HOCHSCHULE DER MEDIEN

Fraunhofer IPA

Bachelor Thesis

My Bachelor Thesis

vorgelegt von

Marcel Pascal Stolin

32168

geboren am 03.04.1993 in Kamen

Verfasst am

Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA

und

Hochschule der Medien Stuttgart

Erstprüfer:	Prof. Walter Kriha
Zweitprüfer:	Prof. Dr. Marco Huber
Betreuung:	M.Sc. Christoph Hennebold

Stuttgart, December 31, 2020

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich, Marcel Pascal Stolin, die vorliegende Bachelor Thesis selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und sowohl wörtliche als auch sinngemäß entlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum: _____

Unterschrift: _____
Marcel Pascal Stolin

Zusammenfassung

Hier kommt eine deutschsprachige Zusammenfassung hin.

Abstract

Abstract in English.

List of Figures

List of Tables

Contents

Notation

Konventionen

x	Skalar
$\underline{x}$	Spaltenvektor
$\boldsymbol{x}, \underline{\boldsymbol{x}}$	Zufallsvariable/-vektor
$\hat{x}, \hat{\underline{x}}$	Mittelwert/-vektor
$x^*, \underline{x}^*$	Optimaler Wert/Vektor
$x_{0:k}, \underline{x}_{0:k}$	Folge von Werten $(x_0, x_1, \ldots, x_k)$ / Vektoren $(\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_k)$
$\mathbf{A}$	Matrizen
$\mathcal{A}$	Mengen
$\preceq, \prec$	schwache/strenge Präferenzrelation
$\mathbb{R}$	Reelle Zahlen
$\mathbb{N}$	Natürliche Zahlen
■	Ende eines Beispiels
□	Ende eines Beweises

Operatoren

$\mathbf{A}^{\mathrm{T}}$	Matrixtransposition
$\mathbf{A}^{-1}$	Matrixinversion
$\lvert\mathbf{A}\rvert$	Determinante einer Matrix
$\lvert\mathcal{A}\rvert$	Kardinalität der Menge $\mathcal{A}$
$\mathrm{pot}(\mathcal{A})$	Potenzmenge von $\mathcal{A}$
$\mathrm{E}\{\cdot\}$	Erwartungswertoperator
$\mathcal{O}(g)$	O-Kalkül entsprechend der Landau-Notation bei welcher beispielsweise $f(x) \in \mathcal{O}(g(x))$ besagt, dass die Funktion $f(x)$ die Komplexität $\mathcal{O}(g(x))$ besitzt

Spezielle Funktionen

$\Pr(\mathcal{E})$	Wahrscheinlichkeitsmaß, welches die Wahrscheinlichkeit angibt, dass Ereignis $\mathcal{E}$ eintritt
$p(\underline{x})$	(Wahrscheinlichkeits-)Dichtefunktion für kontinuierliche $\underline{x}$

	und Zähldichte für diskrete $\underline{x}$
$p(\underline{x}\|\underline{y})$	Bedingte Dichtefunktion
$P(\underline{x})$	(Wahrscheinlichkeits-)Verteilungsfunktion
$\mathrm{erf}(x)$	Gauß'sche Fehlerfunktion
$\exp(x)$	Exponentialfunktion e^x
$\mathcal{N}(\underline{x};\underline{\hat{x}},\mathbf{C}_x)$	Gaußdichte, d. h. Dichtefunktion eines normalverteilten Zufallsvektors $\underline{\boldsymbol{x}}$ mit Mittelwertvektor $\underline{\hat{x}}$ und Kovarianzmatrix $\mathbf{C}_x$

Chapter 1

Introduction

1.1 Distributed Computing

Machine Learning and Big Data projects consist of a combination of extract-transform-load (ETL) pipelines and compute intensive algorithms to create meaningful informations from massive data sets QULLE DevOps: CON.. . Performing ETL applications on a single machine, limits the scale and performance of the application. Given this limitation, the idea of distributed computing evolved, where multiple machines with commodity hardware form a cluster to utilize their resources to solve high complex problems [GOKB16].

Several companies utilized the idea of distributed computing to solve some of their business problems. Google developed the MapReduce [DG04] framework. MapReduce gave the opportunity to solve massive complex problems in parallel on a cluster of single machines. Yahoo published an ecosystem platform for distributed computing called Hadoop QUELLE. Hadoop contributed to create a cluster to process massive amounts of data in parallel. Implementing data pipelines requires to chain multiple MapReduce jobs together. This causes a huge amount of writing and reading operation to the disk with bad impact on the overall performance. Another framework called Apache Spark was developed to simplify writing and executing parallel application at scale while keeping the benefits of MapReduce‘s scalability and fault-tolerant data processing. Apache Spark provides a performance improve of 10x in iterative Machine Learning algorithms over MapReduce [ZCF+10] and has evolved as a replacement for MapReduce as the distributed computing framework of choice.

1.2 Computing Acceleration with GPUs

Distributed computing frameworks like Apache Spark perform applications on a huge amount of CPU cores to enable parallization. A CPU is build of multiple cores which are optimized for sequential serial processing QUELLE. Performing computationally intensive applications on an Apache Spark cluster, consumes a huge amount of CPU cycles with bad impact on the

overall performance QUELLE HETEROSPARK. To handle the complexity of Big Data applications, from executing Machine Learning algorithms or training Deep Loearning models, a remaining option of distributed computing clusters is ti scale-up individual nodes. Scaling-up is limited by resource capacity and can be become uneconomically at a specific point. To perform computationaly complex applications with better performance, GPUs have become first class citizens in modern data centers. The architecture of a Graphical Process Unit (GPU) consists of a huge amount of smaller and more efficient cores which are optimized for parallel data processing (handling multiple tasks simoultaneously). In general, GPUs process data at a much faster rate than CPUs are capable. Leveraging GPUs for distributed computing frameworks like Apache Spark, can boost the overall performance of performing complex algorithms on large datasets.

1.3 Auto-Scaling

Adjusting the resources in a computing environment is not an easy task. To do it manually, a system administrator needs deep knowledge about hte environment and has to watch performance spikes regualarly. This is a resource wasting process. In an optimal way, an automized process would watch the computing enviroment, analyze perofmrance metrcis and automatically add or remove resource to optimize the performance and cost. This process is called an Auto-Scaler.

Hiring experts to manually watching an application and scaling an computing environment is a waste of reources and cost. AN Auto-Scaler takes care of watching the environment and adding and removing resources to the computing needs. An Auto-Scaler can be configured to take care of optimal resource allocation and keeping the cost of running low.

Auto-scaling a computing environment can be achieved with two different scaling approaches: Vertical-scaling and horizontal-scaling. Vertical scaling refers to adjusting the hardware resources of an individual node in the environment. Hardware adjustments can include adding (scale-up) or removing (scale-down) resources like memory or CPU cores [Wil12]. By adding more powerful resources to a node, a node can take more throughput and perform more specialized tasks [LT15]. Adjusting the nodes in a computing environment is referred to horizontal scaling [Wil12]. Increasing the number of nodes in an environment, increases the overall computing capacity and in addition, the workload can be distributed across all nodes [Wil12, LT15]. Both scaling approaches are not exclusive. A computing environment can be designed to scale vertically, horizontally or both [Wil12]. Vertical scaling is limited by the maximum hardware capacity. In addition, a point can be reached where more powerful hardware resources become unaffordable or are not available [LT11]. Therefore, horizontal scaling is the preferred approach to enable auto-scaling.

1.4 DevOps

DevOps (Development plus Operations) is way that automates process during the development life cycle of a product. Development phases can be testing, monitoring or deployment [Vad18]. The automation of development processes leads to an overall improvement of development. It reduces human errors and the downtime for upgrades as well benefits to the improvement of productivity [Vad18].

Machine Learning and Big Data Projects benefits from DevOps [Vad18].

To integrate DevOps methodologies into a projects, an optimized strategy is key. The strategy needs to provides automation for each aspect of the projects life cycle. Therefore, it needs to integrate the correct tools, platforms and technologies [Vad18]. An integration of DevOps methodologis into the life cycle of a software project can serve several purposes [Vad18]:

- Automating the management of the infrastructure and configuration
- Automating testing and build workflows of source ode repositories
- Continuous integration and deployment
- Virtualization and containirazion

1.5 Problem Statement

1.5.1 Performance

For complex machine learning applications working with massive datasets, it is necessary to distribute the workload to enable parallel execution. Therefore, a cluster has to be created to enable nodes to distribute their workload. Apache Spark will be used to create such a cluster. Apache Spark distributes the workload on multiple CPU cores on each node to enable parallelism. Machine Learning algorithms benefits from GPUs which enable data parallelism. In this context, the cluster needs to be extendend with GPUs and Apache Spark needs be aware how to distribute workload on GPUs as well. Since

1.5.2 Auto-Scaling

Performing complex Machine Learning algorithms on large datasets requires a huge amount of resources. Those resources need to be monitored frequently to be able to detect performance spikes. If performance spikes are detected, the system needs to calculate how many nodes to add to keeping the performance on a normal level.

1.6 Research Questions and Research Objective

The goal of this thesis is to implement a distributed computing environment to perform machine learning applications. The environment should be able to scale itself automatically in accordance to the workload caused by performing the Machine Learning applications. In addition, the Machine Learning application should be performed automatically in a DevOps process. To address the goal of this research work, the following two research question will be investigated:

1. Is it possible to extend the number of Spark Worker in accordance to performance?
2. Can we extend Apache Spark to accelerate execution with GPU support?

The first research question searches for the possibilities to create an elastic Apache Spark cluster. This question also builds the foundation of the computing environment used for this thesis. To answer this question, state-of-the-art computing architectures have to be investigated. During the investigation, tools have to be evaluated that enable a simple deployment of Apache Spark worker instances and monitoring the performance metrics of the computing environment. In Addition, horizontal auto-scaling approaches that fit into the computing environment have to investigated.

For the second research questions, approaches need to be investigate to extend Apache Spark to perform operation on GPUs. As mentioned in SEC X, Apache Spark only knows about CPUs. The aim to answer this question, is to find a way to add GPUs to the computing cluster and make them available for Apache Spark.

1.7 Thesis Structure

1.8 Research Methodologies

The first step is to read state-of-the-art literature. Literature about elastic computing architectures needs to be investigated to compare different tools for monitoring an environment and deploying computing nodes on demand. In addition, literature about extending an Apache Spark cluster with GPU support needs to be investigated. The collected literature needs to be classified into different categories. First all architecture related paper should be summarized, second all literature about horizontal auto-scaling needs to be summarized and last all literature about GPU support for Apache Spark needs to be summarized. After gaining knowledge about state-of-the-art technologies and approaches, the Apache Spark cluster needs to be implemented. After an Apache Spark cluster is available, the Auto-Scaler has to be implemented. If the Auto-Scaler is available, it needs to be tested.

GPU support has to bee added to the cluster. With GPU support, the scalable environment can be evaluated with Machine Learning algorithms.

Chapter 2

Theoretical Foundation

This chapter provides background information about ...

TODO: Describe Chapter

2.1 Scalability

Scalability defines the ability of a computing system to handle an increasing amount of load [Far17]. The main reasing for scaling is High-Availability TOOLKIT 2.1. To choose the right approach of scaling an environment is essential. To scale the computing capacity of a system, different approaches exists. A major scaling approach is horizontal scaling.

