional intermediate layers are referred to as hidden layers since the computations performed at these levels are not visible to the user. The multi-layer architecture is often referred to as a feed-forward network, as the outputs from nodes in one layer are fed into the succeeding nodes in a direction towards the output. The most common architecture of a feed-forward neural network assumes that every node in one layer is connected to every node in the next layer. This architecture is often referred to as a fully connected neural network and is depicted in Figure 3.3.[5]

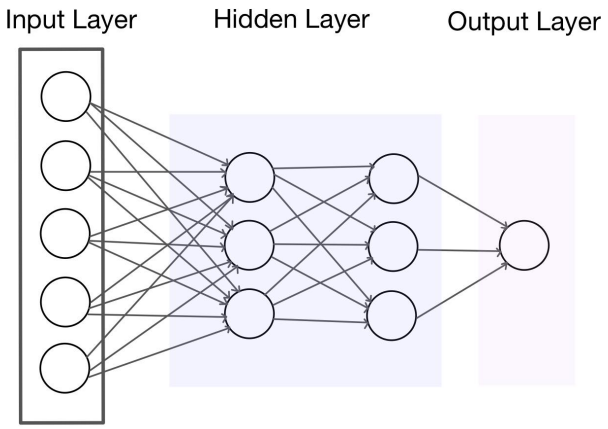


Figure 3.3: Illustration of a fully connected neural network with input layer, hidden layer and output layer.

When training a multi-layer neural network, as opposed to the single-layer neural network where the loss is a direct function of the weights, the loss is a complicated composition function of the weights in earlier layers. To train a multi-layer neural network, the backpropagation algorithm is used. The algorithm consists of two phases: the forward phase and the backward phase. During the forward phase, the input is propagated through the neural network and produces an output using the current weights. The output is then compared to the target value by calculating the derivative of the loss function with respect to the output. Here is where the backward phase takes over, where the derivative of the loss with respect to the weights in all layers is computed. This is accomplished by utilising the chain rule of differential calculus. The gradients are learned by starting from the output node and propagating them backwards through the network, hence the name *backwards phase*. One iteration of this process of forward- and backward propagations is referred to as an epoch. There are many other types of neural networks that can implement multiple layers, one of them being recurrent neural networks (RNN). [5]

Recurrent Neural Network

Recurrent neural networks (RNNs) are a type of network that is specifically designed for sequential data, such as time series data. In sequences, the successive data is dependent on one another, making it essential for the model to incorporate a memory unit. In Figure 3.4, the architecture of a recurrent neural network is illustrated. The memory unit, or units of a multi-layer network, are hidden states within the network. They can be seen as the red loops in the figure. These hidden states store information from previous time steps and allow the network to retain and utilise this information in its computations.[5] The hidden state is given by the function:

$$\bar{h}_t = f(\bar{h}_{t-1}, \bar{x}_t) \tag{3.3}$$

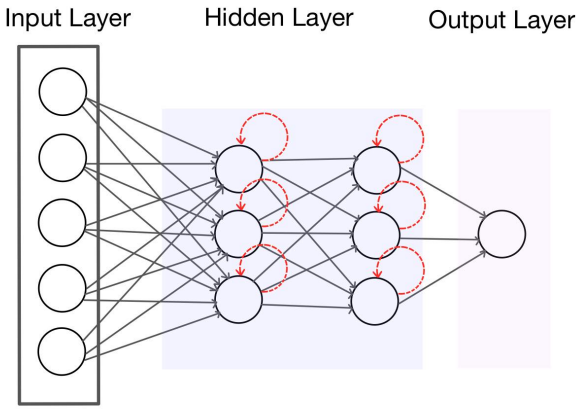


Figure 3.4: Illustration of a recurrent connected neural network with input layer, hidden layer and output layer.

Long Short-term Memory (LSTM)

Similar to neural networks consisting of many layers, recurrent neural networks have stability issues associated with the training. One prominent issue is that the gradients tend to vanish when performing backpropagation on long sequences. This is the result of successive multiplications with the hidden state vectors at various time-stamps. To address this issue, a more advanced type of RNN known as Long Short-Term Memory (LSTM) is preferred. In LSTM, the recurrence equation for the hidden state vector is replaced, this allows for a more precise control over the data to be stored in the long-term memory.[5]

3.3 Anomaly Detection Techniques

In this section we will first give an overview of the anomaly detection techniques used in this thesis. We will further present the advantages and disadvantages of each technique as well as when it is suitable to use. Finally, the specific models used in the thesis and which detection technique it belongs to will be explained.

3.3.1 Classification-Based Anomaly Detection Techniques

In classification-based anomaly detection techniques anomalies can be identified by building a model from labelled data that is able to classify data points as either normal or anomalous. This means that this technique only can be applied to supervised or semi-supervised anomaly detection. The technique consists of two phases; training and testing phase. In the training phase the model is trained with the available labelled data. In the testing phase the model is used to classify unseen data as normal or anomalous and its performance can be evaluated. Advantages of classification based techniques include its ability to utilise labelled data as well as having a fast testing phase as each test point only has to be compared to the prede-

fined model. However, classification-based approaches also have limitations. They rely on availability of labelled data and may struggle with imbalanced data sets where anomalies are rare compared to normal instances. Some examples of classification-based anomaly detection techniques are decision trees and deep learning models such as neural networks and recurrent neural networks. [16]

3.3.2 Autoencoder Approach for Classification

While not inherently a classification-based technique, autoencoders can be effectively used for classification purposes. As depicted in Figure 3.5, an autoencoder consists of two modules: an encoder and an decoder. The encoder learns the underlying features of a process and reduces the dimensionality of the input data. Using the captured features, the decoder is trained to recreate the data to match the original input data. This process involves bringing the dimensionality back to its original shape. [46]

When using an autoencoder for anomaly detection, the model is trained on normal data, making it well-suited for reconstructing data from the normal state of the process. In order to evaluate how successful a reconstruction is, a reconstruction error is calculated. This is done by comparing the prediction, i.e. the reconstruction, to the true value, i.e. the input value. A common approach is to define the reconstruction error as the mean squared error (MSE) between the prediction and the input data. A small reconstruction error indicates data originating from normal process behaviour, while a high reconstruction error suggests data from a rare event or anomaly. By setting a threshold, the reconstruction error can be used to classify data as either normal or anomalous. [46]

The autoencoder model can be extended to handle not only multivariate data but also incorporate the temporal information contained in time series. To achieve this, the encoder and decoder modules need to be based on recurrent neural networks (RNNs). By using the RNN layers, an additional dimension is added to the input data, as the number of points to include in the look-back needs to be specified. The reconstruction error is now calculated over multiple data points, as the entire look back window will serve as the target value. When designing an RNN autoencoder, one option is to employ LSTM layers in the model architecture, as this enables modelling dependencies for longer sequences. [46]

3.3.3 Statistical Anomaly Detection Techniques

Statistical anomaly detection techniques identify anomalies in data based on their deviation from the statistical properties of the majority of the data and is based on the following key assumption:

- *Normal data instances occur in high probability regions of a stochastic model, while anomalies occur in the low probability regions of the stochastic model.*

By assuming that normal data points follow a specific statistical distribution, such as mean, standard deviation or Gaussian distribution, anomalies can be identified as data points that significantly deviate from this distribution. Depending on chosen distribution, data points that fall outside of a certain range or have unusual patterns can be flagged as anomalies.

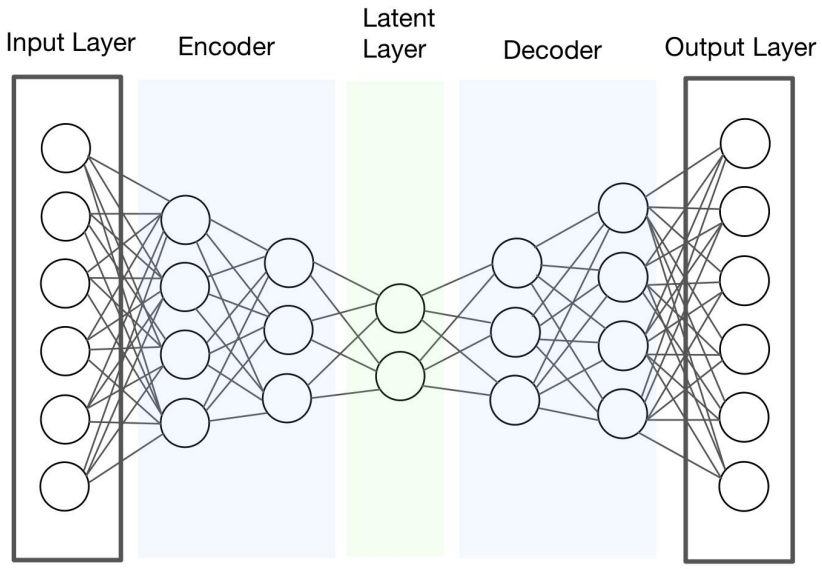


Figure 3.5: Illustration of an autoencoder

Limitation of statistical anomaly detection is primarily the assumption that data follows a specific statistical distribution, which may not always be the case. In the case of multivariate data it is unable to handle dependence between variables. An anomaly could have variable values that individually are very common, but their combination is infrequent. Examples of statistical anomaly detection techniques include *gaussian based*, *regression model based* and *histogram-based*. These techniques can all be used to analyse both univariate and multivariate data. [16]

Histogram-based Outlier Score (HBOS)

Histogram-based Outlier Score or HBOS is a histogram based statistical anomaly detection technique. Given N variables, this algorithm produces N histograms independently where each histogram is a representation of how likely a particular data point is to fall within its bins. The higher the bin, the more observations belong to that bin. This means that values classified as anomalies are found in the lowest bins of the histogram. The outlier score for univariate HBOS is the inverse of the height of a bin calculated as: [35, 15]

$$HBOS(p = 1/hist(p) \tag{3.4}$$

Where $1/hist(p)$ is the height of the bin of variable. A resulting large value will indicate the point p being an outlier.

This algorithm is suitable to handle both univariate and multivariate unsupervised data. In order to calculate a point's multivariate outlier score, the univariate score for each variable is calculated separately, meaning the algorithm assumes independence between the variables. By summing up the N univariate anomaly scores the multivariate HBOS can be calculated.

This is visualised in figure 3.6 where P, the point we want to examine, is first shown as to a normal point as it appears in the high bins for each variable and secondly as an anomalous point as it belongs to the low bins. It is worth noting that P does not need to always belong to the highest bin in order to be classified as normal and vice versa for being an anomaly. HBOS of an observation p defines the sum of the N logarithmic univariate anomaly scores. The reason for using the logarithmic value is to make it less sensitive to extremely small bin values as well as errors due to floating-point precision in unbalanced distribution causing very high scores. [15, 23]The formula is formulated as following:

$$HBOS(p) = \sum_{i=1}^{N} log\left(\frac{1}{hist_i(p)}\right) \tag{3.5}$$

Where $hist_i(p)$ is the height of the bin of variable i for the given observation p.

Determining the size of the bins used to build the histogram is critical in order to achieve effective anomaly detection. If the bins are too small, a large number of normal test points may be placed into rare or empty bins, resulting in a high false alarm rate. Similarly, if the bins are too large it may cause a large number of anomalous test points to be placed in frequent bins, resulting in a high false negative rate. Therefore, determining an optimal bin size for the histogram that can achieve both low false alarm rate and low false negative rates is a significant challenge when using HBOS. [16]

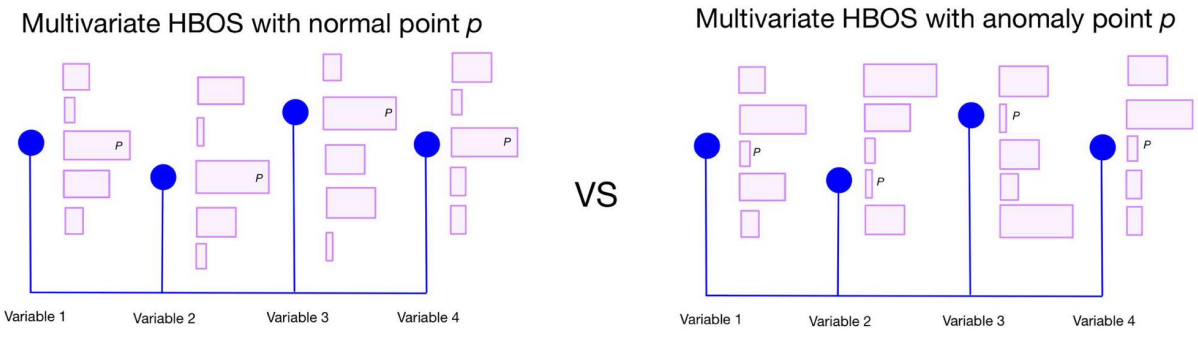


Figure 3.6: Illustration of multivariate HBOS that shows a point P being classified as normal VS anomalous, depending on what bins it belongs to.

3.4 Data Processing

Before applying data to the models described above it is important to prepare and process the data. Data preprocessing is an integral part of data preparation. In this step, various techniques are used to transform and modify the raw data in order to make it suitable for further analysis or processing procedures. In this section we will talk about how the data can be processed to be better suited for anomaly detection.

3.4.1 Data Preprocessing

Data preprocessing is the step after the data has been gathered before entering it into a machine learning model, this step is essential for the success of the model. Insufficient or irrelevant data can lead to machine learning algorithms producing inaccurate and incomprehensible results, or even failing to uncover anything valuable. There are several practices in data preprocessing for example data cleaning or data transformation. The resulting data is the final training set for the model. [31] In this section we will go through data processing practises that had to be considered and evaluated during the experiment of this thesis.

There are several different methods to handle if a metric is missing a value. When working with time-series data this will occur when a specific metric lacks a value for a certain point in time. One way to handle this scenario is to fill the missing value with an arbitrary value. This is a good method for when you have a known default value and said value can be inserted without corrupting the entire data set. The second approach is to calculate a default value based on the metrics statistic. If possible, this is often considered a better opinion. The final way is to simply ignore said value. This can be applied if working in a large data set where single values can be disregarded. [51]

3.4.2 Feature Scaling

Feature scaling is a data transformation process where the aim is to transfer the data into forms suitable for the chosen model. The reason behind this is that features often have different scales, for example one feature can have values ranging from 0 to 1 and another from 0 to 1000. This can result in the machine learning algorithm giving higher weight to features with higher magnitude, (making it unbiased). [14]

Capture Pattern in Data

As the data being used in this thesis is time series data it becomes important to be able to capture this behaviour. Moving average is a statistical technique that can help identify the general direction or movement of the data over time.

Simple Moving Average (SMA)

Simple Moving Average (SMA) is a simple and rapid method to capture patterns in time series data. SMA is calculated by taking the mean value of the past N data points throughout

the timeseries, according to the following formula:

$$SMA = \frac{P_M + P_{M-1} + \cdots + P_{M-(N-1)}}{N} \tag{3.6}$$

Where P_M is the value of the datapoint at time M and N is the number of datapoints used in the calculation. [24]

Besides gaining insight in the data patterns, SMA can be used to detect anomalies. By calculating SMA for every point in the timeseries you get a smoothed graph that displays how the graph is estimated to behave. In comparing the SMA for each point in the time series with the actual value a distribution over its deviance can be composed. By identifying points with high deviance anomalies can be identified. [34]

Exponential Moving Average (EMA)

In SMA the position of the points are not taken into account when calculating the mean value, as a result, past observations are weighted equally. In EMA, the weight of past observations decreases exponentially, meaning recent points have more of an impact when calculating the EMA. Given a time series with observations beginning at time $t = 0$, the output of EMA can be calculated using the following formula:[24, 34]

$$F_t = \alpha y_t + (1 - \alpha)F_{t-1}, t > 0 \tag{3.7}$$

Where F_t is the forecast of the next value, y_t is the value of the time series in time t and α is the smoothing constant which determines the degree of weighting decrease, factoring between 0 and 1. With a high smoothing constant, the model mainly takes the most recent observations into account, whereas with a low value the model takes more of the history of observations into account. [24] As a default the smoothing constant is often set as following:

$$\alpha = \frac{2}{windowsize + 1} \tag{3.8}$$

3.5 Evaluating Anomaly Detection

In order to evaluate anomaly detection performance the first step is to identify the following outcome from classifications obtained from the anomaly detection model [36]:

- True positives, (TP). Number of correctly predicted outliers.
- True negatives, (TN). Number of correctly predicted normal points.
- False positives, (FP). Number of false alarms, namely when normal points are predicted as outliers.

- False negatives, (FN). Number of missed outliers, namely when outliers are predicted as normal points.

These measurements are used to compose performance metrics that evaluates the anomaly detection model. The most common are *Accuracy*, *Recall*, *Precision* and *F1 Score* and are calculated in the following way, where the higher its score, the better the model performs. [12]:

- **Accuracy**: The proportion of correctly identified predictions.

