An Architecture for Resource Analysis, Prediction and Visualization in Microservice Deployments

Xavier Geerinck

Supervisors: Wim Van Den Breen, dhr Pieter Leys (Cisco)
Counsellor: dhr. Stefaan Vander Rasieren (Cisco)

Master's dissertation submitted to obtain the academic degree of
Master of Science in Information Engineering Technology

Department of Information Technology
Faculty of Engineering and Architecture
In conjunction with Cisco Systems
Academic Year 2016 - 2017

An Architecture for Resource Analysis, Prediction and Visualization in Microservice Deployments

Xavier Geerinck

Supervisors: Wim Van Den Breen, dhr Pieter Leys (Cisco)
Counsellor: dhr. Stefaan Vander Rasieren (Cisco)

Master's dissertation submitted to obtain the academic degree of
Master of Science in Information Engineering Technology

Keywords

Architecture, Machine Learning, Prediction, Regression, Analysis, Visualization

Abstract

Foreknowledge is one of the critical factors for any enterprise to outdistance the competition in a timely manner. When a cloud enterprise is not able to detect scaling problems within a timely manner, this can lead to decreasing revenues and the occurrence of potential bottleneck issues. Resulting in a decreased customer satisfaction and Service Level Agreement violations due to the decreased uptime of these services.

This thesis tackles this by introducing a multi-layered architecture that is able to analyse, visualize and predict the resource usage of microservices in a large scale deployment. Through the use of this analysis, prediction models can be created that are able to issue warnings before the occurrence of bottlenecks. The enterprise can then use these warnings to devise a new strategy for the microservices deployments to comply to the Service Level Agreements and increase the customer satisfaction.

From the results obtained in this thesis, it was made clear that the introduced architecture is able to grow, iteratively learn by itself and return the models needed. All of this is achieved, while still maintaining an architecture that is modular enough to adapt to the different complex enterprise structures.

Xavier Geerinck, Januari 2017

Abstract Dutch

Voorkennis is een van de meest kritische factoren die beslissen of een onderneming in staat is om op tijd te reageren op de competitie en deze voor te blijven. Wanneer een cloud-onderneming dit niet op tijd detecteert, kan dit leiden tot verminderde inkomsten en het voorkomen van mogelijke bottleneckproblemen. Wat leidt tot verminderde klantentevredenheid en een mogelijke impact op de verschillende Service Level Agreements omwille van de verminderde uptime van deze services.

Deze masterproef pakt dit aan door een meerlagenarchitectuur te introduceren die het mogelijk maakt om microservices te analyseren, visualiseren en hun resourcegebruik te voorspellen in grote microservice omgevingen. Door gebruik te maken van deze analyse en voorspellingsmodellen kunnen we waarschuwingen weergeven vóór deze problemen zich stellen. Een onderneming kan deze waarschuwingen dan gebruiken om een nieuwe strategie uit te denken om aan hun Service Level Agreements te voldoen en de klantentevredenheid te verbeteren.

Uit de gevonden resultaten kan duidelijk gemaakt worden dat de geïntroduceerde architectuur kan groeien, zelfstandig kan bijleren en de noodzakelijke modellen kan teruggeven. Dit terwijl deze nog steeds modulair genoeg is om aan de complexe eisen van verschillende bedrijfsstructuren te voldoen.

Xavier Geerinck, Januari 2017

Acknowledgments

I would like to start by thanking Cisco Systems for giving me the chance and providing the utilities needed to create this thesis.

Next to that I would like to start by thanking Mr. Pieter Leys who was my promotor at Cisco Systems, whom without I would never have been able to get this far and create such a finetuned thesis.

I am also grateful to Mr. Stefaan Vander Rasieren, who guided me during my thesis at Cisco. With his help, I was able to find a subject that closely matches my interests. He also brought me into contact with the right persons whenever I wanted to discuss an idea. His continued support and availability towards me contributed to this thesis in an unimaginable way.

Thanks to Mr. Wim Van Den Breen who was my promotor assigned to me by my University, for his continued guidance and help.

Also a big thanks to my family and close friends, who have been a big help on both the emotional and technical side. Without them I would never have had the courage or the perseverance required to continue this dissertation for such a long time.

Finally the greatest thanks goes towards my parents, who supported me through the life choices I made and who raised me as the person that I am today. Without them I would never have had the chance to come this far.

Xavier Geerinck, Januari 2017

License

Xavier Geerinck, Januari 2017

Contents

Chapter 1

Introduction

The Information Technology environment is an ever-growing environment that is highly volatile, where new technologies are introduced at an ever expanding rate. Because of this expansion speed, there is a need for a clearer and easier to manage infrastructure that is able to keep up with these developments.

This chapter provides an overview of the technologies used in this thesis, as well as the company where this thesis was performed. It also includes the motivation that guided the effort, its key findings and how this thesis is organized.

1.1 Background

1.1.1 Microservices

Looking at the different application structure models throughout history, we can conclude that there were three big structure changes that happened.

In the 1990s applications were developed as one big monolith. This meant that different business components were all woven together as one component. Often referred as creating 'spaghetti' code, because components were so tightly coupled that it was impossible to separate them.

This changed for the first time in the 2000s. Companies started moving towards more loosely coupled components. To grow more easily and expand their businesses more rapidly. This trend was mostly lead by Amazon who referred to this structure as the 'Pizza Model'. Which stated that new business ideas should lead by a small team that can be fed by two pizzas. [1]

This trend changed for the last time in the 2010s. Applications became completely decoupled from each other and run as different independent components called 'microservices'. Microservices can be created by developers who do not need to have prior knowledge about the other different microservices within the infrastructure. Microservices communicate with one another through the use of a protocol defined at layer 7 of the OSI Model. [2]

1.1.2 Containers

To easily manage and deploy microservices, the concept of containers was introduced. A container is an enclosed space that contains a microservice its code that is required to run it. This enclosed space can then be executed within an isolated environment within the system it is running on. In contrast to a virtual machine, containers do not require a complete operating system to be provided. They will instead utilize and share the underlying operating system. [3]

1.1.3 Cisco Systems

Cisco Systems is an Information Technology company, focused on Network and Communications equipment, with a growing interest in the Software-as-a-Platform landscape. It is currently ranked number 54 on the Fortune 500 [4]. With its vision to change the way we work, live, play and learn, Cisco tries to connect the world in a high-speed way. [5].

Because of its global scale, Cisco has the need to manage and deploy Microservices more optimally, which is why it initially developed an open-source platform called Mantl and acquired a company called ContainerX. Through these recent developments, Cisco is now able to manage its datacenters more optimally, resulting in a larger ever-growing consumer base.

1.1.4 Datacenters

Datacenters are facilities where a collection of servers is brought together to be 'hosted' under climatological ideal circumstances. These facilities try to accomplish an around the clock operation time for these servers resulting in an ideal experience for the end-user.

In recent years different business models have been created for these datacenters. The most important ones being described below:

- **Infrastructure-as-a-Service:** this model allows the end-user to specify the amount of resources that he or she wants to use of the different servers hosted within the datacenter. The end-user then only has to pay for the resources that are used instead.
- **Platform-as-a-Service:** when the end-user does not wanted to manage the infrastructure, this model can be used. Here the code of the application is hosted on an infrastructure that is automatically provided by the provider. Popular example providers for this model are 'Heroku' and 'Google App Engine'.
- **Software-as-a-Service:** here the application is hosted and managed for the end-user. Common applications utilizing this model are blogs, databases and others.

By analyzing the different models above we can detect a re-occurring problem within the Platform-as-a-Service and the Software-as-a-Service model. These two models their efficiency and cost margins are dependent on how well the different resources available are utilized. By providing a finer utilization of these resources, an increase in revenue can be realized.

1.1.5 Container Orchestration Services

With the growth of the amount of microservices and containers being deployed, the need for a service that is able to manage the different microservices was created. Orchestration can be seen as a management layer on top of the different microservices, performing tasks such as grouping, replicating, deploying and monitoring the different microservices.

This service has three core properties that can be identified within a microservice deployment:

- **Affinity:** affinity tells us where a specific type of microservice should be deployed. This can be useful for grouping the same types of microservices on the same machines, or spread them across different machines. For example, cAdvisor is a logging microservice, which will use anti-affinity to spread the different instances over all our different machines, while as a Wordpress microservice will have a shared affinity to group them on the same machine as much as possible, reutilizing shared packages as much as possible.

- **Replication:** replication is a system used to create fault-tolerant and high-availability systems. Through Replication, the system can be told how many instances of a specific microservice should be running at a given time. By combining replication with affinity, we can spread these replications over our different machines, giving us full control over the placement of these microservices. [6]

- **Monitoring:** monitoring is a critical part of any system that changed with the introduction of microservices. The Orchestration Service has to monitor itself and the microservices to measure different statistics such as resource usage and the number of created microservices.

Today there are multiple different high quality orchestration services available which are developed by high reputation vendors such as 'Google' and 'Docker'. For example: Kubernetes, Marathon and Docker Swarm. The choice was made to continue with Docker Swarm in this thesis for several reasons. First of all the deciding reason is that Docker Swarm is an open-source project which allows for a continued development without a vendor lock-in disadvantage. Secondly, Docker Swarm is built in with the Docker Engine, removing the need for an extra dependency that had to be installed seperately. Finally, Docker Swarm is a relatively new project which increases the research and contributions that can be made in this thesis.

1.2 Motivation

By reading the sections above, it can be made clear that microservices solve an important aspect in the Information Technology sector. However as with each technology there are several downsides of it that should be taken into account.

While before there was only one service that included all of the application logic and frameworks required to run the application. There are now different smaller services that all need to re-include certain parts of the application to allow communication between one another. This creates an overhead on the storage layer, increasing the need for a distributed storage pool that can be accessed by the different microservices.

Another critical aspect that changed because of the introduction of microservices is monitoring. Before we only had to monitor the system that was running the monolithic application, whereas now we need to monitor the different containers, the orchestration service and the machine that is running these microservices. This increases the amount of metrics that are collected from the different services. Increasing factors such as the bandwidth required to export these metrics and the storage needed to store the collected data.

The most critical aspect that changed is the increasingly distributed nature of these microservices. In a monolithic model, the application was scaled by introducing more resources on the machine such as memory, processing power and disk storage, oftenly referred as 'vertical scaling'. While as now these services are replicated over multiple different machines, which can be referred as 'horizontal scaling'.

By combining the different aspects above, several problems can be identified that require new solutions to be created. This thesis will look into creating an architecture that is able to efficiently monitor the different microservices, process the different monitored metrics in a real-time fashion and automatically update models that are able to predict resource usage through the use of machine learning.

1.3 Objectives

Looking at the different problems that were identified, several key performance indicators can be extracted that are used to create the objectives of this thesis:

- **Latency:** the latency between the different microservices. Every additional latency created results in the loss of customers.
- **Errors:** how many errors appear on the different systems and microservices?
- **Traffic:** is our traffic trend growing or decreasing? When do microservices have to be upscaled or downscaled?
- **Saturation:** is a microservice satured such as that no traffic is able to be consumed anymore?

From these different performance indicators, we can summarize different objectives that can be seen in Table 1.1.

	Objective	Description
1	**Metric Collection**	How can we create an architecture that is able to collect and store metrics in a real-time fashion, without hitting processing power and storage limits.
2	**Metric Analyzation**	How do we analyze the collected metrics to extract relevant data from them that can solve the different problems.
3	**Visualization**	Can we create a visualization infrastructure that is able to show the different microservices running on the different machines, and gather detailed graphs about these.

Table 1.1: Objectives and Research Goals

1.4 Thesis Organization

The first chapter covered the basic introduction, background, motivation and objectives for this thesis that will be looked at in the following chapters.

In Chapter 2 a detailed description will be given that leads to the creation of the testing environment that was used to generate the acquired results in this thesis.

Chapter 3 will provide a detailed overview of the created platform that solves the first objective posed earlier in this chapter.

Chapter 4 is the core of this thesis. Here a detailed analysis will be done on the data gathered in Chapter 3. Through the use of Regression algorithms, a Proof Of Concept will be demonstrated that is able to provide a solution for the second objective posed.

The visualization step will be covered in Chapter 5. This will cover the ability to visualize the cluster that has been created, showing the different microservices that are created and showing the ability for showing detailed information about them.

This thesis ends with Chapter 6, where a summary of the topics that were covered, as well as the conclusions that can be drawn from it, the contributions made and future work to look into.

Chapter 2

Test Infrastructure

2.1 Introduction

Creating a good testing infrastructure is not an easy task. It should be able to replicate an environment that matches the real world environment, to validate the different objectives in an accurate way to minimize the issues that can arise when deploying the infrastructure to a production environment.

To minimize these issues, several requirements have to be identified and met by the end of this chapter. There are first of all that the infrastructure should be able to scale, such that it can grow over time when more components are introduced. Second, it should be distributed to increase the bandwith, once this becomes a problem. Third, managing this infrastructure should be easy enough such that we are able to lower the maintenance costs and finally, the infrastructure should be able to integrate into an already existing one.

This Chapter offers a detailed overview of the infrastructure that was created to generate the results obtained in this thesis. These results were obtained by a theoretical model, which was implemented by utilizing new technology components such as Docker Swarm, cAdvisor and Apache Kafka which will be explained further on. These components can however be interswapped by their respective alternatives.

For example, within a Google infrastructure it would be beneficial to swap the Docker Swarm Orchestration Service with a Kubernetes one.

2.2 Technologies Used

2.2.1 ESXi

Figure 2.1: ESXi

VMWare vSphere Hypervisor is a type 1 hypervisor that virtualizes servers on the hardware itself rather than on the Operating System. When virtualization on top of an Operating System happens, this is called a type 2 hypervisor. The advantage of utilizing virtualization is that it allows multiple operating systems to be run on the same hardware available. [7]

2.2.2 Docker

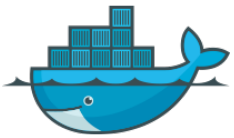

Figure 2.2: Docker

Docker introduced a way to run applications as an isolated container, bundled with all its code and dependencies. By doing this it improves the ability to manage and scale distributed applications.

These improvements include the easiness to deploy the application to different machines, test the applications for different errors, run the container on different machines without the need to reinitialize this machine after each run, remove the need for a hypervisor by utilizing the underlying kernel and many others.

All these improvements allow us to get more out of the existing hardware by moving the application from a virtual or physical machine towards a container. [8]

Figure 2.3: Vagrant

2.2.3 Vagrant

Vagrant is a command line utility used to generate lightweight and portable development environments. It does this by creating a Vagrantfile that defines how the development environment looks in terms of allocated resources and operating systems. Whereafter it will use a Virtual Machine provider to create this environment. [9]

2.2.4 Docker Swarm

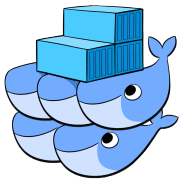

Figure 2.4: Docker Swarm Logo

Docker Swarm is a clustering engine for Docker delivered as a standalone component for Docker. Since version 1.12.0, Docker decided to include this clustering engine within its core package. The goal of Docker Swarm is to manage a pool of different Docker hosts through one of its different manager hosts. These manager hosts contain the state of the swarm with the different services running on it. They can also be managed through the Docker Application Interface. By utilizing clustering, a system is able to easily distribute service over multiple hosts.