2.1.1 Horizontal Scaling

Horizontal scaling is accomplished by duplicating nodes in the computing environment [Wil12]. Dplucation the nodes in a computing environment increases the computational capacity of the environment. In addition, the workload can be distributed across all clones to handle and balance and increasing load of tasks [Wil12, LT15]. To increase the efficiency of horizontal scaling, all nodes should be homogeneous. Homogeneous nodes are able to perform the same work and response as other nodes [LT15].

2.1.2 Limitations of Scalability

The limit of scalability is reached, when a computing system is not able to serve the requests of it‘s concurrent users [Wil12]. If the scalability of a computing environment is reached, an option is to add more powerful hardware resources to the system. This approach is called vertical scaling QULLE??. By adding more powerful hardware resources, the point can be reached, where more powerful hardware becomes unaffordable or not available [Wil12]. To overcome the limits of hardware cpacity, a computing system shuld be designed to scale horizontally in the first place [LT15].

2.2 Autonomic computing

Autonomic computing is the ability of an IT infrastructure to automatically manage itself in accordance to high level objectives defined by administrators [KC03]. Autonomic computing gives an IT infrastructure the flexibility to adapt dynamic requirements quickly and effectively to meet the challenges of modern business needs [Mur04]. Therefore, autonomic computing environments can reduce operating costs, lower failure rates, make systems more secure and quickly respond to business needs [JSAP04].

Computing systems need to obtain a detailed knowledge of it‘s environment and how to extend it‘s resources to be truly autonomic [Mur04]. An autonomic computing system is defined by four elements:

- **Self-configuring:** Self-configuring refers to the ability of an IT environment to adapt dynamically to system changes and to be able to deploy new components automatically. Therefore, the system needs to understand and control the characteristics of a configurable item [Mur04, Sin06].
- **Self-optimizing:** To ensure given goals and objectives, a self-optimizing environment has the ability to efficiently maximize resource allocation and utilization [JSAP04]. To accomplish this requirement, the environment has to monitor all resources to determine if an action is needed [Mur04].
- **Self-healing:** Self-healing environments are able to detect problematic operations and then perform policy-based actions to ensure that the systems health is stable [Sin06, JSAP04]. The policies of the actions have to be defined and should be executed without disrupting the system [Sin06, JSAP04].
- **Self-protecting:** The environment must identify unauthorized access and threats to the system and automatically protect itself taking appropriate actions during its runtime [Sin06, JSAP04].

2.2.1 Autonomic computing concept

Figure 2.1 demonstrates the main concept of an autonomic computing environment. The autonomic computing architecture relies on monitoring sensors and an adoption engine (autonomic manager) to manage resources in the environment [GBR11]. In an autonomic computing environment, all components have to communicate to each other and can manage themselves. Appropriate decisions will be made by an autonomic manager that knows the given policies [JSAP04].

The core element of the autonomic architecture is the control-loop. Figure 2.2 illustrates the concept of a control-loop. The control-loop collects details about resources through monitoring and makes decisions based on analysis of the collected details to adjust the system if needed [Mur04].

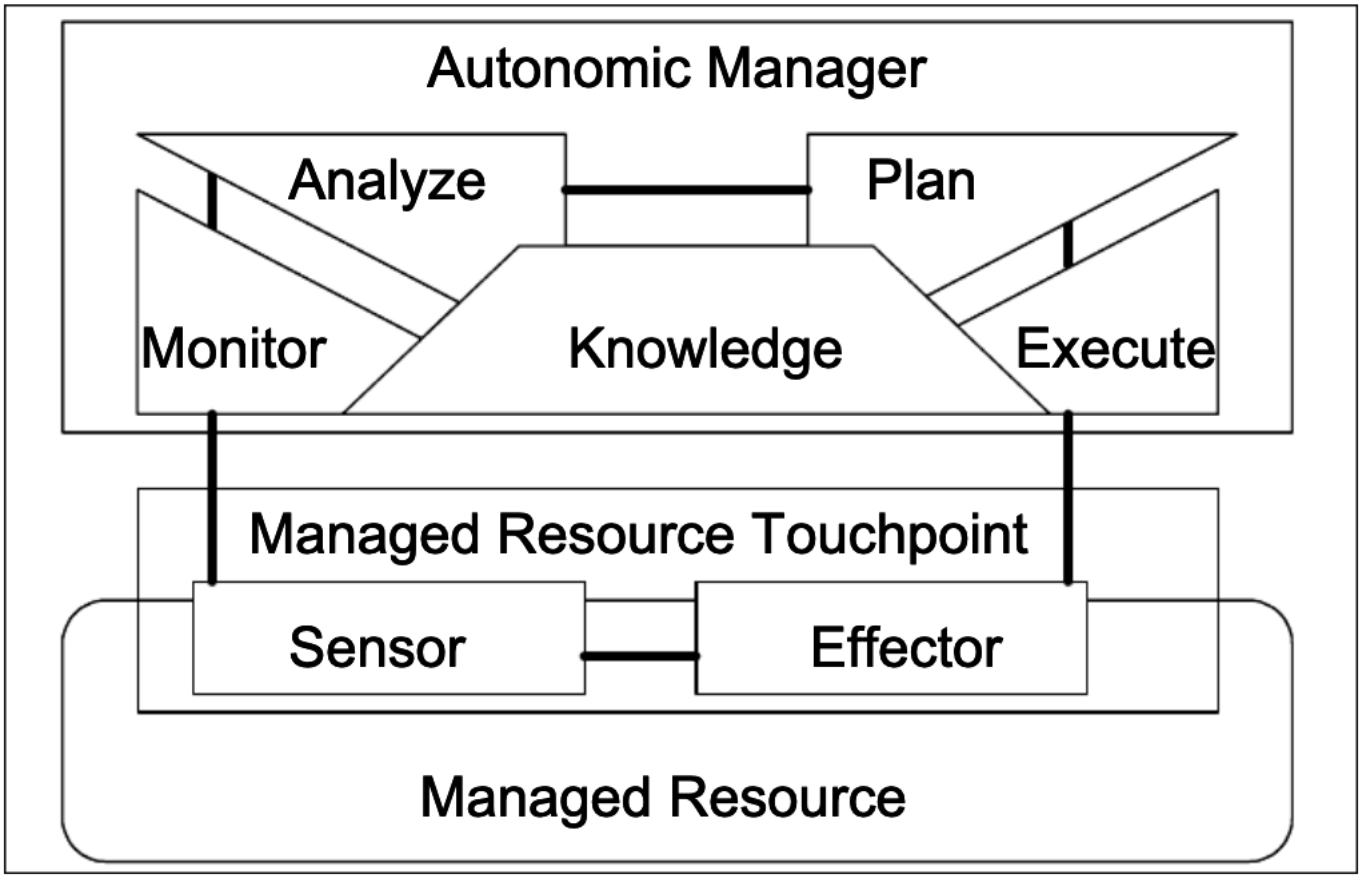

Figure 2.1: Autonomic computing concept - Source: Authors own model, based on [JSAP04].

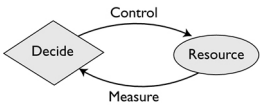

Figure 2.2: The control-loop concept - Source: Authors own model, based on [Mur04].

2.2.2 Managed resources

A managed resource is a single component or a combination of components in the autonomic computing environment [Mur04, JSAP04]. A component can be a hardware or software component, e.g. a database, a server, an application or a different entity [Sin06]. They are controlled by their sensors and effectors, as illustrated in Figure 2.3. Sensors are used to collect information about the state of the resource and effectors can be used to change the state of the resource [JSAP04]. The combination of sensors and effectors is called a touchpoint, which provides an interface for communication with the autonomic manager [Sin06]. The ability to manage and control managed resources make them highly scalable [Mur04].

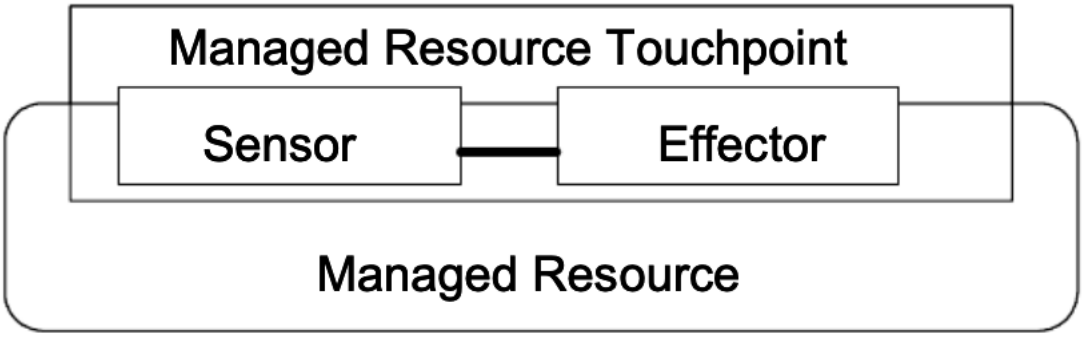

Figure 2.3: Managed resource - Source: Authors own model, based on [JSAP04].

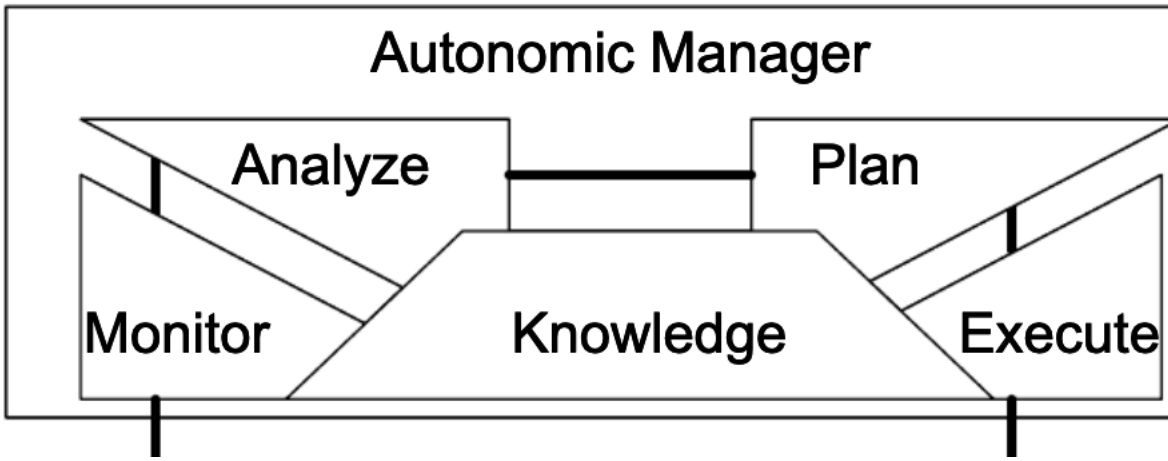

Figure 2.4: Autonomic manager - Source: Authors own model, based on [JSAP04].