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

- **Recall**: The proportion of true anomalies that was identified.

$$Recall = \frac{TP}{TP + FN}$$

- **Precision**: The proportion of identified anomalies that are true anomalies.

$$Precision = \frac{TP}{TP + FP}$$

- **F1 Score**: The overall performance of the anomaly detection model, calculated by using the harmonic mean of precision and recall.

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

A model is said to be under-predicting if its performance results in high precision and low recall, meaning anomaly predictions are often accurate but only a small proportion of them are predicted. Similarly, a model is said to be over-predicting if it has low precision but high recall. In this case the model correctly identifies a large proportion of the anomalies, but a lot of its predicted anomalies are miss-classified and are instead normal values. Often a model can be manipulated to increase precision at a cost of lower recall, and same the other way around. [36]

F1 score is the computed average of precision and recall, meaning it gives equal weight to precision and recall. As Precision and Recall are both rates, harmonic mean is a rational choice. Harmonic mean is a type of average, best suited when the average rate is desired. It is calculated by taking the number of observations and dividing it with the reciprocal of each observation. A high F1 score is obtained if the models Recall and Precision are both high, similarly its score will be low if they are both low. A medium F1 score is obtained if the model precision or recall is low while the other is high, meaning the model is under-predicting or over-predicting. [30]

Chapter 4
Method

In this chapter we aim to provide a detailed description of the data collection, data preprocessing, metric selection, the anomaly detection process, and methods to evaluate our results. This will give the reader an understanding of the challenges that needed to be addressed at the different stages as well as necessary context to follow the subsequent presentation of results and discussion. Based on our research questions, the scope of our work can essentially be divided into two main categories; metric selection and anomaly detection. Different methods of how to reduce the number of metrics included in our experiments are discussed. The possible architectures of previously mentioned anomaly detection models are described as well as the process of evaluating the results.

4.1 Project Process

In the start of our thesis a literature study was conducted in order to determine the best practices for anomaly detection when using metrics. Through this study, we discovered that when working with metrics generated from monitoring the system, it is crucial to identify a subset of monitoring metrics that are relevant to our objective. In our case, it was finding metrics whose anomalies indicated that the system was losing fault intolerance. The literature suggested several state-of-the-art methods, including Support Vector Machine (SVM), Mutual information (MI), finding correlation between error logs and metrics, and finally Filter methods.

In order to determine which methods would be suitable for our experiment, we analysed the data, specifically focusing on how fault intolerance was represented. The fault tolerance of a database, referred to as *db*, is reported using the metric *neo4jdatabases.condition.duration/db/condition:faulttolerant*. When a database is operating normally, i.e. all pods are up and running, the value of this metric is 0.0. However, if the database loses fault tolerance, this metric indicates the duration for which the database has been without fault tolerance. Therefore, this metric provides valuable information about the occurrence of fault intolerance loss and subsequent recovery. The significance of this information had a large impact on what methods we could use to identify our relevant metrics.

SVM was withdrawn from consideration due to the method relying on separate classes. Since our data exhibited severe class imbalance, with fault intolerance only comprising a small fraction of the data set, this method was deemed not optimal for our data. Mutual information, on the other hand, relies on finding a correlation between a feature and the target value, which in our case was fault tolerance. However, as our metric monitoring fault tolerance only provides information when fault tolerance is lost, rather than offering a continuous representation of the system's state, calculating a relationship with other features would be challenging.

Since filter methods simply rank the features according to some criteria, independent of the target feature, we could disregard the limitations imposed by the representation of our target feature, which prevented us from using the methods mentioned above. Therefore, we decided to implement a filter method to identify the most relevant metrics. The selection of interesting metrics was made using the filter method, which involved a statistical analysis of standard deviation and metric selection based on domain knowledge.

After identifying the interesting metrics, our focus shifted to how they could be utilised for anomaly detection. To address this, an additional literature study was conducted, with the aim of identifying state-of-the-art anomaly detection algorithms that would be suitable for our specific scenario. We paid particular attention to algorithms capable of handling time series data and high dimensionality, caused by us aiming to incorporate multiple metrics in the detection process. Ultimately, we selected HBOS and an LSTM-based autoencoder as our chosen algorithms.

While the LSTM-based autoencoder algorithm is constructed to capture the interdependen-

cies among the data, HBOS did not take the data points surroundings into consideration. As a result, HBOS was found to be less suitable for handling time series data. To address this limitation, a customised vrsion of HBOS was developed. This customised HBOS, incorporated a moving average technique to capture the estimated behaviour of the data points based on their moving average values. The difference between the actual value and the moving average estimation was then used as input for the HBOS algorithm.

The models were trained and tested using real data collected during one month of normal operation from 30 chosen databases in the AuraDB system. The data set was divided into two equal parts: one for training and one for testing. The models were evaluated based on their capacity to detect anomalies that resulted in a database losing fault tolerance. The evaluations were conducted based on metrics such as *recall*, *precision* and *F1-score*.

4.2 Metrics Selection

A significant part of the thesis project involved extracting and processing metric data from the Neo4j AuraDB system. In the following section, we outline the steps taken to reduce the number of metrics that were later used in the anomaly detection phase. In order to scale down the amount of data, our first step was to examine what data was available. Subsequently, we analysed the metric data in several steps to finally arrive at a manageable set of metrics on which we could focus our experiments.

4.2.1 Identify Relevant Metrics

The first step was to obtain a list of available metrics used to monitor the system. The metrics were obtained by querying *get_active_metrics_list* to the Datadog API, which provided us with a comprehensive list of all actively reporting metrics from a specified time until the present. We decided to limit the timeframe to metrics active in the last 30 days, resulting in 2,591 different metrics. Our next task was to identify which of these metrics that appeared to correlate with databases becoming fault intolerant. In collaboration with our advisory at Neo4j, we were able to eliminate metrics that solely monitored the system separately from the databases or were known to have no correlation with the state of the databases. This reduced the number of relevant metrics to a total of 1,158. To continue our analysis and further narrow down the metric selection, we began examining the metric data, leading us to the next step in the process of metric selection.

4.2.2 Data Querying

To be able to conduct further analysis, we now needed to query for actual data. This might seem like a trivial step, but due to limitations of the Datadog API and the complexity of the system there were a number of factors that we needed to take into account when designing the pipeline. To gain an understanding of the data before proceeding to large-scale querying, we used the Datadog metric explorer to visualise selected metrics. This was especially beneficial as it allowed us to gain an understanding of the frequency of database fault intolerance. We could see that it happened a couple of times a week, and that it stayed fault intolerant for

up to a couple of minutes. This information was useful when we had to decide on the time to query data and the sampling rate, the number of samples taken per unit of time. Since the number of metrics was still quite large, we decided to query data for a total time of one week, with a sampling rate of one minute per sample.

The first step was to extract the metric data from the system by constructing queries to send to the Datadog API. The required arguments in the request were the start and end time, as well as the actual query string. Unfortunately, the version of the Datadog API that was used did not allow us to specify the sampling rate, instead this was decided on the start and end time that was given. In order to sample data with a rate of one minute, the start and end time was set with a four hour interval in between. To query for more data than four hours while still maintaining the same sampling rate, we needed to make several queries and concatenate the results as the metric data was returned from the API.

The actual query string was created by composing a string with the information on what to query for. The format of the string can be seen below.

$$<grouping\_function>:<metric> \{<from>\} \ by \ \{<groups>\}$$

The available grouping functions to choose from was; *average by, max by, min by, and sum by*. "The averaging function was used when collecting our metric data as it is the most generic and suitable for most types of metrics. The Datadog API only allows querying for one metric at a time, and is specified by giving the name of the metric. For instance, to narrow down the scope, we decided to query data for databases belonging to the same orchestra, specifically *orch-1*. Therefore, our *from* argument would be *orchestra:orch-1*. Lastly, we could include a grouping parameter to group the data by. Since we knew that a specific database consisted of several pods which in turn reside on different hosts, we decided to group the time series data on *dbid*, *podname*, *pod_name*, and *host*.

A problem that arose when querying the data was the inconsistency in the names of the groups from which we queried the desired metric data. For example, one metric may have used *orchestra* as the group name, while another metric used *aura_cluster_name*, or *cluster-name*, all belonging to the same group in principle. If we included a group in the *from* field that could not be found by the Datadog API for the specified metric, an error was returned. As we could not determine in advance which metric used what group name, we had to adopt a trial-and-error approach. We started by making a test query to check if the query was successful, thus indicating that the specified group name was included as part of the metric tags. If an error was returned, we would modify the group name and try again.

Similarly, the groups (*dbid*, *podname*, *pod_name*, and *host*), for which we were to group the time series by, also had slight differences in naming for different metrics. E.g. sometimes the name *podname* was used by a metric, other times it was called *pod_name*. However, even if we included a name that was not part of the metric tags, we still obtained a result. The only difference was that there would be no grouping on that specific subgroup name. This enabled us to include all the metric tags to group by for every single query, even if it was not always possible to group by them. For example, some metrics could only be grouped by *host*,

while others could only be grouped by *dbid*, or *podname*. To give an example of how a query string looks like, we can study the following query:

$$avg : jvm.heap\_memory\{orchestra : orch-1\}\ by\ \{dbid, podname, pod\_name, host\}$$

Taking all these aspects into account when querying for metric data, in order to get data for one week with a sample rate of one minute, we needed to make more than 42 000 queries. The high number is the result of the inability to specify sampling rate, as well as the large number of metrics. Sampling at a rate of one minute we could only query for data at a four hour interval which results in 6 queries per day, making it 42 queries for one week. Multiplying this by the number of metrics, which was over 1000, gave us the final number.

The time series resulting from the querying process were stored in files, with each file containing one time series. Each time series represents an unique combination of values for the groups that we used for grouping. For example, we could have time series matching the following two set of values.

```
metric:     haproxy.backend.session.current
host:       gke-prod-1-XXXX
dbid:       0001
podname:    NA

metric:     haproxy.backend.session.current
host:       gke-prod-1-YYYY
dbid:       0001
podname:    NA
```

We can observe that the two time series originate from different hosts while still sharing the same dbid. This is a result of the system's complexity described earlier in Section 2.4.1. The pod name is set to NA since these groups were not applicable to the specific metric during the querying process. The fact that database cores, each with its own pod, can change host or spin up and down over time contributes to the fragmented nature of the collected time series data. An illustrative example on how the resulting fragmented time series can look like, can be seen in Table 4.1. The table presents data from six time series, all from the same metric. The example also illustrates the different types of pods that can be connected to a database. This includes the core pods, given the annotation *neo4j-core* where the database cores reside, and the backup pods with the annotation *backup*, which are used for backing up a database. Some data cannot be connected to a specific pod, resulting in a time series with the pod name *NA*.

In contrast to the previous example where the metric could be grouped by *host*, *dbid* and *podname*, there are metrics which can only be grouped by host, meaning they cannot be directly linked to a database. The hosts form the lower level of the system, meaning there can be no pods not related to any host. This provides a challenge of connecting time series from metrics that lacked information regarding the database they belonged to. Our proposed solution to this challenge is described in section 4.2.4.

4.2.3 Database Selection

When querying for data, an assumption was made that the data from different databases belonged to the same distribution. Therefore, in order to obtain a larger data set for training and testing the anomaly detection models, we decided to query data from several database instances. Initially, the number of databases belonging to orchestra 1 were 124. However, this provided us with to much data to query so the number of databases was narrowed down to 30. All the databases belonging to orchestra 1 were part of the AuraDB Professional plan. As the objective of the thesis is to predict fault intolerance when a database is operating normally, we opted to select databases based on how long they had been active. When a new database is first created, it may affect the metric values as it takes a while for the database to stabilise. To exclude data from these initial events, the databases were sorted according to how long they had been active, and the 30 databases with the longest lifetime were chosen. The selected databases have varying instance sizes, ranging from 1GB to 64GB of RAM. Further investigation showed that one of the databases had been paused during the examined interval, and was therefore excluded from our experiments. As a result, we ended up with a total of 29 databases to process.

4.2.4 Data Preprocessing

With the data now locally available, our next step was to further reduce the number of metrics before proceeding with anomaly detection. As mentioned previously, there were several issues that first needed to be addressed. Firstly, we had to tackle the problem of labelling the time series with no dbid connected to them. Secondly, we needed to address the problem of the varying number of time series for a specific database and metric over time as well as the fragmented characteristics of the time series.

Creating Time Series from Metrics without dbid

When querying the data, the collected time series could be associated to one or several of the different groups: host, database or pod. While the groups *database* and *pod* could include the dbid, the host did not always contain this information. This was expected since a host can encompass multiple pods connected to different databases, and the pods themselves can move between different hosts. Thus, having database id as part of the host name would be impractical. An example of a metric that did not keep information on what databases that were related to it was *kubernetes.kubelet.pod.worker.duration.sum*. For this metric we would only have a number of time series with information regarding what hosts they were queried from.

In order to utilise the metrics lacking information regarding dbid, we needed to associate each databases with a set of hosts on which they were active. This was achieved by studying the time series from metrics that included both the host name and dbid. For each database, the time series were sorted based on the host from which they were queried. Once the time

	TS1	TS2	TS3
host	gke-prod-1-000X	gke-prod-1-000Y	gke-prod-1-000Z
dbid	0001	0001	0001
podname	neo4j-core-0001-0A	neo4j-core-0001-0B	NA

	TS4	TS4	TS6
host	gke-prod-1-00XX	gke-prod-1-00YY	gke-prod-1-00ZZ
dbid	0001	0001	0001
podname	neo4j-core-0001-0C	neo4j-core-0001-0D	backup-0001-0A

time	TS1	TS2	TS3	TS4	TS5	TS6
t_0	0.0	-	0.0	0.0	0.0	0.0
	0.0	-	0.0	0.0	0.0	0.0
	0.0	-	0.0	0.0	0.0	0.0
	0.0	-	0.0	0.0	0.0	0.0
	0.0	-	0.0	0.0	0.0	2152.31
	0.0	-	0.0	0.0	0.0	0.0
	22.81	-	22.81	0.0	0.0	0.0
	0.0	-	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0
	-	0.0	-	0.0	0.0	0.0
	-	0.0	-	0.0	0.0	0.0
	-	0.0	-	0.0	0.0	0.0
	-	0.0	-	0.0	0.0	0.0
	-	0.0	-	0.0	0.0	0.0
	1850.54	1850.54	-	1850.54	0.0	0.0
	1850.54	1850.54	-	1850.54	0.0	-
	-	0.0	-	0.0	0.0	-
	-	0.0	-	0.0	0.0	-
t_n	-	0.0	-	0.0	273.66	-

Table 4.1: Illustrative example showing six time series from the metric *kubernetes.io.read_bytes*, all which are connected to the same database with id 0001. Pod names and hosts related to the specific time series is given. Missing values are denoted as "-"

series were sorted, they were merged and saved to different files, where each file only contained time series queried from the same host and dbid. In Table 4.2 we can see how time series from host *gke-prod-1-000X* and dbid *0001* were collected together.

In the Table 4.2, it can be observed how the data points are limited to specific time intervals. This is due to the fact that the time series only contains data when the pod is active on a host. Looking at these time series, we can determine the periods during which a database is active on a specific host. Using this information, we can generate new time series for the metrics lacking information about the dbid. This process is explained through Figure 4.3. The time series from the specific metric that we wanted to connect to a dbid was copied. From the merged time series that was the result of the process shown in 4.2, we used the *has data* column to identify the time indices during which the database was active on that specific host. If the value of *has data* was 0, the row's value was excluded from said copy. After having excluded these values we had a new time series that was representative for the specific metric and database.

Handling Missing Values and Outliers

As outlined in Section 3.4.1, there are various approaches to handling missing values. Given that a metric missing one or several values can potentially be an indication that the database the metric describes is about to lose fault tolerance, we opted not to fill the missing values with a default or arbitrary value. Since we had access to a substantial data set, even if some values were missing, we would still have sufficient data for our experiments. This resulted in a time series as illustrated in figure 4.1, where missing values were left empty. The same argument can be made regarding outlier data points, as they too can be an indication concerning the state of the database. Hence, no outliers were removed from the data sets.

Aggregating the Time Series Data

Before utilising the time series data for analysis, we made the decision to further reduce and aggregate our data. As depicted in Figure 4.1, there are different types of pods represented in the data set; primarily cores and backups, but there can also be data not associated to any pod. Given our research objective of predicting fault intolerance, which refer to a core pod going down, we opted to only include data from core pods. Data from backup pods and pods not associated to a specific database was therefore disregarded. Some metrics could not be grouped on pod name, resulting in the time series having the pod names being set to *NA*. For these metrics, all time series were considered and used for anomaly detection. To ensure we only had one time series per database and metric, we performed data aggregation in a convenient manner. Specifically, we calculated the average of all available time series data for each point in time, per database and metric, resulting in one single time series.

