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, November 19, 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.

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
$\blacksquare$	Ende eines Beispiels
$\square$	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
$\mathcal{N}(\underline{x}; \hat{\underline{x}}, \mathbf{C}_x)$	Gaußdichte, d. h. Dichtefunktion eines normalverteilten Zufallsvektors $\underline{\boldsymbol{x}}$ mit Mittelwertvektor $\hat{\underline{x}}$ und Kovarianzmatrix $\mathbf{C}_x$

Chapter 1

Introduction

Storing huge amounts of data has become inexpensive in recent years, but processing it, requires parallel computations on clusters with multiple machines [CZ18]. Complex computation operations rely on a IT infrastructure with the ability to perform operations on scale.

TODO: EVTL DATUM AUF NOVEMBER ÄNDERN; WEGEN STARTDATUM!! in den online quellen

In order to achieve high scalability, computing systems need to adapt dynamically to demands and conditions of the workload.

1.1 Problem statement

ETL [1] operations are compute-intensive. During the execution of analytic applications, performance thresholds can be reached and the computing system can become out-of-order.

In addition, many algorithms profit from data-parallelism.

1.2 Research questions

1.3 Thesis structure

1 Extract transform load

Chapter 2

Related Work

This chapter provides background information about ...

TODO: Describe Chapter

2.1 Scalable container architectures

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. First, Barna et al. [BKFL17] primarily focused on a design for an auto-scaling architecture of containers with state-of-the-art technologies. Second, Srirama et al. [SAP20] researched auto-scaling algorithm and policies for efficient resource utilization and cost reducing.

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.

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%.

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.

2.2 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 3

Background

In this chapter bla bla

TODO: Describe Chapter

3.1 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].

