Bachelor Thesis

My Bachelor Thesis

Submitted by

Marcel Pascal Stolin

32168

born at 03.04.1993 in Kamen

Written at

Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA

and

Stuttgart Media University

First Examiner:	Prof. Walter Kriha
Second Examiner:	Prof. Dr.-Ing. Marco Huber
Supervisor:	M.Sc. Christoph Hennebold

Stuttgart, January 28, 2021

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

Listings

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

In this chapter the concepts of distributed computing, GPU acceleration, auto-scaling, and automated deployment pipeline will be introduced. Next, the research objective and the research questions as well as the problem statement of this thesis will be described. Finally, the structure of this thesis is being explained.

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 information from large datasets [Vad18]. Because of its computing intensive nature, Big Data is mostly processed in parallel on distributed hardware. Both concepts of distributed computing and parallel processing follow a divide-and-conquer principle [KBE16]. Distributed computing is achieved by forming a cluster of multiple machines with commodity hardware to utilize their resources to solve highly complex problems [GOKB16]. To process Big Data in parallel, a larger task will be divided into smaller sub-tasks that run concurrently. In general, one of the two following approaches can be used to achieve parallel processing [KBE16]:

- Task Parallelism: This approach refers to enabling parallelization by dividing a task into multiple sub-tasks. Each sub-task performs a different algorithm with its own copy of the same data in parallel. The result is created by joining the output of all sub-tasks together [KBE16].
- Data Parallelism: This approach is achieved by dividing a dataset into a series of smaller sub-datasets to process each sub-dataset in parallel. The sub-datasets are processed using the same algorithm across different nodes. The final output is joined together from each sub-dataset [KBE16].

Various tools and frameworks such as MapReduce, Apache Hadoop and Apache Spark have been created to facilitate distributed computing. The

MapReduce[DG04] framework gives the opportunity to solve massive complex problems in parallel on a cluster of single machines. Apache Hadoop[1] is an ecosystem platform for distributed computing. It contributes to create a cluster to process massive amounts of data in parallel by implementing the MapReduce processing framework [KBE16]. Implementing data pipelines with MapReduce 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 applications 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 parallelism. A CPU is build of multiple cores which are optimized for sequential serial processing. Performing computationally intensive applications on an Apache Spark cluster, consumes a huge amount of CPU cycles with negative impact on the overall performance [PYNY15]. To handle the complexity of Big Data applications, from executing Machine Learning algorithms or training Deep Learning models, an option of distributed computing clusters is to scale-up individual nodes. Scaling-up is limited by resource capacity and can be become uneconomically at a specific point. To perform computationally complex applications with better performance, Graphical Process Units (GPUs) have become first class citizens in modern data centers. The architecture of a GPU consists of a large amount of smaller and more efficient cores which are suitable for data-parallel data processing (handling multiple tasks simultaneously) [YSH+16]. In general, GPUs process data at a much faster rate than CPUs are capable. Apache Spark applications have a data-parallel nature. Therefore, enabling Apache Spark to leverage GPUs to perform complex ML algorithms on big datasets can have a huge positive impact on the performance [YSH+16].

1.3 Auto-Scaling

Adjusting the resources in a computing environment is not an easy task. To do it manually, a system administrator needs a deep knowledge about the environment and has to watch performance spikes regularly. This is a resource wasting process. In an optimal way, an automized process would watch the

1 Apache Hadoop - `https://hadoop.apache.org/` (Accessed: 2020-01-08)

computing environment, analyse performance metrics and automatically add or remove resources to optimize the performance and cost of running. This process is called auto-scaling.

Hiring experts to manually watching an application and scaling an computing environment is a waste of resources. An *Auto-Scaler* takes care of watching the environment by adding and removing resources to adapt to the computing needs. The *Auto-Scaler* can be configured to take care of optimal resource allocation and keep the cost of running at low point.

There exist two different scaling approaches to scale resources in a computing environment: 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 as horizontal scaling [Wil12]. Increasing the number of nodes in an environment, increases the overall computing capacity and additionally, the workload can be distributed across all nodes [Wil12, LT15]. It is important to note, that both approaches are not exclusive from each other and a computing environment can be designed to combine both approaches [Wil12]. Vertical scaling is limited by the maximum hardware capacity. Furthermore, 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 Automated Deployment Pipeline

Building, testing and releasing software manually is a time-consuming and error-prone process. To overcome this issue, a pattern called deployment pipeline automates the build, test, deploy, and release processes of an application development cycle. The concept of deployment pipelines is based on automation scripts which will be performed on every change on an applications source code, environment, data or configuration files [FH10]. A fully automated deployment pipeline has many improvements over deploying applications manually:

TODO: Eher automated software deployment nennen, dann warum das nötig ist und dann auf die pipeline eingehen.