2.2.3 Autonomic manager

The autonomic manager implements the control-loop to collect, aggregate, filter and report system metrics from the managed resources. It can only make adjustments within it‘s own scope and uses predefined policies to make decisions of what actions have to be executed to accommodate the goals and objectives [Mur04, Sin06]. In addition, the autonomic manager gains knowledge through analyzing the managed resources [Mur04]. The autonomic computing concept digests the MAPE-K model to implement an autonomic manager, as illustrated in Figure 2.4 [GBR11].

- **Monitor:** The monitor phase is responsible to collect the needed metrics from all managed resources and applies aggregation and filter operations to the collected data [Sin06].
- **Analyze:** The autonomic manager has to gain knowledge to determine if changes have to made to the environment [Sin06]. To predict future situations, the autonomic manager can model complex situation given the collected knowledge [JSAP04].
- **Plan:** Plans have to be structured to achieve defined goals and objectives. A plan consists of policy-based actions [JSAP04, Sin06].

- **Execute:** The execute phase applies all necessary changes to the computing system [Sin06].

Multiple autonomic manager can exist in an autonomic computing environment to perform only certain parts. For example, there can be one autonomic manager which is responsible to monitor and analyse the system and another autonomic manager to plan and execute. To create a complete and closed control-loop, multiple autonomic manager can be composed together [Sin06].

2.3 System Performance

2.3.1 Performance Metrics

Performance metrics are statistics that describe the system performance. These statistics are generated by the system, applications or other tools [Gre20]. The following are examples of common types for performance metrics:

- **Throughput:** Volume of data or operations per second [Gre20].
- **Latency:** Time of operation [Gre20].
- **Utilization:** Usage of a component [Gre20].

Measuring performance metrics can cause an overhead. To gather and store performance metrics, CPU cycles must be spent. This can have a negative affect on the target performance [Gre20].

2.3.2 Time-Based Utilization

Utilization is a performance metrics that describes the usage of a device, e.g. CPU device usage. A time-based utilization describes the usage of a component during a time period where the component was actively performing work [Gre20]. When a resource approaches 100% utilization, the performance of that resource can degrade. If a component can process operations in parallel, the performance does not have to degrade much at 100% utilization and the component can process more work [Gre20].

2.4 Monitoring

Monitoring is a process, that aims to detect and take care of system faults. In a dynamic environment, becoming aware of the system is trivial [Lig12]. A monitoring system consists of a set of different tools. The tools are responsible to perform measurements on components in the computing environment and collect, store and interpret the monitored data [Lig12].

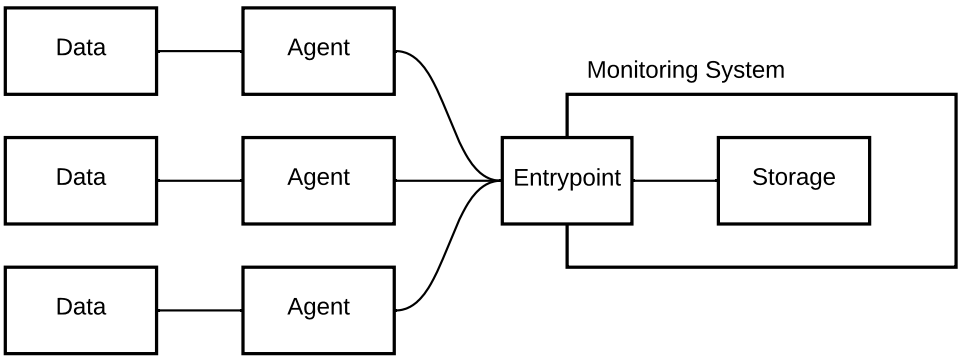

Figure 2.5: The monitoring process - Source: Authors own model.

In the monitoring process, illustrated in Figure 2.5, data is continuously collected by agents. An agent is a process that continuously gathers metrics. Data can be device statistics, logs or system measurements. Agents will group these data into metrics and submit them to the monitoring system via a protocol. The monitoring system will store the metrics in its database [Lig12].

The requirements for a monitoring system, that is able to monitor a dynamic changing environment, are the following:

- An efficient database to store metrics
- A push or pull based way of gathering metrics [Far17]
- Multi-dimensional metrics [Far17]
- A powerful query language [Far17]

2.4.1 Database

Saving continous data needs to be done efficiently. For time-based metrics, a time series database is he most effective way to save the data [Far17]. To store a huge amount of data, data is stored in a very compact format. A BISL MEHR ÜBER METRICS UND TIMESERIES.

2.4.2 Push and pull

The approach how the monitoring systems gathers metrics to store in the database plays a significant role. Push and pull based systems are the two primary approaches to gather metrics from services. Push based monitoring systems expect services to push metrics to their storage. On the other hand, pull based monitoring systems scrape metrics from all defined targets. Targets do not know about the existence of the monitoring system and only need to collect and expose metrics [Far17].

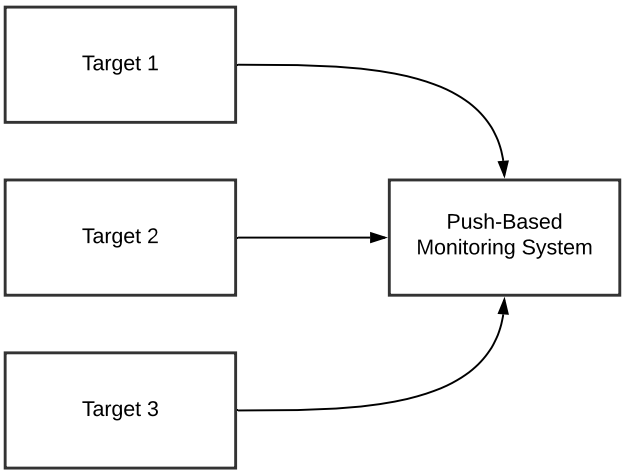

Figure 2.6: Push-based monitoring approach - Source: Authors own model.

Service discovery is an important aspect to decide whenever to use a pull or push based monitoring system [Far17].

In a push-based environment, services only need to know the address of the monitoring service to push their data to the storage [Far17].

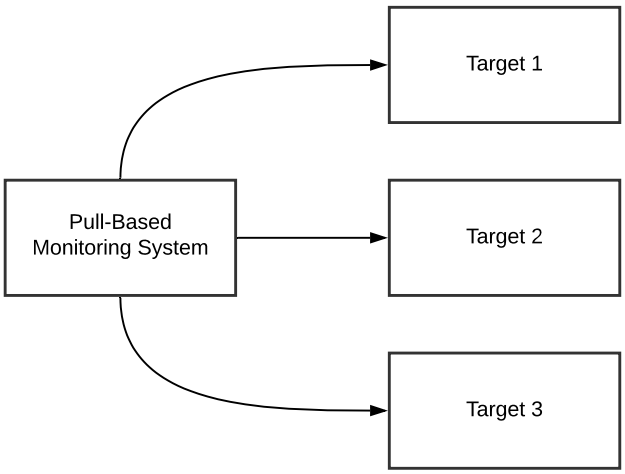

Figure 2.7: Pull-based monitoring approach - Source: Authors own model.

A pull-based monitoring tool needs to know the address of each target in the environment. The advantage of a pull-based monitoring systems is the simplicity to detect whenever a target has failed or is not available [Far17].

2.4.3 Multi-dimensional Metrics

For query languages to be effective, metrics need to be dimensional. Metric without dimensions, are limited in their capabilities. In a dynamic environment, services are dynamically added and removed. Therefore, a dynamic

environment needs dynamic analytics where metrics represent all dimension in the environment [Far18].

```
container_memory_usage
```

Listing 2.1: Example of a dimensionless-metric

```
container_memory_usage{service="my_service"}
```

Listing 2.2: Example of a metric with dimensions

As the exasmple Listing 2.1 and Listing 2.2 show, the metric with dimension provide more efficient querieing to gather informations about the environment.

2.4.4 Query Language

2.4.5 Choosing a Monitoring Tool

Graphite or Prometheus, Prometheus it its! <- Eher in Design

Chapter 3

Related Work

This chapter provides background information about ...

TODO: Describe Chapter

3.1 Elastic Environments

In recent years, container technologies have been used efficiently in complex IT environments. Dynamic scaling of containerized applications is an active area of research. The studied research can be divided in two parts.

3.1.1 Architecture

In the work by Lorido-Botrán et al. they reviewed state-of-the-art literatures about auto-scaling and proposed a process for auto-scaling homogeneous elastic applications. They mentioned three different problems, auto-scaler face while remaining the Quality of Service (QoS): Under-provisioning, over-provisioning and oscillation. Under-provisioning refers to, if not enough resources are available, over-provisioning means that more resources are available than needed and oscillation occurs when the environment gets scaled too quickly before the impact is clear. They mentioned the MAPE-Loop which consists of four different parts: Monitor, Analyze, Plan and Execute. The Auto-Scaler is part

3.1.2 Auto-Scaler

Srirama et al. [SAP20] designed a heuristic-based auto-scaling strategy for container-based microservices in a cloud environment. The purpose of the auto-scaling strategy was to balance the overall resource utilization across microservices in the environment. The proposed auto-scaling strategy performed better results than state-of-the-art algorithms in processing time, processing cost and resource utilization. The processing cost of microservices could be reduced by 12-20% and the CPU and memory utilization of cloud-servers have been maximized by 9-15% and 10-18%.

Lorido-Botrán et al. [LBMAL13] compared different representative auto-scaling techniques in a simulation in terms of cost and SLO violations. They compared load balancing with static threshold-based rules, reactive and proactive techniques based on CPU load. Load balancing is based on static rules defining the upper and lower thresholds of a specific load. For example *if CPU > 80% then scale-out; if CPU < 20% then scale-in.* The difficulty of this technique is to set the ideal rules. False rules can lead to bad performance. Proactive techniques try to predict the future values of performance metrics based on historical data. Reactive techniques are based on control theory to automate the systems management. In Addition, the authors proposed a new auto-scaling technique. To overcome the difficulties of static thresholds, the authors proposed a new auto-scaling technique using rules with dynamic thresholds. The results showed, that for auto-scaling techniques to scale well, it highly depends on parameter tuning. The best result was achieved with proactive results with a minimum threshold of 20% and a maximum threshold of 60%.

3.1.3 Auto-Scaling Algorithms

Barna et al. [BKFL17] proposed an autonomic scaling architecture approach for containerized microservices. Their approach focused on creating an autonomic management system, following the autonomic computing concept [KC03], using a self-tuning performance model. The demonstrated architecture frequently monitors the environment and gathers performance metrics from components. It has the ability to analyze the data and dynamically scale components. In addition, to determine if a scaling action is needed, they proposed the *Scaling Heat Algorithm.* The Scaling Heat Algorithm is used to prevent unnecessary scaling actions, which can throw the environment temporarily off.

Casalicchio et al. [CP17] focused on the difference of absolute and relative metrics for container-based auto-scaling algorithms. They analysed the mechanism of the *Kubernetes Horizontal Pod Auto-scaling* (KHPA) algorithm and proposed a new auto-scaling algorithm based on KHPA using absolute metrics called *KHPA-A*. The results showed, that KHPA-A can reduce response time between 0.5x and 0.66x compared to KHPA. In addition, their work proposed an architecture using cAdvisor for collecting container performance metrics, Prometheus for monitoring, alerting and storing time-series data and Grafana for visualizing metrics.