3.1.1 Autonomic computing concept

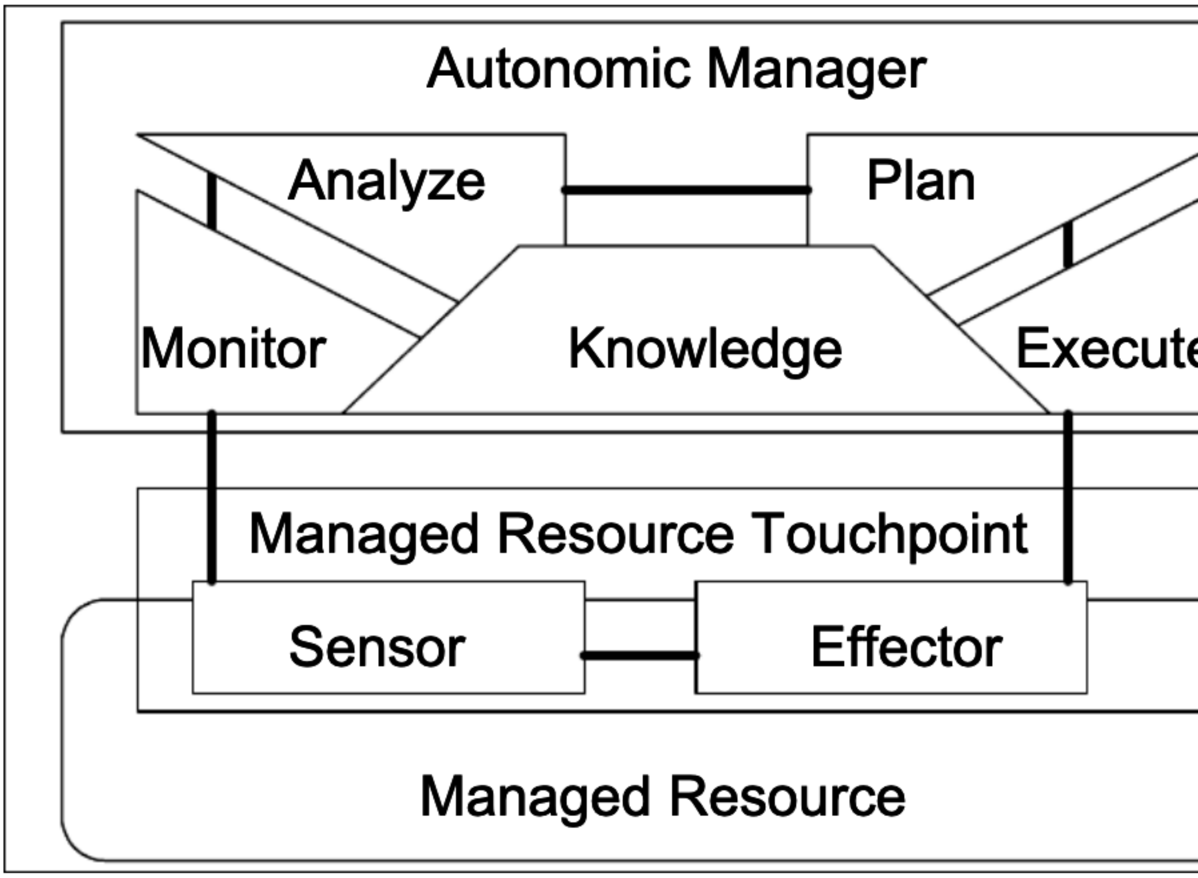

Figure 3.1: Autonomic computing concept - Source: Authors own model, based on [JSAP04].

Figure 3.1 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].

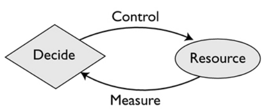

Figure 3.2: The control-loop concept - Source: Authors own model, based on [Mur04].

The core element of the autonomic architecture is the control-loop. Figure 3.2 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].

3.1.2 Managed resources

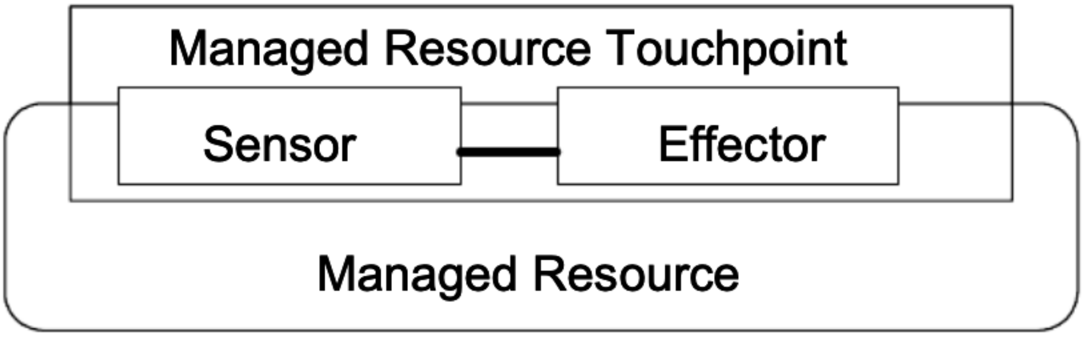

Figure 3.3: Managed resource - Source: Authors own model, based on [JSAP04].

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 3.3. 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].