4.2.5 Further Reduction of Metric Count

Since we now had the time series for the metrics saved on a suitable format, we were still faced with a high number of metrics. To narrow down the scope, we made the decision to

	TS1	TS2	TS3
metric	container.cpu.system	neo4j.transaction.rollbacks	kubernetes.io.read_bytes
dbid	0001	0001	0001
podname	neo4j-core-0001-0E	neo4j-core-0001-0E	NA

	TS4	TS4	TS6
metric	jvm.heap_memory	containerd.mem.cache	neo4j.transaction.tx_size_heap.count
dbid	0001	0001	0001
podname	neo4j-core-0001-0E	neo4j-core-0001-0E	neo4j-core-0001-0E

time	TS1	TS2	TS3	TS4	TS5	TS6	has data
t_0	2470568.14	4279.25	0.0	162005910.0	45659750.4	432.75	1
	2267125.72	4285.25	13.2	162303648.0	45659750.4	433.25	1
	2463920.82	4297.25	0.0	160784824.0	45659750.4	433.75	1
	-	-	0.0	-	-	-	1
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	-	-	-	-	-	-	0
	2597467.84	1008.75	0.0	162841248.0	-	242.0	1
	10352518.61	1014.25	0.0	163784272.0	45686784.0	242.5	1
	10143984.33	1020.25	0.0	161018500.0	45686784.0	243.0	1
	9811915.01	1026.25	0.0	164899958.0	45666508.8	243.5	1
t_n	10150722.33	1032.25	0.0	163611446.0	45632716.8	244.0	1

Table 4.2: Illustrative example showing six time series from the same host *gke-prod-1-000X*, all which are connected to the same database with id 0001. Missing values are denoted as "-".

	TS1
metric	kubernetes.kubelet.pod.worker.duration.sum
dbid	NA
podname	NA

	TS1 for dbid 0001
metric	kubernetes.kubelet.pod.worker.duration.sum
dbid	0001
podname	NA

time	TS1	has data	*time*	TS1 for dbid 0001
t_0	158465982.55155748	1	*t_0*	158465982.55155748
	158468921.16411304	1		158468921.16411304
	158471806.52155754	1		158471806.52155754
	158474314.00066864	1		158474314.00066864
	158476484.04189083	0		-
	158479531.2624464	0		-
	158481879.8714464	0		-
	158484205.97222412	0		-
	158486860.0660019	0		-
	158489579.06711298	0		-
	158491743.23955742	0		-
	158493944.63633522	0		-
	158493944.63633522	0		-
	158497326.02989075	0		-
	158500337.03477967	0		-
	158504770.67000186	1		158504770.67000186
	158507204.13500193	1		158507204.13500193
	158510325.085113	1		158510325.085113
	158513013.07166857	1		158513013.07166857
t_n	158515644.0610019	1	*t_n*	158515644.0610019

Table 4.3: Illustrative example on how time series for metrics without dbid were processed and connected to a specific database. The original time series TS1 is copied and data points in the rows where there was no data for any of the metrics containing the host and database specified (has data column) with were removed. Missing values are denoted as "-". How the *has data* column is put together is illustrated through Table 4.2.

perform analyses aimed at further reducing the number of metrics. As a result, we compiled two lists of metrics: one based on domain knowledge and the other on statistical analysis.

Selecting Metrics Based on Domain Knowledge

As mentioned in Section 1.2, a common approach to selecting monitoring metrics is by leveraging domain knowledge of the system. In order to obtain such a selection of metrics, we consulted our advisory at Neo4j and requested their input on identifying metrics that correlate with the event of a database becoming fault intolerant, as outlined in research question **RQ1**. The resulting list can be viewed in its entirety in table 4.4.

Selecting Metrics Based on Statistical Analysis

The process of identifying relative metrics was conducted iteratively, by applying different requirements for the data to meet. Firstly, we took a look at missing values. As a missing value could potentially indicate an anomaly, discarding metrics with missing values should be done with care. However, there were metrics that continuously reported no values for extended periods or had entirely missing time series for certain databases, indicating that no values were reported for those specific metric and databases. Since our aim was to find a unified anomaly detection model applicable to all databases, we decided to discard metrics that did not have data for all databases being studied. This resulted in 307 remaining metrics to further examine.

One method described in Section 1.6 is the *Filter method*, where the selection of metrics is based on ranking them according to some criteria. In order further reduce the number of metrics, our objective was to evaluate and rate them using this criteria. The chosen criteria should indicate potential deviation in behaviour between the period before a database becomes fault intolerant and the periods when the database was running with all three cores, demonstrating fault tolerance. In order to rate the metrics in this way, we made the following assumption:

> *Metrics containing data which behaviour diverge leading up to the event of fault intolerance, in comparison to a randomly selected time interval, holds information that can be used to predict when a database becomes fault intolerant.*

To identify changes in the behaviour of the data we used standard deviation. We examined the time periods preceding the loss of fault tolerance for the databases and designated them as one category. For the other category, we randomly selected intervals from clean data. All time intervals span over 30 minutes. During our previous analysis of the time series, we observed that the time series could vary in behaviour over longer time periods. Therefore, comparing the standard deviation of the intervals preceding fault intolerance events to the intervals from clean data could potentially overshadow the small changes in the data, as longer trends may dominate. To address the problem of comparing behaviour between intervals over shorter time periods, we divide each interval into two parts: the first being 20 minutes, and the second one 10 minutes. An illustrative example of such an interval can be seen in 4.1.

The reason for dividing the intervals in this manner was based on the assumption that the

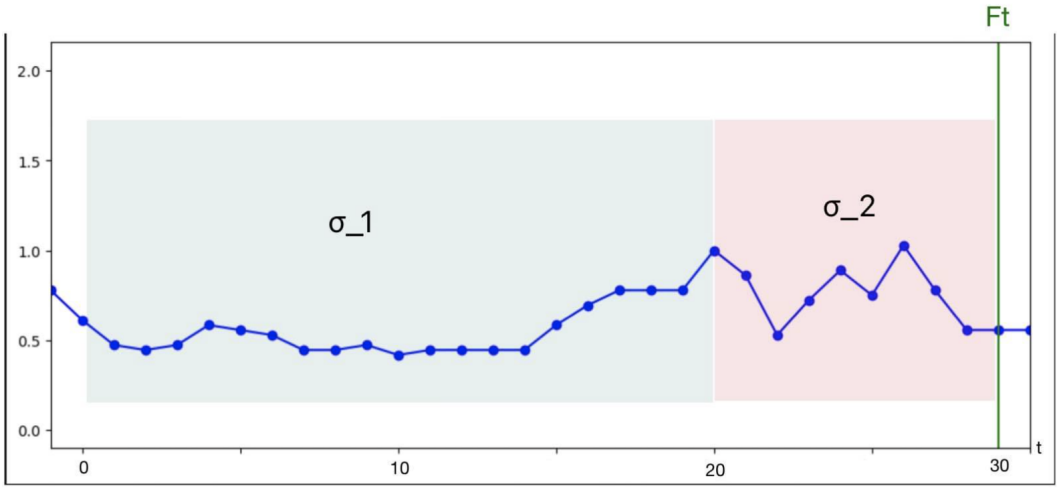


Figure 4.1: An illustrative example of a time interval leading up to a database loosing fault intolerance at time *Ft*.

data would have a higher likelihood of exhibiting varying behaviour when approaching fault intolerance. This assumption relied on the data coming from a metric that could be correlated to the fault intolerance events. After splitting the interval into two parts, we calculated the standard deviation for each part, denoted as σ_1 and σ_2,respectively, and calculated the percentage difference between the two resulting in a ratio. This ratio was calculated for all intervals, both in the set of intervals preceding fault intolerance and the clean set. The average ratio of the two sets were calculated and compared them by determining the percentage difference between the averages. A low value indicated a smaller difference in how the data behaved between the two sets, while a higher value indicated a larger difference in behaviour. We conducted this analysis for all metrics and picked the 20 metrics with the highest percentage difference. The resulting list can be viewed in Table 4.5.

4.3 Anomaly Detection

Now that we had the lists of metrics, we could proceed with designing the anomaly detection pipeline. Since we had significantly reduced the number of metrics, it was now feasible to query for a larger amount of data. To ensure sufficient data for training, testing, and validating our anomaly detection models, we made the decision to query data for a time period of one month. This extended time frame would provide us with a substantial data set to work with.

kubernetes.cpu.usage.total
neo4j.transaction.active
neo4j.transaction.started
neo4j.transaction.rollbacks
neo4j.transaction.tx_size_heap.quantile
neo4j.db.query.execution.latency.millis.quantile
neo4j.check_point.total_time
haproxy.backend.session.current
haproxy.backend.errors.con_rate
jvm.heap_memory
jvm.heap_memory_max
neo4j.page_cache.page_faults
neo4j.vm.gc.time.g1_young_generation
neo4j.vm.gc.time.g1_old_generation
kubernetes.io.write_bytes
kubernetes.io.read_bytes
kubernetes.network.tx_bytes
kubernetes.network.rx_bytes
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage

Table 4.4: Metrics selected based on domain knowledge.

neo4j.transaction.tx_size_heap.count
neo4j.transaction.committed
container.uptime
neo4j.transaction.tx_size_native.count
neo4j.page_cache.unpins
jvm.gc.cms.count
container.memory.cache
neo4j.vm.memory.pool.compressed_class_space
neo4j.vm.memory.pool.metaspace
jvm.gc.metaspace_size
jvm.non_heap_memory
jvm.loaded_classes
kubernetes.memory.cache
containerd.mem.cache
container.memory.oom_events
containerd.mem.current.failcnt
gcp.networking.pod_flow.rtt.sumsqdev
container.cpu.limit
kubernetes.io.read_bytes
container.io.read

Table 4.5: Metrics selected based on statistical analysis.

After preprocesed the new data using the same method as we described in Section 4.2.4 for metric selection, we observed a few data points for some of the metrics and databases that

had *not a number* (NaN) values. These NaN values indicated that there was no values reported for these specific metrics at that given time. Since some anomaly detection models require for each data point to be a value, we had to replace these NaN values with numeric ones. We decided to replace them with 0, as this would describe the behaviour of no reported values.

The available data set had to be divided into a training set and a testing set. This division involved splitting the data set into two parts: one for training and the other for testing its performance. This resulted in data corresponding two weeks was used for training and the other two weeks for testing. To obtain clean or normal data for training the models, a certain part of data points were excluded. The data points corresponding to times when the database was experiencing fault intolerance, as well as data points within 30 minutes preceding and following loss of fault tolerance was removed. By removing these periods, we aimed to focus the models training on normal operational states of the database.

4.3.1 Anomaly Detection using HBOS

One of the anomaly detection models used in our pipeline was the HBOS method, as described in Section 3.3.3. For the implementation of this model, we used a module called PyOD, (Python Outlier Detection). As previously mentioned, the number of bins used in the HBOS model can impact its effectiveness. Since we were working with data from multiple time series, each with distinct characteristics, we adopted a strategy where we constructed a HBOS model using a range of histogram widths. This approach allowed us to generate multiple scores which could be aggregated to create a single, stable model. This approach would thus reduce the risk of overfitting, i.e. making the model good at predicting values for the training set but bad for making predictions on the test set, and enhance the prediction accuracy. In our case, we developed 10 HBOS models using the range of bins [5, 10, 15, 20, 35, 30, 50, 60, 75, 100]. From these models, 10 predictions were made and normalised to zero-mean and unit variance before calculating the average prediction, which served as our final prediction

When conducting experiments with the HBOS method, we discovered that our initial assumption, stating that time series originating from different database instances would belong to the same distribution might not hold true. Therefore, we opted up train the HBOS model using two different approaches. The first method involved training a HBOS model for each individual database, to use for anomaly detection on that specific database. The second method instead trained one single HBOS on data from all available databases and applying said model to predict the anomalies for every database.

After fitting the HBOS model to the training data, we proceeded to make predictions on the test data. As outlined in Section 3.3.3, the HBOS will generate outlier scores for each data point in the test set during the prediction process. By setting a threshold to determine what outlier score should be classified as an anomaly and which should be considered normal, a list of indices corresponding to data points classified as anomalous was created. The threshold was set based on the number of fault intolerant events observed in the training data set. This number was multiplied by 30, as we aimed to account for the assumption that multiple data points in the period preceding a fault intolerance event would exhibit anoma-

lous behaviour. Finally, this was dividing it by the total length of the training time series. This provided us with a ratio regarding how frequent fault intolerance events occurred. The produced list of indices of anomalous points, was later used in the evaluation process.

Using HBOS with Moving Average

Since the HBOS method consider each data points individually and does not take the values of surrounding data points into account, the time series aspect of the data in which we train our model is lost. As we believed that the data points were not independent of their surroundings, we wanted to preprocess the data in a way that incorporated information about how each data point behaved relative to its surroundings, before using it on our model. To achive this, we used two types of moving averages: Simple Moving Average (SMA) and Exponential Moving Average (EMA), as described in Section 3.3.3. For each time series, we created two new time series: one based on SMA and another on EMA. First, the moving average was calculated for each data point, then the difference between the moving average the actual value was calculated. A large difference would indicate that the data point deviated from its surrounding points. To calculate the SMA, a window size of 30 minutes was used. For the EMA, a smoothing level of 0.065 was used, which was determined based on the equation in 3.4.2 and a window size of 30 minutes.

The resulting time series consisted of the calculated differences between the SMA and the original data, and the EMA and the original data. These time series were used to train the HBOS models, following the same methodology as described above for the original time series.

4.3.2 Anomaly Detection using LSTM Autoencoder

The second method tested for detecting anomalies was an LSTM autoencoder, as described in section 3.3.2. An LSTM autoencoder can be used to capturing dependencies and patterns over longer time series, making it suitable for anomaly detection.

Architecture

To implement our LSTM autoencoder, we used Keras. The model summary of the architecture used is depicted in table 4.6. The architecture was derived from attempting different architectures and hyperparameters, and studying the result. The architecture is as follows: two LSTM layers acting as encoders, followed by a RepeatVector. Then, we had two more LSTMs acting as decoders, and finally a TimeDistributed layer. The output shapes of the different layers can be viewed in the model summary. As we studied the data in intervals of 5 and the number of metrics studied was 19 or 20 depending on metric set, the input shape was (1, 5, 19) or (1, 5, 20). The loss function used to calculate the error was MSE and the optimiser used was Adam. Since we designed and used the LSTM autoencoder for anomaly detection, the input sequence was also the target sequence in the training process. This means that we are training the model to detect changes in the data, not to predict future values.

After fitting the model to the training data, we proceeded to make predictions on the test

Layer (Type)	Output Shape	Param #
encoder_1 (LSTM)	(None, 5, 32)	6656
encoder_2 (LSTM)	(None, 16)	3136
encoder_decoder_bridge (RepeatVector)	(None, 5, 16)	0
decoder_1 (LSTM)	(None, 5, 16)	2112
decoder_3 (LSTM)	(None, 5, 32)	6272
time_distributed (TimeDistributed)	(None, 5, 19)	627

Table 4.6: Summary of the LSTM auto-encoder used.

data. The LSTM autoencoder will try to encode and decode a given sequence, and by comparing the predicted output to the target and observing the deviation, we can detect rare patterns in the data. The Mean Average Error (MAE) loss was therefore computed for all the predicted outputs and their target values. A small MAE would indicate that the pattern in the sequence had been observed before in the normal training data, while a high MAE would indicate an anomalous pattern. By setting a threshold regarding what MAE would count as normal and what should be considered an anomaly, we could label the data point indices as either predicted to be normal or predicted to be anomalous. The threshold was determined in the same way as we did when setting it for the anomaly detection using HBOS, as described in Section 4.3.1. This involved counting the number of fault tolerance events in the training data and then multiplying it by 30 and dividing it by the total number of data points.

4.4 Evaluation Method

As our research question **RQ2** states, we want to be able to predict when the database is going to become fault intolerant. Now that we had the models produce predictions that classified the data points as either normal or anomalous, we needed a way to compare these points to the events where a database became fault intolerant. There are several aspects which need to be taken into consideration when developing such an evaluation model. It is important to define the time horizon within which we can accurately predict fault intolerance. Additionally, we need to establish clear criteria for determining true positives, true negatives, false positives, and false negatives in order to evaluate the performance of our prediction models.

When considering the time aspect, our aim was to investigate not only if fault intolerance could be predicted, but also how long in advance. However, to ensure meaningful results, we decided to analyse multiple data points simultaneously when making classifications close to the fault intolerance events. To achieve this, we employed time windows in our evaluation, extending up to one hour before a database became fault intolerant. Each time window had a length of 10 minutes, equivalent to 10 data points. Given the data points proximity to a fault intolerance event, we desired these time windows to be classified as anomalous in order to trigger an alert. As a consequence, these time windows can only be classified as either true positives or false negatives. All data points outside of the studied time windows will be classified as either false positives or true negatives.

In our work we decided to investigate the following time intervals before a fault intoler-

ance event occurred; 0-10, 5-15, 10-20, 15-25, 20-30, 25-35, 30-40, 35-45, 40-50, and 45-55. Considering the time intervals, such as 0-10 minutes and 45-55 minutes, implies different expectations regarding the system's ability to alert for databases becoming fault intolerant within those respective time frames. In the 0-10 minute interval, we anticipate the system to detect and alert promptly for fault intolerance events occurring within the next 10 minutes. On the other hand, within the 45-55 minute interval, the system should be capable of predicting and alerting for fault intolerance events expected to happen between 45 and 55 minutes from the present moment.

