Diego Martínez Baselga

Detección de anomalías con Prometheus

Escuela de Ingeniería y Arquitectura (EINA) - Grado en Ingeniería Informática
Trabajo de Fin de Grado - Tutor: Javier Fabra Caro - Director: Davide Taibi
Realizado durante programa Erasmus en la Universidad de Tampere
Junio 2020

Resumen - Detección de anomalías con Prometheus

En los últimos años, el auge de la informática y el incremento de las prestaciones de los ordenadores y de las necesidades de los usuarios han potenciado el uso de sistemas distribuidos. Ofrecen características como fiabilidad, escalabilidad, tolerancia a fallos y disponibilidad. En particular, Kubernetes se ha convertido en una alternativa muy común.

Kubernetes es una plataforma que administra contenedores, normalmente a través de Docker. Los contenedores son un software de virtualización a nivel de sistema operativo que permiten desplegar aplicaciones fácil y rápidamente, encapsulando la aplicación junto con sus dependencias, sus necesidades y la imagen del sistema operativo necesaria para su despliegue. Por ello, son comúnmente utilizados a la hora de encapsular microservicios.

El servicio de monitorización más utilizado en Kubernetes es Prometheus. Es una plataforma de código abierto que permite recopilar estadísticas de otras aplicaciones o de los elementos de Kubernetes. Estas métricas se pueden visualizar a través de gráficos o se pueden solicitar a través de APIs para operar con ellas.

Además, Prometheus ofrece un administrador de alertas: Prometheus Alertmanager. Su propósito es encender alarmas o establecer acciones a realizar cuando se detecta una anomalía en el sistema. Se considera anomalía a un evento que provoca un comportamiento extraño en el sistema, por lo que detectarlas es muy importante. Sin embargo, Prometheus Alertmanager solo es capaz de detectar anomalías utilizando expresiones matemáticas, y en realidad éstas son mucho más complicadas de detectar.

El objetivo de este trabajo es explicar y desplegar un sistema de Kubernetes realista similar a un entorno de producción, generar un conjunto etiquetado de datos obtenido con Prometheus con anomalías producidas por un microservicio; así como analizar las formas de las que podría implementarse un plug-in de detección de anomalías. De esta manera, el método podría utilizarse para predecir anomalías en el propio sistema desarrollado o extrapolarse para utilizarse en otra implementación completamente distinta con otro tipo de anomalías.

El primer paso seguido en la metodología fue ejecutar los programas a utilizar localmente, antes de pasar a un entorno distribuido. Además de Prometheus y Prometheus Alertmanager, se decidió utilizar Kafka para asemejar más el sistema a uno real y no uno de simulación. Kafka es una aplicación de transmisión de mensajes basada en la arquitectura "publish-subscribe". En esta arquitectura, una serie de procesos publican mensajes asociados a determinados temas, y los procesos que se suscriben a esos temas reciben los mensajes. Una vez se hubieron ejecutado

las aplicaciones, se implementaron unos programas que las conectaban, generaban transmisiones de datos a través de Kafka, solicitaban y examinaban métricas de Prometheus y generaban alertas. El objetivo de este paso era desplegar el sistema en un entorno más simple y aprender cómo interaccionar con él. Algunas partes de los programas implementados fueron más tarde utilizados en propósitos más complejos.

Posteriormente se crearon 4 máquinas virtuales, con las que se formó un cluster de Kubernetes, formado por 1 maestro y 3 trabajadores. En el cluster se desplegaron Prometheus, Kafka y Zookeeper (necesario para ejecutar Kafka) para obtener un sistema distribuído con las aplicaciones que se habían probado en el entorno local. Adicionalmente, se utilizaron Kube-state-metrics y Node Exporter, dos aplicaciones oficiales de Kubernetes para exportar más métricas. Se proveyó al cluster de una estructura de almacenamiento persistente y fiable.

Una vez el sistema estaba en correcto funcionamiento, se generaron las anomalías. Para ello, se implementó un microservicio como un contenedor de Docker que generaba un problema de memoria. En primer lugar, se anotaba el momento en el que el contenedor se ejecutaba en un fichero y se descargaban los binarios para ejecutar Kafka. Después, se volvía a anotar la hora en otro fichero y se comenzaban a transmitir datos a través de Kafka. Finalmente, después de un número aleatorio de segundos, se producía la anomalía, anotando el momento en el que se empezaba a producir. La anomalía consistía en una combinación de instrucciones que solicitan una cantidad de memoria cada vez mayor al sistema, hasta que excede los recursos disponibles para el contenedor y es destruído. La ejecución se configuró a través de Kubernetes para que los tres ficheros generados se almacenaran y se editaran de forma persistente siempre en la misma máquina virtual.

El miscroservicio diseñado se introdujo como un Pod a Kubernetes. Una vez se crea el Pod, genera métricas recopiladas por Prometheus que pueden ser extraídas a través de su API. Se ejecutó durante dos intervalos de tiempo diferentes, resultando en un total de 1010 segundos de ejecución. El volumen de datos solicitado provocaba latencias que resultaban en fallos en Prometheus, por lo que fueron necesarios 3 programas distintos para crear las tablas de datos. Un primer programa recopiló los datos almacenándolos en ficheros y carpetas según la métrica y el intervalo de tiempo que fuera, utilizando 10 carpetas para los 10 intervalos en los que se dividieron. El segundo programa buscaba métricas inconsistentes y borraba todos los ficheros generados con esas métricas. Por último, el tercer programa generó dos tablas que contenían todos los datos y la información, utilizando los datos recopilados, así como los ficheros de horas generados por el microservicio.

Se obtuvieron dos tablas diferentes, una describiendo las anomalías generadas y la otra el conjunto de datos. Ambas fueron subidas a Gitlab para entrenar un algoritmo de aprendizaje automático para predecir anomalías. La tabla de anomalías está formada por las 30 generadas, indicando los momentos en los que se inician y se terminan, así como otros campos descriptivos. En la tabla del conjunto de datos, se indica para cada dato el segundo correspondiente a ese dato, si corresponde a una anomalía o no (1 ó 0), el estado del microservicio en ese instante, y el valor de cada una de las métricas tomadas. El resultado son 10010 puntos, 600 de los cuales son anomalías, con 7062 métricas por punto; además de información complementaria y explicativa sobre los datos.

El propósito conjunto de datos generado es utilizarse para entrenar un algoritmo de aprendizaje automático e implementar un plug-in que prediga las anomalías. Para implementar el algoritmo, podría realizarse primero una selección de métricas, ya que algunas de ellas tienen información más importante en relación con las anomalías, mientras que otras no guardan relación con el hecho de que el punto sea una anomalía o no. Por lo tanto, para implementar un algoritmo más eficiente, algunas podrían combinarse o eliminarse.

Hay una gran variedad de opciones para desarrollar el algoritmo. Una opción muy común para aprendizaje supervisado son las redes neuronales, que además pueden ser muy útiles en problemas de una gran comlejidad. La red neuronal se entrenaría con el conjunto de datos obtenido para clasificar los puntos y determinar si son anomalías. Adicionalmente, otras técnicas o mecanismos podrían utilizarse para mejorar su exactitud. Por ejemplo, podría reentrenarse la red con falsos positivos para evitar clasificar como anomalías puntos que no lo son.