3.1.3 Autonomic manager

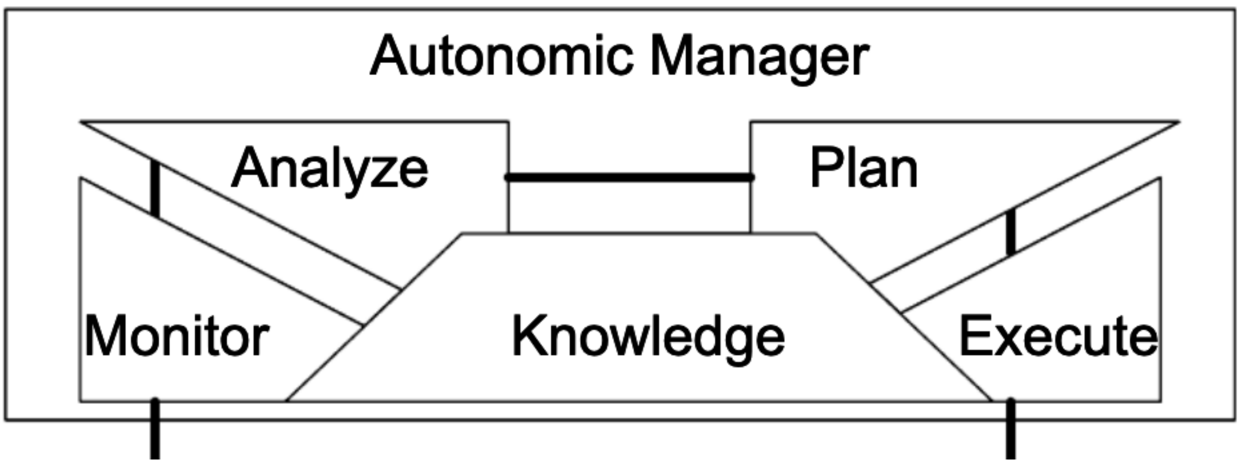

Figure 3.4: Autonomic manager - Source: Authors own model, based on [JSAP04].

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 3.4 [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].

3.2 Docker

TODO: Mehr zum Thema DevOps

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].

3.2.1 Docker architecture

Figure 3.5 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 [Inc20].

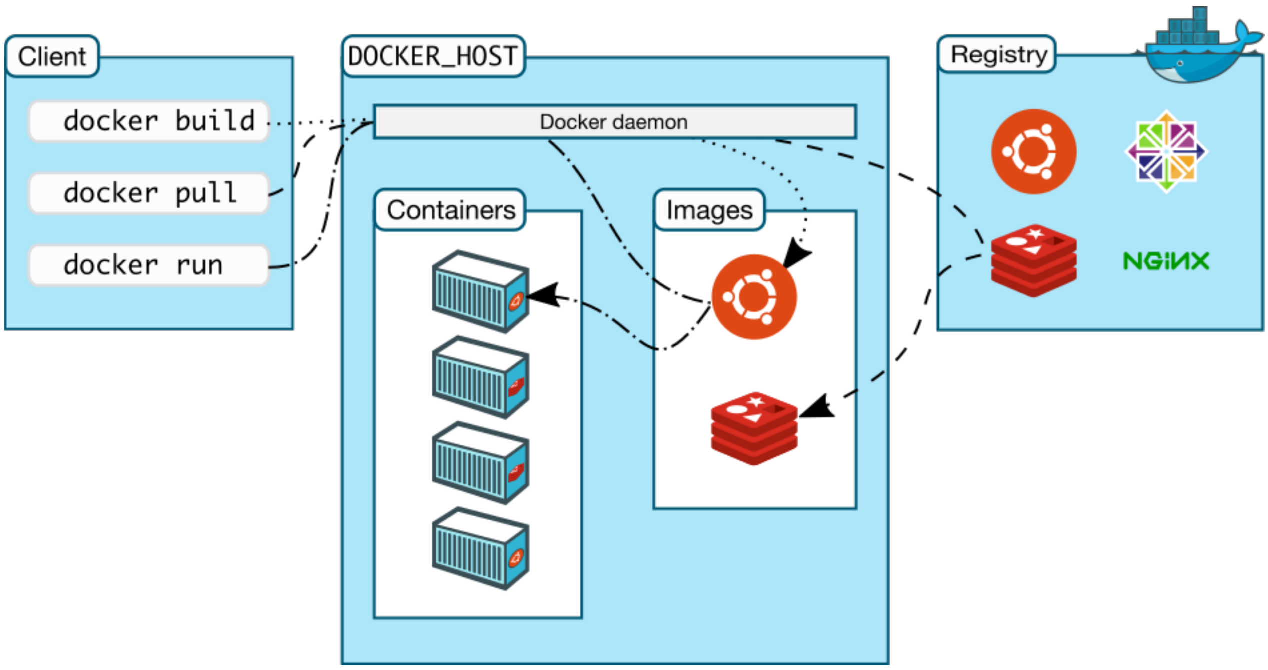

Figure 3.5: Docker architecture - Source: Authors own model, based on [Inc20].

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 [Inc20].

3.2.2 Docker image

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 [Inc20, 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 3.1: Example of a Dockerfile

Listing 3.1 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.