The different time windows can be classified as follows:

- **True Positives:** All time windows which have at least one anomaly within them, and lies within the specified time from a fault intolerance event.
- **False Negatives:** All time windows with no anomalies, and which lies within the specified time from a fault intolerance event.
- **True Negatives:** All data points not predicted as anomalous, and which do not lie in the specified time window currently studied.
- **False Positives:** All data points predicted as anomalous, and which are not in the specified time window currently studied.

Chapter 5
Result

The purpose of this chapter is to present the result of the anomaly detection and provide necessary information to substantiate the following discussion. We begin by providing the results from the HBOS method, followed by the results obtained from the LSTM autoencoder. It is important to note that the results presented in this section represent only a part of the information gathered from the conducted experiments. We have selected the findings that showed the best results or provided us with insights which could be used in the discussion and the validation of the thesis.

5.1 Data Attributes

In the previous chapter we described the process we went through in order to get the data that we used to train and test our models on. We first started by reducing the metric count to 19 ones chosen by a developer at Neo4j with domain knowledge of the system, and 20 ones selected through the metric reduction process described in section 4.2. Note that there is one metric overlapping the two sets of metrics, therefore the total number of metrics studied is 38. The listed metrics are shown in the Tables 4.4 and 4.5. From these metrics, data for a total of four weeks was queried and aggregated to become one time series per metric and database, as described in section 4.2.4.

In order to get a better understanding of the results, a brief description of the attributes of the final time series is given. For all 29 databases there was a total number of 194 instances in which a database lost fault tolerance during the two weeks used for training, and 137 times during the two weeks that was used for testing. The distribution of fault intolerance events between the different databases did not align: one database could have become fault intolerant only 3 times during the training period and 3 times during the testing period, while another database became fault intolerant 23 times during the training period and 17 times during the test period. Observe that the training data did not include any of the fault intolerance data during training, this is only to illustrate how fault intolerance was distributed. The number of times the databases became fault intolerant could also differ a lot between the two time periods used for testing and training of a database. For example, there was one database that became fault intolerant 10 times during the training period and only one time during the testing period. In general, there were more instances of databases becoming fault intolerant during the training period than during the testing period. Additionally, only six databases exhibited a higher count of fault intolerance events in the test data compared to the training data.

Four example of time series from different metrics can be seen in the figs. 5.1 to 5.5. From these figures we can see that the time series behaves differently depending on the metric. Some of them reports no values, or very low, but can suddenly spike before going back to reporting low values again, this behaviour can be seen for the *haproxy.backend.errors.con_rate* metric in figure 5.1. Other metrics reports values that fluctuate a lot, as is the case for the values coming from the metric *jvm.heap_memory* seen in figure 5.2. Additionally, it can be observed that for some metrics the data is spiking but continues to report the higher values for some time before going back to report lower values again, these plateau like shapes can be seen in the time series coming from the metric *neo4j.page_cache.page_faults* shown in figure 5.3. Other time series shows some kind of combination of the two last mentioned, they are fluctuating but stays at plateaus at different stages for a while, such as the time series coming from neo4j.transaction.rollbacks shown in figure 5.4. Another thing worth noticing is that the values that are reported from the metric can differ a lot between databases, for example for metric *haproxy.backend.session.current* shown in figure 5.5 we can see that the values lies between 0 and approximately 4 for one database, while it for another varies between 0 and approximately 40. Last it is worth pointing out that the trends in the data can change with time, as for metric *neo4j.page_cache.page_faults*, data from the two first weeks have values lower than data from the two last weeks, this trend is clearly shown in Figure 5.3.

In Figures figs. 5.1 to 5.5 not only the time series for the different metrics are shown, but also the periods where the database has been fault intolerant. It is hard to see any correlation between the fault intolerance events and the time series data just by looking at them. In order to better get a view over how the data changes close to a fault intolerance event a few close ups are shown in figure 5.6. The data comes from the metric *haproxy.backend.error.con_rate*. The reason for why this metric was chosen for close ups was because it looked like there could be a correlation between the spikes in a time series to when a database becomes fault intolerant. Worth noticing is that the two close ups presented highest up in the figure are overlapping, meaning that the second spike in the first time series is the same as the second one in the second time series which also shows a fault intolerance event in it. For the third spike in these two time series at the top, there was no fault intolerance reported from the database. The following three time series in the figure are not overlapping any other.

The figures presented above are only a few of the 1102 (38 metrics for each of the 29 databases) time series generated through the metric selection and the data preprocessing. Counting the time series derived when applying SMA and EMA the number is even higher. In the next section we will provide the results from using these time series in the anomaly detection process.

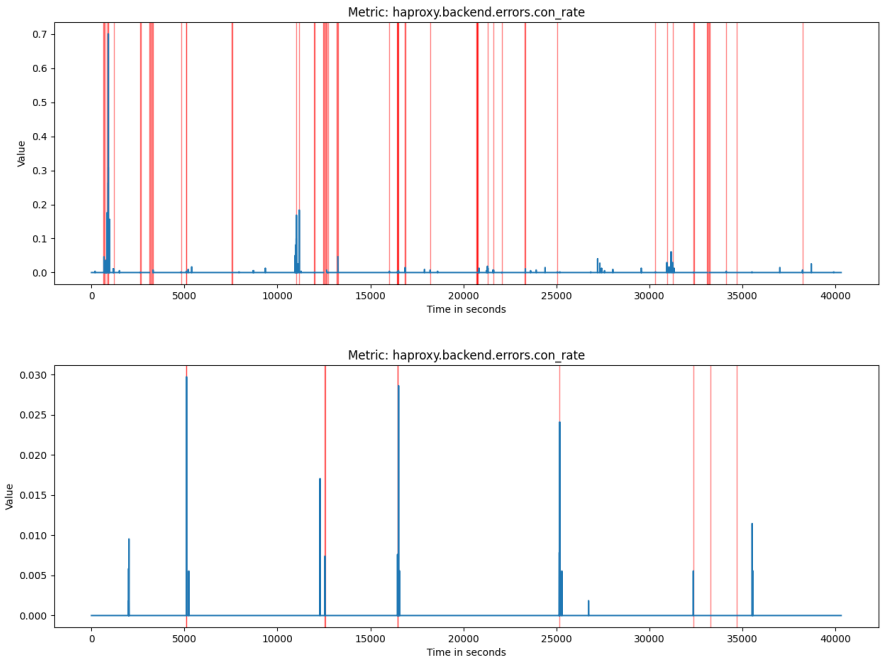


Figure 5.1: Two time series from different databases showing the values of *metric haproxy.backend.errors.con_rate* in blue and fault intolerance in red. Note the contrast in frequency of fault intolerance between the databases.

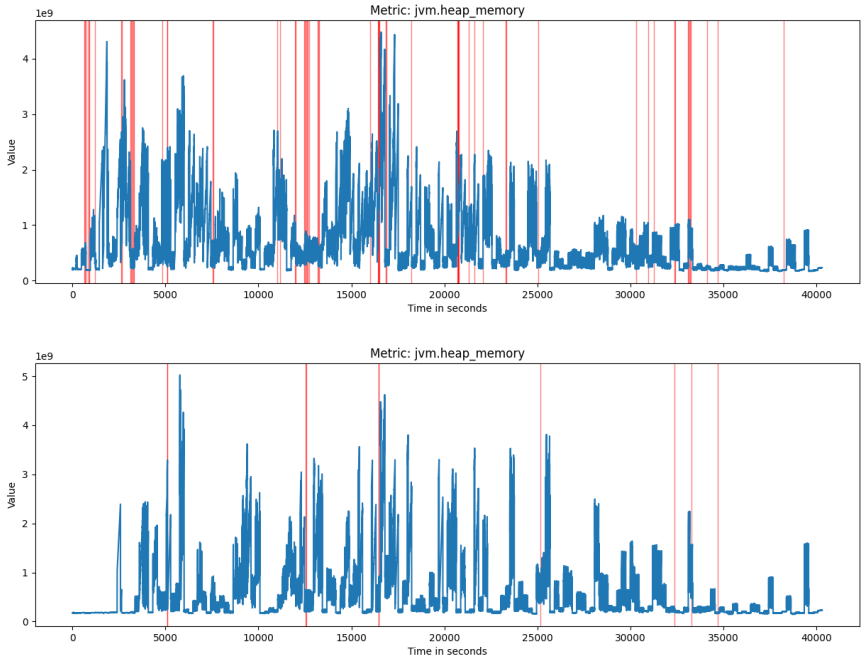


Figure 5.2: Two time series from different databases showing the values of metric *jvm.heap_memory* in blue and fault intolerance in red. Note the higher level of fluctuation in the data.

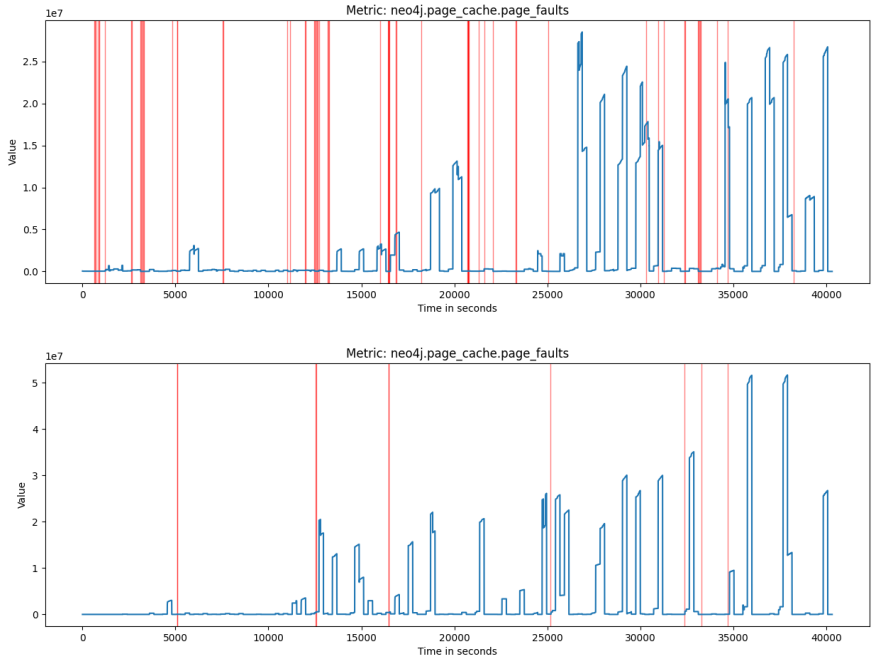


Figure 5.3: Two time series from different databases showing the values of *neo4j.page_cache.page_faults* in blue and fault intolerance in red. Note the uneven distribution of the data between the first half and the second half.

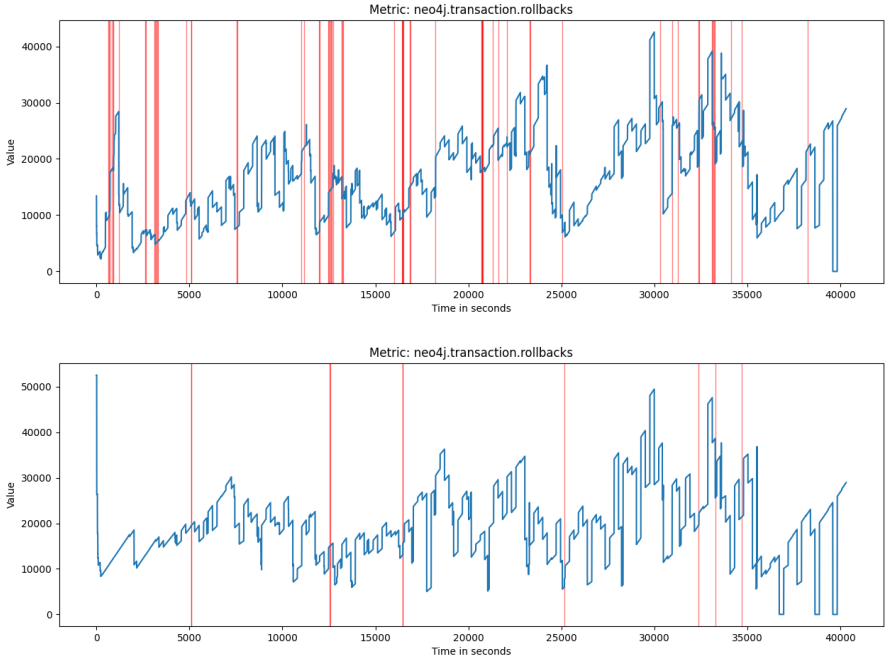


Figure 5.4: Two time series from different databases showing the values of metric *neo4j.transaction.rollbacks* in blue and fault intolerance in red. The lower level of fluctuation in the data.

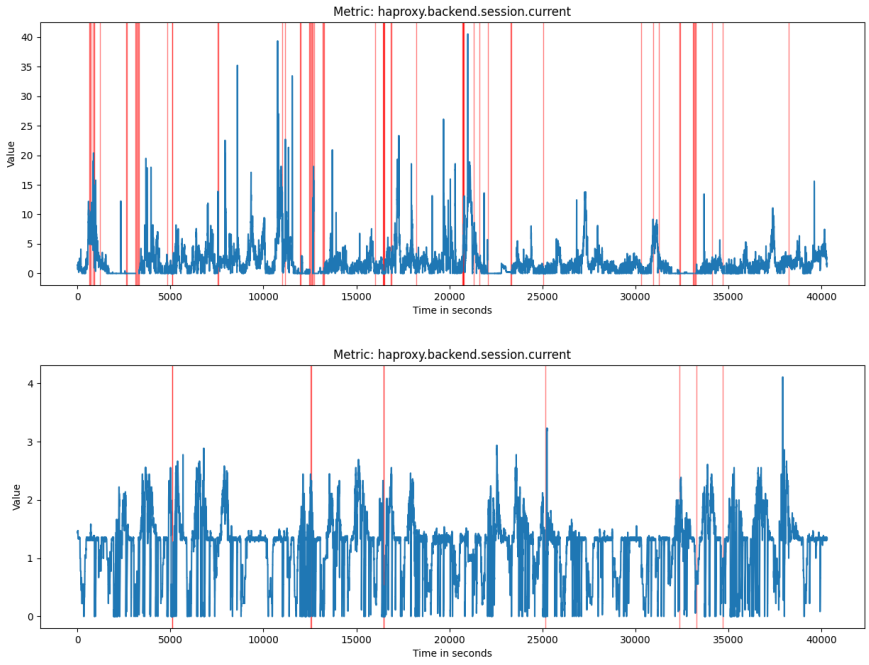


Figure 5.5: Two time series from different databases showing the values of *haproxy.backend.session.current* in blue and fault intolerance in red. Note the substantial disparity in the range of values between the two databases.

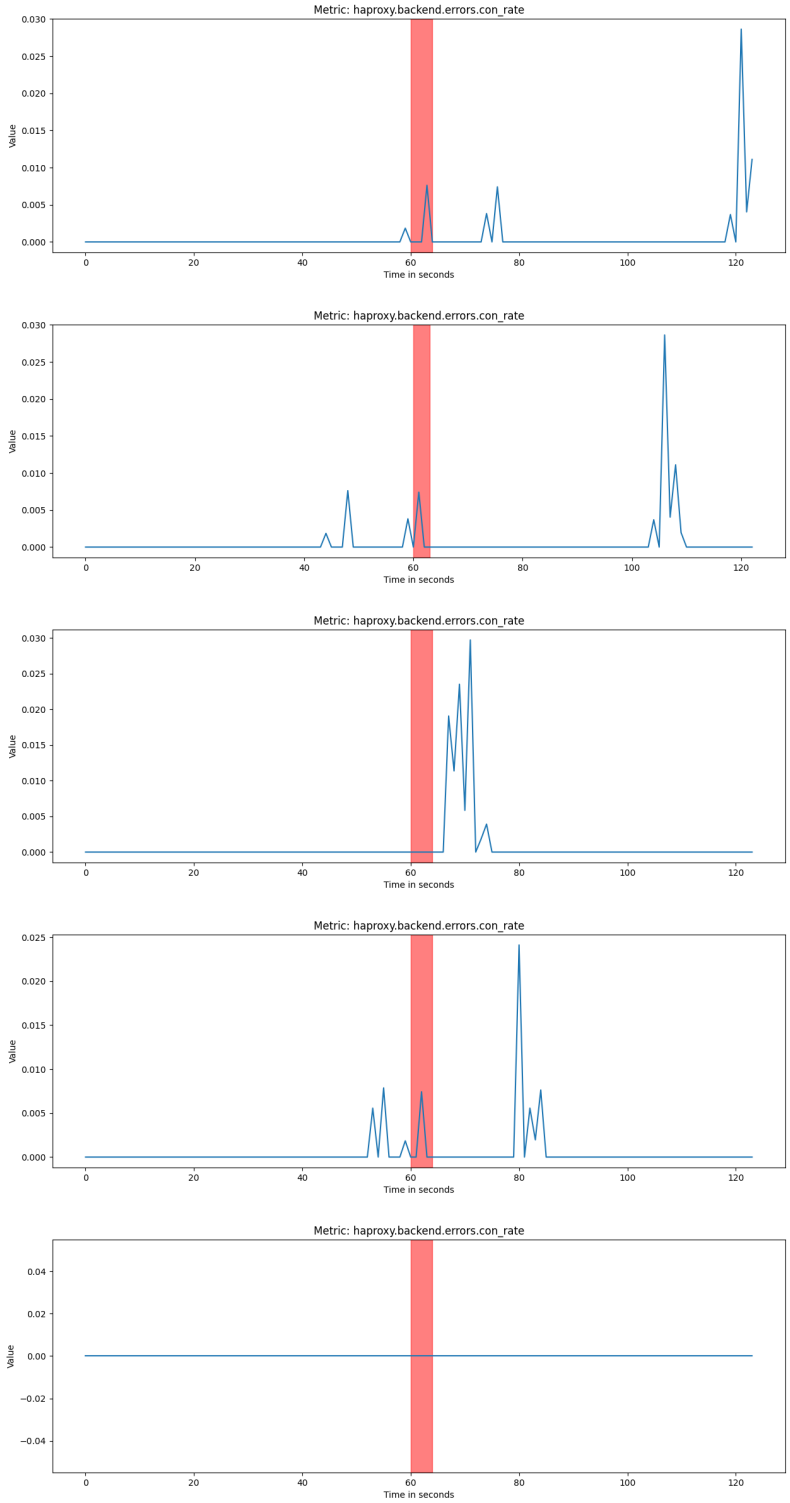


Figure 5.6: Four time periods of one specific database showing the values of *haproxy.backend.errors.con_rate* in blue and fault intolerance in red. Note how the values of the metric correlates to the fault intolerance instances and the overlap of the top two graphs.