Figure 2.5 illustrates how the Docker Swarm clustering engine is used in combination with two different hosting platforms called 'AWS' and 'Microsoft Azure'. The setup exists out of two environments each containing three docker engines. The first three docker engines are being created on the Amazon AWS Cloud whereas the other three will be created on a Microsoft Azure Cloud environment. Of these different nodes, one node will be selected as the 'manager node'. The core responsibility of this manager node is to dispatch different service tasks towards the other 'worker nodes' within the cluster. Worker nodes will execute tasks as soon as they receive these. [10] [11]

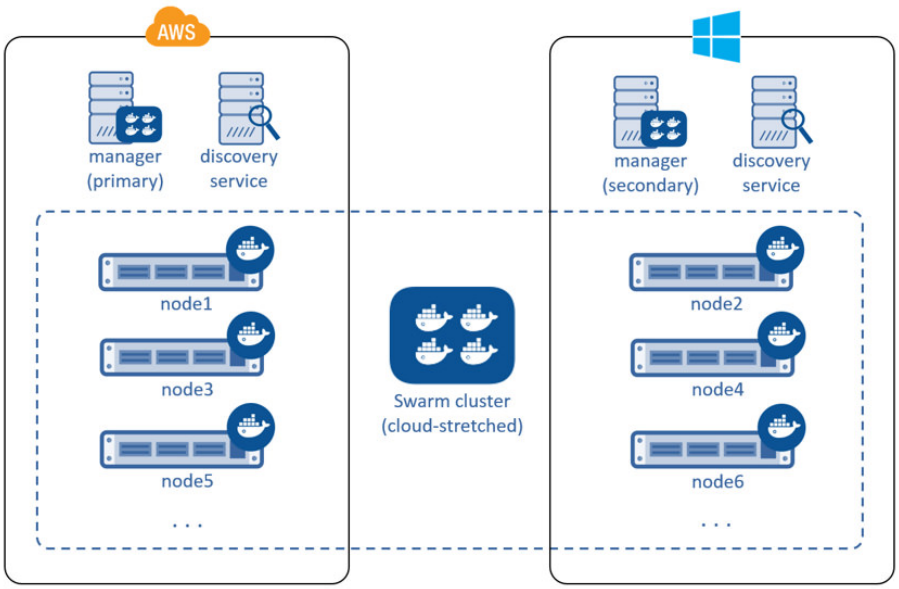

Figure 2.5: Docker Swarm Infrastructure Example

To create a Docker Swarm cluster, the Docker Command Line interface is used. This Command Line interface allows us to configure the different nodes within the cluster and the roles of these nodes. This way we are able to tell which node will act as the manager and which ones will be the agent node. When a node joins the swarm, it will act as a worker node by default.

To create this swarm, we first start by initializing it through the 'docker swarm init' command. After running this command, we get appointed a swarm token that is used to let other nodes join this swarm. With this token, other nodes can join this swarm by running 'docker swarm join token' where 'token' is the token returned by the initial creation. An example of these different steps can be found in listing 2.2.4

```
#!/bin/bash
# Create a manager node and get the token
sudo docker swarm init --advertise-addr 10.0.7.11 --listen-addr 10.0.7.11
SWARM_TOKEN=‘sudo docker swarm join-token -q worker --quiet‘

# Add a node to this manager
ssh vagrant@10.0.7.12 "sudo docker swarm join --token ${SWARM_TOKEN} 10.0.7.11:2377"
```

Figure 2.6: Docker Swarm Initialization

2.3 Infrastructure Creation

To generate the results obtained in this thesis, the choice was made to create a portable and isolated environment that can be set-up in a matter of hours on a single system. Allowing it to be used on development machines while still providing the provisioning of real-scale infrastructures, without requiring a lot of configuration changes.

The foundation of this setup is created on a Bare metal physical server preconfigured with the ESXi virtualization platform, which connects to the network through Network Address Translation to make its IP address available in the lab environment.

A Virtual Machine containing the latest version of CentOS 7 is spun up. We can see this Virtual Machine as a new Physical Server that could be our Developer Machine or our Physical server used to host the entire infrastructure.

After this initial creation, the Vagrant provisioning system is used to create three Virtual Machines on top of the CentOS system. These Virtual Machines represent the ability to distribute the system over different physical hosts, while still maintaining the portability for the developers. Vagrant will create these Virtual Machines through the use of a Vagrantfile as represented in listing 2.3

This script will specify specific parameters for our Virtual Machine such as the available memory, CPU, IP address and hostname. By specifying a 'provisioning script' it is also able to execute certain commands after the initial booting of these Virtual Machines, giving the ability to initialize the systems for first use. Running 'vagrant up' will then start and provision Virtual Machines with names 'swarm_manager_00', 'swarm_agent_00', 'swarm_agents_01' and their respective IP addresses '10.0.7.11', '10.0.7.12' and '10.0.7.13'. Now a Docker Swarm Cluster is created that is composed of these Virtual Machines with the 'swarm_manager_00' node as the leader. The code to accomplish this, is made available online [12].

Figure 2.8 illustrates this setup with the respective IP addresses that are assigned to each VM and container.

2.4 Conclusion

This chapter introduced the building blocks of a test setup that is scalable and dynamic enough to be expanded to meet the criteria needed in this thesis. The next chapter will use this foundation to create an architecture that is able to solve the introduced problems.

```
$prepare_swarm_manager_script = <<SCRIPT
# Install Docker
curl -sSL https://get.docker.com/ | sh
SCRIPT

// ... the same is done for a $$prepare_swarm_node1_script and
    $$prepare_swarm_node2_script

Vagrant.configure(2) do |config|
  config.vm.define "swarm_manager_00" do |config|
    config.vm.box = "ubuntu/trusty64"
    config.vm.hostname = "swarm-manager-00"
    config.vm.network "private_network", ip: "10.0.7.11"
    config.vm.provision "shell", inline: $prepare_swarm_manager_script
    config.vm.synced_folder "../", "/vagrant"

    config.vm.provider "virtualbox" do |v|
      v.memory = 1024
      v.cpus = 2
    end
  end

  config.vm.define "swarm_agent_00" do |config|
    config.vm.box = "ubuntu/trusty64"
    config.vm.hostname = "swarm-agent-00"
    config.vm.network "private_network", ip: "10.0.7.12"
    config.vm.provision "shell", inline: $prepare_swarm_node1_script
    config.vm.synced_folder ".", "/vagrant", disabled: true

    config.vm.provider "virtualbox" do |v|
      v.memory = 4096
      v.cpus = 2
    end
  end

  // ... another swarm_agent_01 is created that uses $prepare_swarm_node2_script
end
```

Figure 2.7: Vagrantfile Code

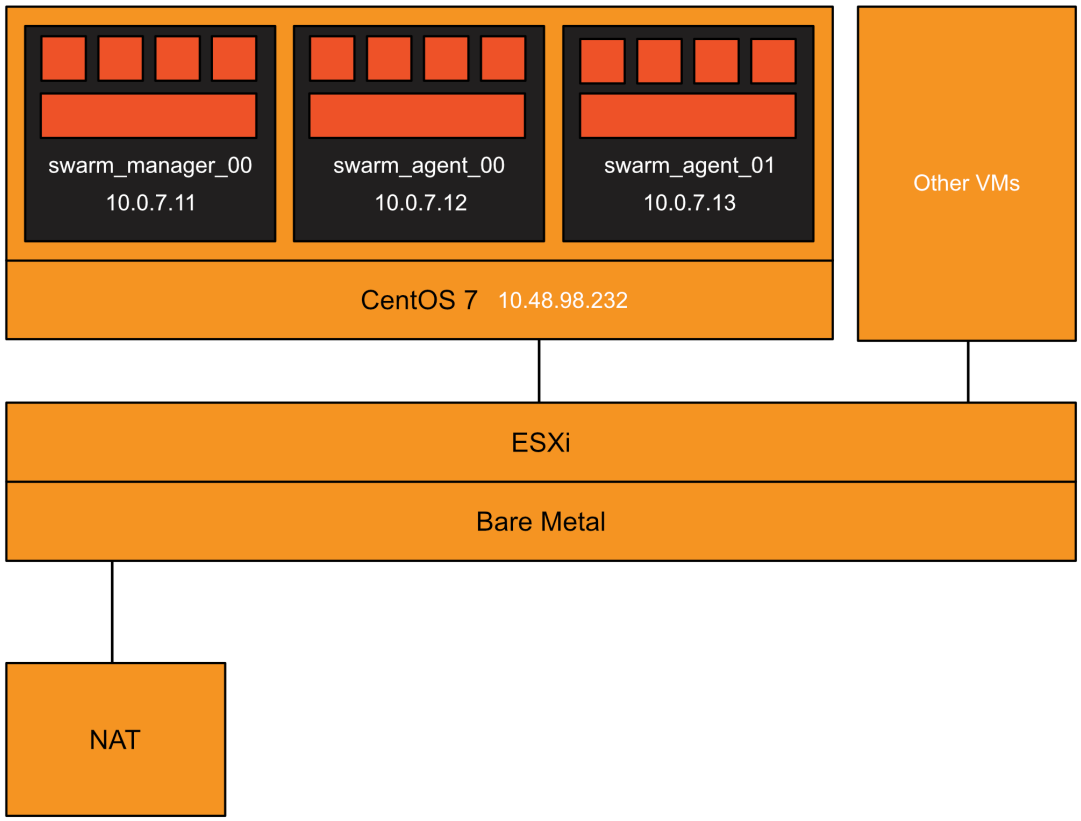

Figure 2.8: Test Infrastructure Setup

Chapter 3

Platform Architecture

Data Collection is a task that is becoming more import every day. Big data warehouses are not unusual anymore and are more and more becoming data lakes, increasing the need to store and process this data within given time constraints.

This thesis is no different from that, the main part being the collection of data and analyzing this data to predict the future and what should happen in it.

In this chapter a platform architecture is discussed that is able to process, analyze and visualize data that is being produced by different microservices with the goal to predict their resource usage and to issue alerts once a microservice should be scaled up to handle the increased load on the service.

3.1 Introduction

Before we are able to create a platform that is able to process, analyze and visualize big amounts of data, we first need to set goals which we want to achieve with this data. In the case of this thesis, these goals are clear.

First of all, the platform should be able to process huge amounts of incoming data in a real-time distributed way that is able to scale over different machines.

Secondly, the architecture should then be able to analyze this incoming data through the use of machine learning algorithms to extract predictions from it.

As a last goal, these extracted predictions have to be combined with the created microservices architecture to visualize them in one overview.

These different goals are achieved through usage of technologies and frameworks such as Apache Kafka [13], Apache Flink [14], [15] and the Lambda Architecture [16], which are explained in Sections 3.2 and 3.3.

Section 3.4 will then introduce the platform architecture that achieves those goals, which is then concluded in Section 3.6.

3.2 Technologies Used

3.2.1 Apache Kafka

Figure 3.1: Apache Kafka

Apache Kafka is an open-source platform for processing real-time data streams that are coming in from multiple producers. [13] These data streams are then fed into different 'Topics' that collect the real-time data in a time ordered way for being processed by multiple Consumers. Apache Kafka ensures the delivery of messages through the use of a commit log for every topic [17]. This commit log will keep track of the messages being received and will only mark a message as received once an acknowledgment has been received. Every message being received by the Apache Kafka platform is also retained for a specific amount of days before being removed from the system. To gather the information with the state of the Apache Kafka Platform, Apache Zookeeper is used. Apache Zookeeper is a service to maintain configuration information. [18]

3.2.2 Apache Flink

Figure 3.2: Apache Flink

Apache Flink is a open source framework for big data processing. It supports batch- and stream processing allowing for data to be processed in an efficient way. On top of this framework, Apache Flink also offers implementations to make Machine Learning, Graph Processing (Gelly) and Event processing (CEP) easier for their respective domain uses. [14]

By utilizing Apache Flink in this thesis, we are able to effectively process both real-time data and historical data. Making it the perfect framework for detecting trends through analyzing our historical data, as well as updating our historical model to adapt in real-time to the newer data that is coming in.

3.2.3 cAdvisor

Figure 3.3: cAdvisor

cAdvisor is an open-source project created by Google that gathers information about containers running on the host where cAdvisor is running. cAdvisor has been built to work with different types of containers in mind, such as the Docker containers used within this thesis.

Besides gathering resources, cAdvisor is also able to export these resources to different consumers through the use of a storage layer. Some of the included export consumers include 'Apache Kafka', 'InfluxDB', StatsD' and others. [15]

All of this together makes cAdvisor the perfect tool to gather information about the resource usage of the different microservices being analyzed within this thesis. By exporting the data to a kafka queue, we are then able to perform statistic processing and predict certain bottlenecks.

3.3 Related Works

3.3.1 The Lambda Architecture

Different options are available to perform Big Data analysis. One of the most widely used architectures is the 'Lambda Architecture' created by Nathan Marz [16] who came up with it while working on systems at Twitter [19] and Backtype. This architecture is able to work with realtime data, as well as collecting information on historical data by using different layers.

The Lambda Architecture is a multi-layered architecture consisting of a 'Batch', 'Speed' and 'Serving' layer as illustrated in Figure 3.4.

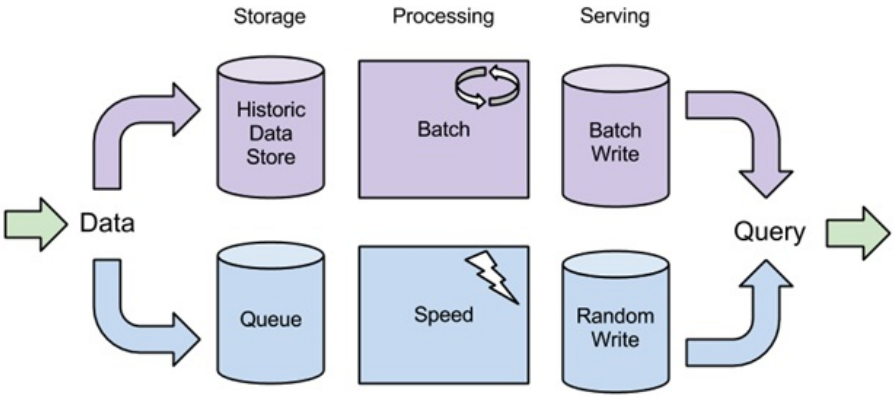

Figure 3.4: Lambda Architecture

The first layer is the **Batch Layer**. This layer will work with a collection of datapoints that is called a 'batch' rather than with individual datapoints. These datapoints can be acquired by buffering incoming data for a certain time interval; or by retrieving previously stored historical data. After this data has been acquired, an algorithm is run on this batch that is able to detect anomalies, peaks and other pattern related information to create predictions that are as accurate as possible.