3.2.3 Docker Container

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

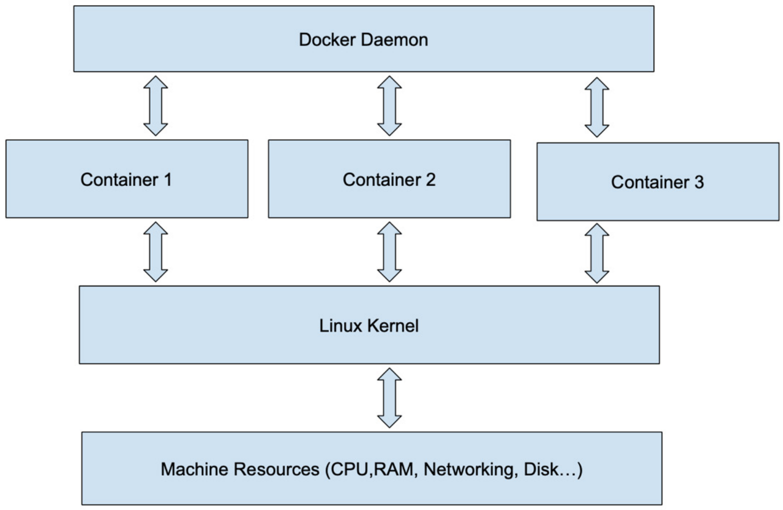

Figure 3.6: Docker basic container structure - Source: Authors own model, based on [BMDM20].

The advantage of a container is its lightweight nature. As illustrated in Figure 3.6, containers take advantage of OS-level virtualization instead of hardware-virtualization without the need of a hypervisor [Inc20, 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 [Inc20]. 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].

3.2.4 Docker Compose

Docker Compose is a tool to run multi-container applications on a single host. A multi-container application consists of a stack of services, where each service is deployed as a container [BMDM20, Inc20]. Services can be configured in a YAML[1] file called *docker-compose.yml*. This file defines the requirements of each service and determines how a service communicates to other services [KK18].

TODO: Beispiel docker-compose

1 YAML Ain't Markup Language - https://yaml.org/

3.3 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[2], a JVM-based language and provides a programming interface for Scala, Java[3], Python[4] and R[5]. In addition, Spark includes an interactive SQL shell and libraries to implement Machine Learning and streaming applications [CZ18]. It was developed in 2009 as the Spark research project at UC Berkeley and became an Apache Software Foundation project in 2013 [CZ18].

3.3.1 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,

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

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 3.7 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].

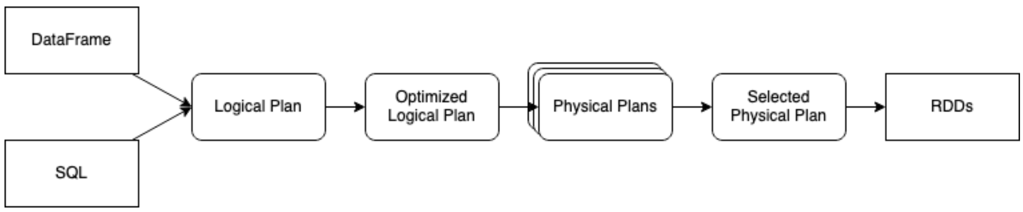

Figure 3.7: Optimization process of the Spark Catalyst - Source: Authors own model, based on [Luu18].

TODO: Bild nochmal machen mit abstand + QUELLE anpassen

3.3.2 Spark application architecture

TODO: Lieber Doch Master/Slave ???. Was ist mit dem CLuster-Manager

TODO: Nicht ganz richtig, Master und Worker sind machines und driver und executor sind prozesse

Figure 3.8 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].

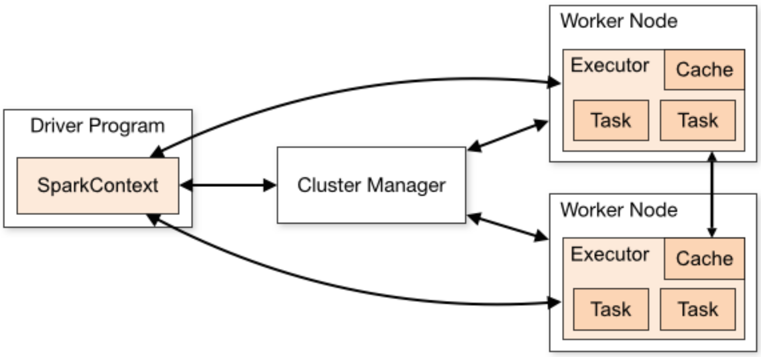

Figure 3.8: Overview of a Spark cluster architecture - Source: Authors own model, based on [Apa20].