5.2 Results for Fault Intolerance Predictions

In this section, results from the evaluation of the anomaly detection experiments are given. As our research questions **RQ1** and **RQ2** state, we are interested in finding metrics that correlate with databases becoming fault intolerant, and if we can, based on anomaly detection, predict when a database is at risk of becoming fault intolerant. In order to answer these questions we designed an evaluation method to be able to label the predicted anomalies as true positives, false positives, true negatives, and false negatives. It is this process, described in section 4.4, that process that the resulting scores are based on. The models that were designed and built for the anomaly detection were based on the HBOS method and the LSTM autoencoder method. The metrics used for training and testing these models are shown in the Tables 4.4 and 4.5.

5.2.1 Results for Metrics Selected on Domain Knowledge

The results given in this section derives from models trained and tested on metrics selected based on domain knowledge. The list of metrics can be seen in Table 4.4.

Results based on HBOS Anomaly Detection

In Table 5.1, combined results of having trained and tested multiple HBOS models, one for each database are shown. The time series used in this part are the original ones, i.e. not the ones derived from combining SMA and EMA. We can see that all scores are low for all metrics and time intervals. The table also shows results for the multivariate HBOS model, where all metrics are used to build a single model. Worth noticing is that the multivariate model did not perform better than the single best univariate model which was trained on the *haproxy.backend.errors.con_rate metric*.

In contrast to the result described above, the results shown in Table 5.2 comes from training one single HBOS model on the data from all databases; the single HBOS model was then used by all databases to test on. Note that a multivariate HBOS model was not tested for this set up. In the table we can see, again, that all scores are low for all metrics and time intervals. Additionally, we can observe that the performance of the best metric, *haproxy.backend.errors.con_rate*, is worse than for the ensemble HBOS model.

For the time series derived from the process of using moving averages in combination with HBOS as described in Section 4.3.1 we can see the result in Table 5.3. The table only shows the five best performing metrics of each moving average. As we can see, the scores are low for all metrics and time windows. Later we will present the result based on a multivariate HBOS model trained on the three best performing metrics from the metrics selected based on domain knowledge, and the three best metrics based on the statistical analysis.

Metrics	F1-scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.001
neo4j.transaction.active	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
neo4j.transaction.started	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.rollbacks	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.db.query.execution.latency.millis.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.check_point.total_time	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001
haproxy.backend.session.current	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
haproxy.backend.errors.con_rate	0.065	0.026	0.013	0.007	0.007	0.020	0.013	0.013	0.013	0.007	0.013
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.005	0.005	0.000
jvm.heap_memory_max	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.page_cache.page_faults	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_young_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_old_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.write_bytes	0.001	0.001	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.001	0.001
kubernetes.io.read_bytes	0.014	0.018	0.006	0.013	0.009	0.009	0.005	0.005	0.001	0.004	0.004
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.005	0.005	0.002
kubernetes.network.rx_bytes	0.004	0.004	0.004	0.007	0.004	0.005	0.005	0.000	0.000	0.001	0.001
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Multivariate analysis	0.0006	0.0006	0.0006	0.0006	0.0006	0.0006	0.0007	0.0007	0.0006	0.0007	0.0007

Metrics	Recall										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.072	0.094	0.109	0.159	0.159	0.152	0.174	0.145	0.159	0.087	0.130
neo4j.transaction.active	0.094	0.087	0.094	0.094	0.101	0.101	0.109	0.109	0.094	0.094	0.101
neo4j.transaction.started	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.022	0.022
neo4j.transaction.rollbacks	0.116	0.116	0.116	0.123	0.101	0.101	0.101	0.101	0.101	0.101	0.101
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007
neo4j.db.query.execution.latency.millis.quantile	0.080	0.080	0.080	0.080	0.072	0.072	0.072	0.094	0.101	0.101	0.101
neo4j.check_point.total_time	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.058	0.051	0.058
haproxy.backend.session.current	0.065	0.036	0.029	0.029	0.036	0.051	0.051	0.058	0.043	0.043	0.036
haproxy.backend.errors.con_rate	0.072	0.029	0.014	0.007	0.007	0.022	0.014	0.014	0.014	0.007	0.014
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.007	0.007	0.000
jvm.heap_memory_max	0.261	0.261	0.261	0.261	0.275	0.275	0.275	0.275	0.290	0.283	0.290
neo4j.page_cache.page_faults	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.072	0.043	0.043
neo4j.vm.gc.time.g1_young_generation	0.094	0.094	0.094	0.109	0.109	0.109	0.109	0.116	0.123	0.123	0.123
neo4j.vm.gc.time.g1_old_generation	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007
kubernetes.io.write_bytes	0.029	0.022	0.007	0.014	0.014	0.014	0.000	0.000	0.007	0.014	0.014
kubernetes.io.read_bytes	0.080	0.101	0.036	0.072	0.051	0.051	0.029	0.029	0.007	0.022	0.022
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.029	0.036	0.036	0.014
kubernetes.network.rx_bytes	0.058	0.058	0.065	0.101	0.065	0.080	0.080	0.000	0.000	0.022	0.022
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.116	0.116	0.116	0.116	0.116	0.116	0.116	0.116	0.101	0.101	0.101
Multivariate analysis	0.0362	0.0362	0.0362	0.0362	0.0362	0.0362	0.0435	0.0435	0.0362	0.0435	0.0435

Metrics	Precision										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000
neo4j.transaction.active	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.transaction.started	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.rollbacks	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.db.query.execution.latency.millis.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.check_point.total_time	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
haproxy.backend.session.current	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000
haproxy.backend.errors.con_rate	0.059	0.024	0.012	0.006	0.006	0.018	0.012	0.012	0.012	0.006	0.012
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.004	0.000
jvm.heap_memory_max	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.page_cache.page_faults	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_young_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_old_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.write_bytes	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.read_bytes	0.008	0.010	0.003	0.007	0.005	0.005	0.003	0.003	0.001	0.002	0.002
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.003	0.003	0.001
kubernetes.network.rx_bytes	0.002	0.002	0.002	0.004	0.002	0.003	0.003	0.000	0.000	0.001	0.001
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Multivariate analysis	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003

Table 5.1: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one HBOS model per database and combining the results.

Metrics	F1-scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.000	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.001
neo4j.transaction.active	0.003	0.002	0.002	0.002	0.002	0.003	0.003	0.003	0.002	0.002	0.003
neo4j.transaction.started	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.rollbacks	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.db.query.execution.latency.millis.quantile	0.001	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.check_point.total_time	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001
haproxy.backend.session.current	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
haproxy.backend.errors.con_rate	0.036	0.009	0.000	0.000	0.009	0.009	0.000	0.000	0.000	0.009	0.018
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.heap_memory_max	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.page_cache.page_faults	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_young_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_old_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.write_bytes	0.005	0.004	0.000	0.000	0.004	0.004	0.000	0.000	0.000	0.003	0.003
kubernetes.io.read_bytes	0.018	0.024	0.008	0.015	0.011	0.011	0.005	0.005	0.000	0.005	0.005
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.004	0.006	0.002
kubernetes.network.rx_bytes	0.006	0.006	0.002	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001

Metrics	Recall										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.036	0.065	0.087	0.116	0.130	0.116	0.159	0.138	0.130	0.065	0.109
neo4j.transaction.active	0.072	0.043	0.043	0.043	0.043	0.072	0.072	0.072	0.058	0.058	0.080
neo4j.transaction.started	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.000	0.000
neo4j.transaction.rollbacks	0.065	0.065	0.065	0.065	0.043	0.043	0.043	0.043	0.043	0.036	0.036
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.db.query.execution.latency.millis.quantile	0.101	0.101	0.101	0.101	0.094	0.094	0.094	0.094	0.094	0.094	0.094
neo4j.check_point.total_time	0.036	0.036	0.036	0.029	0.029	0.029	0.036	0.036	0.065	0.058	0.065
haproxy.backend.session.current	0.029	0.029	0.022	0.022	0.014	0.022	0.022	0.014	0.022	0.022	0.029
haproxy.backend.errors.con_rate	0.029	0.007	0.000	0.000	0.007	0.007	0.000	0.000	0.000	0.007	0.014
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.heap_memory_max	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.page_cache.page_faults	0.022	0.022	0.022	0.022	0.022	0.022	0.022	0.022	0.022	0.014	0.014
neo4j.vm.gc.time.g1_young_generation	0.029	0.029	0.029	0.029	0.029	0.029	0.029	0.036	0.036	0.029	0.029
neo4j.vm.gc.time.g1_old_generation	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007
kubernetes.io.write_bytes	0.029	0.022	0.000	0.000	0.022	0.022	0.000	0.000	0.000	0.014	0.014
kubernetes.io.read_bytes	0.080	0.109	0.036	0.065	0.051	0.051	0.022	0.022	0.000	0.022	0.022
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.022	0.029	0.036	0.014
kubernetes.network.rx_bytes	0.022	0.022	0.007	0.007	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.094	0.094	0.094	0.094	0.094	0.094	0.094	0.087	0.087	0.087	0.087

Metrics	Precision										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.active	0.002	0.001	0.001	0.001	0.001	0.002	0.002	0.002	0.001	0.001	0.002
neo4j.transaction.started	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.rollbacks	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_heap.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.db.query.execution.latency.millis.quantile	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.check_point.total_time	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
haproxy.backend.session.current	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001
haproxy.backend.errors.con_rate	0.047	0.012	0.000	0.000	0.012	0.012	0.000	0.000	0.000	0.012	0.023
jvm.heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.heap_memory_max	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.page_cache.page_faults	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_young_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.gc.time.g1_old_generation	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.write_bytes	0.003	0.002	0.000	0.000	0.002	0.002	0.000	0.000	0.000	0.001	0.001
kubernetes.io.read_bytes	0.010	0.014	0.005	0.008	0.006	0.006	0.003	0.003	0.000	0.003	0.003
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.002	0.003	0.001
kubernetes.network.rx_bytes	0.004	0.004	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4jcloud.neo4j_disk_usage_sidecar_disk_usage	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table 5.2: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training one single HBOS model on data from all database and then using that model to evaluate the test data from the different databases.

Metrics	F1-scores for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
haproxy.backend.errors.con_rate	0.019	0.006	0.005	0.003	0.003	0.006	0.005	0.003	0.005	0.005	0.005
jvm.heap_memory_max	0.003	0.003	0.003	0.003	0.000	0.001	0.001	0.001	0.001	0.001	0.001
kubernetes.io.read_bytes	0.005	0.006	0.004	0.004	0.004	0.004	0.001	0.001	0.001	0.001	0.001
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.001	0.001	0.000	0.001	0.003	0.004	0.005	0.003
kubernetes.network.rx_bytes	0.005	0.005	0.004	0.005	0.004	0.004	0.004	0.000	0.000	0.001	0.001
Metrics	Recall for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
haproxy.backend.errors.con_rate	0.087	0.029	0.022	0.014	0.014	0.029	0.022	0.014	0.022	0.022	0.022
jvm.heap_memory_max	0.022	0.022	0.022	0.022	0.000	0.007	0.007	0.007	0.007	0.007	0.007
kubernetes.io.read_bytes	0.101	0.116	0.087	0.087	0.072	0.072	0.029	0.029	0.022	0.022	0.022
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.007	0.007	0.000	0.007	0.029	0.036	0.043	0.022
kubernetes.network.rx_bytes	0.116	0.116	0.101	0.123	0.094	0.101	0.094	0.000	0.000	0.022	0.022
Metrics	Precision for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
haproxy.backend.errors.con_rate	0.011	0.004	0.003	0.002	0.002	0.004	0.003	0.002	0.003	0.003	0.003
jvm.heap_memory_max	0.002	0.002	0.002	0.002	0.000	0.001	0.001	0.001	0.001	0.001	0.001
kubernetes.io.read_bytes	0.003	0.003	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.002	0.003	0.001
kubernetes.network.rx_bytes	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.000	0.000	0.000	0.000

Metrics	F1-scores for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.001	0.001	0.001	0.002	0.002	0.003	0.003	0.003	0.001	0.001
haproxy.backend.session.current	0.004	0.003	0.003	0.004	0.003	0.003	0.003	0.004	0.003	0.004	0.003
haproxy.backend.errors.con_rate	0.020	0.008	0.005	0.003	0.003	0.005	0.003	0.003	0.003	0.003	0.003
kubernetes.io.read_bytes	0.006	0.007	0.005	0.005	0.004	0.004	0.002	0.002	0.002	0.001	0.001
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.001	0.002	0.004	0.005	0.006	0.002
Metrics	Recall for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.007	0.029	0.036	0.036	0.036	0.029	0.072	0.072	0.080	0.036	0.029
haproxy.backend.session.current	0.065	0.051	0.065	0.080	0.058	0.043	0.043	0.080	0.072	0.080	0.065
haproxy.backend.errors.con_rate	0.087	0.029	0.022	0.014	0.014	0.029	0.022	0.014	0.022	0.022	0.022
kubernetes.io.read_bytes	0.101	0.116	0.087	0.087	0.072	0.072	0.029	0.029	0.022	0.022	0.022
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.007	0.007	0.000	0.007	0.029	0.036	0.043	0.022
Metrics	Precision for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
kubernetes.cpu.usage.total	0.000	0.000	0.001	0.001	0.001	0.000	0.001	0.001	0.001	0.001	0.000
haproxy.backend.session.current	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
haproxy.backend.errors.con_rate	0.011	0.004	0.003	0.002	0.002	0.004	0.003	0.002	0.003	0.003	0.003
kubernetes.io.read_bytes	0.003	0.003	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001
kubernetes.network.tx_bytes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.002	0.003	0.001

Table 5.3: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. Only the five best performing metrics are shown. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one HBOS model per database and combining the results. The data used was the SMA and EMA time series produced in the data preprocessing phase.

Results Based on the LSTM Autoencoder Anomaly Detection

In addition to the HBOS anomaly detection method, we also evaluated an LSTM anomaly detection method. The results can be seen in Table 5.4. A seen in the table the scores are low for all time windows. In Table 5.5 the same model is used but the threshold for a point to be classified as an outlier is lowered, producing 3 times more outliers than before. Even though a slight improvement can bee seen, the scores are still low.

	Scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
F1-score	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0003	0.0002	0.0005	0.0005	0.0002
Recall	0.0072	0.0072	0.0072	0.0072	0.0072	0.0145	0.0217	0.0145	0.0362	0.0362	0.0145
Precision	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0002	0.0002	0.0001

Table 5.4: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one multivariate LSTM autoencoder model.

	Scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
F1-score	0.0003	0.0003	0.0003	0.0003	0.0002	0.0003	0.0003	0.0003	0.0004	0.0003	0.0002
Recall	0.0797	0.0797	0.0797	0.0652	0.0507	0.0652	0.0652	0.0870	0.0942	0.0870	0.0580
Precision	0.0002	0.0002	0.0002	0.0001	0.0001	0.0001	0.0001	0.0002	0.0002	0.0002	0.0001

Table 5.5: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one multivariate LSTM autoencoder model.