Una vez se ha implementado el clasificador, se puede implementar un plug-in conectado a Prometheus. El principal problema que superar son las latencias al recopillar las métricas de Prometheus. Para ello hay distintas opciones, como una selección de atributos, solicitar las métricas en intervalos de tiempo mayores (en lugar de cada segundo) o estudiar la producción de anomalías después y no a tiempo real. La generación de alarmas se puede realizar a través de la API de Prometheus Alertmanager cuando una anomalía es detectada.

Finalmente, se ha logrado cumplir el objetivo del trabajo ilustrando un procedimiento para implementar un plug-in de detección de anomalías. Se ha descrito el sistema y su despliegue y se ha generado y analizado un conjunto etiquetado de datos con anomalías. Los resultados sugieren que este conjunto de datos o uno obtenido de forma similar se pueden utilizar para entrenar satisfactoriamente un algoritmo de aprendizaje automático que recopile datos de Prometheus y active alarmas cuando se detecten anomalías; proporcionando una solución aplicable para mejorar la monitorización en sistemas distribuidos.

Abstract

Diego Martínez Baselga: Anomaly detection with Prometheus
Bachelor's thesis
Tampere University
Bachelor's Degree Programme in Computer Science
April 2020

Prometheus is a widely used application to monitor Kubernetes systems. Nevertheless, it does not provide a suitable solution to detect complex anomalies. This thesis discusses the deployment of a Kubernetes system that uses Kafka. Moreover, a microservice is implemented to generate anomalies and a labeled time-series dataset is generated.

The produced dataset can be used to develop a machine learning algorithm for anomaly detection. In addition, the study explains the tools to understand the dataset and how to use it to develop a plug-in that predicts anomalies and fires Prometheus Alertmanager alarms.

Keywords: Kubernetes. microservice, anomaly, dataset, Kafka, Prometheus.

Contents

1 Introduction

In the recent years, distributed systems have become more important and are the main option in production environments. They offer reliability, efficiency and scalability, as well as features to deal properly with big data. Among distributed systems, containers and particularly Kubernetes, are a usual choice to deploy applications.

The monitoring system most used in Kubernetes is Prometheus. Prometheus is an open-source platform that is able to scrape metrics from other applications and Kubernetes elements. Those metrics may be graphed or requested to visualize and operate with them. In addition, it has an alert manager that is able to fire alarms and set triggers when any metric is out of some specified values.

Prometheus Alertmanager is only able to fire alarms if metrics do not fulfill mathematical expressions. However, anomalies are much more complicated than that. Anomalies are events that provoke a strange behavior of the system. Therefore, it is important to know when an anomaly has been produced. Unfortunately, what Prometheus offers may not be enough to do it.

The aim of this thesis is to explain and deploy a real Kubernetes system that could be used in production, generating a labeled time-series dataset with anomalies produced by a microservice, as well as it aims to analyse the ways an anomaly detection plug-in could be implemented to detect those anomalies. Hence, it could be used to predict anomalies in the system deployed or the approach could be extrapolated to be used in another different system. Kafka, a distributed streaming application, runs in that system too, to make it as close as possible to a non-simulated environment.

This document is structured as follows. Chapter 2 states the background of the thesis, as well as studies some important definitions and concepts related to it. Additionally, it provides some related works so that the approach may be examined and compared with others. Chapter 3 explains the approach selected for this research, all the implementations and designs, and the deployment of the system. Chapter 4 shows and illustrates the results. Finally, Chapter 5 analyze the results pointed out before and their importance for anomaly detection solutions.

2 Theoretical background

The background of this thesis describes the different technologies that have been used and related works that have been considered as inspiration or may be examined as further research.

2.1 Docker

Docker is an open-source project that allows packaging and deployment of applications in containers. Containers are virtualization software that differ from virtual machines in the way they obtain resources from the computer host they are running on. On one hand, virtual machines use a hypervisor, which sets resources at hardware level, running each virtual machine completely isolated from the others. On the other hand, containers virtualize at operating system level. Hence, they share resources without being aware of it.

Docker provides a lightweight way to deploy applications. They only use the resources they need and they do not need to start and destroy a whole operating system to work, as they share the host's operating system kernel. Docker containers also have a huge number of images to work as the basis of the containers. Thus, containers are a fast and easy way to deploy applications, defining its dependencies and its needs. That is why they are commonly used to encapsulate microservices to be deployed and maintained. [21]

2.2 Kubernetes

Kubernetes is an open-source platform to manage containerized services. It is used to automate the deployment and scaling of applications, to transform applications into a complete, reliable and scalable distributed systems.

With Kubernetes, more than one node may be connected to a single cluster to provide a fully distributed environment. It may be done also using cloud solutions like Amazon Web Services or Microsoft Azure.[3]

In Kubernetes architecture, there is one master node, which manages the cluster and its communication, and the worker nodes, which are the ones that execute the workload. It provides different abstractions to deploy microservices:

- Pods: Groups of containers with different images grouped in the same unit. They are the smallest deployable unit.[17]

- ReplicaSets: Controllers that ensure that the desired number of replicas of the pod are working.[18]

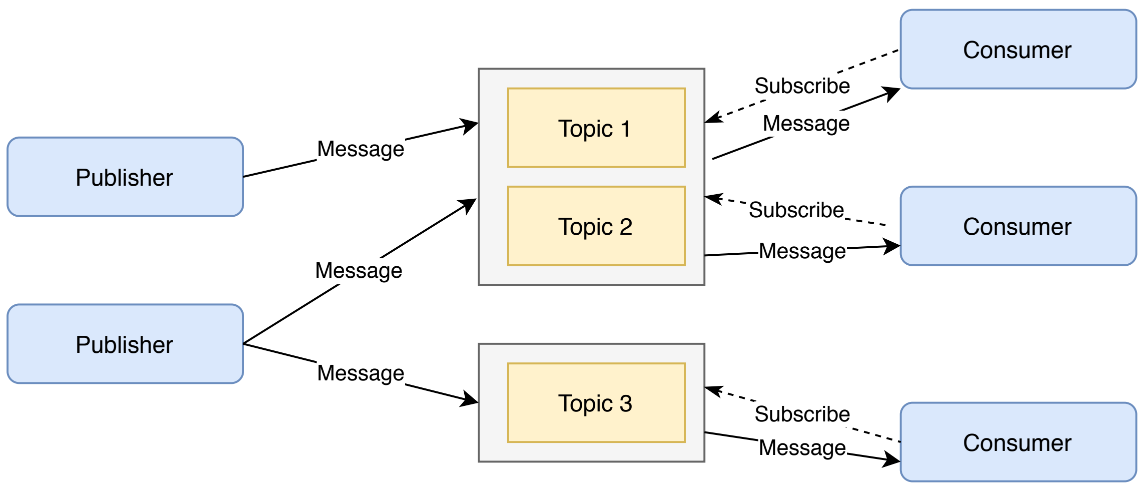


Figure 2.1 *Publish-subscribe pattern. It is used in Kafka application.*

- Deployments: They are declarations that update the state of the cluster to the one stated. For instance, they create ReplicaSets and ensure that they work.[16]

- Services: They isolate microservices from one another and provide Pods their own IP so that they can be externally accessed. They also provide load-balancing across them.[19]

There are more different concepts, but these are the main ones used in the thesis and the basic ones to help to understand how Kubernetes work.