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, Apa20]. 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[6], Hadoop YARN[VMD+13] and Kubernetes[7] [Apa20]. 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].

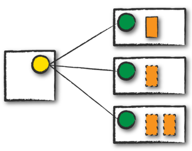

Figure 3.9: Spark‘s cluster mode - Source: Authors own model, based on [CZ18].

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

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].

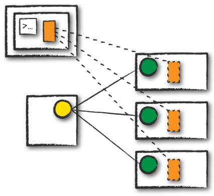

Figure 3.10: Spark‘s client mode - Source: Authors own model, based on [CZ18].

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].

3.3.3 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 3.2: Example of a Python3 Spark application

Listing 3.2 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 image

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

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

Listing 3.3 provides an example how the spark-submit executable can be used to launch a Spark Python application.

3.3.4 Spark standalone cluster deployment

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].

TODO: Why only standalone

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.

3.4 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 [Spa20]. To accelerate computing operations, it uses the RAPIDS[8] libraries and extends the Spark programming model (see Subsection 3.3.1) [Spa20, McD20].

3.4.1 Extension of the Spark programming model

The plugin suite extends the Spark programming model with a new DataFrame based on Apache Arrow[Fou20] 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 [Fou20].

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

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

operations [McD20]. Figure 3.11 illustrates, how a query plan gets optimized with the RAPIDS accelerator for Spark enabled.

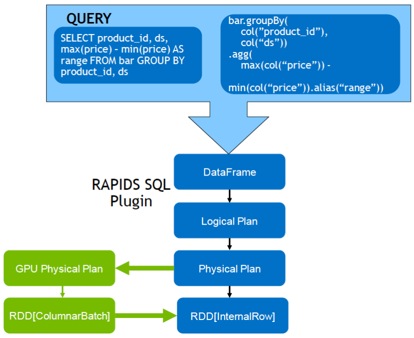

Figure 3.11: Catalyst optimization with RAPIDS accelerator for Apache Spark - Source: Authors own model, based on [McD20].

3.5 Prometheus

Prometheus is an open-source monitoring and alerting system [Pro20]. To collect and store data, Prometheus supports a multi-dimensional key-value pair based data model which can be analyzed in real-time using the PromQL query language [SP20]. It follows the pull-based approach to scrape metrics from hosts and services [BP19].

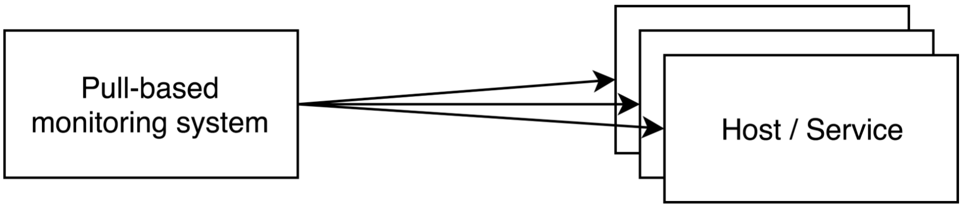

Figure 3.12: Pull-based approach to scrape metrics - Source: [BP19].

As Figure 3.12 demonstrates, a pull-based monitoring system scrapes metrics from services which makes them available for the monitoring system. In this case, the monitoring system needs a list of all hosts and services to monitor [BP19].

A monitoring system keeps track of the health status of components in a computing cluster. Because of the complexity of modern computing clusters, the effort is to high to observe components manually [BP19].

A metric is an observed property of software or hardware, e.g. the usage of a CPU core. To measure metrics, data-point‘s will be recorded at a fixed time interval. A data-point is a pair of a value and a timestamp. The combination of multiple data-point‘s is called a time-series [Tur16].