An example of an algorithm that can be run on batches of data is 'Normal Linear Regression'. Linear Regression will go through all the different datapoints and find a curve that goes through these points. Whereafter it is able to 'predict' future points by extending this line.

In contrast to the batch layer, the **Speed Layer** will not buffer or retrieve historical, but will instead deal with recent data only. This makes the layer ideal for 'iterative' trainable algorithms which continue on improving the already existing model by continuously feeding new data towards them. This is what the 'SimpleRegression' algorithm does [20]. It uses newly fed

datapoints to calculate the new curve through our points, without the need of saving all the previous points.

To generate an answer with the extracted data from the batch layer, a new layer is introduced that is called the **Serving Layer**. This layer will index the data gathered from the batch and real-time layer to answer specific questions while still keeping a low latency. When a query is now sent towards this serving layer, it will collect the relevant data from this index and return the answer.

3.4 Methodology

Before the layers can be created in this architecture, data collection has to happen first.

By creating a cAdvisor monitoring container which will be replicated in global mode, we are able to distribute a monitoring container to each server within the architecture as described in Figure 3.5. These monitoring containers will in turn, send data towards the Apache Kafka platform, which will process it in a centralized way.

From this central platform, data is forwarded towards our different layers described in subsections 3.4.1, 3.4.2 and 3.4.3

```
sudo docker service create --mode global --name cadvisor \
    // ...
    google/cadvisor:latest \
  -storage_driver=kafka \
  -storage_driver_kafka_broker_list=kafka:9092 ‘ \ # The Kafka Address
  -storage_driver_kafka_topic=container_stats ‘# The Kafka Topic name‘
```

Figure 3.5: Start cAdvisor in Global Mode

Seeing that our data is now flowing into the Apache Kafka Platform, we are able to create the processing layers for this data. These layers are built on the initial idea described in the Lambda Architecture.

3.4.1 Real Time Layer

Starting with the **Real Time Layer**, we need to process the data one item at a time, coming in on the Kafka Platform. This is done by creating a consumer through an Apache Flink job, that is able to create 'Direct Streams' to our Apache Kafka platform. This Apache Flink job then also applies different algorithms that will create our results.

Once Apache Flink is done processing the data, we can forward the found results into a different topic on the Apache Kafka platform.

The code illustrated in Figure 3.6 is used to open a stream that allows us to read data from our Apache Kafka Platform in real-time.

```
// ...
// Open Kafka Stream
FlinkKafkaConsumer08<cAdvisor> consumer = new FlinkKafkaConsumer08<>(
        parameterTool.getRequired("topic"), // The Topic name
        new cAdvisorDeserializer(), // Deserialize incoming JSON
        data
        props // Connection details
);
// ...
```

Figure 3.6: Creating a Apache Kafka consumer in Apache Flink

3.4.2 Batch Layer

For the **Batch Layer** data first has to be collected through a Flink Replayer Job. This will create a consumer on the Apache Kafka Platform such as in Figure 3.7 that will forward the data into a database that is chosen by us. In this case the choice was made to use InfluxDB as the database system because it is ideal for storing time related data. This forwarding is done by creating a 'sink' where all our processed data will be sent to. Seeing that this is a custom sink which is not available as a library, custom code was written to do this as illustrated in Figure 3.8

This layer is introduced to create a base model that is able to predict patterns that occurred in the history of the different microservices and that can then be updated through the use of the real-time layer. This can be seen as creating a preconfiguration model that will be utilized when starting a new microservice. This preconfiguration model will allow the real time layer to create more accurate predictions from the beginning. It is important to note that this layer should not be a continuous running process but rather a process that should be run after a specific period.

3.4.3 Visualization Layer

The visualization layer takes care of visualizing the microservices and their respective details. More information about this layer is described in Chapter 5.

```
// ...
// Open Kafka Stream
// ...
// Process data and map to simpler object
// ...
// Send to Sink
dataStream.addSink(new InfluxDBSink("http://<host>:<port>", "<username>",
    "<password>", "<db_name>"));
// ...
```

Figure 3.7: Replaying data to InfluxDB

```
public class InfluxDBSink extends RichSinkFunction<data.Measurement> {
    // Initialize parameters and open the connection in the Constructor and void open

    @Override
    public void invoke(Measurement measurement) throws Exception {
        // ...

        // Add fields through a while loop
        Iterator it = (measurement.getColumns().entrySet().iterator());
        // ...

        // Add tags (These get indexed)
        Iterator it2 = (measurement.getTags().entrySet().iterator());
        // ...

        influxDB.write(dataBaseName, "autogen", p);
    }
}
```

Figure 3.8: Apache Flink InfluxDB Sink

3.5 Scaling the Platform Architecture

Should bottlenecks in this architecture appear, then there are different solutions available to scale this architecture:

- **Split Apache Kafka and Zookeeper:** In the proposed architecture, Apache Kafka and Zookeeper are on the same node. By separating those two applications, more resources can be made available on the Apache Kafka node, allowing for an increase in resources for the Apache Kafka node and higher processing power.
- **Distributing Apache Kafka:** Apache Kafka has been built as a scalable solution on its own [13]. By distributing it over different nodes, we are able to distribute the 'monitoring' topic and the processed data topic on different nodes, allowing for an increase in topic size for both these topics.
- **Scale Apache Flink:** Apache Flink jobs can be submitted on a cluster infrastructure that will distribute the job over a set of machines [21]. This allows larger jobs to be done by horizontally scaling instead of vertically scaling them.

3.6 Conclusion

This chapter introduced an architecture as shown in Figure 3.9 that is able to gather and transform data in a scalable and real-time environment while still being able to detect patterns occurring in historical collected data. This platform has several core concepts that were taken into account:

- **Scaling:** The system can be scaled to work on any scale, creating an architecture that can be used for small to large sized enterprises.
- **Modular:** By using different layers and components, the whole architecture has been built in a modular way. This allows for an inter-swapping of the different layers and applications within.

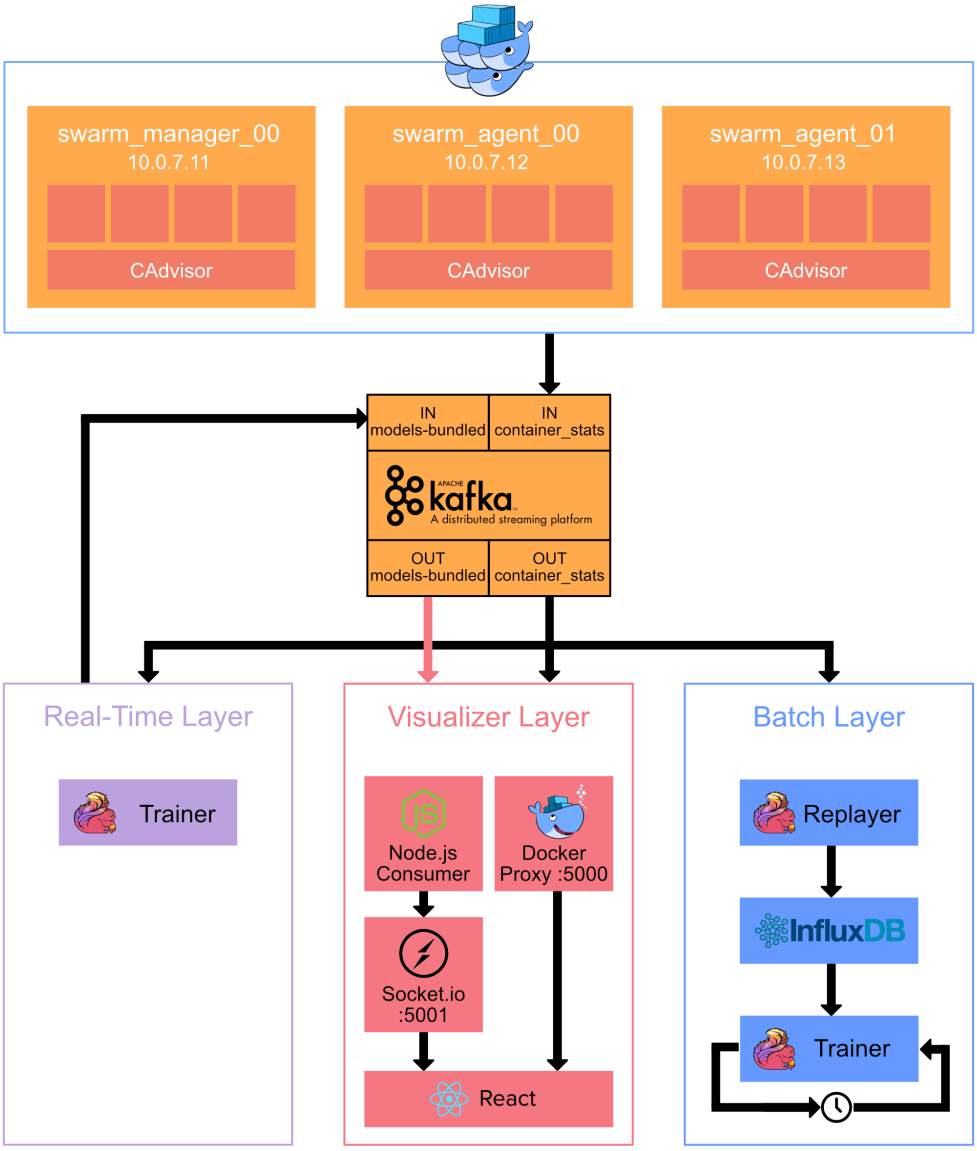

Figure 3.9: Platform Architecture

Chapter 4

Machine Learning

4.1 Introduction

In recent years, machine learning has found its way to the enterprise, being used for a broad range of applications, going from fraud detection systems, auto-pilot technology and ranking webpages to creating artificial minds that are able to learn on their own and beating the highest ranking players at a complex game called Go [22]

Within enterprises, an important aspect of staying ahead of the competition is being able to predict the future. By using this future, enterprises are able to use prior knowledge to extract and gain information regarding the paths that they should take to stay ahead.

Machine Learning algorithms work by returning an output value for a set of given input values. To achieve this goal, machine learning algorithms commonly use a training set that holds a set of values where the end result is known. By utilizing this training set, we are then able to finetune the internal parameters between input and output to achieve a specific result.

For example, when we have a training set of two input nodes and one output node, where the input node represents the hours of work performed on an application and the hours of uptime. With the output node representing the revenue earned on that day, we can train our model to predict the revenue in the future based on the hours worked and the hours of uptime. This training set has two input parameters and one output parameter.

Today there are many different machine learning algorithms that are able to predict the most plausible future. These algorithms are categorized as 'Regression' algorithms. They will assess a relation between different datapoints in our dataset resulting in a continuous response variable [23].

Table 4.1 describes the different regression machine learning algorithms available that are commonly used today. For the scope of thesis the choice was made for Linear Regression which will be explained in Section 4.2. This choice was made because this thesis is about creating the architecture rather than focusing on the prediction algorithm. However when needed, this algorithm can easily be replaced by a more advanced algorithm that is more accurate [24], [25].

Algorithm	**Description**
Ordinal Regression	When our data is in rank ordered categories
Poisson Regression	Used for predicting counts and contingency tables
Fast Forest Quantile Regression	Predicts values for a number of quantiles
Linear Regression	Creates a linear model through a set of datapoints
Bayesian Linear Regression	Bayesian networks will combine linear regression and additional information from a prior probability distribution form
Neural Network Regression	Instead of using pixel data, datapoints are now used to train a Neural Network and predict values based on this training data.
Decision Forest Regression	Decision Trees will traverse a Binary Tree path until a leaf node is reached. Based on this it will create predictions.
Boosted Decision Tree Regression	Boosted Decision Trees will use the MART gradient boosting algorithm to improve the decision tree algorithm.

Table 4.1: Machine Learning Regression Algorithms

The remainder of this chapter is organized as follows. Section 4.2 provides a basic introduction to normal linear regression, gradient descent and linear regression through stochastic gradient descent. This section only covers the pure basics seeing that the scope of this thesis is creating an architecture to perform these algorithms as accurate as possible.

Section 5.3 will explain the method used to apply this algorithm to the different layers within the created architecture. After applying this method, a prediction can be made that is able to pinpoint when a certain microservice is going to create a bottleneck in the future.

The evaluation and results of this implementation are then discussed in Section 5.4 following a conclusion of it in Section 5.5.

4.2 Prediction Algorithms

4.2.1 Simple Linear Regression

Linear Regression is an predictive analysis approach that falls in the category of 'Regression' algorithms. Regression algorithms try to predict results for a given set of values.

For Linear Regression, this prediction is done by finding a polynomial curve of the form $y = slope * x + intercept$ through a given set of scalar datapoints. In a 2-dimensional space, these points would be defined by a (x, y) coordinate as illustrated in Figure 4.1. [20]

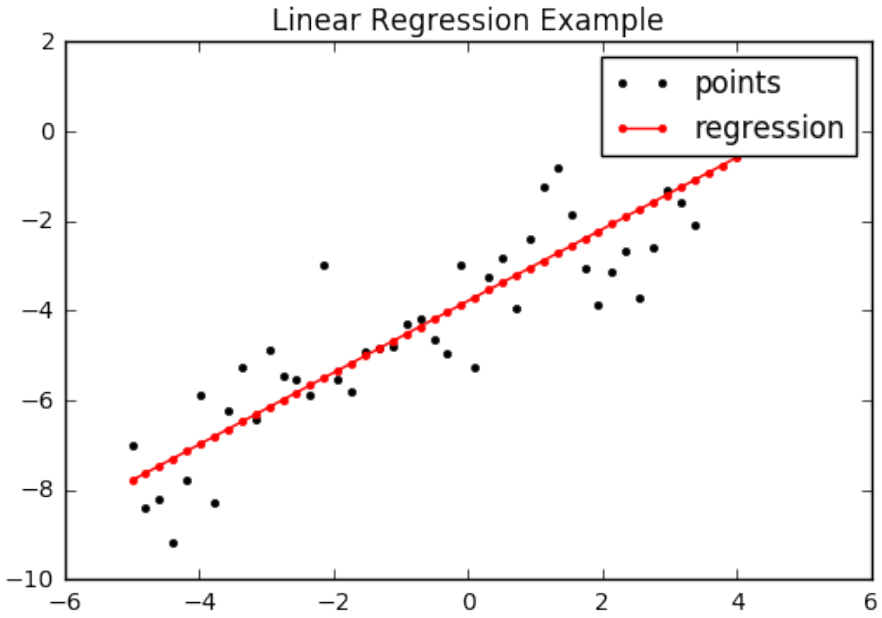

Figure 4.1: Linear Regression

The way Linear Regression works is that it will calculate the sum of squares of the deviations from the mean as described with the following formula:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2$$

This has as disadvantage that it needs to go through each point, increasing the computation time for large N. To prevent the need to recalculate the Linear Regression equation, a method has been found that is able to iteratively update this equation, resulting in an algorithm that is ideal for Real-Time learning and that can be used in our **Speed Layer** [26].