5.2.2 Results for Metrics Selected by Statistical Analysis

The results given in this section derives from models trained and tested on metrics selected based on statistical analysis. The list of metrics can be seen in Table 4.5.

Results based on HBOS Anomaly Detection

In Table 5.8, combined results of having trained and tested multiple HBOS models, one for each database are shown. The time series used in this part are the original ones, i.e. not the ones derived from combining SMA and EMA. We can see that all scores are low for all metrics and time intervals. The table also shows results for the multivariate HBOS model, where

all metrics are used to build a single model.

In contrast to the result described above, the results shown in Table 5.9 comes from training one single HBOS model on the data from all databases; the single HBOS model was then used by all databases to test on. Note that a multivariate HBOS model was not tested for this set up. In the table we can see, again, that all scores are low for all metrics and time intervals. That the performance of the best metric, container.io.read shows has higher scores than it did for the ensemble model accounted for above.

For the the time series derived from the process of using moving averages in combination with HBOS as described in Section 4.3.1 we can see the result in Table 5.10. The tables only shows the five best performing metrics of each moving average. As we can see, the scores are low for all metrics and time windows.

Results Based on the LSTM Autoencoder Anomaly Detection

In the same way as we did train and test an LSTM autoencoder model for the metrics derived from the domain knowledge, we did with the metrics selected based on the statistical analysis. The model trained on these metrics did not succeed in finding any anomalies close enough for our evaluation method to count them in. The fact that the model did not find any anomalies close to the fault intolerance events is illustrated by Table 5.6, where all scores are zero for all time intervals. Again we tried to improve the results by lowering the threshold for a point to be classified as an outlier, which results in 3 times more outliers than before. The results from this change are shown in Table 5.5. Now there are data points classified as outliers that lies close enough to a fault intolerance event. However, the scores are still low.

	Scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
F1-score	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Recall	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Precision	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 5.6: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one multivariate LSTM autoencoder model.

	Scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
F1-score	0.0004	0.0004	0.0004	0.0008	0.0007	0.0006	0.0004	0.0006	0.0005	0.0008	0.0007
Recall	0.0563	0.0563	0.0704	0.1338	0.1127	0.0986	0.0634	0.0986	0.0845	0.1268	0.1127
Precision	0.0002	0.0002	0.0002	0.0004	0.0004	0.0003	0.0002	0.0003	0.0003	0.0004	0.0004

Table 5.7: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one multivariate LSTM autoencoder model.

Multivariate on the Six Best Performing Metrics

From the results we got from the different approaches made with HBOS we trained one multivariate HBOS model based on six of the top performing metrics. The metrics and type of time series chosen is shown in Table 5.11. The time series data used as an input came from either the original time series data, the SMA, or the EMA time series data. One multivariate HBOS model per database is trained and tested, and the combined results from those models can be viewed in Table 5.12.

Metric	Moving Average
haproxy.backend.errors.con_rate	no
kubernetes.network.rx_bytes	no
kubernetes.io.read_bytes	no
container.io.read	no
gcp.networking.pod_flow.rtt.sumsqdev	EMA
containerd.mem.cache	SMA

Table 5.11: The six best performing metrics. The time series data used comes from either the original time series, the EMA or the SMA, depending on which one was used to produce the best results.

	Scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
F1-score	0.008	0.008	0.010	0.010	0.002	0.006	0.006	0.004	0.006	0.006	0.006
Recall	0.029	0.029	0.036	0.036	0.007	0.022	0.022	0.014	0.022	0.022	0.022
Precison	0.005	0.005	0.006	0.006	0.001	0.004	0.004	0.002	0.004	0.004	0.004

Table 5.12: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one multivariate HBOS model per database and combining the results. The six metrics used for the multivariate HBOS model can be seen in Table 5.11.

Metrics	F1-scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
neo4j.transaction.committed	0.000	0.000	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
container.uptime	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_native.count	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.page_cache.unpins	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.cms.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.001
neo4j.vm.memory.pool.compressed_class_space	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.vm.memory.pool.metaspace	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000
jvm.gc.metaspace_size	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.non_heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.loaded_classes	0.001	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.memory.cache	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
containerd.mem.cache	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
container.memory.oom_events	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
containerd.mem.current.failcnt	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
gcp.networking.pod_flow.rtt.sumsqdev	0.003	0.001	0.002	0.002	0.003	0.004	0.003	0.001	0.001	0.001	0.001
container.cpu.limit	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.read_bytes	0.014	0.018	0.006	0.013	0.009	0.009	0.005	0.005	0.001	0.004	0.004
container.io.read	0.007	0.008	0.011	0.007	0.008	0.006	0.001	0.000	0.001	0.006	0.008
Multivariate analysis	0.0009	0.0008	0.0008	0.0005	0.0008	0.0007	0.0005	0.0003	0.0004	0.0004	0.0002

Metrics	Recall										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.077	0.077	0.077	0.077	0.077	0.077	0.077	0.077	0.077	0.070	0.063
neo4j.transaction.committed	0.035	0.028	0.049	0.049	0.049	0.056	0.056	0.056	0.056	0.042	0.049
container.uptime	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014
neo4j.transaction.tx_size_native.count	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028
neo4j.page_cache.unpins	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007
jvm.gc.cms.count	0.028	0.028	0.035	0.035	0.035	0.035	0.042	0.049	0.049	0.049	0.042
container.memory.cache	0.028	0.028	0.028	0.028	0.035	0.035	0.035	0.042	0.042	0.049	0.049
neo4j.vm.memory.pool.compressed_class_space	0.141	0.148	0.162	0.169	0.190	0.162	0.155	0.162	0.169	0.169	0.169
neo4j.vm.memory.pool.metaspace	0.127	0.162	0.169	0.162	0.113	0.099	0.085	0.099	0.113	0.092	0.021
jvm.gc.metaspace_size	0.035	0.028	0.021	0.014	0.021	0.021	0.028	0.021	0.042	0.035	0.042
jvm.non_heap_memory	0.077	0.099	0.070	0.120	0.106	0.092	0.063	0.063	0.077	0.070	0.063
jvm.loaded_classes	0.063	0.042	0.077	0.070	0.063	0.021	0.028	0.028	0.014	0.014	0.007
kubernetes.memory.cache	0.232	0.232	0.225	0.225	0.225	0.225	0.225	0.225	0.225	0.225	0.225
containerd.mem.cache	0.246	0.246	0.246	0.246	0.239	0.239	0.239	0.254	0.254	0.246	0.232
container.memory.oom_events	0.007	0.007	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014
containerd.mem.current.failcnt	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.035	0.035	0.035
gcp.networking.pod_flow.rtt.sumsqdev	0.042	0.021	0.028	0.035	0.049	0.070	0.049	0.021	0.014	0.014	0.014
container.cpu.limit	0.063	0.077	0.070	0.106	0.092	0.099	0.063	0.077	0.035	0.070	0.063
kubernetes.io.read_bytes	0.080	0.101	0.036	0.072	0.051	0.051	0.029	0.029	0.007	0.022	0.022
container.io.read	0.035	0.042	0.056	0.035	0.042	0.028	0.007	0.000	0.007	0.028	0.042
Multivariate analysis	0.0845	0.0704	0.0704	0.0493	0.0775	0.0634	0.0423	0.0282	0.0352	0.0352	0.0141

Metrics	Precision										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.committed	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.uptime	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_native.count	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.page_cache.unpins	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.cms.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.memory.pool.compressed_class_space	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.memory.pool.metaspace	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.metaspace_size	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.non_heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.loaded_classes	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.memory.cache	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
containerd.mem.cache	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
container.memory.oom_events	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
containerd.mem.current.failcnt	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
gcp.networking.pod_flow.rtt.sumsqdev	0.001	0.001	0.001	0.001	0.002	0.002	0.002	0.001	0.000	0.000	0.000
container.cpu.limit	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.io.read_bytes	0.008	0.010	0.003	0.007	0.005	0.005	0.003	0.003	0.001	0.002	0.002
container.io.read	0.004	0.005	0.006	0.004	0.005	0.003	0.001	0.000	0.001	0.003	0.005
Multivariate analysis	0.0005	0.0004	0.0004	0.0003	0.0004	0.0003	0.0002	0.0002	0.0002	0.0002	0.0001

Table 5.8: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one HBOS model per database and combining the results.

Metrics	F1-scores										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.committed	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.uptime	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_native.count	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.page_cache.unpins	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.cms.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.memory.pool.compressed_class_space	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.vm.memory.pool.metaspace	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000
jvm.gc.metaspace_size	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.non_heap_memory	0.000	0.001	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.000
jvm.loaded_classes	0.003	0.001	0.004	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.memory.cache	0.005	0.005	0.005	0.005	0.005	0.004	0.004	0.005	0.005	0.005	0.006
containerd.mem.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.oom_events	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
containerd.mem.current.failcnt	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
gcp.networking.pod_flow.rtt.sumsqdev	0.001	0.000	0.000	0.002	0.002	0.000	0.001	0.001	0.001	0.001	0.001
container.cpu.limit	0.001	0.001	0.001	0.002	0.002	0.002	0.001	0.002	0.001	0.002	0.002
kubernetes.io.read_bytes	0.018	0.024	0.008	0.015	0.011	0.011	0.005	0.005	0.000	0.005	0.005
container.io.read	0.008	0.010	0.013	0.008	0.007	0.005	0.002	0.002	0.005	0.010	0.010

Metrics	Recall										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.007	0.007
neo4j.transaction.committed	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.uptime	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_native.count	0.035	0.035	0.035	0.035	0.028	0.028	0.028	0.028	0.028	0.028	0.028
neo4j.page_cache.unpins	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.cms.count	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007	0.007
container.memory.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.memory.pool.compressed_class_space	0.035	0.035	0.028	0.028	0.028	0.021	0.021	0.042	0.042	0.042	0.042
neo4j.vm.memory.pool.metaspace	0.077	0.106	0.106	0.092	0.049	0.056	0.042	0.049	0.085	0.085	0.021
jvm.gc.metaspace_size	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.non_heap_memory	0.028	0.049	0.021	0.063	0.049	0.049	0.000	0.000	0.000	0.021	0.021
jvm.loaded_classes	0.042	0.014	0.056	0.049	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.memory.cache	0.190	0.183	0.183	0.183	0.183	0.155	0.155	0.169	0.183	0.197	0.211
containerd.mem.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.oom_events	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
containerd.mem.current.failcnt	0.014	0.014	0.014	0.014	0.014	0.014	0.014	0.028	0.035	0.035	0.035
gcp.networking.pod_flow.rtt.sumsqdev	0.014	0.007	0.000	0.028	0.028	0.007	0.014	0.014	0.014	0.014	0.014
container.cpu.limit	0.021	0.035	0.035	0.049	0.049	0.056	0.028	0.049	0.021	0.049	0.049
kubernetes.io.read_bytes	0.080	0.109	0.036	0.065	0.051	0.051	0.022	0.022	0.000	0.022	0.022
container.io.read	0.035	0.042	0.056	0.035	0.028	0.021	0.007	0.007	0.021	0.042	0.042

Metrics	Precision										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.transaction.tx_size_heap.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.committed	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.uptime	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.transaction.tx_size_native.count	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
neo4j.page_cache.unpins	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.gc.cms.count	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
neo4j.vm.memory.pool.compressed_class_space	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.001
neo4j.vm.memory.pool.metaspace	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.001	0.001	0.000
jvm.gc.metaspace_size	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.non_heap_memory	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jvm.loaded_classes	0.001	0.000	0.002	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000
kubernetes.memory.cache	0.003	0.003	0.003	0.003	0.003	0.002	0.002	0.002	0.003	0.003	0.003
containerd.mem.cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
container.memory.oom_events	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
containerd.mem.current.failcnt	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
gcp.networking.pod_flow.rtt.sumsqdev	0.000	0.000	0.000	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000
container.cpu.limit	0.000	0.001	0.001	0.001	0.001	0.001	0.000	0.001	0.000	0.001	0.001
kubernetes.io.read_bytes	0.010	0.014	0.005	0.008	0.006	0.006	0.003	0.003	0.000	0.003	0.003
container.io.read	0.005	0.006	0.007	0.005	0.004	0.003	0.001	0.001	0.003	0.006	0.006

Table 5.9: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training one single HBOS model on data from all database and then using that model to evaluate the test data from the different databases.

Metrics	F1-scores for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.003	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.000
neo4j.vm.memory.pool.metaspace	0.002	0.002	0.002	0.002	0.001	0.001	0.002	0.002	0.002	0.002	0.000
containerd.mem.cache	0.006	0.007	0.006	0.006	0.004	0.003	0.003	0.003	0.003	0.003	0.002
gcp.networking.pod_flow.rtt.sumsqdev	0.003	0.002	0.003	0.002	0.005	0.007	0.005	0.002	0.001	0.000	0.000
kubernetes.io.read_bytes	0.005	0.006	0.004	0.004	0.004	0.004	0.001	0.001	0.001	0.001	0.001
Metrics	Recall for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.127	0.120	0.099	0.106	0.063	0.056	0.056	0.063	0.063	0.035	0.021
neo4j.vm.memory.pool.metaspace	0.092	0.113	0.106	0.092	0.070	0.077	0.092	0.092	0.120	0.106	0.021
containerd.mem.cache	0.176	0.190	0.176	0.176	0.106	0.099	0.099	0.092	0.099	0.077	0.056
gcp.networking.pod_flow.rtt.sumsqdev	0.049	0.035	0.042	0.035	0.077	0.106	0.077	0.035	0.021	0.007	0.007
kubernetes.io.read_bytes	0.101	0.116	0.087	0.087	0.072	0.072	0.029	0.029	0.022	0.022	0.022
Metrics	Precision for SMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
neo4j.vm.memory.pool.metaspace	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000
containerd.mem.cache	0.003	0.003	0.003	0.003	0.002	0.002	0.002	0.002	0.002	0.001	0.001
gcp.networking.pod_flow.rtt.sumsqdev	0.002	0.001	0.001	0.001	0.002	0.003	0.002	0.001	0.001	0.000	0.000
kubernetes.io.read_bytes	0.003	0.003	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001

Metrics	F1-scores for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.003	0.003	0.002	0.002	0.001	0.001	0.001	0.002	0.001	0.001	0.000
neo4j.vm.memory.pool.metaspace	0.002	0.002	0.002	0.002	0.001	0.002	0.002	0.002	0.002	0.002	0.000
containerd.mem.cache	0.006	0.006	0.006	0.005	0.004	0.002	0.004	0.003	0.003	0.002	0.002
gcp.networking.pod_flow.rtt.sumsqdev	0.004	0.004	0.008	0.008	0.008	0.010	0.010	0.008	0.004	0.004	0.004
kubernetes.io.read_bytes	0.006	0.007	0.005	0.005	0.004	0.004	0.002	0.002	0.002	0.001	0.001
Metrics	Recall for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.148	0.141	0.106	0.106	0.063	0.063	0.056	0.077	0.070	0.035	0.021
neo4j.vm.memory.pool.metaspace	0.092	0.113	0.113	0.099	0.070	0.085	0.092	0.099	0.113	0.099	0.021
containerd.mem.cache	0.176	0.176	0.162	0.155	0.106	0.070	0.106	0.092	0.092	0.070	0.056
gcp.networking.pod_flow.rtt.sumsqdev	0.014	0.014	0.028	0.028	0.028	0.035	0.035	0.028	0.014	0.014	0.014
kubernetes.io.read_bytes	0.101	0.116	0.087	0.087	0.072	0.072	0.029	0.029	0.022	0.022	0.022
Metrics	Precision for EMA										
	0-10	5-15	10-20	15-25	20-30	25-35	30-40	35-45	40-50	45-55	50-60
neo4j.vm.memory.pool.compressed_class_space	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
neo4j.vm.memory.pool.metaspace	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.000
containerd.mem.cache	0.003	0.003	0.003	0.003	0.002	0.001	0.002	0.002	0.002	0.001	0.001
gcp.networking.pod_flow.rtt.sumsqdev	0.002	0.002	0.005	0.005	0.005	0.006	0.006	0.005	0.002	0.002	0.002
kubernetes.io.read_bytes	0.003	0.003	0.002	0.002	0.002	0.002	0.001	0.001	0.001	0.001	0.001

Table 5.10: This table shows F1-score, recall, and precision for evaluation of the anomalies found depending on how long time before the prediction of a fault intolerance event is going to be made. Only the five best performing metrics are shown. The time windows evaluated are 10 minutes long and range between 0 minutes up to 50 minutes before a fault intolerance event occurred. The results comes from training and testing one HBOS model per database and combining the results. The data used was the SMAand EMA time series produced in the data preprocessing phase.

Chapter 6
Discussion

This chapter serves several purposes. Not only will it disclose and discuss the results presented previously, but it will also aim at providing the reader with a reflection on the entire work. The results will be discussed in relation to our initial assumptions and the methods used to achieve them will be analysed. We will also discuss the validity of the results and to what extent they can be generalised. In the end, we will provide the reader with examples of input for further work, based on our findings.