3.2 Heterogenous GPU aware Spark systems

Apache Spark is a computing framework that distributes tasks between CPU cores. Data and compute intensive applications profit from GPU acceleration. Therefore, various research projects took effort to bring GPU acceleration to Apache Spark.

Li et al. [PYNY15] developed a middleware framework called *HeteroSpark* to enable GPU acceleration on Apache Spark worker nodes. HeteroSpark listens for function calls in Spark applications and invokes the GPU kernel for acceleration. For communication between CPU and GPU, HeteroSpark uses the Java RMI[1] API to send data from the CPU JVM to the GPU JVM for execution. The design provides a plug-n-play approach and an API for the user to call functions with GPU support. Overall, HeteroSpark is able to achieve a 18x speed-up for various Machine Learning applications running on Apache Spark.

Klodjan et al. [HBK18] introduced HetSpark a heterogeneous modification of Apache Spark. HetSpark extends Apache Spark with two executors, a GPU accelerated executor and a commodity class. The GPU accelerated executor is based on VineTalk[MPK+17] for GPU acceleration. The authors observed, that for compute intensive tasks GPU accelerated executers are preferable while for linear tasks CPU-only accelerators should be used.

Yuan et al. [YSH+16] proposed SparkGPU to enable parallel processing with GPUs in Apache Spark and contributes to achieve high performance and high throughput in Apache Spark applications. SparkGPU extends Apache Sparks to determine the suitability of parallel-processing for a task to enable task scheduling between CPU and GPU. SparkGPU accomplished to improve the performance of machine learning algorithms up to 16.13x and SQL query execution performance up to 4.83x.

1 Java Remote Method Invocation

Chapter 4

Technical Background

In this chapter bla bla

TODO: Describe Chapter

4.1 Docker

Docker is an open-source platform that enables containerization of applications. Containerization is a technology to package, ship and run applications and their environment in individual containers. Docker is not a container technology itself, it hides the complexity of working with container technologies directly and instead provides an abstraction and the tools to work with containers [NK19, BMDM20, PNKM20].

TODO: Mehr zum Thema DevOps

4.1.1 Docker architecture

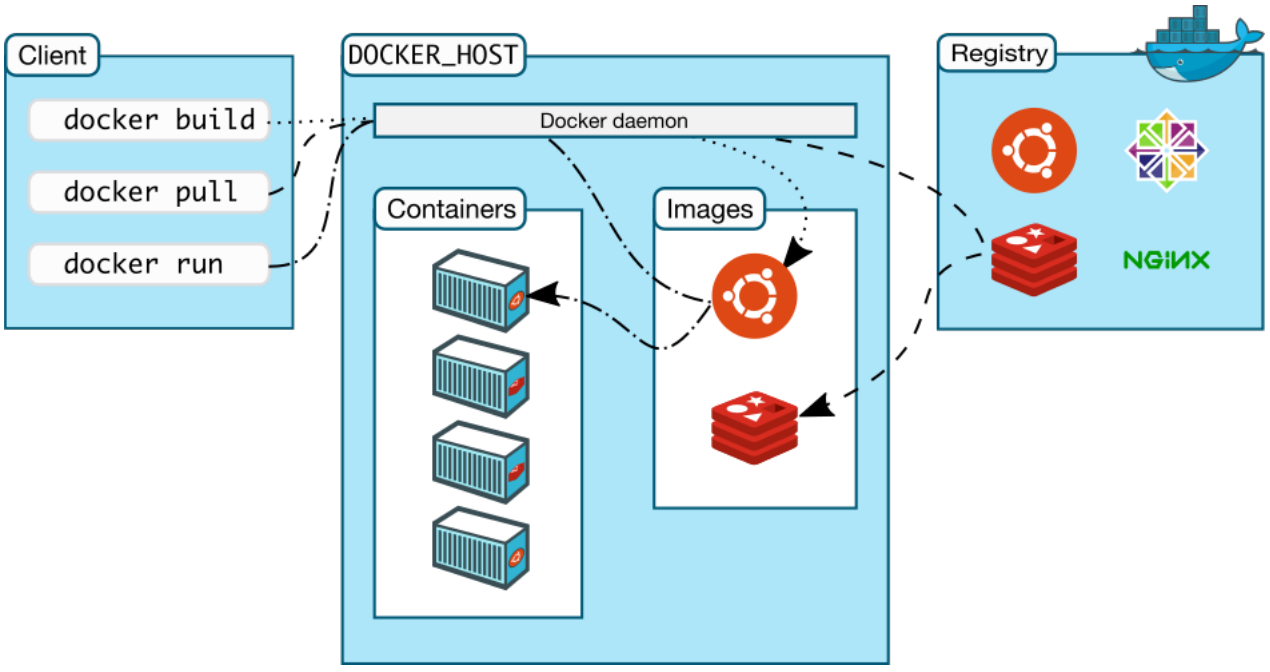

Figure 4.1: Docker architecture - Source: Authors own model, based on [Doc].

Figure 4.1 illustrates the client server architecture of Docker which consists of of a Docker client, the Docker deamon and a registry.

Docker client: The Docker client is an interface for the user to send commands to the Docker deamon [Doc].

Docker deamon: The Docker deamon manages all containers running on the host system and handles the containers resources, networks and volumes [BMDM20].

Docker Registry: A Docker registry stores images. Images can be pushed to a public or private registry and pulled from it to build a container [Doc].

4.1.2 Docker I mage

An Image is a snapshot of the environment that is needed to run an application in a Docker container. The environment consists of all files, libraries and configurations that are needed for the application to run properly [Doc, NK19]. Images can be created from existing containers or from executing a build script called Dockerfile. A Dockerfile is a text file consisting of instructions for building an image. The Docker image builder executes the instructions of a Dockerfile from top to bottom [NK19].