- Makes every process until release visible to all developers [FH10]
- Errors can be identified and resolved at an early stage [FH10]
- The ability to deploy and release any version of an application to any environment [FH10]
- A non automated deployment process is not repeatable and reliable [FH10]
- The automation scripts can serve as documentation [FH10]

- If an application has been deployed manually, there is no guarantee that the documentation has been followed [FH10]

The automated deployment pipeline is based on the Continuous Integration (CI) process. Furthermore, the deployment pipeline is the logical implementation of CI [FH10].

1.5 Research Objective and Research Questions

The thesis work will be implemented at the Center for Cyber Cognitive Intelligence at the Fraunhofer IPA[2]. At the IPA, developers train ML models on Docker container running on a NVIDIA DGX[3] workstation. To optimize the training of ML applications, developers combine CPU and GPU resources only limited. Therefore a prototype of an Apache Spark cluster prototype has to be implemented which has the ability to automatically allocate resources according to the computing needs to scale its performance.

The following three research question will be investigated to implemented the mentioned prototype:

- RQ1: Is it possible to scale the number of Apache Spark Worker in accordance to performance utilization?
- RQ2: How can Apache Spark be extended to accelerate application execution with GPU support?
- RQ3: Is it possible to automate the deployment process of applications to a running Apache Spark cluster?

The first research question searches for concepts to create a self-adapting computing environment. To answer this question, state-of-the-art computing architectures have to be investigated. Monitoring tools to collect performance metrics need to evaluated. Additionally, tools which enable fast deployment of computing units. Furthermore, a suitable scaling approach has to be investigated.

The main goal of the second research question is to enable Apache Spark to perform algorithms with GPU acceleration included. Therefore, a concept needs to be investigated to extend Apache Spark to use GPUs for suitable algorithms in addition to the available CPUs.

2 Fraunhofer Institute for Manufacturing Engineering and Automation IPA - `https://www.ipa.fraunhofer.de/` (Accessed: 2021-01-07)

3 The Universal System for AI Infrastructure - `https://www.nvidia.com/en-us/data-center/dgx-a100/` (Accessed: 2020-01-09)

The last research question has a more applied nature. Automating the development cycle of an application is a well investigated topic. The IPA is using a platform called GitLab (will be introduced in Section 4.6) which provides an API to build automated pipelines. To answer this research question, GitLabs functionality will be investigated to find a solution that fits the need of this project work.

1.6 Problem Statement

Given the previously introduced research questions and the research objective, this thesis work will provide a solution to the following three problem statements:

1. Developers at the Fraunhofer IPA perform ML model training on several Docker containers running on a DGX with limited usage of available GPUs. Apache Spark can be used to optimize the model training by distributing the workload. Additionally, the Apache Cluster should be aware about available GPUs to accelerate the model training.

2. To enable GPU acceleration for Apache Spark alone is not sufficient to increase the performance. At some point, an Apache Spark Worker can reach the limit of its available computing resources. If this point is reached, the environment should automatically scale the number of Apache Spark worker to distribute the workload.

3. To perform an Apache Spark application to the cluster, developers have to submit the application manually. With an automated deployment pipeline, developers can submit an application by pushing changes to the code base. Additionally, a deployment pipeline will contribute to the reliability of executing applications and reduces the development time.

1.7 Thesis Structure

Chapter 2 provides the theoretical foundation about concepts which have been introduced in this chapter. Chapter 3 focuses on related work which provides solutions to solve the given problems of this thesis introduced in Section 1.6. In Chapter 4 all used technologies to implement the objective of this thesis are being introduced. Afterwards in Chapter 5, a conceptual design of a dynamic computing environment and an automated deployment pipeline is being described. Chapter 6 contains the implementation of the computing environment and how the deployment pipeline is being used to automate the deployment of applications to the computing environment. In Chapter 7 the results of the implementation are being presented and analysed. Chapter 8 introduces further work, which has been discovered during the work of this thesis, as well as improvements for the implementation. Finally Chapter 9 ...

TODO: Chapter 9

Chapter 2

Theoretical Foundation

This chapter provides the theoretical foundation to understand concepts that will be used in this thesis. First, the concept of Scalability will be described. Second, the theory behind a deployment pipeline is explained. Third, the concept of autonomic computing is introduced. Fourth, the theory of measuring system performance is explained. Lastly, the concept of monitoring is being described.

2.1 Scalability

Scalability defines the ability of a computing system to handle an increasing amount of load [Far17]. The limit of scalability is reached, when a computing system is not able to serve the requests of its concurrent users [Wil12]. Different approaches exist to increase the scalability of a system. The two main approaches are vertical scaling and horizontal scaling.