6.1 Reflections

When starting our thesis the objective was to predict faults in Neo4j's AuraDB. In a distributed system like this, there are many types of faults that can occur which can in turn cause failures of different types. In order to narrow down the scope of the issue stated, we decided to interpret this into being able to alert when a database is at risk of losing fault tolerance. This decision was based on Neo4j's desire to be alerted before a database core went down, since this would decrease the risk of a failure. Additionally, the database's metrics included information regarding if it was fault tolerant or not. This enabled us to validate our findings with real data in respect to our research question. While the process of testing if a database actually lost fault tolerance following an alert was straightforward, finding correlations from other metrics that could have an implication on the system and thus be used to trigger these alerts, provided a much harder problem.

Since we were dealing with real data from a system in production we had no control over the processes resulting in the values reported from the metrics. Different processes could have an effect on the values which caused them to deviate from a more common pattern without it causing fault intolerance in the system. When and how often such processes occur is not known to us. Instead, assumptions had to be made about what data would be regarded as normal data. In our work we simply defined normal, or clean data, as data that did not lie in the specified proximity of a database becoming fault intolerant, both before and after. If this was a valid definition depends on what the normal data is later used for. In our case it would be used for training anomaly detection models. An additional consequence of using data from a real system was that we had no way of impacting the data set distribution. As fault intolerance was a rare occurrence in the system we were limited in obtaining anomalous data. This led to the data gathered being highly imbalanced. That was another important characteristic of the data that had to be regarded when applying a model to it.

One of the biggest challenges we encountered during our work was how we would handle the huge amount of data that was made available to us. In theory we could query data from thousands of databases, thousands of metrics, for several months back in time, and with sampling frequencies as small as a second. In practice this would not be possible due to limitations in computing capacity and to the fact that there was a limit on how fast we could query the data. In order to limit the scope we had to reduce the number of metrics and databases, as well as limit the data to be queried. When studying the metrics in order to bring the scale the count down, we only queried for one week.

In the beginning of our work we had made the assumption that all databases could be treated as if they were coming from the same distribution, i.e. the data queried from the databases would behave in a similar way given that the same processes were running on them. As all databases are built on the exact same software, we expected this to be the case. This assumption was of importance since we wanted to create and train one single model that would work for any database in the AuraDB system. Creating and training a single model for each database at Neo4j would be much less efficient than having one model that could monitor all instances. In such a case, every time a new database was added to the system a new model had to be trained for that specific database. However, by looking at the attributes of the

data in section 5.1 we can see that the behaviour of the data can differ a lot between different databases. Especially from figure 5.5 we can see that the time series data of the two databases differs a lot. After having disclosed our results with our advisory at Neo4j we discussed the possibility that this assumption was incorrect.

The primary arguments to why databases could not be viewed as coming from one and the same distribution were identified to be differences in the amount of resources assigned to the database, and how the database was used by the customer. Customers' usage of the database may cause a difference in how the database operates. For example, one customer might have a database with the resources to handle sudden spikes in workload, like the spike in e-commerce as a result of Black Friday. In this case, the database is constantly operating on a low workload in regards to its capacity, resulting in very low values on metrics (even though the usage and processes vary over the normal season.) Another customer might have slimmed down their database, and allocated just enough resources to handle a normal workload. In this instance the database is always working and the workload is expected to stay consistent. In the latter case, a small increase in the workload could result in the database losing fault tolerance, while the customer that had allocated much more resources can continue operating as normal. In these two cases, the metric data leading up to the databases losing fault tolerance would probably look very different. The customer with a much slimmer database would lie much closer to the limit of different metrics, such as CPU usage, while the customer with a lot of resources might report low numbers for the corresponding time period. If we were to treat these two databases as coming from the same distribution, and simply use the data from the two as if they came from the same instance, it would be much harder to create a satisfying performing model.

By looking at the figures in Section 5.1, and especially Figure 5.3 which shows two time series from the metric *neo4j.page_cache.page_faults*, we can see that the trend represented in the data is the values increasing over time. In the first two weeks, which is the first half of the time series which has a total length of four weeks, the periods of higher values are fewer and values are lower compared to the second half. We chose to train on the time series for the first two weeks, and test on data from the last two. The reason behind this was because the aim of this thesis was to create a model that will be able to predict fault intolerance events in the future. Thus, training on data that comes before the testing data in time would be most suitable for the validity of the results in relation to the objective. However, by looking at the figure displaying the time series from *neo4j.page_cache.page_faults*, questions regarding if the amount of data queried and used was enough to be able to adequately train and test a model. Are the higher values reported a common occurrence, or is it a new phenomenon? Either way, the fact that the two time periods used for training and testing deviates could have an impact on the performance of the model.

As stated before, finding correlations from metrics that could have an implication on the system and thus be used to trigger alerts for upcoming events of fault intolerance was one of our main problems. One theory we established was that databases lose fault tolerance from different anomalous events in the system. Therefore, one way forward was to look for anomalies in the data and use these as predictors. From the literature study conducted we found that a common technique used for anomaly detection was semi-supervised methods. In order

for semi-supervised models to work, there is a need for training data that consists solely of normal data. This prerequisite raises the question if our definition of normal data is valid for our purpose. As we could not control the operation of the databases from which we collected our data, we could not know if there occurred other anomalous events that would show in the metrics without causing the database to become fault intolerant. Further discussion of this limitation is given in Section 6.6.

In order to complete the evaluation of the anomalies found in the anomaly detection phase we needed to design an evaluation method where we would classify the anomalies. The method for this is described in Section 4.4. There are many ways in which to classify the anomalies, the way we chose to do it was based on the needs of Neo4j. It was of interest to them to see not only if, but also how long in advance it would be possible to predict that a database would become fault intolerant. More about the limitations of the evaluation method is given in Section 6.3. Having given some background on the challenges that had to be addressed during the course of the work, we now come to the part where we discuss the results achieved from applying our method on the problem.

6.2 Discussion on Fault Intolerance Predictions Results

In this section we will present and discuss the results from the evaluation of the anomaly detection methods used to predict fault intolerance. We start by discussing the results from applying HBOS in different ways. In the end we talk about the results derived from the LSTM autoencoder approach.

6.2.1 HBOS

While it can simply be observed that the result obtained by the experiments in this thesis left much to be desired there are still some interesting observations that can be made. By looking at the resulting scores shown in the tables 5.1 to 5.3, 5.8 to 5.10 and 5.12, we can see that our model perform vastly superior in recall rather than precision. This means that while our model could identify some anomalies that indicated loss of fault tolerance, there would be too many false alarms for it being able to be used efficiently. We can also identify the metric with best performance based on domain knowledge was *haproxy.backend.errors.con_rate* and the best metric based on statistical analysis was *kubernetes.io.read_bytes*.

If we look at the Tables table 5.1 and 5.8 and at the multivariate HBOS models created for these two sets of metrics, we see that the F1-score generated by metrics generated by our statistical analysis goes from 0.0009 for the time window 0-10, down to 0.0002 for the time window 50-60. The score becomes increasingly higher for the time windows closer to fault intolerance. For metrics generated by domain knowledge the F1-score varies between 0.0006-0.0007. This means that while metrics based on domain knowledge had better overall performance when looking at the multivariate HBOS analysis: but the metrics selected by the statistical analysis performed better up to 35 minutes before the database lost fault

tolerance. While this could be an indication that our statistical analysis was successful in finding metrics that could be used for predicting leading up to losing fault tolerance, the poorness of the overall result along with the small differences, makes it impossible to draw any conclusions.

As can be seen in the results, HBOS models were constructed both individually for each database where the results were combined to an ensemble, as well using data from all databases to construct one single HBOS model. We expected that the model trained on all available data would perform better, however the results showed that the model constructed individually for each database overall performed better. While some metrics performed better when being trained on all available databases, others performed worse. Most noticeable was that the performance of the overall best metric identified *haproxy.backend.errors.con_rate*, was significantly reduced. Our best result was from training HBOS models individually. This could indicate that the behaviour of the database was too diverse for the use of one unified model. As all databases use the exact same software, we expected to be able to use one model on all databases. As mentioned in section 6.1, the assumption made about the databases originating from one and the same distribution might not hold, thus affecting the result.

HBOS with Moving Average

The results from moving average being used in combination with HBOS can be viewed in the tables tables 5.3 and 5.10. These tables show the five best performing metrics from each experiment. While some metrics performance did improve, others declined. Again, the scores from our best performing metric *haproxy.backend.errors.con_rate*, decreased. We have found that the approach of combining the use of moving averages with HBOS is unique for our thesis. However, our poor overall results do not allow for drawing any conclusions about the benefit of combining moving averages and HBOS for it to work better on time series data.

HBOS on our Best Results

The final experiment conducted with the HBOS anomaly detection method was by creating a multivariate model trained on our six best performing metrics from the previous experiments. The results can be viewed in table 5.12. As we can see, the scores for this multivariate model are much better than the ones from the multivariate models trained on all the selected metrics. This is expected as the evaluation of the anomalies found by the HBOS model is carried out after the model is fit, and no learning from the results is taking place. When we select the metrics by hand and use them to produce a multivariate model, we give the model the metrics which results in the prediction of anomalies lying before the fault intolerance events.

6.2.2 LSTM based Autoencoder

The results of the LSTM based autoencoder trained and tested on metrics extracted by domain knowledge can be viewed in tables 5.4 and 5.5. The results for the metrics selected from the statistical analysis are shown in tables 5.6 and 5.7. As can be seen, this model performance was secondary to HBOS. When training and testing the model with metrics from statistical

analysis with the initial threshold set as described in section 4.3.1, it was unable to predict any anomaly that lied within the hour leading up to a database losing fault intolerance. When lowering the threshold so that three times more data points were classified as anomalies, we got non-zero scores. However, the scores are still very low. The effect of changing the threshold is discussed further in Section 6.3. In contrast to the HBOS model, which assumes no relation between features, the LSTM autoencoder weighs the features and reduces the dimensionality. The fact that there can be correlations between different metrics that we use for the anomaly detection motivates the use of the LSTM autoencoder. In addition to this, the fact that it is developed especially for the use of modelling sequence data made us believe it would perform well. There are several possible reasons why the scores achieved by the LSTM autoencoder approach are so low. In Section 6.6 we discuss further why we achieved low scores both in general, and in regards to the LSTM autoencoder approach.

6.3 Discussion on the Evaluation Method

An important part of our work was the classification of the predicted anomalies produced in the anomaly detection phase. The method that was used is described in Section 4.3. As the scores produced are based on the rate of true positives, false positives, and false negatives, the classification process had a big impact on our results. There are a few limitations to our evaluation method that is worth discussing. In the evaluation process, no regard was given the fact that a database could become fault intolerant repeatedly and that this could happen with little time in between. If we look at Figure 5.6, the three spikes visible in the top two graphs are actually representing the same data points, meaning that the two are overlapping each other. Thus, the fault intolerance event shown on the second (main) spike in the second graph is happening less than 20 minutes after the fault intolerance event shown on the first (main) spike in the top graph. Worth noticing is that the third higher spike has no fault intolerance in its closer proximity. Let us say all the spikes were to be classified as anomalies, then it would be difficult to tell which anomalies should be said to belong to which fault intolerance instance. It all comes down to the on how we define a true positive, false negative, false positive and true negative. By simply defining them as we did in Section 4.3, some of the anomalies are going to be predictive of multiple fault intolerance events, but in time windows lying differently close to the events. This will happen given that the fault intolerance events occur with less than one hour distance. Similarly, these anomalies will be classified as false positives when lying outside these time windows.

There could also be a question whether the chosen size and placement of the time windows studied are appropriate. If the size were to increase, more anomalies would be classified as true positives. However, by studying the results, changes to the windows would not result in any big change to the scores. By increasing the window size, fewer windows would be classified as false negative since the criteria we use to classify a window as true positive is if there is at least one predicted anomaly within it. This would increase the recall, but the change to the precision would be small. To improve the precision we would need to classify fewer anomalies as false positives which would require us to extend the window sizes in an unreasonable way. By our way of categorising the anomalies using time windows, we limit the way we can interpret the results. If we look at Table 5.1 and imagine designing a prediction

model based on the metric *haproxy.backend.errors.con_rate*, we can see that the probability of a fault intolerance happening within 0-10 if an alert would be 5.9 percent. But we cannot tell if we got the alert the minute before or 10 minutes before a database lost fault intolerance. If the window size would be extended this uncertainty would be even bigger, defeating the purpose of an alarm system.

Another thing that has an impact on the scores is the number of data points that we label as a predicted anomaly, which in turn depends on the threshold chosen. The threshold is set as a function of the percentage of fault intolerance data points in the training set, as described in Section 4.3. If the threshold would be set higher, leading to fewer anomalies classified, it could cause the recall to decrease and the precision to increase. This is due to the fact that fewer points can be classified as anomalies, which in our case likely would lead to a decrease in the number of false positives at a higher rate than true positives. However, the effect of higher threshold would have on the poor result we obtained is not going to be overplayed. In the case of the LSTM autoencoders, it was instead necessary to lower the threshold in order to improve the results, but to lower it further than we already did would not generate results superior to those we presented. The number of anomalies that can cause a window to be classified as a true positive is limited. There are only so many fault intolerance events to be predicted.

6.4 Discussion of Metric Selection

Maybe the most important part of the prediction process is the selection of data used for the anomaly detection phase. In this section we discuss the metrics used and the methods to select them.

6.4.1 Metric Selection

From the metric selected by domain knowledge and our statistical analysis only one metric appears in both selections, *kubernetes.io.read_bytes*. While this is the only identical metric there are several that appear to monitor similar behaviour. Both selections contain metrics monitoring the databases' transactions and the heap memory of the JVMs (Java Virtual Machine). The biggest difference between the two collections is that our statistical analysis gave metrics monitoring the *container* and *containerd*.

We further analysed how the metrics selected by domain knowledge scored according to our statistical analysis. As described in 4.2.5 we first removed metrics that on any database were missing metric values of the entire interval. Later we scored them according to their behaviour leading up to the database losing fault tolerance. Surprisingly, not all metrics extracted by domain knowledge were apparent here. This means that for our week of queried data, these metrics generated no values for at least one of the database instances studied. To be able to capture these metrics it could be beneficial to extend the period of the data we conduct our analysis on. The remaining metrics were found on our list, but their score was

not high enough to be included in the 20 best.

6.4.2 Discussion of our Statistical Analysis

Limitations of this approach include simply letting standard deviation represent the behaviour of the graph. In implementing this approach we gain limited information concerning the graph. Standard deviation simply gives us an average regarding how far each value lies from the mean. This method is for example very sensitive to outliers, which causes a disproportionate effect, making it hard to model the graph characteristics. This limitation could disrupt the guarantee of the statistical analysis finding metrics that both have a different behaviour leading up to fault intolerance as well as that conformation is an abnormal behaviour in the metric.

6.5 Our Findings Compared to Related Work

In related work, HBOS was listed as a fast anomaly detection algorithm with performance as reliable as state-of-the-art algorithms in detecting point anomalies. To capture other types of anomalies, temporal information needed to be taken into account. Papers in related work suggested to use an LSTM autoencoder and found it to perform better than classical anomaly detection, for example statistical methods. Based on the findings in related work, it was expected for our LSTM autoencoder to perform better than the statistical model HBOS. Instead we found that our HBOS method performed better, contradicting the related work. However, related work also found limitations of the LSTM autoencoder. One paper found that if the normal and anomalous data was too similar it would result in miss-classifications, subsequently an LSTM autoencoder would be unsuitable. As discussed in Section 6.6, this limitation seemed to be present in our data as well.

Related work further discussed if metric information was sufficient for detecting anomalies in a complex cloud system. They proposed to include logs in addition to metrics when identifying anomalies. Due to time restriction, we did not include logs when conducting our thesis. This claim from related work could be an implication that metrics was inadequate when detecting anomalies in our thesis. The process of incorporating logs is debated in our future work in Section 6.7.

6.6 Possible Factors Contributing to Poor Results

From the results and discussion above, it is clear that our work did not produce the results that we had hoped for. In this section, we aim to examine the factors and circumstances that could have contributed to this outcome. By analysing the limitations, challenges and unforeseen complications we seek to grasp the reasons behind our results. Additionally, we aim to provide valuable insights for further research in a similar setting.

One factor that could have an impact on the result is the choice of metrics. The metrics chosen to conduct the experiments on were wrongly chosen and some metrics that could be used were overlooked. While this could have an impact on the result, we believe it is unlikely to have caused our low scores. Instead, we consider the quality of data to be the issue with the highest impact.

When approaching our research questions we chose to apply semi-supervised models to our problem in question. As mentioned in Section 6.1, in choosing this tactic it became essential to identify normal data from our data sets. When identifying normal data there are two aberrations that can profoundly affect the success of the models; not all normal behaviour is captured, and data is missclassified. We believe that these problems could have an impact on the success of our models. Due to the time restriction imposed on this thesis, we only had time to train and test our models on a time-frame of one month. We consider this to possibly be an insufficient time to capture all normal behaviour of a database. For instance, by observing the graphs in Figure 5.3 we can clearly see that the graphs do not conform to the same behaviour throughout the time series. Since we carry out training with data from the first half and testing on the second, a consequence would be that normal behaviour would be missclassified as anomalous leading to an abundance of false positives (FP). As the formula when calculating the precision of a model contains false positives, this is likely a contribution to our low precision in the results.