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y git

ENTRYPOINT ["git"]
```

Listing 4.1: Example of a Dockerfile

Listing 4.1 provides an example of a Dockerfile with three instructions.

1. `FROM ubuntu:latest` - This image is build on top of the latest Ubuntu image. Dockerfiles have to start with a `FROM` instruction [NK19].
2. `RUN apt-get update && apt-get install -y git` - Update the package manager and install Git.
3. `ENTRYPOINT ["git"]` - Set the git command as the entrypoint of this image.

TODO: Vielleicht noch erklären wie Images mit Docker build erstellt werden.

4.1.3 Docker Container

A container is an execution environment running on the host-system kernel.

The advantage of a container is its lightweight nature. As illustrated in Figure 4.2, containers take advantage of OS-level virtualization instead

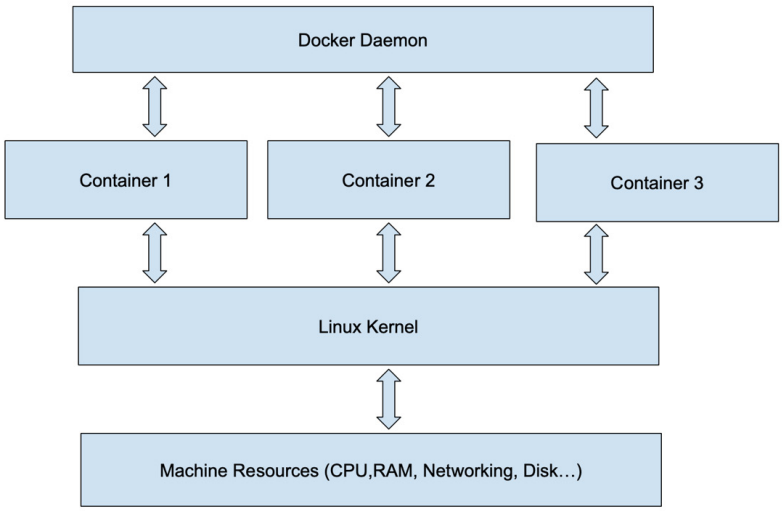

Figure 4.2: Docker basic container structure - Source: Authors own model, based on [BMDM20].

of hardware-virtualization without the need of a hypervisor [Doc, NK19]. Containers share the resources of the host-system instead of using reserved resources [BMDM20]. Multiple containers can run on the host-system kernel and are by default isolated from each other [Doc]. In Docker, a container is a runnable unit of an image and is used for distributing and testing applications. A container can be configured to expose certain resources to the host system, e.g. network ports [BMDM20].

4.1.4 Docker Swarm Mode

Docker Swarm mode is the native cluster orchestration and management tool embedded in the Docker engine. In Docker Swarm mode, a cluster of multiple nodes is called a swarm. All nodes run in Swarm mode and act as managers or workers. In a swarm, multiple services can be deployed. The manager node is responsible to maintain the desired state of a service [Doc].

Docker Swarm mode will be used to create a self-healing and a self-adapting environment as described in Section **??**. Many properties of Docker Swarm mode contribute the fact that it is an ideal candidate to create self-healing and self-adapting environment:

- **Desired state:** The manager node monitors the state of each service in the swarm and adapts the environment to maintain the desired state [Doc].
- **Cluster management and orchestration:** Docker Swarm mode is integrated with the Docker engine. A swarm can be created and managed using the Docker CLI [Doc].
- **Service model:** The Docker engine allows to define the desired state of a service. The manager node maintains the desired state of all services in the swarm [Doc].

- **Scaling:** The number of replicas can be defined for each service. The manager node will automatically adapt the number of replicas for a service to keep the desired state [Doc].
- **Multi-host networking:** A swarm runs all services in an overlay network. New services will automatically be added to the overlay network [Doc].

Nodes

A Docker engine participating in the swarm is called a node. Nodes can act as manager nodes, worker nodes or both [Doc].

The manager node is responsible for cluster orchestration an management. It maintains the desired state of all services and tasks in the swarm. In addition, the manager node dispatches tasks to worker nodes when service definitions will be submitted to the manager node [Doc].

Worker nodes are responsible to execute the tasks receives by the manager node. While performing the tasks, the worker node notifies the manager node about the tasks state [Doc].

Services and Tasks

A services defines the desired state of a task. The state is defined by the number of replicas of a service and the configuration for the Docker container, e.g. Docker Image, resources, network, and more [Doc].

A task is a running Docker container. The task is defined by the corresponding service and will be managed by the manager node. A task can be performed on worker and manager nodes [Doc].

4.2 Apache Spark

Apache Spark is an open-source computing framework for parallel data processing on a large computer cluster. Spark manages the available resources and distributes computation tasks across a cluster to perform big-data processing operations at large scale [CZ18]. Before Spark was developed, Hadoop MapReduce [DG10] was the framework of choice for parallel operations on a computer cluster [ZCF+10]. Spark accomplished to outperform Hadoop by 10x for iterative Machine Learning [ZCF+10]. It is implemented in Scala[1], a JVM-based language and provides a programming interface for Scala, Java[2], Python[3] and R[4]. In addition, Spark includes an interactive SQL shell and libraries to implement Machine Learning and streaming applications [CZ18].

1 Scala programming language. https://www.scala-lang.org/
2 Java programming language. https://www.oracle.com/java/
3 Python programming language. https://www.python.org/
4 R programming language. https://www.r-project.org/

It was developed in 2009 as the Spark research project at UC Berkeley and became an Apache Software Foundation project in 2013 [CZ18].

4.2.1 Spark programming model

Spark provides resilient distributed datasets (RDDs) as the main abstraction for parallel operations [ZCF+10]. Core types of Spark's higher-level structured API are built on top of RDDs [CZ18] and will automatically be optimized by Spark's Catalyst optimizer to run operations quick and efficient [Luu18].

TODO: Master-Slave Architektur + Bild

Resilient distributed datasets

Resilient distributed datasets are fault-tolerant, parallel data structures to enable data sharing across cluster applications [ZCD+12]. They allow to express different cluster programming models like MapReduce, SQL and batched stream processing [ZCD+12]. RDDs have been implemented in Spark and serve as the underlying data structure for higher level APIs (Spark structured API) [ZCD+12]. RDD's are a immutable, partitioned collection of records and can only be initiated through transformations (e.g. map, filter) on data or other RDD's. An advantage of RDDs is, that they can be recovered through lineage. Lost partitions of an RDD can be recomputed from other RDDs in parallel on different nodes [ZCD+12]. RDDs are lower level APIs and should only be used in applications if custom data partitioning is needed [CZ18]. It is recommended to use Sparks structured API objects instead. Optimizations for RDDs have to be implemented manually while Spark automatically optimize the execution for structured API operations [CZ18].

Spark structured API

Spark provides high level structured APIs for manipulating all kinds of data. The three distributed core types are Datasets, DataFrames and SQL Tables and Views [CZ18]. Datasets and DataFrames are immutable, lazy evaluated collections that provide execution plans for operations [CZ18]. SQL Tables and Views work the same way as DataFrames, except that SQL is used as the interface instead of using the DataFrame programming interface [CZ18]. Datasets use JVM types and are therefore only available for JVM based languages. DataFrames are Datasets of type Row, which is the Spark internal optimized format for computations. This has advantages over JVM types which comes with garbage collection and object instantiation [CZ18].

Spark Catalyst

Spark also provides a query optimizer engine called Spark Catalyst. Figure 4.3 illustrates how the Spark Catalyst optimizer automatically optimizes Spark applications to run quickly and efficient. Before executing the user's code, the Catalyst optimizer translates the data-processing logic into a

logical plan and optimizes the plan using heuristics [Luu18]. After that, the Catalyst optimizer converts the logical plan into a physical plan to create code that can be executed [Luu18].

Logical plans get created from a DataFrame or a SQL query. A logical plan represents the data-processing logic as a tree of operators and expressions where the Catalyst optimizer can apply sets of rule-based and cost-based optimizations [Luu18]. For example, the Catalyst can position a filter transformation in front of a join operation [Luu18].

From the logical plan, the Catalyst optimizer creates one ore more physical plans which consist of RDD operations [CZ18]. The cheapest physical will be generated into Java bytecode for execution across the cluster [Luu18].

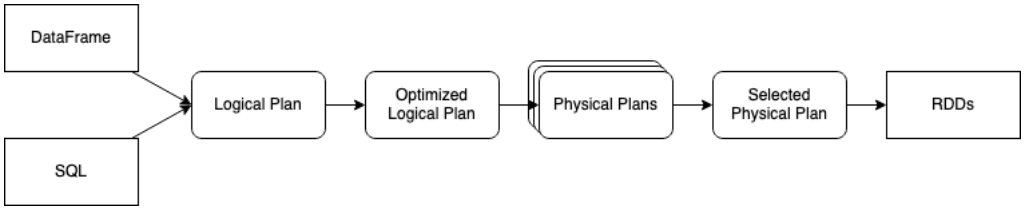

Figure 4.3: Optimization process of the Spark Catalyst - Source: Authors own model, based on [Luu18].

TODO: Bild nochmal machen mit abstand + QUELLE anpassen

4.2.2 Spark application architecture

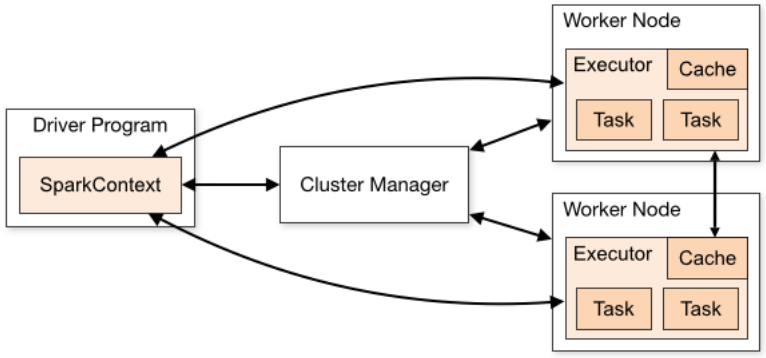

Figure 4.4: Overview of a Spark cluster architecture - Source: Authors own model, based on [Theb].

TODO: Lieber Doch Master/Slave ???. TODO: Nicht ganz richtig, Was ist mit dem CLuster-Manager, Master und Worker sind machines und driver und executor sind prozesse

Figure 4.4 illustrates the main architecture of a Spark cluster. The architecture follows the master-worker model where the Spark driver is the master and the Spark executors are the worker [Luu18].

Spark driver: The Spark driver is a JVM process on a physical machine and responsible to maintain the execution of a Spark application [CZ18]. It coordinates the application tasks onto each available executor [Luu18]. To get launch executors and get physical resources, the Spark driver interacts with the cluster manager [CZ18, Luu18].

Spark Executor: A Spark executor performs the tasks given by the Spark driver [CZ18]. It runs as a JVM process and runs until the Spark application finishes [Luu18]. After the executor finishes, it reports back to the Spark driver [CZ18]. Each task will be performed on a separate CPU core to enable parallel processing [Luu18].

Cluster manager: The cluster manager is an external service that orchestrates the work between the machines in the cluster [Luu18, Theb]. The cluster manager knows about the resources of each worker and decides on which machine the Spark driver process and the executor processes run [Luu18, CZ18]. Spark supports different services that can run as the cluster manager: Standalone, Apache Mesos[5], Hadoop YARN[VMD+13] and Kubernetes[6] [Theb]. The cluster manager provides three different deploy modes for acquiring resources in the cluster.

TODO: Explain standalone

- Cluster mode
- Client mode
- Local mode

To run an application in cluster mode, the user has to submit a precompiled JAR, python script or R script to the cluster manager [CZ18]. After that, the cluster manager starts the driver process and executor processes exclusively for the Spark application on machines inside the cluster [CZ18, Luu18].

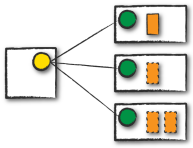

Figure 4.5: Spark's cluster mode - Source: Authors own model, based on [CZ18].

The difference between the client mode and the cluster mode is that, the driver process runs on the client machine outside of the Spark cluster [CZ18].

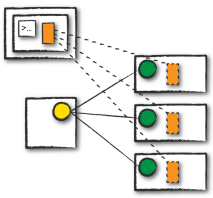

Figure 4.6: Spark's client mode - Source: Authors own model, based on [CZ18].

The local mode starts a Spark application on a single computer [CZ18]. It is not recommended to use the local mode in production, instead it should be used for testing Spark applications or learning the Spark framework [CZ18].

5 Apache Mesos. https://mesos.apache.org/
6 Kubernetes. https://kubernetes.io/

4.2.3 Spark application implementation

The concept of a Spark application consists of calling transformations and actions. A transformation creates a DataFrame or a Dataset, the logical data structures of a Spark application. The computation of a Spark application gets processed when an action gets called in the application. The transformations of a Spark application build up a directed acyclic graph (DAG) of instructions. By calling an action, the DAG will break down into stages and tasks to create a single job for execution [CZ18].

```
# Initialize a SparkSession
sparkSession = SparkSession\
    .builder\
    .getOrCreate()

# Create a dataframe with a transformation
dataframe = sparkSession.range(1, 1000)
# Apply another transformation
dataframe = dataframe.filter(dataframe.id % 2 == 0)
# Call an action
count = dataframe.count()
```

Listing 4.2: Example of a Python3 Spark application

Listing 4.2 demonstrates an example implementation of a Spark application. At first a SparkSession gets initialized. Each Spark application must include a SparkSession to initialize the application driver and executors [CZ18]. In Addition, the SparkSession provides an API for data-processing logic and configuration of the Spark application [Luu18]. After that, a DataFrame gets created with the range transformation to include each number from 1 to 1000 in the DataFrame. Next, a filter transformation is applied on the DataFrame to sort out any odd number. At the end, the number of rows gets saved in a variable with the count action.

TODO: Create listing macro
TODO: Tabelle von Luu18 vll als Anhang(Table 2-1 Subdirectories Inside the spark-2.1.1-bin-hadoop2.7 Directory)

The Spark framework provides a spark-submit executable to launch a Spark application inside a cluster.