4.2.2 Gradient Descent Introduction

The Linear Regression algorithm described above is an iterative algorithm that is being updated in real-time based on the previous parameters. A disadvantage of this method is that when our dataset becomes more complex and larger, we need to iterate through each point in this dataset, increasing the time required to find our result.

To solve the above disadvantage, a more optimized and faster method is required. Which is why Gradient Descent is used within this thesis. Gradient Descent is a method that tries to optimize the time required to find a result. This method can be seen as a method that is able to find a needle in a haystack in the fastest time.

Gradient Descent works by defining a **cost function J** that illustrates the *amount of wrongness* for a certain value. This cost function is the sum of the different error values in our model.

If $\mathbf{y}$ represents our actual score and $\hat{\mathbf{y}}$ represents our predicted value that should match this actual score, then the error for these two values can be represented as $y - \hat{y}$.

By applying the definitions above, we are able to define the cost function more formally and apply the squared deviation function that we know from the statistics field. Resulting in the formula described below. The reason for using squared deviation is explained further on in this section.

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

Gradient Descent will now try to find values that result in the cost function becoming minimal. In a normal use case we would iterate through each different parameter that results in our predicted value to see which of these parameters would result in the smallest value for our cost function. However this means that when this has to be done on a function with three dimensions, that we would have to check $3 * 3 * 3 = 27$ different parameters.

By using mathemetics, the time required to find these parameters can be optimized. Through calculation of the derivative of the cost function, we are able to determine where our cost function will have a minimum or a maximum. After finding this minimum or maximum, we now need to validate if we have to follow the left or the right side of this equation to end in a minimum. To find this minimum or maximum, **numeric estimation** can be used. Numeric estimation will look directly at the left and right of our equation to see which side results in a smaller value, and which way we should follow.

After looking in-depth of the method described above, an important problem can be iden-

tified. What happens when a local minimum is reached instead of a global minimum? This would mean that the algorithm would stop and that the global minimum cannot be found. To overcome this problem, the squared deviation function is introduced. This function will convert the original function into a convex one. Which provides a solution for *almost* every dimension.

When iterating through a lot of values with Gradient Descent the gradient sum value will be large. To keep this value small, 'Batched Gradient Descent' is used. This will allow us to execute the Gradient Descent algorithm a lot faster by using a predefined amount of datapoints that do not span the whole dataset. When using one datapoint at a time, we refer to this as 'Stochastic Gradient Descent'. Allowing for a real-time algorithm to be created.

Everything described above allows us to extract a hypothesis function:

$$h_\theta(x) = \theta^T x = \sum_{i=0}^{n} \theta_i x_i$$

With the update rule for Gradient Descent to minimize this hypothesis resulting in:

Algorithm 1 Gradient Descent

1: **repeat**

2: $\quad \forall j \in \{0, ..., \#features\}, \theta_j = \theta_j - \alpha \frac{\delta}{\delta \theta_j} J(\theta_0, ..., \theta_j)$

3: **until** convergence

4.2.3 Linear Regression with Stochastic Gradient Descent

Applying the 'Stochastic Gradient Descent' algorithm on Linear Regressions means that we first have to write our linear regression algorithm as the cost function for Gradient Descent.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - (\theta_1 * x_i + \theta_0))^2$$

If we then minimize this function through the use of Gradient Descent as described in the previous subsection, we find the line that fits out datapoints and is able to detect the trend line.

A more detailed overview of this method using Stochastig Gradient Descent is described in the paper 'Solving large scale linear prediction problems using stochastic gradient descent algorithms' [27]

4.3 Methodology

Since the Platform Architecture has been built modular, implementing the machine learning algorithms is trivial. By adapting the Apache Flink Jobs that are running within those two layers, we are able to integrate the linear regression machine learning algorithms and retrieve predictions from them.

This chapter will go in depth on how the batch and real time layer models are trained to return models that are able to predict the future.

4.3.1 Batch Layer Implementation

Implementing our Batch Layer is done by gathering the different datapoints from the datastore through the use of a InfluxDB query as illustrated in Figure 4.2. This InfluxDB query will return the given datapoints for a certain period in time, being greater than zero.

Afterwards the actual machine learning algorithm is called that will process this data and run our Gradient Descent algorithm on the datapoints to find the best linear fit through them.

This algorithm is written in the java coding language as described in Figure 4.3.

```
InfluxDB influxDB = InfluxDBFactory.connect("http://<host>:<port>", DB_USER, DB_PASS);
influxDB.createDatabase(DB_NAME);

QueryResult result = influxDB.query(new Query("SELECT * FROM memory_usage ORDER BY
    time DESC LIMIT 50", DB_NAME));
```

Figure 4.2: Connecting to the InfluxDB Data Source

```
MultipleLinearRegression mlr = new MultipleLinearRegression()
                .setIterations(10)
                .setStepsize(0.5)
                .setConvergenceThreshold(0.001);

QueryResult.Series s = result.getResults().get(0).getSeries().get(0);
System.out.println(s.getValues().get(s.getColumns().indexOf("value")));

int valueColumnIndex = s.getColumns().indexOf("value");
int timeColumnIndex = s.getColumns().indexOf("time");

List<LabeledVector> trainingSetList = s.getValues().stream().map(r -> new
    LabeledVector(
        Double.parseDouble(r.get(valueColumnIndex).toString()),
        new DenseVector(new double[] {
            (double)Instant.parse(r.get(timeColumnIndex).toString()).getEpochSecond()
        })
))
.collect(Collectors.toList());

DataSet<LabeledVector> trainingSet = env.fromCollection(trainingSetList);
mlr.fit(env.fromCollection(trainingSetList));
```

Figure 4.3: Training a Batch Model

4.3.2 Real-time Layer Implementation

Our Real-time implementation will use the same linear regression as our Batch layer, seeing that this algorithm can be used in a real-time fashion. By using micro-batched gradient descent, we are able to update the different parameters in real-time and improve the resulting equation.

To start we will open the output file created by the batch gradient descent algorithm if this exist. This ensures us to use preconfigured parameters resulting in a more accurate model from the beginning than when we would start without metrics.

To illustrate this with an example, we will start by taking the assumption that we have an Apache Kafka Queue component that is using almost the maximum CPU usage once it starts to run. When we do not use a preconfigured model through the use of our Batch Layer, we will get predictions that generate a prediction illustrating a need to upscale the microservice within the first few seconds. This happens because our Batch Gradient descent algorithm will see the first two datapoints and plot a line through it, resulting in a steep slope that will hit the threshold almost instantly. In contrast to our Batch Layer, we can now see that the batch layer generates an initial result that contains more datapoints and is thus able to generate a prediction that lies further in the future. By now updating this already preconfigured model in real-time, we are able to achieve an accuracy that lies much higher than when we would start without data.

Implementing this in Apache Flink is done by using the same code as in the Batch Layer. Here we will however use one single value as the batch. The resulting model will then be sent towards Apache Kafka, returning the required parameters to create predictions. To do this, we utilize a data stream mechanism included in Apache Flink that is able to process every value to create an update for the theta values as described in Figure 4.4.

To use these parameters to generate predictions, we will transform the given equation $y = \theta_1 * x + \theta_0$ to return the x value. Thus becoming: $x = \frac{y-b}{a}$ where **x** denotes our epoch time, **y** our predicted value being the threshold that we will use to find the intersection, **b** the intercept and **a** the slope.

```
DataStream<NormalizationParams> normalizedDataStream = dataStream.flatMap(new
    UpdateModel());


// ...

public static final class UpdateModel extends RichFlatMapFunction<CAdvisor,
    NormalizationParams> {
    @Override
    public void flatMap(CAdvisor c, Collector<NormalizationParams> out) throws
        Exception {
        // 1. Load in the model
        // 2. Normalize the datastream point by using the minX, maxX, minY and maxY
        //    from the model
        // 3. Update theta values
        // 4. Add the theta values to the old theta values, recalculating the average
        // 5. Save the new theta values
        // 6. Update the model file
        // 7. Return our updated values
    }
}

public static long normalizeValue(long value, long min, long max) {
    return (value - min) / (max - min);
}
```

Figure 4.4: Stochastic Gradient Descent Adaptation

4.3.3 Normalizing data

One of the most important steps to perform when working with machine learning algorithms is the step of normalizing the data.

Through normalization we are able to provide uniform data towards the algorithm, allowing it to converge faster. This is often called 'feature scaling' [28].

To normalize the data from our dataset, we start by finding the maximum and minimum within the dataset. Whereafter we can apply the following equation:

$$x' = \frac{x - max(x)}{(max(x) - min(x))}$$

Applying this on our incoming data. We can see that we are receiving data from the Apache Kafka component, which is getting parsed through the Apache Flink consumer that was created earlier. Now we find the maximum and minimum timestamp of the dataset being consumed and will normalize the dataset towards it.

Several optimizations were made to improve this process. First the real value was adapted to use megabytes instead of bytes, reducing the computation cost required for these larger values. Secondly, the choice was made to change the timestamp denoted as an epoch value to subtract the startdate from it, also lowering the computation cost due to these large numbers being removed.

4.4 Evaluations and Applications

The structure of this section is as follows. Two subsections will follow detailing the real-time and batch layer.

Within these subsections, four parts are looked at. First a detailed description of how the results for each model are measure. Whereafter an expectation follows that describes the expected result. After the expectation the results are illustrated. These results will then be concluded in the conclusion stating how the results differ from our expectations and the possible reason.

4.4.1 Real-Time Model

Measurement Setup

To analyze the prediction model for the Real-time Model a measurement utility script is written. This script will consume the different predicted parameters for our linear regression model from the 'modelsbundled' Apache Kafka topic.

By using these different parameters, a graph is created illustrating the real usage metrics and the predicted metrics. These real usage metrics will be consumed from the 'container_stats' Apache Kafka topic containing the raw metrics data from our containers and the predicted metrics will be calculated based on the different parameters consumed from the 'modelsbundled' topic. This results in the graph as illustrated in Figure 4.5.

Expectations

Our expectations for the real time algorithm are that the predicted line will start from a predicted value that is not nearly as close as the real metrics being consumed. After a few iterations this predicted line should converge towards the real metrics, illustrating the disadvantage of our linear algorithm.

After running the measurement script the following results were obtained:

Results

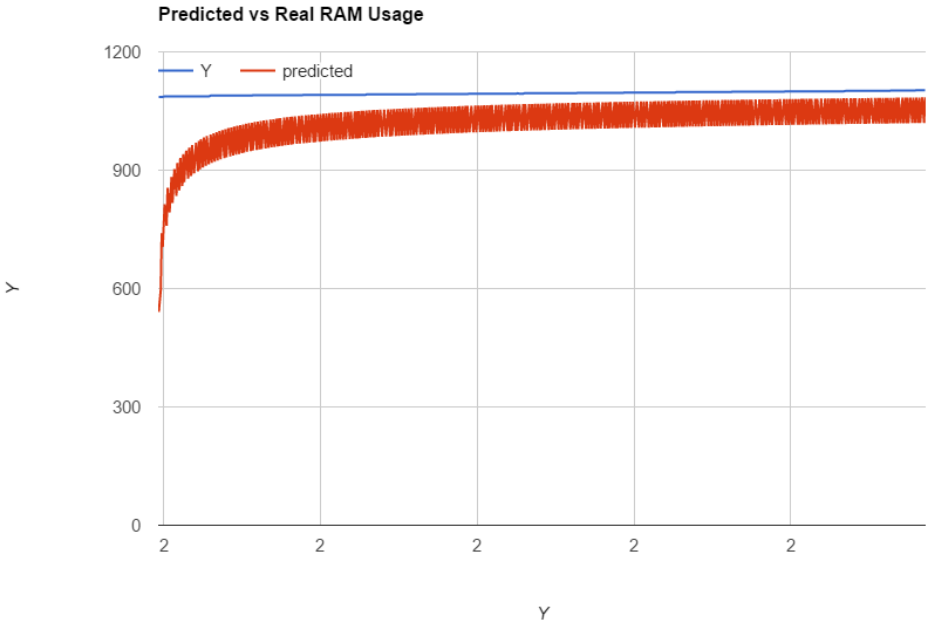

Figure 4.5: Predicted vs Real RAM Usage *RealtimeModel*

Conclusion

Looking at Figure 4.5 , it can be seen that the real-time machine learning algorithm works as expected. Starting from the first point at (0, 0), gradually improving itself while more data is entering from the real time consumer.

We can however detect that our real time algorithm alone is not sufficient. In the illustrated figure, it is made clear that the initial values predict an upscaling that is being needed within the first few minutes of data being gathered. After these few minutes the prediction stabilizes.

Since containers are mostly short lived services, it is not desirable to have predictions matching our expectations after a period of time. Therefor a new layer is needed which will be able to provide a preconfiguration of the parameters used within our regression algorithm. This preconfiguration will also keep detections such as anomalies in mind and will filter out certain spikes happening.

4.4.2 Batch Model

Measurement Setup

To analyze the Batch Model a subset of data will be extracted from the InfluxDB database. This subset of data will show a larger amount of points in time in contrast to our real-time layer, illustrating the historical data with several attention points:

- **Peaks:** resource data should have peaks. These represent spikes happening during intensive processor calculations, garbage collection that happens on the memory or network traffic that suddenly increases.
- **Trend:** a trend line should identifiable, describing how our data increases or decreases. This trend line is important since it will show us if we have to scale-up or scale-down the microservice.
- **Stagnation:** does the data stagnates or not? When it does, we should be able to detect this and our prediction algorithm should show a point that is not in the near future.

For this subset of data, our Batch learning algorithm will be run with the following parameters:

Parameter	Value
Learning Rate	0.1
Training Iterations	1000

Table 4.2: Batch Training Parameters

With these parameters, predictions will be created resulting in the graph illustrated in Figure 4.6.

Expectations

The expectations for our resulting graph is that we will see data containing the attention points described above. Next to these attention points we should see a prediction line that tries to converge to our rising data, showing the general trend line as well as possible. An important thing to pay attention here is that our prediction line should go through the peaks that are created. These peaks do not contribute to the trend line and should be ignored.