2.3 Kafka

Kafka is a streaming application based on the publish-subscribe architecture, where producers may publish streams of records so that consumers may receive them. This pattern is illustrated in Figure 2.1. Messages are organized by topics, which are key-words to group the ones that belong to the same class. Then, brokers store those messages. Consumers subscribe to topics and then receive from brokers all the data published to that topic.[15]

Kafka allows a fault tolerance as well as real-time streaming messages. This is used for two different approaches: Systems that need to share and transfer data in real-time, and systems that need to act when data is transferred.[12] The most common Kafka applications are ”messaging, website activity tracking, metrics, log aggregation, stream processing, event sourcing and commit log”.[14]

2.4 Prometheus

Prometheus is an open-source monitoring system. It provides real-time monitoring data of a system and tools to display the data and operate with it. It is commonly used in container environments and, particularly, to monitor Kubernetes.

In order to monitor a system, Prometheus gathers metrics from it. Metrics are measures that different parts of the system are continuously sending to Prometheus. Some metrics collected are related to Prometheus performance, pods state, nodes usage or applications output. For example, in Kafka case, metrics as bytes consumed or outgoing byte rate may be collected.

Moreover, Prometheus offers more options and resources to try to improve system's performance. It has an API where metrics may be exported to use them outside Prometheus, and it is possible to generate metrics and insert them in Prometheus. Furthermore, it provides a tool, Alertmanager, that allows to fire alerts and set triggers if a metric violates a mathematical expression i.e. when a metric is greater or less than a value. [31]

2.5 Related works

There are several projects related to this thesis. One example is a platform created by Facebook to generate realistic time-series with anomalies called AnoGen. It was developed in order to generate anomalies in a deterministic way due to the absence of labeled datasets and the complicated task of creating one with realistic anomalies. The result is a way to generate a labeled dataset that can be used to improve machine learning methods of anomaly detection. [20]

In another work [27], detecting and categorizing anomalies is discussed using supervised machine learning in a multi-cloud environment with the UNSW dataset. In [9], an Extended Isolation Forest algorithm is used to detect anomalies in a Kubernetes environment.

In [30], a study where a system called KubAnomaly of anomaly detection in Kubernetes is provided. It uses neural network based approaches to detect attacks and vulnerabilities on three types of datasets.

The thesis [5] is about detecting anomalies of microservices deployed in a cloud environment. It states that, as microservices are continuously changing and the changes they undergo are huge, detecting whether they are provoked by anomalies or not is hard. The approach used is tested in a Kubernetes environment but does not use machine learning.

There are a great amount of examples of datasets where anomalies have been generated intentionally. For instance, [28] records a 4998-point dataset of a network that experimented different types of DoS attacks and a Brute Force attack.

In the approach shown in [4], a dataset for NFV (Network Function Virtualization) is created. A Kubernetes environment is simulated to make it similar to a production environment. Then, three types of anomaly datasets are produced using fault injection, measuring workload, fault-load and performance.

More examples of datasets are the datasets UNSW-NB15 and KDD99, which are analyzed and evaluated in [22]. In that paper, both datasets are examined to determine its accuracy and the efficiency of their features.

Finally, as the last example of related work, [2] describes a monitoring system where Prometheus is used as a time-series database. It analyses the usefulness and suitability of Prometheus and Prometheus Alertmanager for that task.

3 Research methodology and materials

The methods used in this thesis are described in this chapter. Every step taken is explained and justified in different sections, from testing and understanding the different technologies, to generating the dataset.

3.1 Deploying Kafka and Prometheus locally

The first step was setting up Kafka locally following the quick-start guide offered by its official site [13]. To do it, it is necessary to start ZooKeeper first. ZooKeeper is an open-source software that provides services to coordinate distributed systems. It helps to provide distributed consensus and other facilities to distributed applications.[1]

Then, Kafka server is started (this is the broker) and a topic called 'test' is created, using the scripts provided by Kafka. Kafka producers and consumers may be created then. The approach made to test it is to implement a simple program in C++ that writes continuously bytes at different frequencies.

Prometheus and Prometheus Alertmanager are installed. In order to configure correctly and easily Prometheus to scrape Kafka elements, a configuration file [26] is used when it is started, as it is showed by Program 3.1.

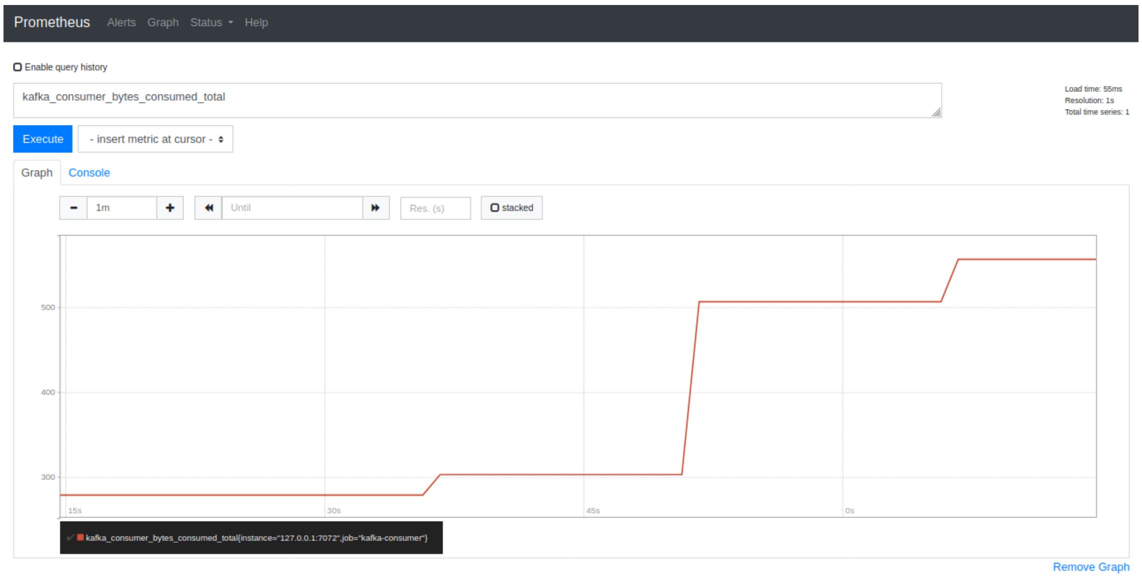
```
global:
  scrape_interval:     15s # Set the scrape interval to every
                           # 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.
                           # The default is every 1 minute.

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - localhost:9093

# A scrape configuration containing endpoints to scrape:
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any
  # timeseries scraped from this config.
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']
  # Kafka added statically for simplicity, instead of using
  # a service-discovering mechanism
  - job_name: 'kafka-server'
    static_configs:
    - targets: ['127.0.0.1:7071']
  - job_name: 'kafka-consumer'
    static_configs:
    - targets: ['127.0.0.1:7072']
  - job_name: 'kafka-producer'
    static_configs:
    - targets: ['127.0.0.1:7073']
```