```
$SPARK_HOME/bin/spark-submit \
    --master spark://spark-master:7077 \
    application.py
```

Listing 4.3: Execution of a Spark Python application using the spark-submit executable

Listing 4.3 provides an example how the spark-submit executable can be used to launch a Spark Python application.

4.2.4 Spark standalone cluster deployment

TODO: Why only standalone

The standalone mode is a basic cluster-manager build specifically for Spark. It is developed to only run Spark but supports workloads at large scale [CZ18].

Spark provides build-in scripts to start a master node and worker nodes in standalone mode. ABC demonstrates how a master node and worker node gets launched using the build-in scripts.

4.3 RAPIDS accelerator for Apache Spark

The RAPIDS accelerator for Apache Spark is a plugin suite to enable GPU acceleration for computing operations on Apache Spark 3.x [NVI]. To accelerate computing operations, it uses the RAPIDS[7] libraries and extends the Spark programming model (see Subsection 4.2.1) [NVI, McD20].

4.3.1 Extension of the Spark programming model

The plugin suite extends the Spark programming model with a new DataFrame based on Apache Arrow[Thea] data structures and the Catalyst optimizer to generate GPU-aware query plans [McD20].

Apache arrow is a data platform to build high performance applications that work with large dataset‘s and to improve analytic algorithms. A component of Apache Arrow is the Arrow Columnar Format, an in-memory data structure specification for efficient analytic operations on GPUs and CPUs [Thea].

Spark‘s DataFrame and SQL use the RAPIDS APIs to run tranformations and actions on a GPU. The Spark Catalyst optimizer identifies operator in a query plan that are supported by the RAPIDS APIs. To execute the query plan, these operators can be scheduled on a GPU within the Spark cluster [McD20]. If operators are not supported by the RAPIDS APIs, a physical plan for CPUs will be generated by the Catalyst optimizer to execute RDD operations [McD20]. Figure 4.7 illustrates, how a query plan gets optimized with the RAPIDS accelerator for Spark enabled.

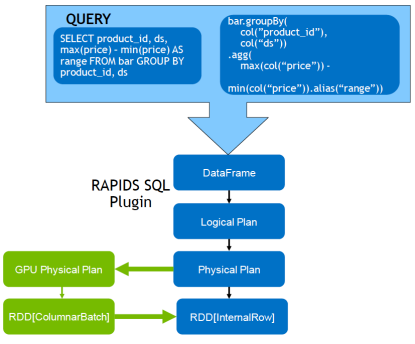

Figure 4.7: Catalyst optimization with RAPIDS accelerator for Apache Spark - Source: Authors own model, based on [McD20].

7 Open GPU Data Science - https://rapids.ai/

4.4 Prometheus

Prometheus is an open-source monitoring and alerting system [Thec]. To collect and store data, Prometheus supports a multi-dimensional key-value pair based data model, according to Subsection 2.4.3, which can be analyzed in real-time using the PromQL query language [SP20]. Prometheus follows the pull-based approach, as described in detail in Subsection 2.4.2, to scrape metrics from hosts and services [BP19].

4.4.1 Prometheus architecture

TODO: Bilder für components

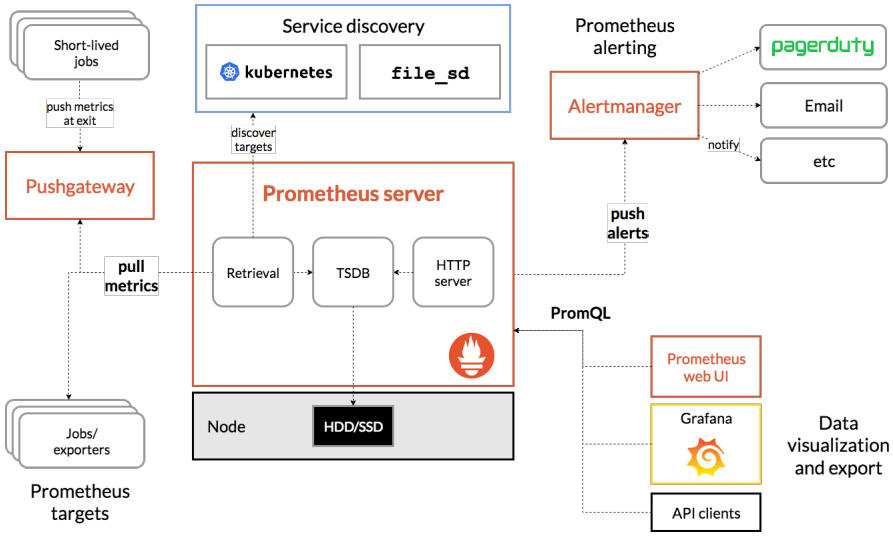

Figure 4.8: Prometheus high-level architecture - Source: Authors own model, based on [Thec, BP19].

Figure 4.8 illustrates the high-level architecture of Prometheus. The Prometheus ecosystem provides multiple components. Components can be optional, depending on the monitoring needs of the environment [BP19]. The main components of a Prometheus system are Prometheus server, Alertmanager, service discovery, exporters, Pushgateway and visualization tools [Thec].

Prometheus server: The Prometheus server is the main component of a Prometheus system. It is responsible to collect metrics as time-series data from targets and stores the collected data in the built-in TSDB [BP19]. Prometheus uses the concept of scraping to collect metrics from a target. A target host has to expose an endpoint to make metrics available in the Prometheus data format [SP20]. Additionally, the Prometheus server triggers alerts to the Alertmanager if a configured condition becomes true [Thec].

Alertmanager: If an alerting rule becomes true, the Prometheus server generates an alert and pushes it to the Alertmanager. The Alertmanager generates notifications from the received alerts. A notification can take

multiple forms like emails or chat messages. Webhooks can be implemented to trigger custom notifications [BP19].

Service discovery: As mentioned before, Prometheus follows a pull-based approach to fetch metrics from a target. To know about all targets, Prometheus needs a list of the corresponding hosts. The service discovery manages the complexity of maintaining a list of hosts manually in an changing infrastructure [BP19].

Exporters: If an application does not support an endpoint for Prometheus, an exporter can be used to fetch metrics and make them available to the Prometheus server. An exporter is a monitoring agent running on a target host that fetches metric from the host and exports them to the Prometheus server [SP20].

Pushgateway: If a target is not designed to be scraped, metrics can be pushed against the Pushgateway[Thec]. The Pushgateway converts the data into the Prometheus data format and passes them to the Prometheus server [SP20].

Visualization: Prometheus supports various tools for virtualization of the scraped data. Grafana[8] is one of the widely used tools for this occasion.

4.5 cAdvisor

Container Advisor (cAdvisor) is a running deamon that collects, aggregates, analyses and exposes performance metrics from running containers. It has native support for Docker container and is deployed as a Docker container. cAdvisor collects metrics from the container deamon and Linux cgroups. Collected metrics will be exposed in the Prometheus file format.

4.6 GitLab CI/CD

GitLab CI/CD is a tool integrated into the GitLab platform that enables Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CD) for software development. The GitLab platform integrates many development features like Git repository management and CI/CD. By pushing code changes to the codebase, GitLab CI/CD executes a pipeline of scripts to automate CI and CD processes of the software development cycle. A CI pipeline will consist of scripts that builds, tests and validate the updated codebase. A CD pipeline is responsible to deploy the application for production after the CI pipeline has executed successfully. Adding CI/CD

8 Grafana: The open observability platform - https://grafana.com/

pipelines to the software development cycle of an application, allows to catch bugs an errors early. This ensures that an application deployed to production will conform to established standards [Git].

4.6.1 CI/CD Pipeline

The fundamental component of GitLab CI/CD is called a pipeline. Pipelines will perform based on conditions. A conditions might be a push to the main branch of the repository [Git]. A pipeline comprises two components:

- **Stages:** A stage consists of one or multiple jobs that run in parallel. Furthermore, a stage defines how jobs will be executed. For example, a build stage only performs after a test stage has performed successfully [Git].
- **Jobs:** Jobs are responsible to perform the scripts defined by administrators. The scripts define necessary actions. For example compiling the source code or performing tests [Git].

Configuration

GitLab CI/CD is configured by a *.gitlab-ci.yml* file. It is necessary that this file is located in the repositories root directory. The configuration file will create a pipeline that performs after a push to the codebase [Git].

Basic Architecture

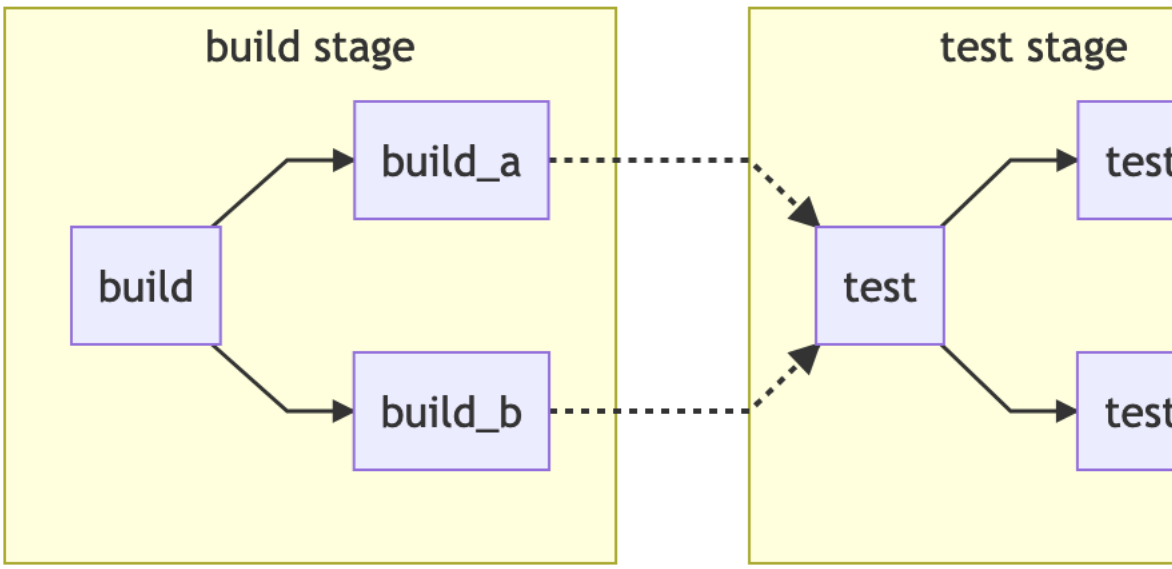

Figure 4.9: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

4.6.2 Job Execution

Jobs which are defined in the configuration file will be performed by GitLab runners. A GitLab runner is an agent that performs the jobs in its own environment and responds the result back to the GitLab instance. A runner is a lightweight and highly scalable application that runs on a server and performs one or multiple executors. An executor provides the environment where jobs will be executed in. GitLab runner provides multiple variants of executors. For example the Docker executor that connects to the underlying Docker engine. In addition, the Docker executor performs a job in a separate and isolated Docker container. GitLab runner can be set up only for specific projects or be available to all project on the GitLab platform [Git].

4.7 K-MEANS

4.8 Naive Bayes Classifier

4.9 Scaling Heat

4.10 KHP

Chapter 5

Conceptual Design

TODO: Describe Chapter

5.1 Design Restrictions

The design of the computing environment will be restricted by several points. These factors are given because of technological choices and requirements from up there.

- **Running on a NVIDIA DGX A100[1]:** The environment will run on a NVIDIA DGX A100 workstation. The workstation has 80 CPUs and 8 GPUs installed. For this computing environment, 2 GPUs will be available.
- **Apache Spark for distributed computing:** Apache Spark will be used as a distributed computing framework.
- **Python as the main programming language:** Python will be used as the main programming language for Apache Spark applications. Therefore, examples will use Python code. Configurations for the system are optimized for using Python in production.

5.2 CI/CD

5.3 Identification of suitable Metrics for Scaling

To analyze the Apache Spark cluster computing performance suitable metrics have to be defined. As mention in SECTION XY, Apache Spark distributes its workload across multiple CPU cores. In addition, one objective of this thesis is to accelerate the computing performance of the Apache Spark cluster with GPUs. Therefore suitable performance metrics are CPU utilization of all Apache Spark worker and the utilization of all available GPUs. These

1 The Universal System for AI Infrastructure - `https://www.nvidia.com/en-us/data-center/dgx-a100/`

utilization metrics will be based on the time, when the Apche Spark cluster is actively performing computations.

5.3.1 CPU Performance

TODO: Hier auf Tabelle verweisen (Anhang) Metriken die von Prometheus + cAdvisor bereitgestellt werden

To adapt to business needs, the CPU percentage of each Spark Worker will be calculated. Prometheus provides several metrics to calculate the CPU percentage. The CPU percentage of all Worker can be calculated as follows:

Each Apache Spark worker has a number of CPU cores to perform work. To calculate the CPU utilization of an Apache Spark worker,