Results

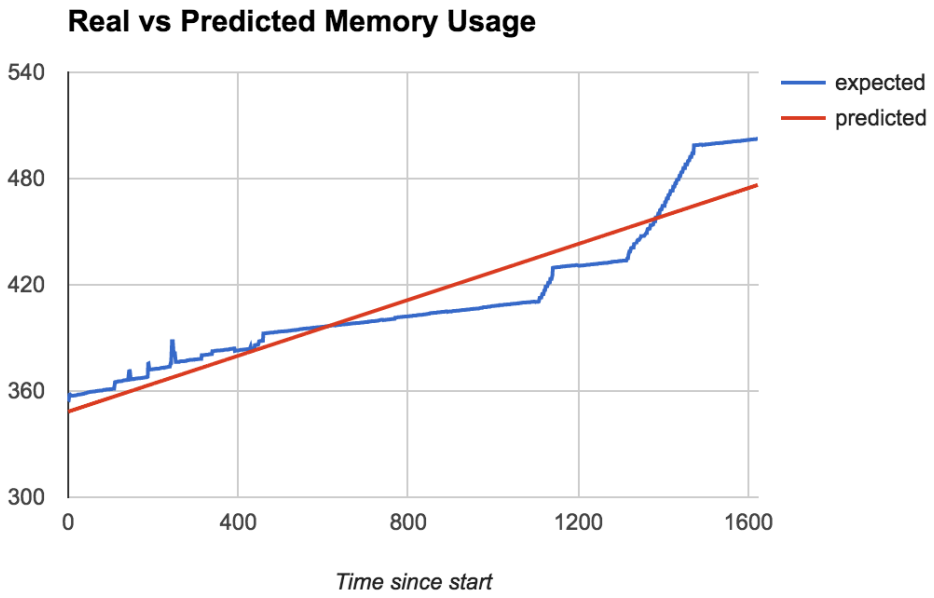

Figure 4.6: Predicted vs Real RAM Usage *BatchModel*

Conclusions

After analyzing Figure 4.6, conclusions can be drawn that our model adapts based on the different peaks appearing in our data. The maximum deviation of this prediction being 34 megabyte, which is reasonable for memory usage prediction.

For a Linear model, these results are acceptable seeing that a linear model is used that does not follow the peaks and patterns exactly.

4.4.3 Combined Model

Measurement Setup

The measurement setup is identical to the real time layer, with the only change being that we start by executing the batch layer first to create the initial preconfiguration of our machine learning model. Hereafter the real time layer is being run, utilizing this created preconfiguration.

Expectations

Our expectations here are that the linear regression algorithm will create predictions which are within a reasonable deviation range from the start. This should eliminate the wrong predictions which occur when only running the real time algorithm.

Results

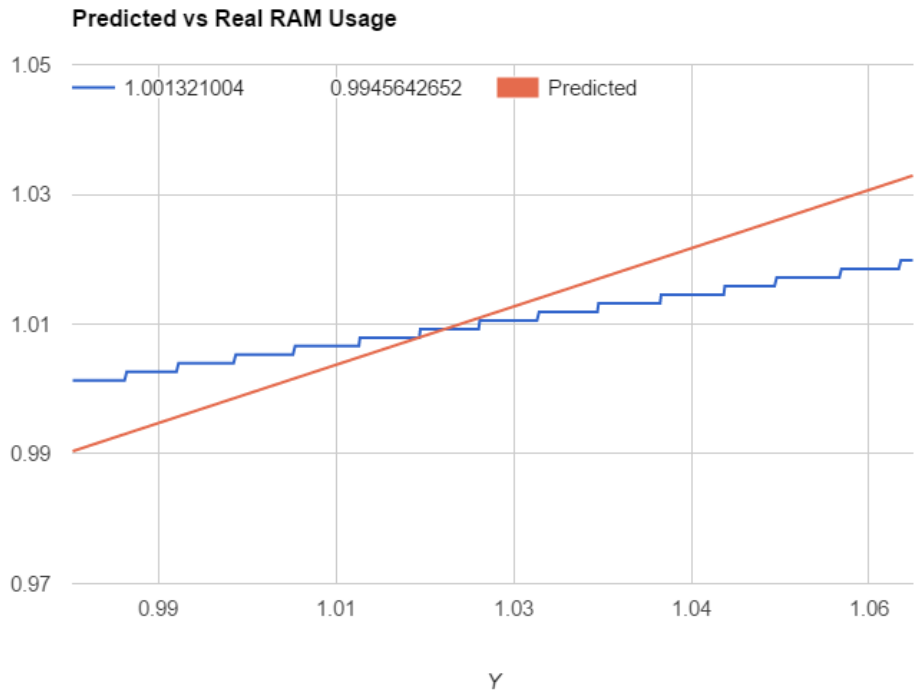

Figure 4.7: Predicted vs Real RAM Usage *BatchRealtimeModel*

Conclusion

Analyzing our results illustrated in Figure 4.7, it can clearly be seen that our expectations match the reality. The preconfiguration provides an initial starting point for our predictions.

4.5 Conclusions

This chapter introduced two different ways to generate a machine learning model that is able to generate predictions for the future. By combining both real time and batch based learning algorithms, we are able to pinpoint the point in time where a bottleneck is going to occur, while still looking at historical data. This allows us to detect when a service should be scaled.

Chapter 5

Visualization of the created Architecture

5.1 Introduction

Visualization is an important aspect in every development project. It allows us to grasp the concept that we are tackling. Different questions should be asked when creating a visualization platform. How do we make an interface for a user that is both intuitive and easy to use? Which technologies do we use that allow for a fast development iteration? Can we create a look and feel that is modern? What features should our platform have?

This chapter will provide an answer to each of these questions. The goal is to create a visualization platform that is easy to use, but still provides all the data needed to understand the platform.

5.2 Technologies

5.2.1 Proxy Server

A proxy server is an intermediate server between the target machine and the client allowing the guest to access the information on the target machine. This proxy server allows the administrator to restrict, monitor and create content that is made available to the guest user.

5.2.2 React

Figure 5.1: React logo

React is a javascript library created with as goal to simplify and modularize User Interface creation. This is done by focusing on three core aspects within the React library. First of all it works **Declarative**, by creating views for each state it is able to update and render the correct components when data changes. Secondly it is **Component-Based**, making it easy to create components that manage their own state. By using components, we are able to create much more complex UIs. The last aspect is the concept of **Learn Once, Write Anywhere**. This says that React can be used within any technology stack to maximize code reuse. React can be rendered on a server, run as a standalone Single Page Application or even create Mobile Apps through React Native. [29]

5.2.3 Redux

Figure 5.2: Redux logo

Redux is created to manage the state of an application in an immutable way. It does this by creating a state tree that keeps track of the different variables within an application, called the state. This state can get updated through different actions that are being created by different events, such as clicking a button, which will then be forwarded towards a dispatcher. This dispatcher will distribute these actions over the different reducers. These listen for a specific action unique for the part of the state tree it is handling and update the state tree based on it.

Once the state tree is updated, the different web components will then compile a new version of the user interface. This will reflect the state change that has happened [30]. A store is connected to this state tree to fetch information of the different states of our components. This store can not be changed directly since that would void the principle of having a state tree, and thus actions are used. This concept is illustrated in Figure 5.3 and Figure 5.4

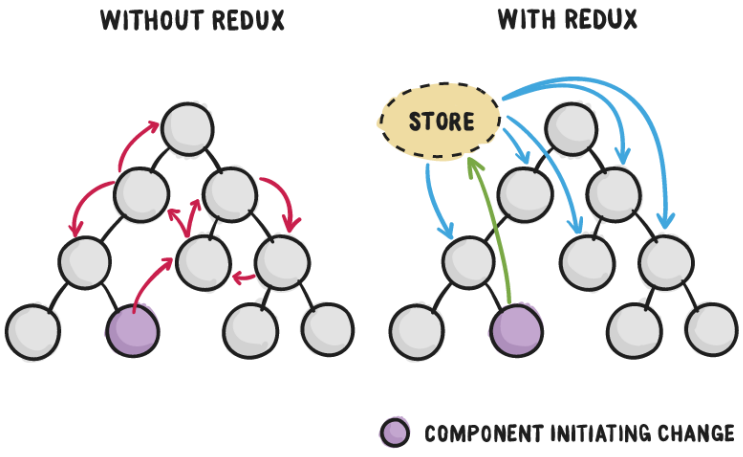

Figure 5.3: Redux State Tree

It is important to note that while the idea of redux is the create reusable components, that is it not always trivial to do this. Sometimes components need to be created that contain logic such as pages and that cannot be reused. This is why the concept of **Smart Components** and

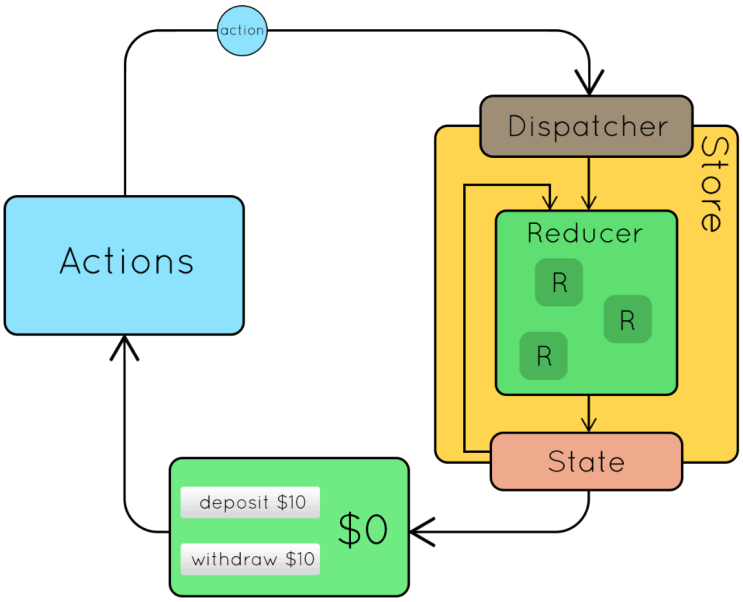

Figure 5.4: Redux Flow

Dumb Components has been introduced. Smart Components are components that contain actions and delegate the results of these actions to the Dumb Components. Whereas Dumb Components only render specific data sent to them.

5.2.4 Socket.io

Figure 5.5: Socket.io logo

Socket.io is a Javascript library that makes real-time data transport between a browser or mobile device and a webserver easy. It does this by utilizing the WebSocket Protocol [31]. Together with this Protocol, Socket.io also provides intelligent 'fallback' algorithms that will try to use different communication protocols if the WebSocket Protocol implementation is not available. [32]

5.3 Methodology

5.3.1 Interfacing with Docker Swarm

Before being able to start working on the visualization part, an important aspect should be solved first. How are we able to fetch the state of the different containers that are running on the different machines? We can do this by creating an interface that is able to interact with our Docker daemon that is running on one of the Manager hosts. This management host has as an advantage that it is a leader in the Docker Swarm infrastructure, thus it is able to manage the different nodes within the Swarm. Docker itself provides a remote API which we can connect to, by using a unix named pipe socket which can be found at '/var/run/docker.sock'. Since this connection is only available on the local client, a proxy server is created that forwards the requests from a remote host towards this unix socket. Here an important security detail is introduced, where the administrator has to refrain users from getting access on a public network, due to the critical endpoints that are included in this proxy server.

5.3.2 Important design aspects

Looking at the data that is available for consumption through the Docker interface, we can see that only some of this data is relevant for the user. The following items are used as main markers that allow us to pinpoint the exact data that should be extracted for visualization:

- **Swarm Nodes:** An overview of the different swarm nodes that are available in the system should be provided. This interface should visualize the different nodes, including their status and memory usage.
- **Containers:** These are running on the different swarm nodes. These should be represented by nodes running on a specific swarm node, each with their own status, memory usage and other statistics.
- **Alerts:** The main goal of this thesis is to detect and represent growing bottlenecks within our infrastructure. To accomplish this, alerts are introduced that can tell the user if and when a certain container is going to reach its predicted resource bottleneck. Through these alerts, an automation interface can be built that allows the system to automatically scale the container once it is going to hit this potential bottleneck.
- **Graphs:** To visualize the data and the analysis done in the next chapter, a graph has to be introduced. This graph visualizes the current data flowing in, and our prediction being made.

5.3.3 Creating the React.js and Redux data flow

Since all our data is now available and the design aspects have been found, the data flow can be created. Our design elements are created through the use of React.js, focusing on components such as a Swarm, SwarmNode, SwarmNodeContainer and Alert component that each visualize a different design aspect.

Data will then be provided to these components through the Redux pattern. This is done by keeping one state tree, containing the information about our different nodes and containers that is being fetched from our proxy interface. When new data is available through an action, our root reducer is being called and compiles a new version of the state tree and updates the components using it.

5.3.4 Alerts

Alerts are time sensitive components that should inform the user about an upcoming event as soon as possible. To create this real-time update, two components are needed:

- **Server:** A server provides data, that it aggregates from different other sources. In our system, this server will read events from kafka and it forwards these events to the Socket.io connection that connects to the frontend.
- **Client:** Our client can be seen as a consumer. It will connect to the Socket.io server and process data when it arrives through an asynchronous event.

Once events are being processed by our client, they can be visualized as an alert component in the Graphical User Interface.

Due to the real time nature of this architecture, an important problem was initially overlooked. What if our data flows in as quickly as possible, and the user interface keeps updating throughout these alerts? This would result in a user interface that would have to redraw itself in real-time, creating possible updates every few milliseconds, which is something our browser is not supposed to handle.

There are different possible solutions for this problem, the events could be delayed in a specific time-window of a few seconds, allowing for an improved refreshing rate of the user interface elements, while still keeping our real-time data as fresh as possible; or we could keep a cache of the different alert components, such that these do not have to be redrawn each time our updates flow in.

In this case, the solution chosen was based on the predictability of our data. Our data does not change within seconds, but grows gradually towards a point where a server should be upscaled. Which is why a small delay can be introduced that will not have an impact on the freshness of our data. If however real-time data is needed for other applications, an extra consumer could be written that reads the data in from the Apache Kafka Infrastructure, still allowing data to be processed in real-time.

5.4 Evaluations and Applications

Looking at the functionality introduced in this Chapter, we can see that our architecture changed. Elements such as a Kafka queue consumer, Socket.IO frontend connections and Proxies have all been introduced allowing for a real-time interaction with the data that is being gathered from different microservices running within our infrastructure.

5.5 Conclusions

To review this chapter, we can see a subsystem that was introduced to visualize the architecture introduced in Chapter 3. The system allows the user to get real-time alerts and information, for a volatile system. The result of this chapter is illustrated in Figures A.1 and A.2.

Chapter 6

Conclusion

6.1 General

This thesis introduced a platform architecture that is able to predict how the resource usage of microservices will change over a period of time. By creating a modular design, we introduced a platform that can achieve higher accuracies by adapting specific components within the architecture.

Through this platform, enterprises are now able to scale their microservice deployments more efficiently. This results in a cost decrease by freeing up resources when they are not needed.