It is not far-fetched to believe that there are other processes in the system that would cause an anomalous behaviour that would display in the metric data. In terms of mistakenly classifying anomalous data as normal this can be observed in Figures 5.1 and 5.6. Based on these figures it is easy to believe that the spikes in the time series are the result of some kind of anomalous behaviour, thus the spikes would be classified as anomalies. However, when we look at the graphs we see that even though some of the spikes appear to correlate with the fault intolerance event shown in red, others lie far from these events in time. This means that even though we removed some of the anomalies from the training data as part of the preprocessing, others would persist and be used in the training of the semi-supervised models. Occurrences like this would cause our models to be trained to interpret anomalous data as normal resulting in a good deal of false negatives (FN) to be identified. As the formula when calculating the recall of a model contains false negatives, this is likely a contribution to our low recall in the results. The importance to clearly separate data for when a database is running normally from when anomalies happens is especially true for the training of the LSTM autoencoder, as it will learn how to encode and decode the data it trains on.

While it is possible that these factors could have contributed to our poor results it is hard to speculate if it would be enough to cause such an immense impact. Another theory is that the approach of training models to perform anomaly detection on metric data originating from a system in production was simply too complex for our purpose. This leads us to the next section where we will present our thoughts and ideas for future research and possible alternative approaches to methods used in our work.

6.7 Future Work

In terms of future work there are several directions that could be interesting to explore further. Even though we did not achieve good results for predicting fault intolerance by applying anomaly detection on metric data in our work, we do not reject the use of metric data as a base for making these predictions. In this section we will suggest alternative ways of creating models fit for the purpose.

As discussed in Section 6.1, one of the big question marks is whether there are other processes running that could be causing errors that would show as anomalies in the metric data. Not all anomalies would lead to a database core going down resulting in the system becoming fault intolerant. In order to have better control over the processes that run on the system and thus be able to monitor the metric behaviour in a better way it could be good to get the data from a controlled environment. Thus, the suggestion would be to use a database in a controlled setup and try to invoke fault intolerance by making the database subject to extreme or anomalous events, such as making a large query, or many queries. By using an approach like this, not only can one correlate different behaviours in the metric to when the database loses fault tolerance, but also know what was the root cause of the anomalies in the metric data.

If instead it is of interest to study data from the real system running in production, a different approach to ours would be to only try to build a model for a single database instance, this would allow for more data to be queried for. As discussed in Section 6.1, the assumption that all database instances would be of the same distribution might not hold. By only studying one database would eliminate this problem.

Additionally, the logs produced by the system could be taken into account. By combining information both from logs and metrics it could be easier to understand the behaviour of the system. As related work suggested this approach could also be used to identify which metrics lead to the failures reported in logs. In future work, this would be an interesting direction to take.

Chapter 7
Conclusion

The aim of this theis was to examine if faults could be predicted in Neo4j AuraDB, a database as a service offered by the company Neo4j. The approach taken was to use anomaly detection with monitoring metrics and train different models to predict when a database core would go down. This approach resulted in the research questions addressing both the challenges of metric selection, as well as the anomaly detection problem. The research questions are as follows:

- **RQ1:** Can we identify metrics that correlate, with the event of a database becoming fault intolerant.
- **RQ2:** From the metrics identified in RQ1, can we based on anomaly detection state when a database is at risk of becoming fault intolerant.

For RQ1 we were not successful in general to identify metrics that correlate with the event of a database becoming fault intolerant, neither by using domain knowledge, nor statistical analysis. The only metric we could identify that would show a reasonable correlation was *haproxy.backend.errors.con_rate*, with an F1-score of 0.065, where the recall was 0.072 and the precision 0.059. This was for values registered 0-10 minutes before fault intolerance events. However, the rest of the 37 metric did not show any correlation to a satisfactory extent.

For RQ2 we were not able to predict when a database is at risk of becoming fault intolerant based on the different anomaly detection methods used in this work. The anomaly detection methods used were based on either HBOS or the LSTM autoencoder approach.

Example of probable explanations for our results are that the data, on which the models were trained, did not capture all of the normal behaviour or that anomalies were present in the normal data from processes not causing fault intolerance. To obtain more reliable data for training and testing, we suggest that further work should utilise data collected from a controlled environment rather than from a system in production.

References

[1] What is cloud computing? https://www.ibm.com/topics/cloud-computing [Accessed: 2023-04-02].

[2] What is database-as-a-service (dbaas)? https://www.ibm.com/topics/dbaas [Accessed: 2023-04-02].

[3] Distributed consensus - raft and jraft, aug 2019. https://www.sofastack.tech/en/projects/sofa-jraft/consistency-raft-jraft/ [Accessed: 2023-05-20].

[4] Database as a service market: Global industry trends, share, size, growth, opportunity and forecast 2023-2028. Technical Report 5732314, IMARC Group, January 2023.

[5] Charu C Aggarwal. *Neural Networks and Deep Learning*. Springer Cham, 2018.

[6] Omar Alghushairy, Raed Alsini, Terence Soule, and Xiaogang Ma. A review of local outlier factor algorithms for outlier detection in big data streams. *Big Data and Cognitive Computing*, 5(1), 2021.

[7] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), feb 2008.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[9] Will Badr. Having an imbalanced dataset? here is how you can fix it., feb 2019. https://towardsdatascience.com/having-an-imbalanced-dataset-here-is-how-you-can-solve-it-1640568947eb [Accessed: 2023-05-08].

[10] Maroua Bahri, Flavia Salutari, Andrian Putina, and Mauro Sozio. Automl: state of the art with a focus on anomaly detection, challenges, and research directions. *International Journal of Data Science and Analytics*, 14, 08 2022.

[11] Caroline Banton. Data science: Overview, history and faqs, 2022. https://www.investopedia.com/terms/d/data-science.asp [Accessed: 2023-05-25].

[12] Julia Bohutska. Anomaly detection — how to tell good performance from bad, aug 2021. https://towardsdatascience.com/anomaly-detection-how-to-tell-good-performance-from-bad-b57116d71a10 [Accessed: 2023-04-15].

[13] Chris Brook. Saas: Single tenant vs multi-tenant - what's the difference?, apr 2023. https://www.digitalguardian.com/blog/saas-single-tenant-vs-multi-tenant-whats-difference [Accessed: 2023-04-25].

[14] Obradovic Z. Cao XH, Stojkovic I. Da robust data scaling algorithm to improve classification accuracies in biomedical data. *BMC Bioinformatics*, 17(359):111–117, 2016.

[15] Anthony Cavin. Fast anomaly detection with python, jul 2022. https://towardsdatascience.com/hbos-vs-iforest-on-macbook-pro-m1-c258d2b5fe6b [Accessed: 2023-04-07].

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41, 07 2009.

[17] Charlie Custer. What is fault tolerance, and how to build fault-tolerant systems, mar 2023. https://www.cockroachlabs.com/blog/what-is-fault-tolerance/ [Accessed: 2023-04-07].

[18] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. *Journal of Systems and Software*, 137:531–549, 2018.

[19] Jake Frankenfield. Data analytics: What it is, how it's used, and 4 basic techniques, 2023. https://www.investopedia.com/terms/d/data-analytics.asp [Accessed: 2023-05-25].

[20] Jake Frankenfield. What is cloud computing? pros and cons of different types of services, 2023. https://www.investopedia.com/terms/c/cloud-computing.asp [Accessed: 2023-04-02].

[21] Song Fu. Performance metric selection for autonomic anomaly detection on cloud computing systems. In *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, pages 1–5, 2011.

[22] Alexander S Gillis. single-tenancy, feb 2023. https://www.techtarget.com/searchcloudcomputing/definition/single-tenancy [Accessed: 2023-04-24].

[23] Markus Goldstein and Andreas Dengel. Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm. 09 2012.

[24] Seng Hansun. A new approach of moving average method in time series analysis. In *2013 Conference on New Media Studies (CoNMedia)*, pages 1–4, 2013.

[25] Red Hat. What is a kubernetes cluster?, jan 2020. https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster [Accessed: 2023-04-07].

[26] Red Hat. What is container orchestration?, may 2022. https://www.redhat.com/en/topics/containers/what-is-container-orchestration [Accessed: 2023-04-05].

[27] Adam Hayes. What is a time series and how is it used to analyze data?, june 2022. https://www.investopedia.com/terms/t/timeseries.asp [Accessed: 2023-04-02].

[28] L. Hermes and J.M. Buhmann. Feature selection for support vector machines. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 2, pages 712–715 vol.2, 2000.

[29] Inc IBM. What is saas – software-as-a-service?, 2023. https://www.ibm.com/se-en/topics/saas [Accessed: 2023-03-04].

[30] Joos Korstanje. The f1 score, aug 2021. https://towardsdatascience.com/the-f1-score-bec2bbc38aa6 [Accessed: 2023-04-16].

[31] Sotiris B Kotsiantis, Dimitris Kanellopoulos, and Panagiotis E Pintelas. Data preprocessing for supervised leaning. *International journal of computer science*, 1(2):111–117, 2006.

[32] Kubernetes. Cluster architecture, june 2021. https://kubernetes.io/docs/concepts/architecture/ [Accessed: 2023-04-06].

[33] Kubernetes. Kubernetes documentation, jan 2023. https://kubernetes.io/docs/concepts/overview/ [Accessed: 2023-04-06].

[34] Chris Kuo. Anomaly detection for time series, apr 2021. https://medium.com/dataman-in-ai/anomaly-detection-for-time-series-a87f8bc8d22e [Accessed: 2023-04-10].

[35] Chris Kuo. Handbook of anomaly detection: with python outlier detection - hbos, oct 2021. https://medium.com/dataman-in-ai/anomaly-detection-with-histogram-based-outlier-detection-hbo-bc10ef52f23f [Accessed: 2023-04-06].

[36] Frank Liang. Evaluating the performance of machine learning models, aug 2020. https://towardsdatascience.com/classifying-model-outcomes-true-false-positives-negatives-177c1e702810 [Accessed: 2023-04-15].

[37] Huan Liu and Rudy Setiono. Chi2: Feature selection and discretization of numeric attributes. In *Proceedings of 7th IEEE international conference on tools with artificial intelligence*, pages 388–391. IEEE, 1995.

[38] Francesco Lomio, Sergio Moreschini, Xiaozhou Li, and Valentina Lenarduzzi. Anomaly detection in cloud-native systems. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 100–103, 2022.

[39] Fan Jing Meng, Xiao Zhang, Pengfei Chen, and Jing Min Xu. Driftinsight: Detecting anomalous behaviors in large-scale cloud platform. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 230–237, 2017.

[40] Joydeep Mukherjee, Alexandru Baluta, Marin Litoiu, and Diwakar Krishnamurthy. Rad: Detecting performance anomalies in cloud-based web services. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 493–501, 2020.

[41] Inc Neo4j. Introducing neo4j aura: A new graph database as a service, 2019. `https://neo4j.com/emil/neo4j-aura-graph-database-as-a-service/` [Accessed: 2023-03-06].

[42] Inc Neo4j. What is a graph database?, 2022. `https://neo4j.com/docs/getting-started/current/get-started-with-neo4j/graph-database/` [Accessed: 2023-03-07].

[43] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8):1226–1238, 2005.

[44] Oleksandr I. Provotar, Yaroslav M. Linder, and Maksym M. Veres. Unsupervised anomaly detection in time series using lstm-based autoencoders. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, pages 513–517, 2019.

[45] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[46] Chitta Ranjan. Extreme rare event classification using autoencoders in keras, may 2019. `https://towardsdatascience.com/extreme-rare-event-classification-using-autoencoders-in-keras-a565b386f098` [Accessed: 2023-05-16].

[47] Muntadher Saadoon, Siti Hafizah Ab. Hamid, Hazrina Sofian, Hamza H.M. Altarturi, Zati Hakim Azizul, and Nur Nasuha. Fault tolerance in big data storage and processing systems: A review on challenges and solutions. *Ain Shams Engineering Journal*, 13(2):101538, 2022.

[48] Mahmoud Said Elsayed, Nhien-An Le-Khac, Soumyabrata Dev, and Anca Delia Jurcut. Network anomaly detection using lstm based autoencoder. In *Proceedings of the 16th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, Q2SWinet '20, page 37–45, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Yang Shen, Jue Bo, Li Kexin, Shuo Chen, Lin Qiao, and Jing Li. *High-Dimensional Data Anomaly Detection Framework Based on Feature Extraction of Elastic Network*, pages 3–17. 10 2019.

[50] Dorjey Sherpa. Introduction to univariate, bivariate and multivariate analysis, mar 2021. https://medium.com/analytics-vidhya/univariate-bivariate-and-multivariate-analysis-8b4fc3d8202c [Accessed: 2023-04-07].

[51] Pallavi Srivastava. How to handle missing data in data preprocessing, jan 2022. https://python.plainenglish.io/handling-missing-data-b92044bc2066y [Accessed: 2023-02-06].

[52] Srikanth Thudumu, Philip Branch, Jiong Jin, and Jugdutt (Jack) Singh. A comprehensive survey of anomaly detection techniques for high dimensional big data. *Journal of Big Data*, 7(1):42, Jul 2020.

[53] Prometheus Authors 2014-2023 | Documentation Distributed under CC-BY-4.0. Overview | prometheus, 2023. https://prometheus.io/docs/introduction/overview/ [Accessed: 2023-03-08].

[54] Yiteng Zhai, Yew-Soon Ong, and Ivor W. Tsang. The emerging "big dimensionality". *IEEE Computational Intelligence Magazine*, 9(3):14–26, 2014.

EXAMENSARBETE Predicting Loss of Fault Tolerance in a Cloud Graph Database
STUDENT Evelina Danielsson, Lisa Franzén af Klint
HANDLEDARE Johan Eker (LTH)
EXAMINATOR Markus Borg (LTH)

Förutsäga förlust av feltolerans i en molntjänst

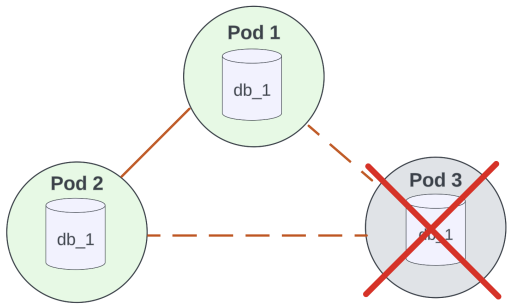
POPULÄRVETENSKAPLIG SAMMANFATTNING **Evelina Danielsson, Lisa Franzén af Klint**

För att en molntjänst ska vara konstant tillgänglig för en användare behöver systemet ha en tolerans mot fel som kan uppstå. Detta examensarbete undersöker om det finns anomalier i de olika mätvärdena från systemet vars information kan användas i syfte att behålla feltoleransen.

Molntjänster är idag det vanligaste sättet för företag att leverera programvara till sina kunder. Tidigare var det vanligt att behöva installera ett program direkt på datorn för att kunna använda det. Idag är vi i stället vana vid att få tillgång till digitala tjänster via en webbläsare. Dessa tjänster kan innefatta allt från lagring av foton, till avancerade tjänster för bildredigering. För att dessa tjänster ska vara tillgängliga för användare behöver systemet vara feltolerant. För att kunna avgöra hur ett system mår övervakas det ofta med hjälp av olika metriker och loggar. Metriker ger kontinuerlig kvantitativ information angående olika processer i systemet.

I vårt examensarbete har vi undersökt om vi med hjälp av olika metriker extraherade ifrån en databas-molntjänst kan förutsäga när fel inträffar som resulterar i att systemet förlorar sin feltolerans. För att en kund ska ha konstant tillgång till sin databas finns det tre kopior av databasen i systemet. Om en utav dessa kopior skulle gå ner är systemet fortfarande operativt då de återstående två kopiorna kan serva kunden. Detta resulterar i att funktionaliteten upprätthålls, men feltoleransen går förlorad.

För att undersöka om det är möjligt att förutsäga då en databaskopia går ner analyserades all tillgänglig metrik för att undersöka vilka som skulle kunna innehålla relevant information. Därefter analyserades värden från dessa metriker genom att applicera olika algoritmer för anomalidetektion. Dessa algoritmer bygger dels på en statistisk metod baserad på HBOS, dels på en autoencoder baserad på artificiella neuronnätverk.



Resultaten visar att genom att applicera de metriker och de metoder som nämnts ovan, inte är möjligt att förutsäga när en databas tappar feltolerans. Det kan finnas många orsaker till detta, några som omnämns i rapporten är processer som orsakar anomalier inte leder till förlorad feltolerans samt att databaserna som undersöks inte besitter gemensamma egenskaper.