$$SparkWorkerCPUUtilization = \frac{\sum CPUCoreUtilization}{NumberOfCPUCores} \tag{5.1}$$

$$OverallCPUUtilization = \frac{\sum SparkWorkerCPUUtilization}{NumberOfSparkWorker} \tag{5.2}$$

5.3.2 GPU Performance

The system has a fixed number of GPUs to use.

$$GPUUtilization = \frac{\sum SparkWorkerGPUUtilization}{NumberOfSparkWorker} \tag{5.3}$$

5.4 Computing environment Architecture

5.4.1 Overall

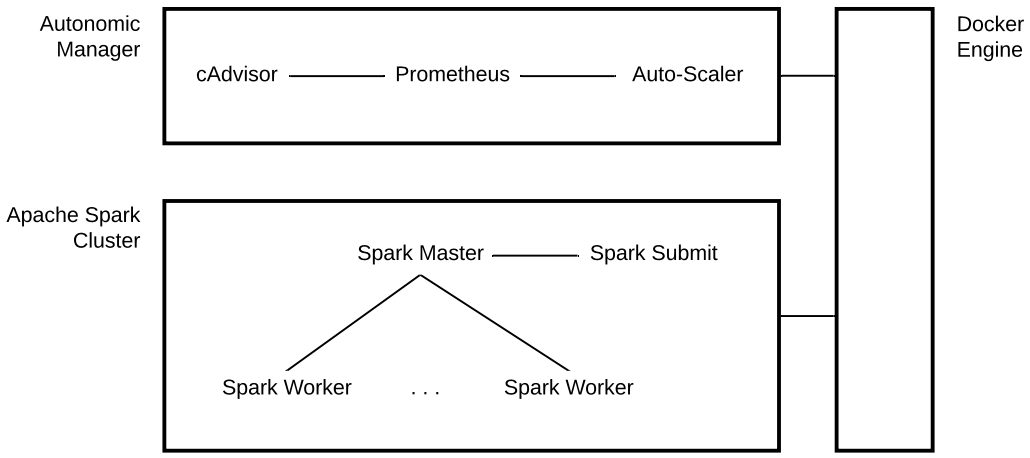

Figure 5.1: Overall cluster architecture - Source: Authors own model.

Figure 5.1 illustrates the overall architecture with all components of the computing environment. The two main components in the environment are an Apache Spark cluster and an autonomic manager. The autonomic manager will be implemented according to the MAPE architecture (introduced in Subsection 2.2.3). It is responsible for monitoring and auto-scaling the Apache Spark cluster. To enable distributed computing, an Apache Spark cluster will be set up to execute machine learning applications. The autonomic manager and the Apache Spark cluster consist each of multiple nodes. Each node of a component will run as a Docker container. All containers will run as services in Docker Swarm mode. As described in Subsection 4.1.4, Docker Swarm mode will maintain a healthy state of all running containers. Therefore, the environment will enable self-healing according to the requirements of Autonomic Computing described in Section 2.2.

5.4.2 Apache Spark Cluster

Master and Worker

The Apache Spark cluster will consist of a Spark master node and a dynamic number of Spark worker nodes. The Spark master node is responsible to distribute the application workload across available Spark worker nodes. A Spark worker node will execute the workload given by the master node. Each Spark worker is homogeneous. Homegenneosity is important to scale the number of worker nodes. To enable homogeneous nodes, each SPark worker node is a Docker container running the same Docker image. In addition, each worker is given the same computing resources. With homogeneous Spark worker nodes, each worker will respond as all other nodes. To enable GPU acceleration, each WORKER/MASTER will have the RAPIDS plugin installed. The cluster will be deployed in standalone mode. To be able to run Python applications

Spark Submit

Because the Apache Spark cluster ill be executed in standalone mode, a node inside the cluster is required to run Spark applications. When a Spark application will be executed, a Spark Submit container will be deployed in the cluster. When the appliccation has finished, the container will be automatically removed. Each app will be executed by a unique Spark Submit node. The Spark Submit node will be deployed via the CI pipeline (SECTION XY). The prupose of this

5.4.3 Autonomic Manager

The design of the autonomic manager needs to fullfill all requirements given by the MAPE architecture (described in Subsection 2.2.3). It will be responsible to monitor the performance of Apache Spark worker nodes in the environment, analyze the metrics and plan and execute scaling actions in

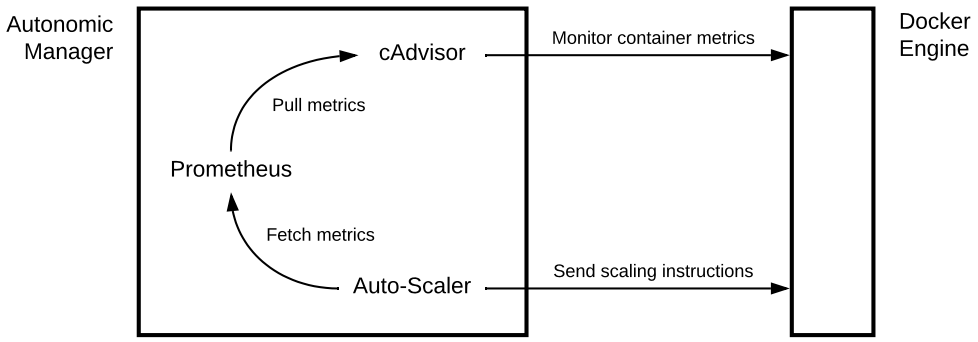

Figure 5.2: Autonomic manager component design - Source: Authors own model.

accordance to fullfill the performance goals. As illustrated in Figure 5.2, the autonomic manager consists of a cAdvisor, Prometheus and Auto-Scaler node. Together, all three nodes build a complete autonomic manager in accordance to the MAPE architecture. Each node will run as a Docker container. For monitoring, cAdvisor collects metrics from all available Docker container in the computing environment. Prometheus pulls metrics from cAdvsior. The Auto-Scaler will fetch metrics from Prometheus and send scaling instructions to the Docker engine.

Monitoring System

The design introduced in Subsection 5.4.1 is a dynamic changing computing environment. To monitor this dynamic environment, a monitoring-system is needed that fullfills the requirements described in Section 2.4.

cAdvisors will be used as an agent to collect performance metrics from all running Docker containers in the environment. Prometheus pulls the collected performance metrics from cAdvisor and stores the data as time-series data in its database. In addition, Prometheus provides a powerful multi-dimensional query language to aggregate and analyze the stored data.

Workflow

The workflow of the autonomic manager is implemented as a loop.

Figure 5.3 illustrates all steps of each component of the autonomic manager process according to the MAPE architecture.

The workflow starts with collecting metrics in the monitor phase. cAdvisor is responsible to collect metrics from the Docker engine. After that, Prometheus stores the collected metrics as time-series data in its database. Next, in the analyze phase, the Auto-Scaler needs to determine if a scaling action is needed. If scaling the Apache Spark worker nodes is not needed, the process has finished and will repeat from the beginning in the next period. If the performance is over- or under-utilized, a scaling action is needed. Then, the Auto-Scaler needs to determine how many Apache Spark worker

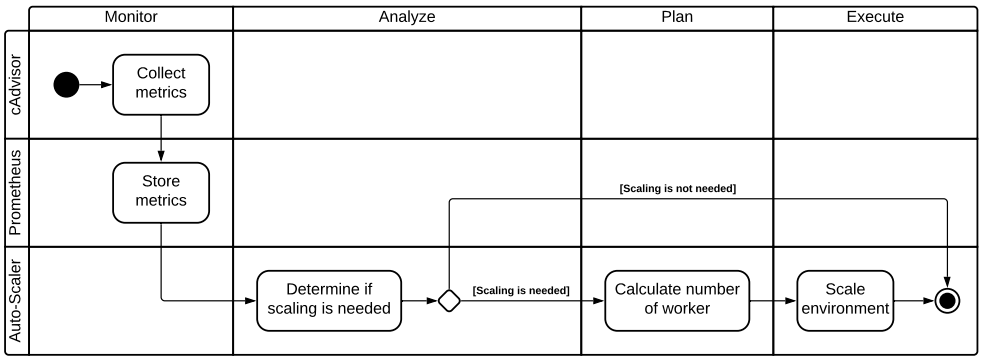

Figure 5.3: UML activity model of the autonomic manager process - Source: Authors own model.

nodes are needed to reach the performance goal. Lastly, the Auto-Scaler is responsible to send the scaling instructions to the Docker engine.

5.5 Auto-Scaler

The Auto-Scaler is a component of the the Autonomic Manager and is responsible for the Analyze, Plan and Execution phase. Together with cAdvisor and Prometheus, the Auto-Scaler builds a complete Autonomic Manager according to the architecture demonstrated in Subsection **??**. In addition, the Auto-Scaler implements the control-loop which is responsible to make adjustments in the environment. Since Prometheus collects and stores available metrics from cAdvisor, the Auto-Scaler has not to communicate with cAdvisor.

5.5.1 Configuration

The Auto-Scaler needs specific configuration properties to be able to collect the correct metrics from Prometheus and deploy new Apache Spark worker container in the environment. The following are properties that have to be defined to ensure that the Auto-Scaler is able to collect meaningful metrics and scale Apache Spark worker as expected.

General properties

- **Interval seconds:** The number of seconds when the loop has to repeat needs to be defined.
- **Cooldown period:** The duration in seconds, the Auto-Scaler has to wait after a scaling action was performed.
- **Recurrence factor:** To prevent to many scaling actions, the autonomic manager should only execute a scaling action, if the utilization thresholds is violated n times.

- **Prometheus URL:** The Auto-Scaler will fetch the configured metrics from the Prometheus REST API.

Metrics

To support to analyze multiple metrics, the user should be able to create a dynamic list if metrics. Each metric needs to have a variety of properties configured.

- **Target utilization:** The relative target utilization of a metrics needs to be defined to calculate the number of Spark worker to add or to remove to reach the defined goal.
- **Utilization thresholds:** To determine if a scaling action is needed, the scaling heat algorithm needs the minimum and maximum utilization defined by an administrator.
- **Query:** A PromQL query needs to be defined to collect the metric for all Spark Worker.

Apache Spark worker properties

- **Worker image:** To guarantee that each Spark worker is homogeneous, all worker container should be created with the same image.
- **Worker network:** To establish communication between all Spark worker and the Spark master, all new Spark worker container should be in the same network.
- **Worker thresholds:** The minimum and maximum number of concurrent Spark worker should be defined. To avoid the cold start effect, the minimum amount of worker should be 1.
- **Apache Spark master URI:** To distribute the workload across all Spark Worker, all Spark Worker need to communicate with the Spark master.

5.5.2 Analyze

In order to determine if a scaling-action is necessary, the Auto-Scaler has to process the collected metrics. During each period, the Auto-scaler queries the Prometheus time-series database with the configured queries to get all needed metrics. After the metrics are received, the Auto-Scaler determines if a scaling action is needed using the Scaling Heat algorithm (introduced in Section AB). If scaling is not necessary, the Auto-Scaler continues to collect metrics from Prometheus.

5.5.3 Plan

If a scaling-action is necessary, the Auto-Scaler is responsible to plan how to scale the number of Spark worker to satisfy the defined utilization goals. A scaling plan consists of instructions to add or remove Spark worker which will be send to the Docker engine. To calculate the number of Spark worker, needed to accomplish the defined target utilization, the Auto-Scaler uses the *Kubernetes Horizontal Pod Auto-Scaling* algorithm. In addition, the Auto-Scaler needs to check if the estimated number of Spark worker fall bellow the minimum threshold or exceed the maximum threshold of concurrent Spark worker.

5.5.4 Execute

After a scaling plan has been created, the Auto-Scaler needs to send the instructions to the Docker engine. After scaling the environment, it needs time for changes to take effect. Therefore a cooldown period will be activated after each scaling action. During the cooldown period, no scaling actions will be forwarded to the Docker engine.

Chapter 6

Implementation

6.1 General

TODO: Describe Chapter

T create an elastic computing environment, all needed components will be run as Docker container in a Docker network.

6.2 Computing environment

To create an elastic environment Docker will be used. With docker Spark worker can be easy scaled. With docker-compose an environment can be created.

6.2.1 Monitoring

cAdvisor will be uses to monitor metrics from the Docker engine. Prometheus collects the metrics defined in Section XY from cAdvisor and stores them in its time-series database. The query used for getting CPU metrics:

Listing 6.1: Python example