6.2 Contributions

6.2.1 Docker swarm deadlock

While working with Docker Swarm, a deadlock was found that occurred in specific events when the swarm and its hosts were terminated within seconds. The swarm would not detect the hosts leaving, and an invalid state was kept.

This was solved by providing a stack trace to the docker swarm developers, who pinpointed this issue and provided a solution for this that got published in version 1.13.0. [33]

6.2.2 Visualizer

A docker swarm visualizer is already existing on Github. This visualizer is built on outdated technologies such as jQuery and D3 graphing. Because of this, the decision was made to start a new project with technologies such as React and Redux to create an architecture that allows

developers to easily add new features and expand the project. This visualizer has been published to the open-source community for other people to use and contribute to.

6.3 Future Work

This thesis only touched the surface of what can be done in the area of microservice analysis. Because of the structure of the architecture that was built, different components can be replaced or upgraded to fit the needs of enterprises. This section will provide an overview of these different components, and the work that can be done on them to improve them.

6.3.1 Machine Learning Algorithm

A major factor within this architecture is the accuracy of the machine learning model. While in this thesis a Linear Model was used to create predictions, it should be noted that this model is only able to detect rising or decreasing trend lines over the span of the collected data. When more accurate algorithms are required that can detect both, other algorithms should be looked at. To transform this thesis towards a production ready architecture, an algorithm should be chosen that is able to create predictions that match the expected value more closely. A good example for such an algorithm could be a Bayesian Network. This model can be preconfigured in the batch layer, while refinement of it can happen in the real-time layer.

Seasonable Buckets

Another concept to improve the accuracy of our machine learning algorithm, is the concept of buckets. By creating our machine learning models for specific day ranges such as days, seasons or holidays, we are able to finetune the model to improve the accuracy on these days.

6.3.2 Apache Flink

Apache Flink was used in this thesis for the training of the batch and linear models. To further improve the portability and modular architecture, it could be beneficial to use an abstraction layer for this framework. This abstraction layer will then allow us to swap Apache Flink with other big data processing frameworks such as Apache Spark or others. Apache SAMOA [34] is a 'distributed streaming machine learning framework that contains a programing abstraction', essentially doing just this.

6.3.3 Visualizer

The visualizer can be updated to include extra functionality. Currently it will fetch the state of the swarm as it is on the initial load. However when this swarm state changes in time, it would be beneficial to update the visualization through its socket data.

Other extra functionality that can be included is a REST API for the visualizer that can send alerts towards different consumers to inform when a service should be upscaled to handle a higher load.

Appendix A

Visualizer

A.1 Frontpage

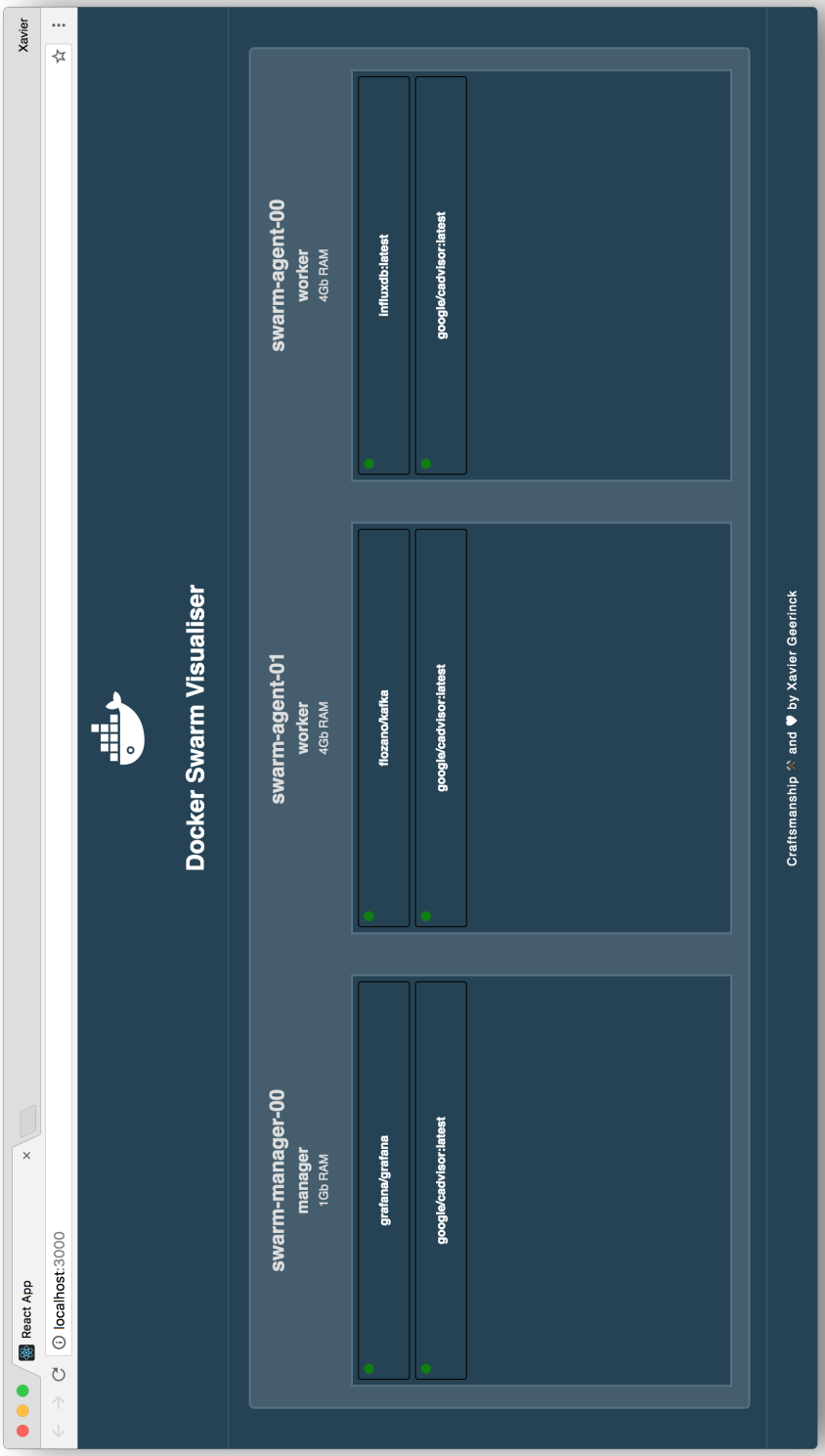

Figure A.1: Frontpage

A.2 Container Detail

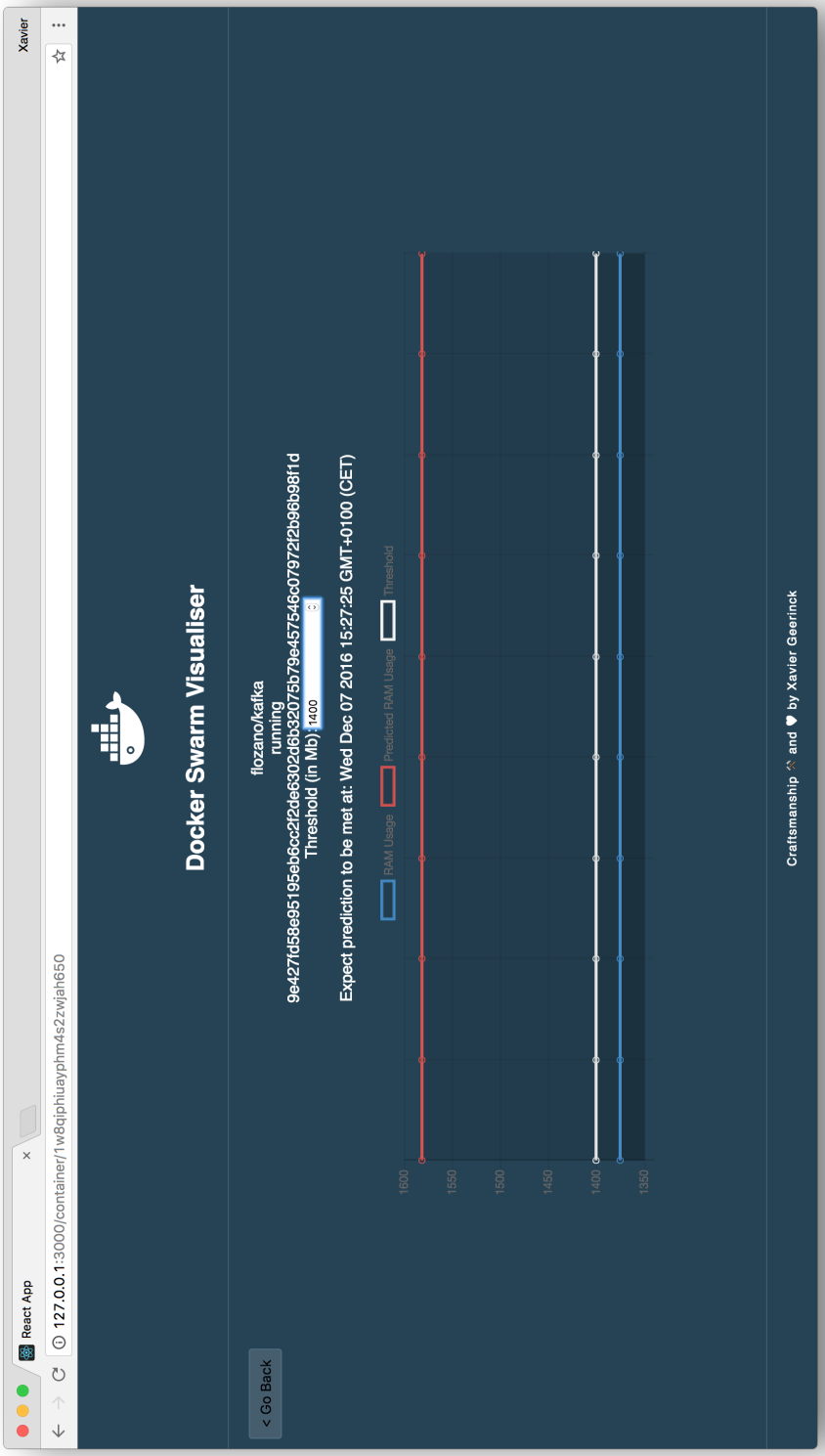

Figure A.2: Container Detail

Bibliography

[1] Microservices. http://microservices.io/patterns/microservices.html. [Online; accessed 13-December-2016].

[2] John Day. The (un)revised OSI reference model. *Computer Communication Review*, 25(5):39–55, 1995.

[3] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[4] Fortune 500. Cisco systems. http://beta.fortune.com/fortune500/cisco-systems-54, 2016. [Online; accessed 14-September-2016].

[5] Cisco Systems. About us - cisco. http://weare.cisco.com/c/r/weare/about-us.html, 2016. [Online; accessed 14-September-2016].

[6] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464–474. IEEE, 2000.

[7] Esxi. http://www.vmware.com/products/vsphere-hypervisor.html. [Online; accessed 14-October-2016].

[8] Docker. https://mesos.apache.org. [Online; accessed 27-September-2016].

[9] Vagrant. https://www.vagrantup.com/. [Online; accessed 29-November-2016].

[10] Docker swarm. https://docs.docker.com/swarm/overview. [Online; accessed 11-October-2016].

[11] Docker swarm key concepts. `https://docs.docker.com/engine/swarm/key-concepts/`. [Online; accessed 29-November-2016].

[12] Vagrant docker swarm code repository. `https://github.com/thebillkidy/vagrant-swarm-cluster`. [Online; accessed 13-October-2016].

[13] Apache kafka. `https://kafka.apache.org/`. [Online; accessed 12-October-2016].

[14] Apache flink. `https://flink.apache.org/`. [Online; accessed 17-October-2016].

[15] cadvisor. `https://github.com/google/cadvisor`. [Online; accessed 01-December-2016].

[16] Nathan Marz. Lambda architecture, 2015.

[17] Apache kafka delivery guarantee. `https://kafka.apache.org/08/documentation.html#semantics`. [Online; accessed 29-November-2016].

[18] Apache zookeeper. `https://zookeeper.apache.org/`. [Online; accessed 26-December-2016].

[19] How twitter uses apache kafka. `https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`. [Online; accessed 29-November-2016].

[20] Apache simple regression. `http://commons.apache.org/proper/commons-math/userguide/stat.html#a1.4_Simple_regression`. [Online; accessed 26-October-2016].

[21] Apache flink clustering. `https://ci.apache.org/projects/flink/flink-docs-release-0.8/cluster_setup.html`. [Online; accessed 01-December-2016].

[22] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[23] Linear regression and modeling. `https://www.coursera.org/learn/linear-regression-model`. [Online; accessed 02-December-2016].

[24] Azure machine learning cheat sheet. http://aka.ms/MLCheatSheet. [Online; accessed 02-December-2016].

[25] Azure machine learning. https://msdn.microsoft.com/en-us/library/azure/dn906022.aspx. [Online; accessed 02-December-2016].

[26] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[27] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 116–, New York, NY, USA, 2004. ACM.

[28] Feature scaling. https://www.coursera.org/learn/machine-learning/lecture/xx3Da/gradient-descent-in-practice-i-feature-scaling. [Online; accessed 04-December-2016].

[29] React. https://facebook.github.io/react/. [Online; accessed 15-November-2016].

[30] Redux cartoon introduction. https://code-cartoons.com/a-cartoon-intro-to-redux-3afb775501a6#.6pllscez4. [Online; accessed 15-November-2016].

[31] Inc. A. Melnikov Isode Ltd. I.Fette, Google. The websocket protocol, 12 2011.

[32] Socket.io. http://socket.io/. [Online; accessed 15-November-2016].

[33] Docker deadlock fix. https://github.com/docker/docker/issues/25432. [Online; accessed 16-November-2016].

[34] Apache samoa. https://samoa.incubator.apache.org/. [Online; accessed 05-December-2016].

List of Figures

List of Tables

Een architectuur voor Resource Analyse, Voorspelling en Visualisatie in Microservice Deployments

Xavier Geerinck

Supervisor(s): Wim Van Den Breen, Pieter Leys (Cisco), Stefaan Vander Rasieren (Cisco)

Abstract— **Dit artikel probeert de betrouwbaarheid van 'High Availability' systemen in grote microservice deployments te verbeteren, door een meerlagen architectuur te introduceren die ons in staat stelt om de 'resource usage' van deze verschillende microservices te analyseren, visualiseren en te voorspellen.**

Keywords— **Architectuur, Machine Learning, Voorspelling, Regressie, Analyse, Visualisatie**

I. Inleiding

VOORKENNIS is een van de meest kritische factoren die beslissen of een onderneming in staat is om op tijd te reageren op de competitie en deze voor te blijven. Wanneer een cloud-onderneming dit niet op tijd detecteert, kan dit resulteren tot verminderde inkomsten. Dit resulteert in een verminderde klantentevredenheid en een mogelijke impact op de verschillende Service Level Agreements omwille van de verminderde uptime.