3.5.1 Prometheus architecture

TODO: Bilder für components

Figure 3.13 illustrates the high-level architecture of Prometheus. The Prometheus ecosystem provides multiple components. Components can

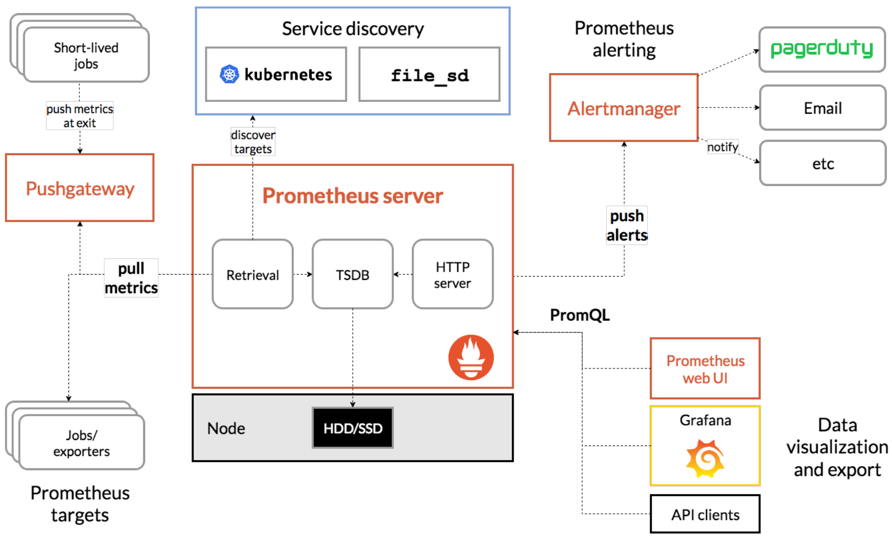

Figure 3.13: Prometheus high-level architecture - Source: Authors own model, based on [Pro20, BP19].

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 [Pro20].

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 [Pro20].

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[Pro20]. 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[9] is one of the widely used tools for this occasion.

3.5.2 Monitoring Docker container

TODO: Wahrscheinlich erst nach der implementation

3.6 Gitlab

3.7 K-MEANS

3.8 Naive Bayes Classifier

3.9 Scaling heat

3.10 KHP

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

Chapter 4

Conceptual Design

4.1 Cluster Architecture

TODO: Describe Chapter

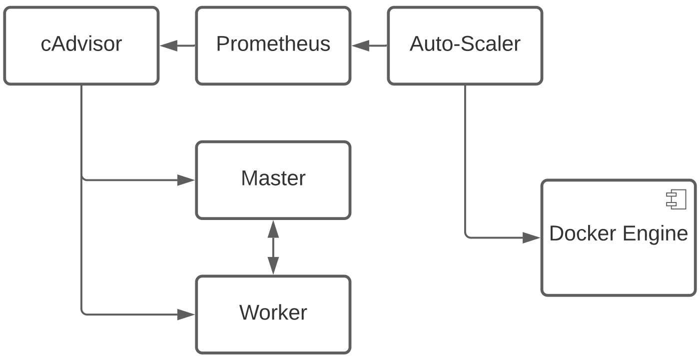

Figure 4.1: Scaling process UML activity model - Source: Authors own model.

The cluster conists of multiple components runnin in a own docker container. Prometheus, cAdvisor, Spark Master, Spark Worker, Auto-Scaler. cAdvisor collects metrics from all available docker container running in the same network. Prometheus scrapes metrics from cAdvisor and stores them as time-series-data in its ts-db. The Auto-Scaler read metric from Prometheus via its REST API. If a scaling action is necessary the auto-scaler scales the number of worker.

4.2 DevOps Process

4.3 Autonomic Manager

As described in section ABC, the Autonomic Manager follows the MAPE Architecture. The Autonim manager implements a Loops which is responsible for collecting and analyzing the metrics.