Program 3.1 *Configuration file of Prometheus to launch it locally with Kafka and Prometheus Alertmanager*

Prometheus GUI can be accessed by a web browser to check that everything is working correctly. An example of how to verify it is given in Figure 3.1, where the metric of bytes consumed by Kafka consumer is set in a graph.



kafka_consumer_bytes_consumed_total

Load time: 55ms
Resolution: 1s
Total time series: 1

kafka_consumer_bytes_consumed_total{instance="127.0.0.1:7072",job="kafka-consumer"}

Figure 3.1 *Prometheus GUI that shows how Kafka consumer is getting bytes by a producer. This shows that the system has been deployed correctly.*

Prometheus offers a HTTP API to get the metrics stored on "/api/v1" [24]. This allows to develop external programs that treat and operate with the metrics obtained. Metrics may be accessed with two different endpoints. The endpoint "/api/v1/query" evaluates the query at a single point, whereas the endpoint "/api/v1/query_range" does it over a range of time.

In addition, there is an endpoint available in the Prometheus Alertmanager that allows to fire alerts. The endpoint may be accessed at "/api/v1/alerts", in the port opened for the Alertmanager. A few programs in Python were implemented in order to check the functionalities of these endpoints. Particularly, programs were implemented to store and get from the disk metrics provided by Prometheus, fire an alert if a specific metric (”jvm_memory_bytes_used”) was above a certain value for 10 seconds or more and fire an alert if the mean of the last 10 seconds of the same metric was above that certain value.

The goal of running Kafka and Prometheus locally was deploying the system in a simple scenario and learn how to interact with it. The Python programs implemented simple aspects that were later used in a more complex purpose.

3.2 Deploying the system in Kubernetes

The approach that was decided to take is deploying Kubernetes in 4 virtual machines, one as the master and the other three as the workers. The computer that was used at the beginning was a 4-core CPU and 8GB RAM computer. Different operating systems for the virtual machines were tried. First, Ubuntu and CentOS did not work properly because of lack of memory. Then, more lightweight operating systems were used: K3s and RancherOS. Nevertheless, they were discarded because they are commonly used in testing environments rather than production environments [10] [23]; the goal was to deploy a system as close as possible to the reality. Hence, the solution was using a 8-core CPU and 16GB RAM computer and Ubuntu server distribution for the virtual machines.

The 4 virtual machines were created. 2 CPUs and 2GiB of RAM were allocated for every of them. Then Ubuntu 18.04 was installed and all packages were updated. All machines were added to the same network.

Kubernetes was set up with docker; using kubeadm, kubelet and kubectl. The machine that had the lowest IP address was set as the master and the other machines joined it. The CNI Flannel was chosen in order to manage the Kubernetes network. Flannel is a network plug-in that allocates subnets and addresses to every component of a Kubernetes cluster. It is used with the Kubernetes API and it is configured at the beginning of the deployment with the tool kubeadm. [6]

To deploy Kafka, helm charts were used. Helm is a tool used to install, define and upgrade Kubernetes resources in packages named charts. It may be used to easily use popular software instead of defining manifests to do it, reducing the errors and helping to build complex distributions. [7]

The chart used to run Kafka is ”incubator/Kafka” chart, which may be found in Github or in Helm hub of projects [8]. This is an implementation of Kafka as a StatefulSet. A StatefulSet maintains identity and uniqueness of the pods used in the Deployment, ensuring that there is the number of pods specified in the manifest is always running. Pods are identical, but they have different identity, so any of them may be used. They are also dynamically scalable, which means that the number of pods may be increased improving efficiency and performance, as well as fault tolerance.

The implementation of Kafka results in a StatefulSet of 3 Kafka brokers pods and a StatefulSet of 3 Zookeeper pods. A service is created to expose Kafka endpoints to a static port, using a NodePort, allowing other applications from the cluster and outside the cluster to access Kafka brokers. Thus, Kafka producers and consumers may be easily created and connected to the brokers. Furthermore, the chart provides options to configure metrics exported that Prometheus collects. In this case, JMX metrics (metrics provided by Java Virtual Machine, which may be provided by any Java application) as well as other metrics provided by Kafka are exported.

Prometheus was deployed using and editing some manifests found in Github repositories [29]. It was defined as a Deployment of 1 replica and exposed as a NodePort similarly as Kafka. Therefore, Prometheus GUI could be accessed by any web browser using master's IP address and the external port provided by NodePort. Prometheus Alertmanager was set up using the same approach of a Deployment and a Service. Additionally, other two services were launched to generate more metrics for Prometheus. Kube-state-metrics is an official metrics exporter that updates the cluster state [11]. It generates metrics about cluster elements, as Pods and Deployments, and was defined as a Deployment. Besides, Node Exporter was the other metrics generator used. This is a Prometheus tool that gathers hardware information about the nodes in the cluster. This information is, for instance, about nodes memory and CPU. [25]. Node exporter is defined as a DaemonSet in order to ensure that there is always one instance in each node.

In order to provide a persistent and reliable structure of storage for all the system deployed, some manifests were implemented defining new elements. First, two different Storage Classes were defined, one for Prometheus and another for Kafka. Then, a Persistent Volume of 40 GB was defined for Prometheus and 3 of 1 GB each were defined for Kafka brokers; each one of them in a different mount path, so that no information could be overwritten. Finally, a Persistent Volume Claim of 40 GB and another one of 1 GB for each Kafka broker were defined. In that way, the applications were provided with the storage requirements.

To conclude this section, the main elements present on the cluster are 4 deployments (Kafka brokers, Zookeeper, Prometheus and kube-state-metrics) and 1 DaemonSet (Node Exporter), as it is shown in Figure 3.2. Anomalies will be generated in this structure and data will be gathered using those elements.