```
SUM(SELECT BLA FROM XYZ)
```

The query used for getting GPU metrics:

6.2.2 Apache Spark Images

The are 4 different Spark images:

1. Spark-Base 2. Spark-Master 3. Spark-Worker 4. Spark-Submit

Spark-Base is the foundation image for all other Spark images. Spark-Master is the image for the master node. Spark-Worker is the image for all SPark worker nodes. With Spark-Submit, a application will be submitted, the container runs as long as the application runs and exits after the execution automatically. Spark-Submit is used, because Cluster runs in Standalaone mode. Python cann submit cannot run in cluster mode QUELLE SPark.

6.2.3 GPU Acceleration

NVIDIA RAPIDS will be used as the plugin for GPU acceleration.

6.3 Auto-Scaler

The Auto-Scaler consists of a Python module which implements the logic for control-loop and a own Docker image. The Auto-Scaler is implemented in the Python programming language. The Auto-Scaler module will be executed in an individual Docker image.

6.3.1 Configuration

All configuration properties defined in the concept of the Auto-Scaler in SECTION A will be defined in a YAML file. To set the configuration properties, the configuration file needs to be set as a command-line argument.

6.3.2 Control-Loop

Control-Loop of the Auto-Scaler implements the Analyze, Plan, and Execute phase of the MAPE architecture.

Analyze

Scaling Heat algorithm will be used to estimate if a scaling action is necessary.

Plan

The KHPA algorithm will be used to calculate how many worker are needed to reach the target utilization. Since we have two different metrics, the highest number of running worker will be used KUBERNETES QUELLE.

Execute

Via the Python Docker SDK new container will be spawned in the network. If worker need to be removed, it is necessary to check if the worker are running any applications at the moment. Spark provides a REST API to check this. After a scaling action has been performed, a cooldown period will be applied so all nodes can relax.

6.4 Automated Deployment

Hier die CI/CD von gitlab.

Chapter 7

Evaluation

TODO: Describe Chapter

7.1 Test Environment

About DGX how powerful is the DGX ...

7.2 Datasets

Description of used datasets

Chapter 8

Outlook

TODO: Describe Chapter

8.1 Optimizing Scaling

Current approach: Wait until all applications have finished. Better approach: Blacklist worker for removing so no executor will be launchend on the worker. Create an external shuffle service, so worker can be remove on runtime.

8.2 Reinforcement Learning for Auto-Scaling

bla bla

Chapter 9

Conclusion

9.1 Cluster architecture

TODO: Describe Chapter

Bibliography

[BKFL17] Barna, C. ; Khazaei, H. ; Fokaefs, M. ; Litoiu, M.: Delivering Elastic Containerized Cloud Applications to Enable DevOps. In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2017, S. 65–75

[BMDM20] Bullington-McGuire, Richard ; Dennis, Andrew K. ; Michael, Schwartz: *Docker for Developers*. Packt Publishing, 2020. – ISBN 9781789536058

[BP19] Bastos, Joel ; Pedro, Araujo: *Hands-On Infrastructure Monitoring with Prometheus*. Packt Publishing, 2019. – ISBN 9781789612349

[CP17] Casalicchio, Emiliano ; Perciballi, Vanessa: Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics, 2017

[CZ18] Chambers, Bill ; Zaharia, Matei: *Spark: The Definitive Guide*. 1st. O'Reilly Media, 2018. – ISBN 9781491912218

[DG04] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, S. 137–150

[DG10] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: A Flexible Data Processing Tool. In: *Commun. ACM* 53 (2010), Januar, Nr. 1, 72–77. `http://dx.doi.org/10.1145/1629175.1629198`. – DOI 10.1145/1629175.1629198. – ISSN 0001–0782

[Doc] Docker Inc.: *Docker Documentation*. `https://docs.docker.com/`. – Accessed: 2020-12-06

[Far17] Farcic, Viktor: *The DevOps 2.1 Toolkit: Docker Swarm*. Packt Publishing, 2017. – ISBN 9781787289703

[Far18] Farcic, Viktor: *The DevOps 2.2 Toolkit*. Packt Publishing, 2018. – ISBN 9781788991278

[GBR11] Goscinski, Andrzej ; Broberg, James ; Rajkumar, Buyya: *Cloud Computing: Principles and Paradigms*. Wiley, 2011. – ISBN 9780470887998

[Git] GitLab Inc.: *GitLab User Documentation*. `https://docs.gitlab.com/`. – Accessed: 2020-12-19

[GOKB16] Ganelin, Ilya ; Orhian, Ema ; Kai, Sasaki ; Brennon, York: *Spark*. Wiley, 2016. – ISBN 9781119254010

[Gre20] Gregg, Brendan: *Systems Performance, 2nd Edition*. Pearson, 2020. – ISBN 9780136821694

[HBK18] Hidri, Klodjan K. ; Bilas, Angelos ; Kozanitis, Christos: HetSpark: A Framework that Provides Heterogeneous Executors to Apache Spark. In: *Procedia Computer Science* 136 (2018), S. 118 – 127. `http://dx.doi.org/https://doi.org/10.1016/j.procs.2018.08.244`. – DOI https://doi.org/10.1016/j.procs.2018.08.244. – ISSN 1877–0509. – 7th International Young Scientists Conference on Computational Science, YSC2018, 02-06 July2018, Heraklion, Greece

[JSAP04] Jacob, Bart ; Sudipto, Basu ; Amit, Tuli ; Patricia, Witten: *A First Look at Solution Installation for Autonomic Computing*. IBM Redbooks, 2004

[KC03] Kephart, J. O. ; Chess, D. M.: The vision of autonomic computing. In: *Computer* 36 (2003), Nr. 1, S. 41–50

[KK18] Kane, Sean P. ; Karl, Matthias: *Docker: Up & Running*. O'Reilly Media, Inc., 2018. – ISBN 9781492036739

[LBMAL13] Lorido-Botrán, Tania ; Miguel-Alonso, Jose ; Lozano, Jose: Comparison of Auto-scaling Techniques for Cloud Environments, 2013

[Lig12] Ligus, Slawek: *Effective Monitoring and Alerting*. O'Reilly Media, Inc., 2012. – ISBN 9781449333522

[LT11] L., Abbott M. ; T., Fisher M.: *Scalability Rules: 50 Principles for Scaling Web Sites*. Addison-Wesley Professional, 2011. – ISBN 9780132614016

[LT15] L., Abbott M. ; T., Fisher M.: *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition*. Addison-Wesley Professional, 2015. – ISBN 9780134031408

[Luu18] Luu, Hien: *Beginning Apache Spark 2: With Resilient Distributed Datasets, Spark SQL, Structured Streaming and Spark Machine Learning library*. 1st. Apress, 2018. – ISBN 9781484235799

[McD20] McDonald, Carol: *ACCELERATING APACHE SPARK 3.X*. 2020

[MPK+17] Mavridis, Stelios ; Pavlidakis, Manos ; Kozanitis, Christos ; Chysos, Nikos ; Stamoulias, Ioannis ; Kachris, Christoforos ; Soudris, Dimitrios ; Bilas, Angelos: VineTalk: Simplifying Software Access and Sharing of FPGAs in Datacenters. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL 2017)* (2017). – ISSN 1946–147X

[Mur04] Murch, Richard: *Autonomic Computing*. IBM Press, 2004. – ISBN 9780131440258

[NK19] Nickoloff, Jeffrey ; Kuenzli, Stephen: *Docker in Action*. Manning Publications, 2019. – ISBN 9781617294761

[NVI] NVIDIA Corporation: *Spark-Rapids Online Documentation*. `https://nvidia.github.io/spark-rapids/`. – Accessed: 2020-12-04

[PNKM20] Potdar, Amit ; Narayan, DG ; Kengond, Shivaraj ; Mulla, Mohammed: Performance Evaluation of Docker Container and Virtual Machine. In: *Procedia Computer Science* 171 (2020), 01, S. 1419–1428. `http://dx.doi.org/10.1016/j.procs.2020.04.152`. – DOI 10.1016/j.procs.2020.04.152

[PYNY15] Peilong Li ; Yan Luo ; Ning Zhang ; Yu Cao: HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. In: *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2015, S. 347–348

[SAP20] Srirama, Satish N. ; Adhikari, Mainak ; Paul, Souvik: Application deployment using containers with auto-scaling for microservices in cloud environment. In: *Journal of Network and Computer Applications* 160 (2020), 102629. `http://dx.doi.org/https://doi.org/10.1016/j.jnca.2020.102629`. – DOI https://doi.org/10.1016/j.jnca.2020.102629. – ISSN 1084–8045

[Sin06] Sinreich, D.: An architectural blueprint for autonomic computing, 2006

[SP20] Sabharwal, Navin ; Pandey, Piyush: *Monitoring Microservices and Containerized Applications: Deployment, Configuration, and Best Practices for Prometheus and Alert Manager*. Apress, 2020. – ISBN 9781484262160

[Thea] The Apache Foundation: *Arrow. A cross-language development platform for in-memory data*. `https://arrow.apache.org/`. – Accessed: 2020-12-03

[Theb] The Apache Software Foundation: *Spark 3.0.1 Documentation*. https://spark.apache.org/docs/3.0.1/. – Accessed: 2020-09-12

[Thec] The Linux Foundation: *Prometheus Online Documentation*. https://prometheus.io/docs/. – Accessed: 2020-12-03

[Tur16] Turnbull, James: *The Art of Monitoring*. Turnbull Press, 2016

[Vad18] Vadapalli, Sricharan: *DevOps: Continuous Delivery, Integration, and Deployment with DevOps*. Packt Publishing, 2018. – ISBN 9781789132991

[VMD+13] Vavilapalli, Vinod K. ; Murthy, Arun C. ; Douglas, Chris ; Agarwal, Sharad ; Konar, Mahadev ; Evans, Robert ; Graves, Thomas ; Lowe, Jason ; Shah, Hitesh ; Seth, Siddharth ; Saha, Bikas ; Curino, Carlo ; O'Malley, Owen ; Radia, Sanjay ; Reed, Benjamin ; Baldeschwieler, Eric: Apache Hadoop YARN: Yet Another Resource Negotiator. New York, NY, USA : Association for Computing Machinery, 2013 (SOCC '13). – ISBN 9781450324281

[Wil12] Wilder, Bill: *Cloud Architecture Patterns*. O'Reilly Media, Inc., 2012. – ISBN 9781449319779

[YSH+16] Yuan, Y. ; Salmi, M. F. ; Huai, Y. ; Wang, K. ; Lee, R. ; Zhang, X.: Spark-GPU: An accelerated in-memory data processing engine on clusters. In: *2016 IEEE International Conference on Big Data (Big Data)*, 2016, S. 273–283

[ZCD+12] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauly, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA : USENIX Association, April 2012. – ISBN 978–931971–92–8, 15–28

[ZCF+10] Zaharia, Matei ; Chowdhury, Mosharaf ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USA : USENIX Association, 2010 (HotCloud'10), S. 10