Deze masterproef pakt dit aan door een meerlagenarchitectuur te introduceren die het mogelijk maakt om microservices te analyseren, visualiseren en hun resourcegebruik te voorspellen in grote microservice omgevingen. Door gebruik te maken van deze analyse en voorspellingsmodellen kunnen we waarschuwingen weergeven vóór deze problemen zich stellen. Een onderneming kan deze waarschuwingen dan gebruiken om een nieuwe strategie uit te denken om aan hun Service Level Agreements te voldoen en de klantentevredenheid te verbeteren.

Uit de gevonden resultaten kan het duidelijk gemaakt worden dat de geïntroduceerde architectuur kan groeien, zelfstandig kan bijleren en de noodzakelijke modellen kan teruggeven. Dit terwijl deze nog steeds modulair genoeg is om aan de complexe eisen van verschillende bedrijfsstructuren te voldoen.

II. Platform Architectuur

A. Inleiding

Het verzamelen van data is een taak die vandaag de dag meer en meer belangrijk wordt. Big data warehouses zijn niet meer onlogisch en worden meer en meer datameren, wat resulteert in vermeerderde capaciteitsbenodigdheden om deze data te bewaren en te verwerken binnen bepaalde tijdsgrenzen.

Deze thesis werkt hier ook mee, met als hoofdzaak om de data te kunnen verzamelen en analyseren om daarna hier mee aan de slag te gaan om de toekomst te voorspellen en wat er kan gebeuren. In dit hoofdstuk wordt er in detail gekeken naar platformarchitectuur die ons toelaat om data die vergaard wordt door verschillende microservices te verwerken, analyseren en te visualiseren. Met als doel om de 'resource usage' in kaart te brengen en op tijd waarschuwingen te genereren wanneer een bepaalde microservice meer resources zal nodig hebben dan er beschikbaar zijn.

B. Orchestratie

De gemaakte architectuur start bij het gebruik van een 'orchestrator'. Deze orchestrator stelt ons in staat om verschillende microservices op verschillende hosts binnen de infrastructuur in kaart te brengen en deze te beheren. Door gebruik te maken van een orchestrator kunnen we globale replicatie gebruiken. Deze gaat een bepaalde microservice repliceren over de verschillende hosts zodat deze beschikbaar is op elke van deze hosts. Hierdoor kunnen we nu de verschillende resources van de verschillende hosts verzamelen en doorsturen naar onze centrale processing hub. In dit artikel wordt er gebruikgemaakt van 'Apache Kafka' als de centrale processing hub.

C. Lagen

Eenmaal de data verzameld is in de centrale processing hub, kunnen we deze verwerken. Om nu betrouwbare voorspellingen te maken, maken we gebruik van drie verschillende lagen binnen onze architectuur.

C.1 Real time Laag

Startend met de Real Time Laag, gaan we data die binnenkomt op de centrale processing hub één per één verwerken in real-time. Dit wordt gedaan door een 'verwerker' te maken die ons toelaat om 'Direct Streams' te maken naar onze processing hub. Deze verwerker kan dan real-time voorspellingsalgoritmen toepassen die onze resultaten gaat genereren. Eenmaal de verwerker klaar is met het verwerken van de data, kunnen we de gemaakte voorspellingen in een ander topic op de processing hub toevoegen voor latere verwerking.

C.2 Batch Laag

Voor onze batch laag wordt data verzameld door een andere verwerker. Deze verwerker gaat data in een databank toevoegen. In dit artikel wordt er gebruikgemaakt van InfluxDB als databank.

Dit opslaan van data gebeurt door het maken van een 'sink' welke de verwerkte data gaat doorsturen naar een ander systeem. In het geval van dit artikel gaat deze verwerkte data dan doorgestuurd worden naar ons databanksysteem. Vermits dit niet aanwezig was in de gebruikte processing hub, is er code geschreven die dit wel kan.

Het uiteindelijk doel van deze laag is om een basismodel te maken dat ons toelaat om patronen te voorspellen die voorgekomen zijn in het verleden van de verschillende microservices. Deze modellen kunnen dan uiteindelijk güpdatet worden door gebruik te maken van onze real-time laag. Deze laag kan dus gezien worden als het aanmaken van een voorgeconfigureerd model dat gebruikt wordt bij de aanmaak van een nieuwe microservice. Dit voorgeconfigureerd model laat ons dan toe om deze aan te passen in real-time met nieuwe voorspellingen om uiteindelijk een betere voorspelling te bekomen. Het is belangrijk om deze laag te aanzien als een laag die niet continu draait maar periodiek.

C.3 Visualisatie Laag

Als laatste laag is er de visualisatie laag. Deze is aangemaakt met de nieuwste technologien zoals react.js en redux, die ons toelaten om gemakkelijk nieuwe 'user interfaces' te maken die overzichtelijk en makkelijk in beheer zijn. Deze laag zorgt ervoor dat onze orchestratie infrastructuur duidelijk gemaakt wordt en de inkomende data in real-time kan verwerkt worden.

III. Model Training

Het meest belangrijke deel in deze architectuur is het gebruikte machine learning algoritme in de verschillende lagen. Dit algoritme gaat resulteren in een model dat ofwel kosteffiçiënt is, ofwel niet.

Voor beide algoritmen is de keuze gemaakt om Lineaire Regressie te gebruiken om de werking van de gemaakte architectuur te demonstreren. Lineaire Regressie gaat proberen een relatie te vinden tussen verschillende punten op een dataplot en zo een trendlijn te voorspellen die ons gaat zeggen waar de data naartoe zal gaan in de toekomst. Dit algoritme heeft de vorm $y = a * x + b$.

Om daarna voorspellingen te maken moeten we enkel de x-component uit deze vergelijking halen voor een specifieke y-waarde.

A. Batch Training

Om de batch laag te trainen gaat het lineaire algoritme gebruik maken van Gradiënt Descent. Gradiënt Descent gaat proberen om de kost functie die geïllustreerd is in Algoritme 1 te minimaliseren. Het doet dit door gebruik te maken van de mathematische gradiënt die snel kan itereren en zo zijn minimum kan bepalen.

Algorithm 1 Gradient Descent voor Lineare Regressie

1: **repeat**
2: $\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$
3: $\theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$
4: **until** convergence

Dit algoritme kan makkelijk geïmplementeerd worden met behulp van Apache Flink. Apache Flink is een big data processing framework, welke ons toelaat om grote hoeveelheden data te verwerken als 'batches' of als real-time streams. [2]. In onze batch laag gaan we deze data verwerken als batches en daarna het gradient descent algoritme toepassen. Dit resultaat wordt uiteindelijk opgeslagen in een lokaal bestand om dan hergebruikt te worden door het real-time trainingsalgoritme.

B. Real-time Training

Real-time training wordt bekomen door het maken van een verwerker op de processing hub. Deze verwerker laat ons toe om de inkomende data in real-time te verwerken. Deze verwerker gaat dan hetzelfde gradient descent algoritme gebruiken als in de Batch Training sectie. Maar met als grootste verschil dat de grootte van de batch maar een enkel datapoint is. Dit wordt ook 'Stochastic Gradient Descent' genoemd. [3].

De functie van deze laag is om zichzelf snel aan te passen aan de inkomende data op een real-time manier. Dit laat ondernemingen toe op snel te reageren op een verandering van de trends die aan het gebeuren zijn binnen de verschillende microservices.

IV. Resultaten

Na de aanmaak van de verschillende lagen kunnen er resultaten behaald worden. Eerst wordt een uiteenzetting gedaan van de gemaakte architectuur, waarna een conclusie getrokken wordt. Daarna wordt er dieper gekeken naar de verschillende lagen en hun gebruikte modellen.

A. Platform Architectuur

Als we kijken naar de gemaakte architectuur, kunnen we zien dat de architectuur bestaat uit verschillende modules die individueel veranderd kunnen worden. Door gebruik te maken van dit design, kan de architectuur gemakkelijk geschaald worden om zichzelf aan te passen aan de complexe noden van de verschillende ondernemingen en hun unieke use cases.

B. Machine Learning Modellen

Kijkend naar de resultaten van de verschillende machine learning modellen die geïllustreerd zijn in de onderstaande figuren kunnen er verschillende conclusies getrokken worden.

Voor ons real-time model, geïllustreerd door Figuur 1. Kunnen we zien dat wanneer er geen voorconfiguratie gemaakt wordt door ons batch model dat er een grotere convergentietijd gaat zijn die resulteert in een minder nauwkeurig model in de startfase van een container. Omwille van de korte levensduur van deze containers is het dus nadelig om dit te hebben, wat de nood aan een real-time laag illustreert.

Deze preconfiguratie kan gemaakt worden door de batch laag, als gellustreerd in Figuur 2. Hier kunnen we zien dat het algoritme probeert om een lijn te trekken door de verschillende datapunten, resulterend in onze voorconfiguratie. Deze voorconfiguratie kan den hergebruikt worden en verder op getraind worden. Dit algoritme neem ook de historische kenmerken in acht zoals trendlijnen, anomalieën en pieken.

Als de voorconfiguratie en het real-time model uiteindelijk gecombineerd worden zoals geïllustreerd in Figuur 3, kunnen we zien dat het algoritme werkt zoals verwacht. De initiële convergentietijd van het algoritme is veel korter en laat ons toe om betere voorspellen te genereren van in het begin.

V. Conclusie

Dit artikel introduceert een platformarchitectuur zoals voorgesteld in Figuur 4. Deze laat ons toe om de resource usage van microservices te voorspellen. Door het maken van een modulair design, is er een platform geïntroduceerd die een hogere nauwkeurigheid teweegbrengt. Deze nauwkeurigheid kan nog verder geoptimaliseerd worden door de verschillende componenten in de architectuur te veranderen. Door gebruik te maken van dit platform kunnen ondernemingen hun microservices beter beheren en de kosten verminderen wanneer bepaalde resources niet meer nodig zijn.

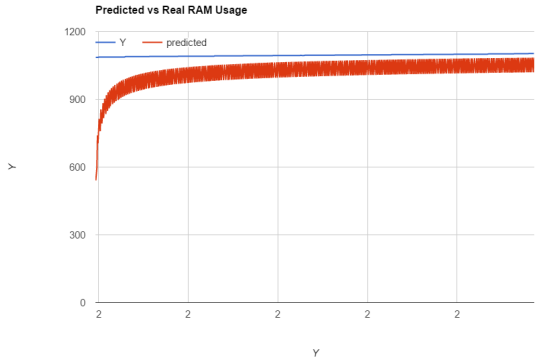

Fig. 1. Realtime Training Model

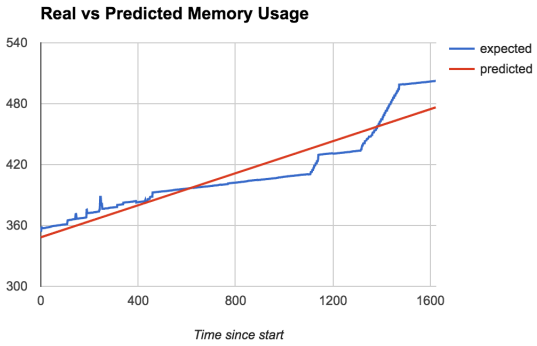

Fig. 2. Batch Training Model

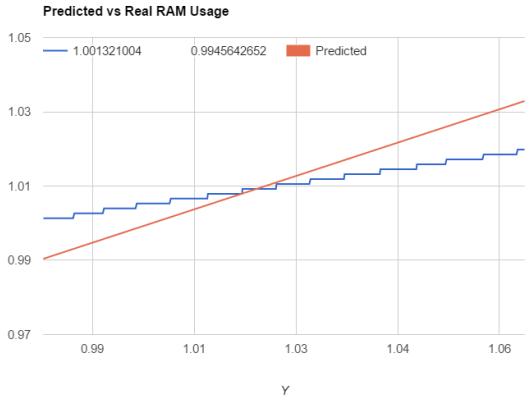

Fig. 3. Voorgeconfigureerd Real-Time Training Model

References

[1] Microservices, http://microservices.io/patterns/microservices.html [Online; accessed 13-December-2016]

[2] Apache Flink, https://flink.apache.org/, [Online; accessed 17-October-2016]

[3] Zhang, Tong. "Solving large scale linear prediction problems using stochastic gradient descent algorithms." Proceedings of the twenty-first international conference on Machine learning. ACM, 2004.

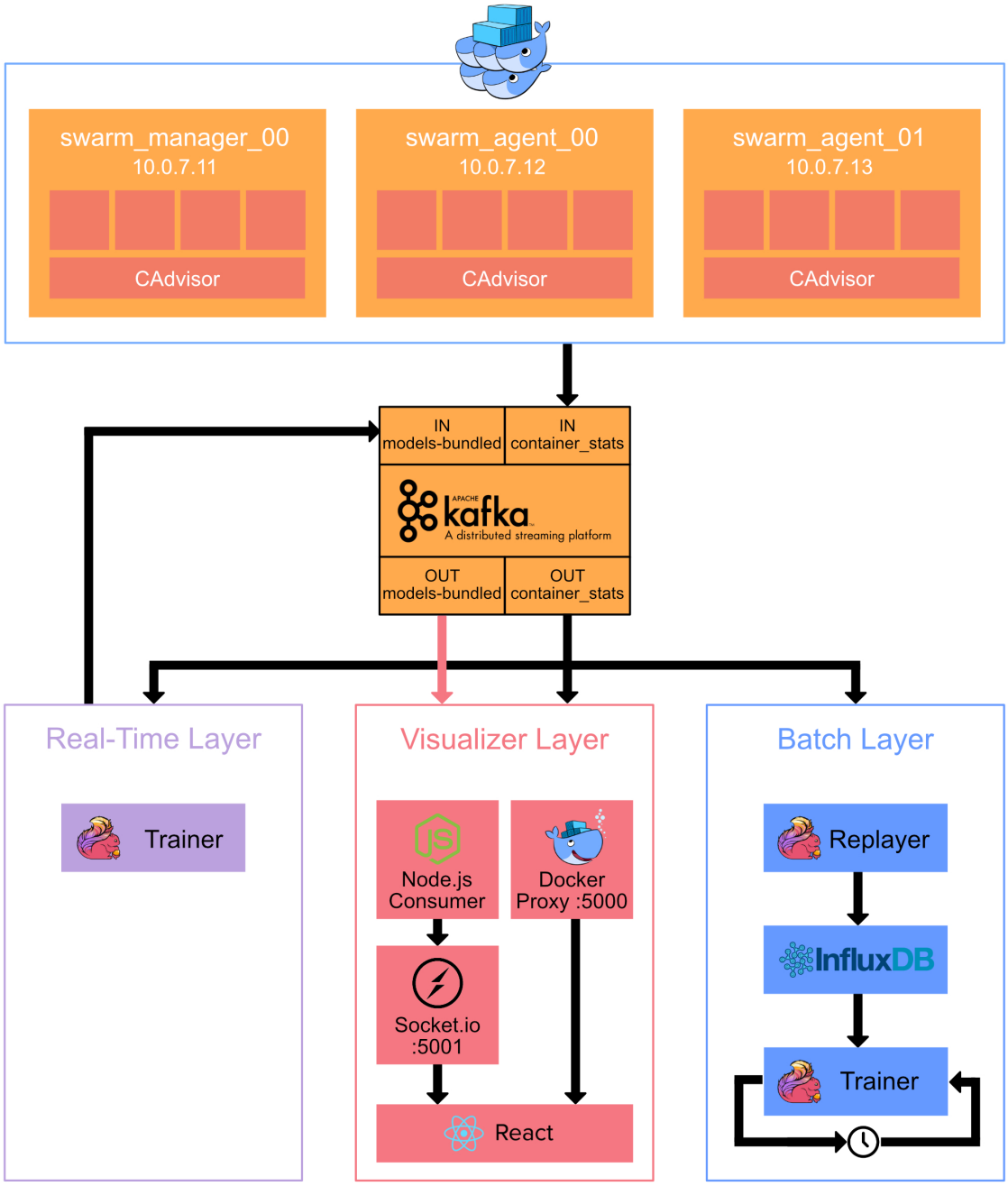

Fig. 4. Gemaakte Architectuur

An Architecture for Resource Analysis, Prediction and Visualization in Microservice Deployments