3.3 Anomalies generation

Once the system was correctly started up and running, anomalies needed to be generated. An anomaly is a strange or unexpected condition that provokes an unusual situation. In this case, the idea is simulating a bad situation, so that

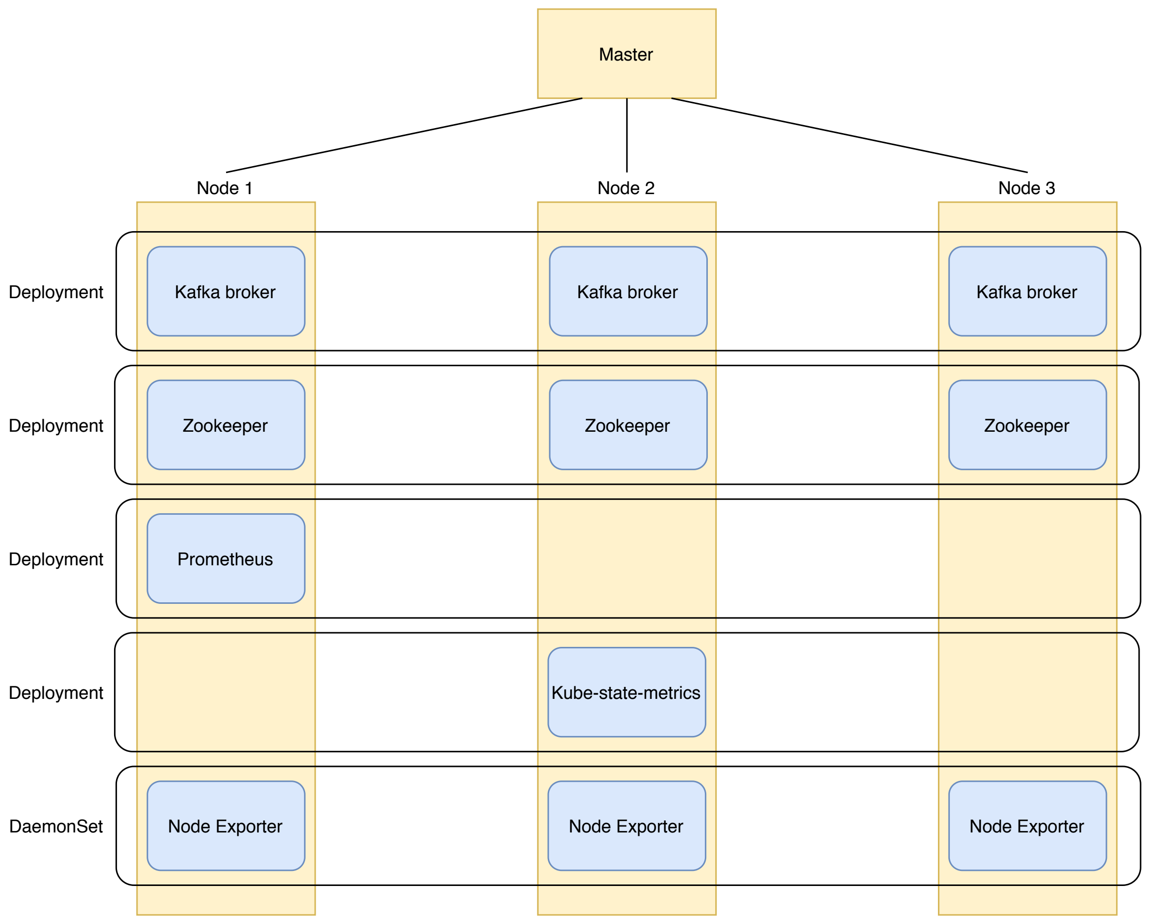


***Figure* 3.2** *Kubernetes cluster. The figure shows all Deployments and DaemonSets of the system deployed in Kubernetes.*

its recognition triggers an alert. Consequently, a specific and controlled problematic anomaly is needed.

The decision was to implement a microservice that would provoke a memory anomaly. The microservice had first to be created and started as a Pod inside the cluster, then connects with a Kafka broker and sends a bytes stream for a few seconds and finally requests memory until the requests exceed the maximum allowed and the container is destroyed. The anomaly was decided to be related to memory because it is a very common problem faced by every kind of organisations and may be effectively monitored.

```
FROM ubuntu:18.04

# Ubuntu operating system and name and email
# of maintainer
MAINTAINER Diego <diego.martinezbaselga@tuni.fi>

# Update tools
RUN apt-get update

# Copy script that produces the anomaly and
# the one that produces the Kafka stream
COPY anomalie_generator.sh /anomalie_generator.sh
COPY kafka_produce.sh /kafka_produce.sh

# Add permissions to execute the scripts
RUN chmod +x /anomalie_generator.sh
RUN chmod +x /kafka_produce.sh

# Start the script as the entrypoint, the main
# application to be executed by the container.
# If the application end, the container is
# destroyed.
ENTRYPOINT /anomalie_generator.sh
```

Program 3.2 *Dockerfile. This is the Dockerfile used to create the container.*

First, the container is implemented. As it is a Docker container, a Dockerfile needs to be designed. In this case, it has to be just a simple one that runs a script that produces the anomalies, as is illustrated in Program 3.2. The directives of the file state the operating system, the maintainer, to copy and change the permissions of the scripts implemented, and finally, to execute the main script as the executable of the container.

The script Program 3.3 is the entry point of the Docker container. It writes in Unix format the moment when the script starts, when the Kafka producer is executed and when the anomaly is generated. This allows to know which data points are considered anomalies and which are regular points. Moreover, it lets the dataset be completed by adding information about the state of the microservice.

In order to run Kafka, it is necessary to install required packages first. The most important one is the Java Runtime Environment, as Kafka is a Java application and needs it to run. After having installed it, Kafka binaries are downloaded and unzipped. When they have been downloaded, they may be executed, as it is done in the script *kafka_produce.sh*. The time is written down in a file before the Kafka producer is started.

The script *kafka_produce.sh* implemented is an infinite loop. It waits for 0.1

seconds and sends 1 byte to a Kafka broker. It uses echo to write a byte and sends the byte through a pipe to the script *kafka_producer_console.sh*, which is included in Kafka binaries. As an argument to the Kafka script, the brokers address is given, which is the IP address of the master and the port provided by the NodePort. Therefore, a rate of 10 bytes per second is obtained.

```
#!/bin/bash

# Write the time when the container starts
date +%s >> /files/times_start.txt

# Install equired packages
apt-get install wget -y
apt-get update
apt-get install default-jre --fix-missing -y

# Download and unzip Kafka executables to send the stream
# to a Kafka broker
wget "https://www.apache.org/dist/kafka/2.1.1/kafka_2.11-2.1.1.tgz"
tar -xzf kafka_2.11-2.1.1.tgz

# Write the time when Kafka stream starts
date +%s >> /files/times_kafka.txt

# Run a Kafka producer in the background
./kafka_produce.sh &

# Sleep for a random number of seconds between 5 and 15
sleep $((5 + RANDOM % 10))

# Write time when anomaly starts
date +%s >> /files/times_anomalies.txt

# Produce anomaly
yes | tr \\n y | head -c $((1024*1024*5000)) | grep a

# Infinite loop. This part of the code never runs.
while true
do
        sleep 1
done
```

Program 3.3 *Script to generate anomalies. This script creates the memory anomaly as well as runs a Kafka producer. It also writes the time when the script starts, when the Kafka producer starts and when the anomaly is generated.*

The Kafka producer is run in the background, so that it sends bytes while the main script is being executed. In order to let the Kafka stream running for a few seconds before the anomaly starts, the function sleep is used. It is forced to wait for a random number of seconds between 5 and 15, so that every execution of the container is a little different. When the waiting time is finished, the time is written down.