4.3.1 Workflow

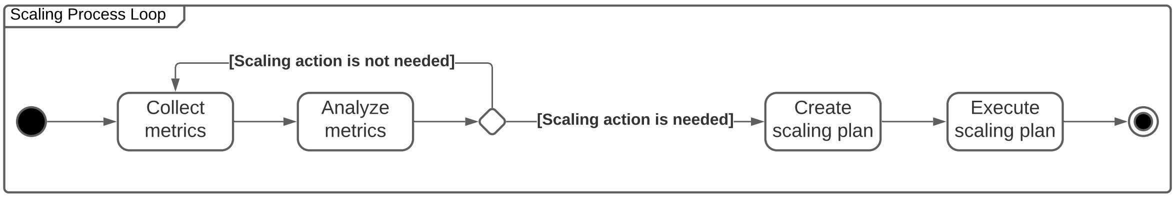

Figure 4.2: Scaling process UML activity model - Source: Authors own model.

As illustrated in Figure 4.2 the workflow consists of four different steps:

1. Collect metrics
2. Analyze metrics
3. Create scaling plan
4. Scale environment

In the first step, all metrics (described in Subsection **??**) are collected using cAdvisor and will be stored in Prometheus. The autonomic manager (described in Section **??**) gathers all metrics from Prometheus REST API and analyzes the data to discover if scaling actions are necessary. According to the data, the manager creates a plan to estimate how many nodes needed to be added or removed. In the last step, the scaling plan will be executed by the manager.

4.3.2 Design

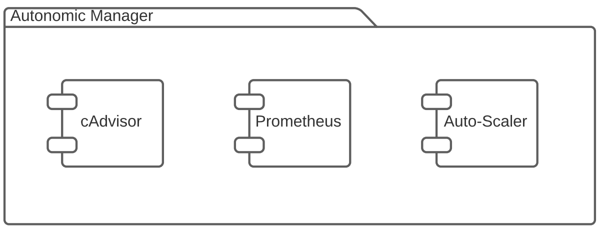

Figure 4.3: Scaling process UML activity model - Source: Authors own model.

The autonomic manager will be compose by several components to build a manager. To collect the metrics, cAdvisor will be used (see Section A), for storing the metrics Prometheus will be used (see Section B) and for auto-scaling, a Auto-scaler will be developed.

4.4 Auto-Scaler

4.4.1 Configuration

The Auto-Scaler needs specific configuration properties to be able to collect the correct metric 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.

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.

TODO: Check nochmal den Cold Start effekt

- **Apache Spark master URI:** To distribute the workload across all Spark Worker, all Spark Worker need to communicate with the Spark master. The Spark master URI starts with *spark://*.

Prometheus properties

- **Prometheus URL:** The Auto-Scaler will collect the configured metrics from Prometheus REST API.
- **Prometheus query:** A PromQL query needs to be defined to collect specific metrics for CPU utilization calculation.

4.4.2 Analyze

4.4.3 Plan

4.4.4 Scaling

If the CPU Utilization goes above 80%, it means the cluster is overloaded. If the Utilization is less than 20% the cluster is underutilized.

4.5 Metrics

4.5.1 CPU Utilization

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:

$$CPUUtilization = \sum ContainerCPUUserSecondsTotal \times 100 / NumberOfWorker \tag{4.1}$$

4.5.2 GPU Utilization

Chapter 5

Implementation

5.1 Cluster architecture

TODO: Describe Chapter

Chapter 6

Evaluation

6.1 Cluster architecture

TODO: Describe Chapter

Chapter 7

Outlook

7.1 Cluster architecture

TODO: Describe Chapter

Chapter 8

Conclusion

8.1 Cluster architecture

TODO: Describe Chapter

Bibliography

[Apa20] *Spark 3.0.1 Documentation*. https://spark.apache.org/docs/3.0.1/. Version: Oktober 2020

[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

[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

[Fou20] Foundation, The A.: *Arrow. A cross-language development platform for in-memory data*. https://arrow.apache.org/. Version: Oktober 2020

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

[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

[Inc20] Inc., Docker: *Docker Documentation*. https://docs.docker.com/. Version: Oktober 2020

[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

[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

[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

[Pro20] *Prometheus Online Documentation*. https://prometheus.io/docs/. Version: Oktober 2020

[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

[Spa20] *Spark-Rapids Online Documentation*. `https://nvidia.github.io/spark-rapids/`. Version: Oktober 2020

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

[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

[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

List of Figures