Xavier Geerinck

Supervisor(s): Wim Van Den Breen, Pieter Leys (Cisco), Stefaan Vander Rasieren (Cisco)

Abstract—**This article tries to improve the reliability of High Availability systems in large microservice deployments by introducing a multi-layered architecture that is able to analyze, visualize and predict the resource usage of microservices.**

Keywords— **Architecture, Machine Learning, Prediction, Regression, Analysis, Visualization**

I. Introduction

FOREKNOWLEDGE is one of the critical deciding factors for any enterprise to outdistance the competition in a timely manner. When a cloud enterprise is not able to detect this within a timely manner, this can lead to decreasing revenues and the occurrence of potential bottleneck issues. Resulting in a decreased customer satisfaction and Service Level Agreement violations due to the decreased uptime of these services.

This thesis tackles this by introducing a multi-layered architecture that is able to analyze, visualize and predict the resource usage of microservices in a large scale deployment. Through the use of this analysis, prediction models can be created that are able to issue warnings before the occurrence of these bottlenecks. The enterprise can then use these warnings to devise a new strategy for the microservices deployments to comply to the Service Level Agreements and increase the customer satisfaction.

From the results obtained in this thesis, it was made clear that the introduced architecture is able to grow, iteratively learn by itself and return the needed models, while still being modular enough to meet complex demands of the different enterprise structures.

II. Platform Architecture

A. Introduction

Data Collection is a task in the world today that is becoming more and more important. Big data warehouses are not unusual anymore and are more and more becoming data lakes, increasing the need to store and process this data within given time constraints. This thesis is no different from that, the main part being the collection of data and analyzing this data to predict the future and its events. In this chapter a platform architecture is detailed that is able to process, analyze and visualize data that is being produced by different microservices with the goal to predict their resource usage and to issue alerts once a microservice should be upscaled to handle the load on the service.

B. Orchestrator

The created architecture starts by the creation of an orchestrator. This orchestrator manages the different microservices running on each host within the infrastructure. By using an orchestrator, we can utilize the global replication feature that will replicate a microservice on each host within the orchestration infrastructure. This allows us to create a microservice that can collect resources from the different hosts within our infrastructure, and send these to a central processing hub. In this article Apache Kafka was used as this central processing hub.

C. Layers

Once this data is being collected in a central processing hub, we are able to process it. To create accurate predictions for the resource usage, three layers are introduced.

C.1 Real time Layer

Starting with the Real Time Layer, we need to process the data coming in on the central processing hub, one item at a time. This is done by creating a consumer that is able to create Direct Streams to our processing hub. This consumer then also applies different prediction algorithms that will create our results. Once the consumer is done processing the data, we can forward the gathered results into a different topic on the processing hub.

C.2 Batch Layer

For our Batch Layer, data has to be collected through a consumer. This consumer will forward the data into a database that is chosen by us. In the case of this article the choice was made to use InfluxDB as this database system.

This forwarding is done by creating a sink where all our processed data will be sent to. Seeing that this is a custom sink which is not available as a library, custom code had to be written to do this.

The eventual goal for this layer is to create a base model that is able to predict patterns that occurred in the history of the different microservices and that can then be updated through the use of the real time layer. This can be seen as creating a preconfiguration model that will be utilized when starting a new microservice. This preconfiguration model will allow the real time layer to create more accurate predictions from the start. It is important to note that this layer should not be a continuous running process but rather a process that should be run after a specific period.

C.3 Visualization Layer

Finally a visualization layer is created through the use of cutting edge technologies such as react.js and redux, that introduces an easy to use interface, detailing how the orchestration infrastructure is built up. This visualization layer is able to process

incoming data in realtime and visualize warnings when a microservice is about to hit its capacity.

III. Model Training

The most critical part of this architecture are the machine learning algorithms used within the different layers. These machine learning algorithms will result in a model that will decide if a cost efficient solution can be found.

For both these algorithms, the choice was made to use Linear regression to prove that both layers work as intended. Linear Regression will try to find the correlation between different datapoints and plot a trend line through these points. As the name indicates, it is a linear algorithm providing a result in the form of $y = a * x + b$.

Finding predictions through this algorithm can then easily be done by extracting the x-component for a specific y-value.

A. Batch Training

To train the batch layer, a linear regression algorithm that utilizes Gradient Descent. Gradient Descent will try to minimize the cost function illustrated in Algorithm 1, by using the mathematical gradient to quickly iterate and converge to the minimum of this function.

Algorithm 1 Gradient Descent for Linear Regression

1: **repeat**
2: $\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$
3: $\theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$
4: **until** convergence

This algorithm can easily be implemented using Apache Flink. Apache Flink is a big data processing framework, that is able to process big data as batches or as realtime streams [2]. In our batch layer we are able to process this data as batches and apply gradient descent on it, resulting in the lowest cost for this batch. This result is then saved towards a local file that can be used as a starting point for the real time training algorithm.

B. Real time Training

Real time training is performed by creating a consumer on the processing hub that allows us to consume incoming data in real time. This consumer will then run the same gradient descent algorithm as described in the batch training section, but with as a difference that a batch size of one datapoint is used. This is also commonly referred as 'Stochastic Gradient Descent' [3].

The function of this layer is to adapt towards incoming data in a real time fashion, allowing enterprises to react quickly to new trends happening within the different microservices.

IV. Results

After creation of the different layers, several results can be extracted. First a discussion is made about the architecture as a whole, whereafter conslusions are drawn. Thereafter a more in-depth review of the different layers is made with their created models.

A. Platform Architecture

Looking at the architecture created, it can be seen that the architecture is built up out of different modules that are interchangeable. By following this design, the architecture is able to scale and adapt to the complex needs of the different enterprise use cases.

B. Machine Learning Models

Looking at the results of the different Machine Learning models as illustrated in the figures below, several conclusions can be made.

For the real time model, illustrated by Figure 1. It can be seen that when no preconfiguration is created by the use of a batch model that a specific convergence time is needed for the model to become accurate. Due to the short lived nature of microservices, this is a time that should be minimalized as much as possible.

This preconfiguration can be made by the batch layer, illustrated by Figure 2. Here it can be seen that the algorithm tries to find a graph through the specific datapoints, resulting in a preconfigured model that can be re-used and trained upon. This algorithm also takes the historical features into account such as trend lines, anomalies, spikes and others.

When the preconfiguration and the real time model are finally combined as illustrated in Figure 3, it can be seen that the algorithm behaves as expected. The initial convergence time of the algorithm is way shorter and allows for better predictions to be made from the start.

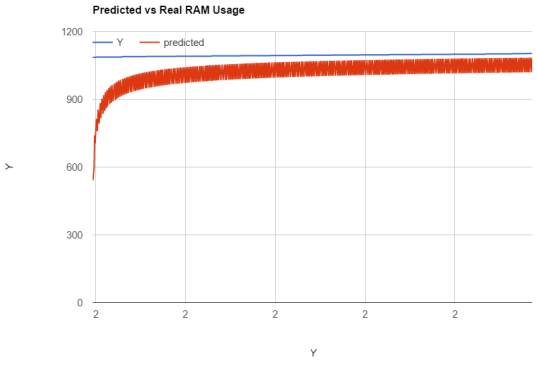

Fig. 1. Realtime Training Model

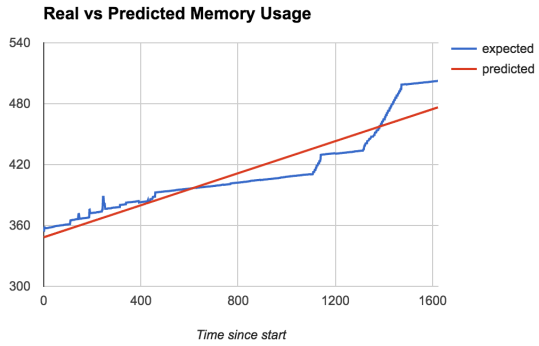

Fig. 2. Batch Training Model

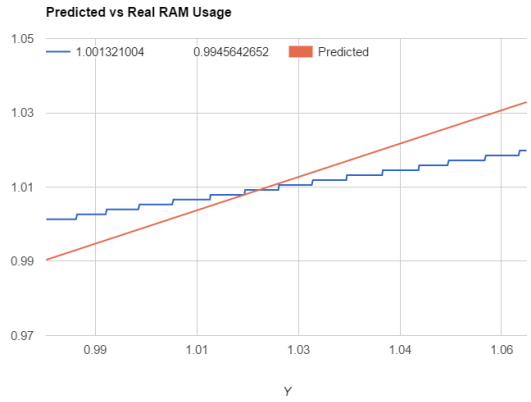

Fig. 3. Preconfigured Real Time Training Model

V. Conclusion

This thesis introduced a platform architecture as visualized in Figure 4 that is able to predict how microservices their resources are going to change over a period of time. By creating a modular design, we introduced a platform that can achieve higher accuracies by adapting specific components within the architecture. Through this platform, enterprises are now able to scale their microservice deployments more efficiently, resulting in a cost decrease by freeing up resources when they are not needed.

References

[1] Microservices, http://microservices.io/patterns/microservices.html [Online; accessed 13-December-2016]

[2] Apache Flink, https://flink.apache.org/, [Online; accessed 17-October-2016]

[3] Zhang, Tong. "Solving large scale linear prediction problems using stochastic gradient descent algorithms." Proceedings of the twenty-first international conference on Machine learning. ACM, 2004.

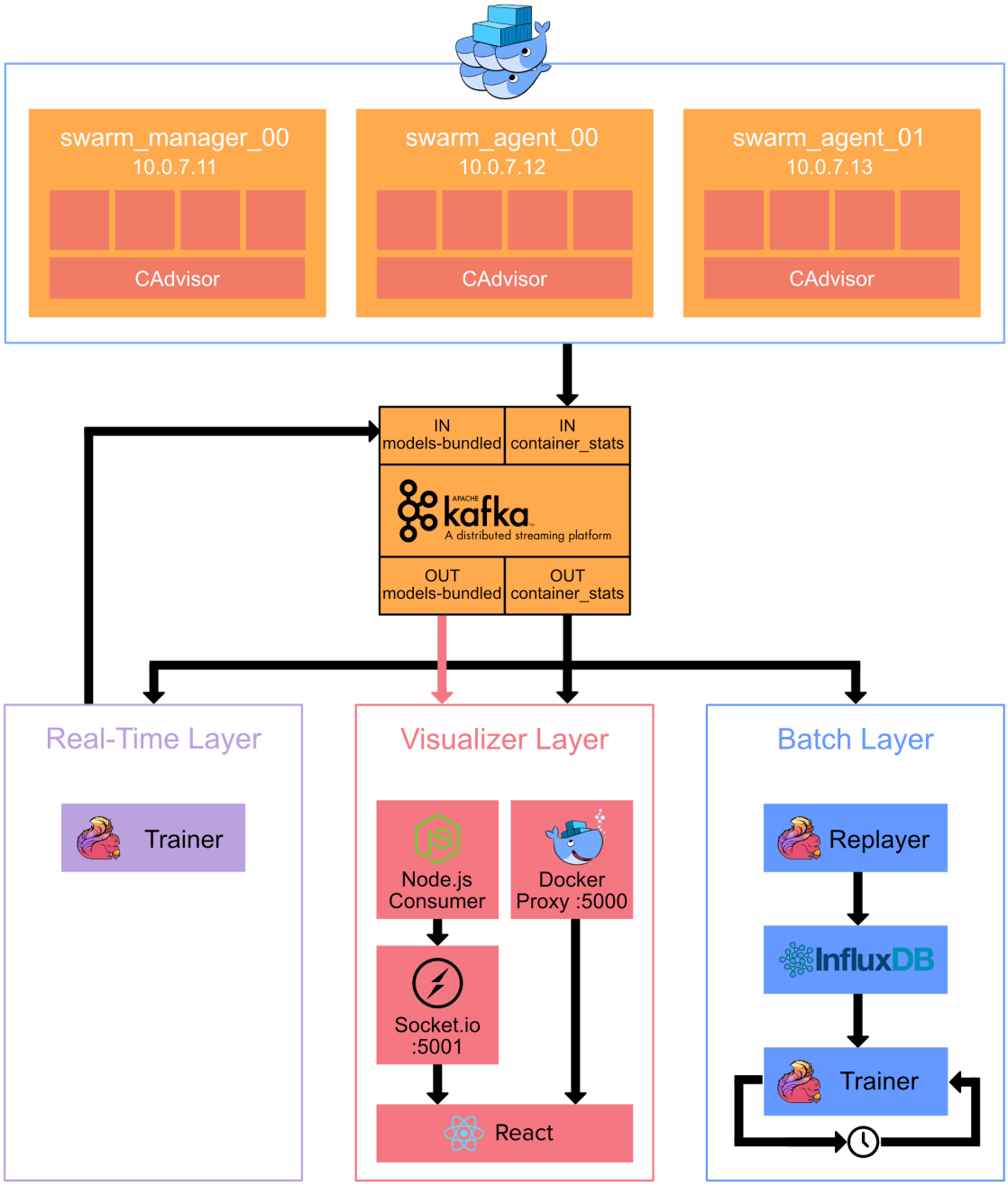

Fig. 4. Created Architecture