After that, the anomalies are generated. The idea of the anomaly is an instruction that incrementally requires memory, until memory requirements exceed the maximum. This is accomplished using the following tools connected by pipes:

- **yes**: It outputs the string 'y' until it is killed. It is used to get bytes indefinitely.
- **tr**: It removes new lines, replacing them with another 'y'. The tool yes produces the character 'y' in different lines. Using tr, a continuous stream of bytes is obtained.
- **head**: It is a control tool as it would not be necessary. It outputs the number of bytes set of the text given. In this case, it sets the limit of bytes generated to 5000 MB. It is set to ensure that a big number of bytes is generated. If the limit was set to 200 MB, the container would not be destroyed.
- **grep**: Grep is a tool that looks for a specified pattern in a text. The reason why it is used is that it loads the text into memory to search the pattern.

The result of using the tools listed is that grep loads into memory up to 5000 bytes. Nevertheless, when the container reaches the maximum allowed, it is destroyed. The last lines of the script are more instructions to prove that the script has performed its function correctly. If the container is destroyed, the infinite loop is not reached and the container restart may be checked. If the container was not destroyed, the script would not finish and it would never be restarted. This may be forced, for example, by setting the bytes limit in the head tool to 200 MB. That would mean that the anomaly would not have been provoked successfully.

The files and scripts were uploaded to the docker repository *dmartinezbaselga*, so that they may be used and accessed by Kubernetes manifests and set up in a Kubernetes cluster. This process of uploading the container to a Docker hub was performed using docker build and docker push commands.

A Storage Class, a Persistent Volume Claim and a Persistent Volume of 2 GB were created in order to provide a persistent storage to the files that record the times of the script. The Persistent Volume manifest (Program 3.4) has a special interest because it was forced to be always in the same node. This is important because Kubernetes does not ensure that when the pod is deleted and the Persistent Volume Claim is rearranged, the files are the ones that were created before in that node instead of new ones in a different node.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  # Name of the Storage Class
  storageClassName: anomalies
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    # Path where is mounted in the physical node
    path: "/mnt/anomalies"
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        # Make the volume be mounted in node1
        - key: kubernetes.io/hostname
          operator: In
          values:
          - node1
```

Program 3.4 *Persistent Volume manifest. Manifest that creates a 2 GB persistent volume that is mounted in the node called node1.*

The container is set up in the cluster as a single pod, with the manifest Program 3.5. In this manifest three aspects are stated. The first one is the container image. The one which implementation is explained before is referred. The latest one is pulled. Secondly, resources limits. The most important one is the memory limits, as the anomaly generated is related to the memory. 400 MB are chosen as the limit. If the container requests more than 400 MB, it will be destroyed. Finally, the Persistent Volume Claim.

The name of the PVC created is the one specified in the manifest. In addition, the volume is mounted in the path where the times files are stored. Consequently, there is a correspondence between the path inside the Pod and the path inside the node. Files stored in */files* inside the Pod are the same as the ones that are present in */mnt/anomalies* in the node named node1. This approach lets the files be modified inside the Pod writing the times when the microservice perform each action, reuse the same files even though the Pod is deleted and created again, and access those files from outside the Pod implementing an external program.

```
apiVersion: v1
kind: Pod
metadata:
  name: anomalies
spec:
  volumes:
  # PVC created
  - name: anomalies-pv
    persistentVolumeClaim:
      claimName: task-pv-claim
  containers:
  - name: anomalies
    # Docker image created
    image: dmartinezbaselga/anomalies
    resources:
      limits:
        # If the pod requires more than this limits,
        # it is destroyed
        memory: "400Mi"
        cpu: "500m"
    volumeMounts:
    # Mount the PV in the path where the times files
    # are stored
    - mountPath: "/files"
      name: anomalies-pv
```

Program 3.5 *Pod manifest. Manifest that creates a Pod using the Docker image designed before. A Persistent Volume Claim is required to run this manifest successfully.*

The microservice designed introduced to Kubernetes as it has been explained allows to generate a time series anomaly dataset that may be extracted from Prometheus as well as from the times files created.

3.4 Dataset generation

The microservice designed is introduced as a Pod to Prometheus. Once the Pod is created, it starts to generate metrics. Metrics are stored in Prometheus and may be extracted using an external program by using Prometheus API.

The Pod was created and continuously being executed in two different time intervals. For every interval, 5005 data points were obtained. A single data point was obtained for each second. Therefore, the microservice had to be deployed for at least 5005 seconds in each time interval.

In order to get the metrics from Prometheus, a Python program that interacted with its API was implemented. In Program 3.5, the fundamental lines of the code of this program are included. They are explained in the following paragraphs.

The endpoint accessed was */api/v1/query_range*. To reach the endpoint, the program uses the master IP address and the eternal port offered by Prometheus service.

```python
for i in range(5):
    # time1 is the time when the range starts and time2 when it ends
    time1 = str(t)
    time2 = str(t + 1000)
    # Open the file that stores metrics names
    with open(file_metrics_names) as fp:
       # Read all metrics names and iterate through them
       line = fp.readline()
       while line:
           # Request the metric for the interval of time specified
           obj = {'query': line.strip(), 'start': time1, 'end': time2,
                  'step': '1s'}
           x = requests.post (url, data = obj)
           # Open a file named with the name of the metric inside the
           # corresponding folder
           f = open(save_path + "/metrics" + str(ext) + "/" +
                    line.strip() + ".txt", "w")
           # Writes the content of the metric
           f.write(str(x.json()))
           f.close()
           line = fp.readline()
    # Get the next 1001 points in the next metrics folder
    ext = ext + 1
    t = t + 1001
```

Program 3.6 *Request metrics from Prometheus. Part of the code of a Python program that requests the metrics from Prometheus as it is explained.*

The query range endpoint allows to get the value of a specified metric from one moment to another. It takes as arguments the name of the metric, the time when the interval starts, the time when it ends and the step. The step means the times difference between one data point and the following. It is set to 1 second, so that 1 data point per second is obtained. The times when the time interval is taken are set manually.

Prometheus API offers and endpoint to get metrics names. However, only local server metrics are listed. Thus, another approach should be taken to get all metrics names. In this case, web browser inspector was used. In Prometheus GUI, there is a form where the metric that is showed may be selected and where all metrics are listed. The inspector is used to get these values and they were copied to a file. This file was formatted so that its content was just one file name per line.

Prometheus resolution for queries is 11000 points. Nevertheless, latencies provoked errors that made Prometheus fail and queries had to request a smaller amount of points each time. 5 requests of 1001 points each were made in each of the time intervals where anomalies were generated.

The data received from Prometheus is in json format. It is stored in the same format in text files. Ten folders of metrics are created, one for each time range. Inside each folder, there is one file for each metric, named as the specific metric name. They contain the json received from Prometheus exactly as it is received.

Then, it was necessary to change the format of the data to make it readable, as well as adding more information about the anomalies using the times files. Three Python programs were implemented to do it.

The first program was implemented to delete inconsistent metrics. Metrics obtained from Prometheus may be in two formats: a vector of tuples, being each tuple the time of the point and the metric value; or a vector of vectors of tuples time-value. Hence, these second metrics had more than one value for each point. For example, Kafka metrics had 3 values, one from each broker. Therefore, the aim of the first program was deleting the metrics that were not present in every point or did not have the same number of values for every point. These inconsistent metrics would not have an important role in an algorithm that predicts anomalies, as they appear just in particular moments.

The second program creates a file that contains a table of the anomalies with the information related to them, as the time when they started and the time when they finished. It is not possible to measure when the container is destroyed. As it is being destroyed externally, the microservice script could not write it down. In addition, logs are deleted when the Pod starts again. Thus, some anomalies were empirically measured in different times and the container was destroyed around 20 seconds after the anomaly started (3-tenths of seconds of error). As a consequence, the end of the anomalies were listed 20 seconds after they started. In order to create this file, metrics were not needed. The only files that were used were the file that registered anomalies times and the file that registered Kafka producer times. They were used to describe the anomalies generated.

The main goal of the last Python script was to get all of the metrics and write them together in a single file. This file had to be structured as a human and computer readable table. Each point needed to have its time, if it is an anomaly or not, the state of the microservice and all its metrics. The table was generated sequentially, setting up all of the table entrances from one of the 10 folders before working with the following one. After obtaining the entrances of the folder, they were appended to the output file.

The first column added was the times of the points; which were taken from a random metric. The table was then initialized containing just the times. Whether the point is an anomaly or not, and the state of the microservice at that moment is derived from the 3 times files. Finally, a big loop iterated the anomalies files, getting all the metrics values and all the vectors of metrics; adding them to the table. When every file has been read, the output file is edited, appending the information to the end of the file.

To conclude, two different tables were the result of this approach: The one that contains anomalies information and the one of the metrics. The data was successfully extracted from Prometheus and formatted so that it is easy to use to design anomaly detection applications.

4 Results

The result of the deployment is 2 different tables. One table describes anomalies generated and the other one states data points with their metrics. Both tables were uploaded to Gitlab so that they were used to train a machine learning algorithm to predict anomalies.

The first file name is table_anomalies.txt. A few entrances of the table are represented as an example in the Table 4.1. The first line of the file is the header, which contains the following fields:

```
Timestamp_from, Timestamp_to, Anomaly type, Anomaly details
```

The table is made up of 30 entrances, corresponding to the 30 anomalies generated. 14 of them were produced in the first interval of time and 16 in the second one.

Table 4.1 *Example of 5 entrances of anomalies dataset.*

Timestamp_from	**Timestamp_to**	**Anomaly type**	**Anomaly details**
1579033135	1579033155	Request memory until container is destroyed	Kafka runs for 11s before anomaly starts
1579033582	1579033602	Request memory until container is destroyed	Kafka runs for 10s before anomaly starts
1579033997	1579034017	Request memory until container is destroyed	Kafka runs for 8s before anomaly starts
1579034411	1579034431	Request memory until container is destroyed	Kafka runs for 7s before anomaly starts
1579034846	1579034866	Request memory until container is destroyed	Kafka runs for 13s before anomaly starts

Timestamp_from and Timestamp_to are the timestamps when each anomaly starts and finishes. From these times, it may be seen that the common behavior is producing the anomalies with a difference of time between 414 and 455 seconds. Nevertheless, there are 5 cases where anomalies were produced with a difference of less than 200 seconds, 1 less than 300 seconds (and more than 200) and 1 less than 400 (and more than 300). In addition, the difference of time between the last anomaly of the first time range and the first anomaly of the second is 43612 seconds.

The Anomaly type field is the same for all the anomalies. It states that is an anomaly that requests memory until the container is destroyed. Its purpose is to clarify that aspect and make the possibility of integrating it with other metrics or anomalies easier.

The last field of the table points out the details of the anomaly. This indicates the number of seconds that the Kafka producer has been sending messages before the anomaly starts. There are all between 5 and 15 seconds.

The second file created is table_metrics.txt. An example of a few entrances of the table are represented in the Table 4.2. Similarly, its content is a table where all the data points are listed, one data point in each row. The header of the table is the following:

```
Time | Anomaly | Instance | Metric 1 | Metric 2 | ... | Metric n
```

The field Time is the timestamp of the data point. One data point is taken each second of the intervals. The first point has a timestamp of 1579033025, the last point off the first interval is at 1579038029; and the second interval range is between 1579081912 and 1579086916. Thus, there is a gap of 43883 seconds between both intervals.

Table 4.2 *Example of 5 entrances of anomalies dataset.*

Time	**Anomaly**	**Instance**	**Node__load1__0**	**Kafka__server__brokertopicmetrics__bytesinpersec__count__2**	**machine__memory__bytes__0**
1579033025	0	Waiting for Kafka stream to start	0.21	207	2089807872
1579033132	0	Kafka running	1.2	276	2089807872
1579033142	1	Generating anomaly	1.47	414	2089807872
1579033167	0	Waiting for the script to start	0.97	552	2089807872
1579034005	1	Generating anomaly	1.36	828	2089807872

The second field, Anomaly, is a Boolean attribute that takes the value 1 if the data point is an anomaly and 0 if it is not. There are 600 points considered anomalies.

The Instance is the state of the microservice in each point. It may take 4 different values:

- **Waiting for the script to start**: The Pod is being created after having been destroyed. The Pod has to be destroyed and associated to the Persistent Volume; and the system has to check whether the container image stored is the last one available and start the Pod. It usually takes between 300 and 350 seconds, but there are some outliers.
- **Waiting for the Kafka stream to start**: Time between the container starts and the Kafka producer starts to send a bytes stream. The bottleneck of this phase is downloading Kafka binaries. There are between 87 and 118 points off this phase each time.
- **Kafka running**: Kafka producer sends bytes but the anomaly is not being generated. The microservice is between 5 and 14 seconds in this phase.
- **Generating anomaly**: The anomaly is being generated. This takes 20 seconds and it is the time during which the anomaly is started and the container is destroyed.

The next fields are the metrics. There is a single value for every point in every metric. Some metrics exported from Prometheus had more than one value. For example, Kafka metrics have a different value from each broker. In those cases, the metric has been named using its name, a backslash and a number meaning the number of the instance. Illustrating it, a metric named Metric would have the following instances : Metric_0 | Metric_1... The number of metrics presented in the table is 7062.

To sum up, two files are the resulting dataset. One file contains the 30 anomalies generated and the other one 10010 data points, 600 of which are anomalies. The metrics dataset has 7062 metrics types, which is a total of 70690620 metrics points. Moreover, an explanation of the system, how to deploy them and its interest is provided. Furthermore, the tools to understand the meaning of the elements of the thesis are offered.

5 Results analysis

In this thesis, a time series dataset has been generated. The aim of this dataset is to be used to train a machine learning algorithm and implement a plug-in that predicts anomalies.

In order to implement an algorithm, a selection of metrics could be done first. It is understandable that there are some metrics that would have more critical information regarding the anomalies than others. Therefore, some metrics could be combined or deleted to find a more efficient algorithm.

There is a huge variety of options to develop the algorithm. A very common technique for supervised learning is using neural networks. Neural networks are a set of perceptrons interconnected in different layers. They have different classification functions and parameters. The idea of using a neural network is training it with the metrics dataset to make it classify points in regular points or anomalies. In addition, some other mechanisms may be used to improve the correctness of the classifier. For example, it could be retrained with false positives, in order to prevent the classifier from labeling as anomalies points that are not.

Once the classifier has been implemented, a plug-in connected to Prometheus may be implemented. A challenge to be faced is the latency of getting the metrics from Prometheus. If a selection of metrics has been done before implementing the classifier, the latency would be smaller. However, a real-time plug-in is a hard task, as getting all the metrics from Prometheus is likely to take more than 1 second. Another possibility is requesting metrics less often, for instance, each 5 seconds; or studying the possible anomalies once they have been produced instead of doing instantly.

After getting the metrics, Prometheus Alertmanager may be used in order to fire alarms, as it has been seen in Section 3.1. With this proposal, the system is alerted and triggers might be set. A complete anomaly detection system would be deployed then.

After all, this dataset is the first step for implementing an anomaly detection plug-in, a system that deals with anomalies or a solid anomaly study. A wide range of possibilities is opened that require more research. Nevertheless, the thesis offers a suitable introduction to a Kubernetes system with Prometheus and anomaly detection in cloud and distributed systems.

6 Conclusions

This thesis has explained an approach to detect anomalies in a Kubernetes system. The system and its deployment have been described, a labeled dataset of Prometheus metrics that contains anomalies has been generated by a microservice and everything has been analyzed.

One of the files created contains information about the anomalies provoked. The other file is a table of data points labeled with whether or not they are anomalies and the state of the microservice that generated them. The dataset is made up by 10010 points, 600 of them anomalies, with 7062 metrics each.

The focus of this thesis has been fulfilled illustrating the procedure of implementing an anomaly detection plug-in and the use of the dataset generated in order to do it. The results suggest that this dataset or a similar one could be used effectively to train a machine learning algorithm that scrapes Prometheus metrics and fires and alarm in Prometheus Alertmanager if an anomaly is detected, providing an applicable solution to improve monitoring in distributed systems.

References

[1] *Apache Zookeeper*. Available: `https://zookeeper.apache.org/`. 2020.

[2] Radu Boncea and Ioan Bacivarov. "A System Architecture for Monitoring the Reliability of IoT". In: *Proceedings of the 15th International Conference on Quality and Dependability*. 2016, pp. 143–150.

[3] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes*. Dpunkt, 2018, pp. 1–11.

[4] Qingfeng Du et al. "An approach of collecting performance anomaly dataset for NFV Infrastructure". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2018, pp. 59–71.

[5] Thomas F Düllmann. "Performance anomaly detection in microservice architectures under continuous change". Available: `https://elib.uni-stuttgart.de/bitstream/11682/9083/1/MScThesis-TFDuellmann.pdf`. MA thesis. 2017.

[6] *Flannel*. Available: `https://github.com/coreos/flannel`. 2020.

[7] Cloud Native Computing Fundation. *Helm*. Available: `https://helm.sh/`. 2020.

[8] Cloud Native Computing Fundation. *Kafka 5.0.1 for Kubernetes*. Available: `https://hub.helm.sh/charts/incubator/kafka`. 2020.

[9] Sahand Hariri and Matias Carrasco Kind. "Batch and online anomaly detection for scientific applications in a Kubernetes environment". In: *Proceedings of the 9th Workshop on Scientific Cloud Computing*. Available: `https://dl.acm.org/doi/pdf/10.1145/3217880.3217883`. 2018, pp. 1–7.

[10] Mehedi Hasan. *Best Linux Server Distro*. Available: `https://www.ubuntupit.com/best-linux-server-distro-top-10-compared-recommendation/`. 2020.

[11] Elana Hashman. "Operating Within Normal Parameters: Monitoring Kubernetes". In: (2019). Available: `https://www.usenix.org/sites/default/files/conference/protected-files/sre19amer_slides_hashman.pdf`, p. 17.

[12] *Kafka Introduction*. Available: `https://kafka.apache.org/intro`. 2020.

[13] *Kafka Quickstart*. Available: `https://kafka.apache.org/quickstart`. 2020.

[14] *Kafka Use cases*. Available: `https://kafka.apache.org/uses`. 2020.

[15] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. Available: `http://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf`. 2011, p. 2.

[16] *Kubernetes Deployment*. Available: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/`. 2020.

[17] *Kubernetes Pod Overview*. Available: `https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/`. 2020.

[18] *Kubernetes ReplicaSet*. Available: `https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/`. 2020.

[19] *Kubernetes Service*. Available: `https://kubernetes.io/docs/concepts/services-networking/service/`. 2020.

[20] Nikolay Laptev. *AnoGen: Deep Anomaly Generator*. Tech. rep. Available: `https://research.fb.com/wp-content/uploads/2018/11/AnoGen-Deep-Anomaly-Generator.pdf`. Technical Report. Facebook. https://research. fb. com/wp-content/uploads ..., 2018, pp. 1–3.

[21] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014). Available: `https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf`, pp. 1–2.

[22] Nour Moustafa and Jill Slay. "The significant features of the UNSW-NB15 and the KDD99 data sets for network intrusion detection systems". In: *2015 4th international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. Available: `https://www.researchgate.net/profile/Nour_Moustafa4/publication/312184006_The_Significant_Features_of_the_UNSW-NB15_and_the_KDD99_Data_Sets_for_Network_Intrusion_Detection_Systems/links/58a288d045851598babaf088/The-Significant-Features-of-the-UNSW-NB15-and-the-KDD99-Data-Sets-for-Network-Intrusion-Detection-Systems.pdf`. IEEE. 2015, pp. 25–31.

[23] Brian Turner Nate Drake. *Best Linux server distro of 2020*. Available: `https://www.techradar.com/best/best-linux-server-distro`. 2020.

[24] *Prometheus HTTP API*. Available: `https://prometheus.io/docs/prometheus/latest/querying/api/`. 2020.

[25] *Prometheus Node Exporter*. Available: `https://github.com/prometheus/node_exporter`. 2020.

[26] *Prometheus, Getting Started*. Available: `https://prometheus.io/docs/prometheus/latest/getting_started/`. 2020.

[27] Tara Salman et al. “Machine learning for anomaly detection and categorization in multi-cloud environments”. In: *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*. Available: `https://arxiv.org/pdf/1812.05443.pdf`. IEEE. 2017, pp. 97–103.

[28] Salvatore J Stolfo, Ke Wang, and Janak Parekh. *Systems, methods, and media for outputting a dataset based upon anomaly detection*. US Patent 8,381,299. Feb. 2013.

[29] Vaibhav Thakur. *K8s monitoring*. Available: `https://github.com/Thakurvaibhav/k8s/tree/master/monitoring`. 2020.

[30] Chin-Wei Tien et al. “KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches”. In: *Engineering Reports* (2019). Available: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12080`, e12080.

[31] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018, pp. 6–8, 18, 170.