2.1.1 Horizontal Scaling

Horizontal scaling is accomplished by adding nodes to the computing environment to increase the overall capacity. Each node typically adds the equal amount of computing capacity (e.g. amount of memory) [Wil12]. By increasing the number of nodes in a computing environment, the workload can be distributed more efficiently across all nodes to handle and balance an increasing workload [Wil12, LT15].

Scaling a computing environment horizontally is limited by the efficiency of each added node. The horizontal scaling approach is more efficient with the simplicity of homogeneous nodes. Homogeneous nodes add the same amount of computing power to the system and are able to perform the same work and response as other nodes. With homogeneous nodes, creating strategies for capacity planning, load balancing, and auto-scaling is more efficient. In an environment with different types of nodes, creating these strategies is more complex due to the need of context [Wil12].

2.1.2 Vertical Scaling

Vertical scaling refers to increasing the overall capacity by improving the computing power with additional hardware of individual nodes (e.g. adding memory, increasing number of CPU cores) [Wil12].

If additional hardware has to be added to a system, it is not guaranteed that more powerful hardware is available or affordable. Therefore, vertical scaling is limited by of available hardware. Additionally, changing physical hardware of a running system can require a downtime. For most system a downtime should be avoided because it will interrupt important services running on the system [Wil12].

2.2 Deployment Pipeline

A deployment pipeline is an implementation of the process of getting software from source code to production. It is based on the concept of Continuous Integration (CI). The process involves building, testing and deploying software through automated scripts [FH10].

2.2.1 Continuous Integration

Continuous Integration is a development practice where each change on a primary code base is validated by automated scripts. This ensures that errors are detected and fixed in an early development stage [DMA07]. The CI process is responsible for building and testing the software to guarantee that it is in a releasable state at all time [Ros17]. CI contributes with the following advantages to the development life cycle of an application:

- Reduce risks: The CI process runs tests and validates the software on each change. Errors are detected in an early stage and can be fixed immediately [DMA07].
- Reduce manual processes: The CI process will perform every time a commit has being made to the code base. Each run is processed the exact same way every time. Therefore, no human intervention is needed to start the process which saves time and cost [DMA07].
- Generate deployable software: If an error occurs during a CI run, developers will be informed and fixes can be applied immediately. This ensures that the software is in a deployable state at all times [DMA07].

2.2.2 Requirements of a Continuous Integration Process

The implementation of a CI process is based on several requirements:

1. Version control repository: To manage changes to the code base, the source code and all other assets like the build script should be hosted on a single version control repository. Each change on the code base

triggers the CI process on the build server to run against the latest version available [DMA07].

2. Build server: The build server is responsible to monitor the code base for changes. If a change is committed, the build server automatically executes the CI scripts in order [Ros17, DMA07].

3. Build script: This includes all automation scripts to validate the source code [DMA07]. Typical examples are:
    - Building the software binaries (e.g. `.jar` binaries for Java source code).
    - Running unit and integration tests.
    - Deploying the binaries to a test or production environment.

2.2.3 Continuous Integration Process Implementation Example

Figure 2.1 demonstrates the CI scenario. First a developer commits changes to the version control repository. The CI server monitors the repository for changes. After the change has been committed, the CI server pulls the latest version of the source code and executes all build scripts in order to integrate the software. Finally, the CI server sends feedback to inform the developer about the build script status [DMA07].

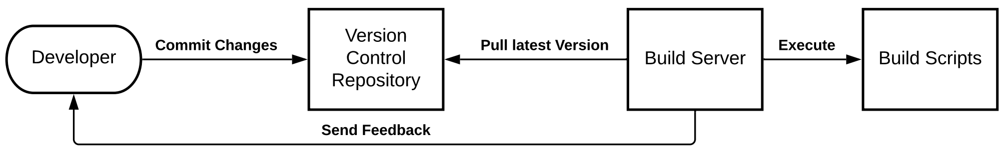

Figure 2.1: Continuous Integration Scenario - Source: Authors own model, based on [DMA07].

A CI run should be executed in a headless automated process. It is not feasible to rely on a manual process. All assets to perform the CI run should be accessed from the repository. Therefore a machine can start the build script process by executing a command script in an automated fashion [DMA07]. An example of a logical build script order is illustrated in Figure 2.2.

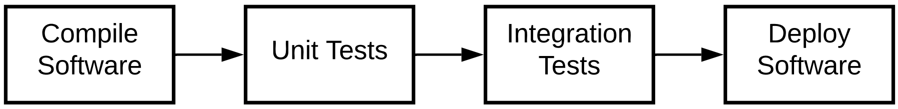

Figure 2.2: An example of a logical build script order for a CI process- Source: Authors own model, based on [DMA07].

2.3 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 its environment and how to extend its 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].

2.3.1 Autonomic Computing Concept

Figure 2.3 demonstrates the main concept of an autonomic computing environment. The autonomic computing architecture relies on monitoring

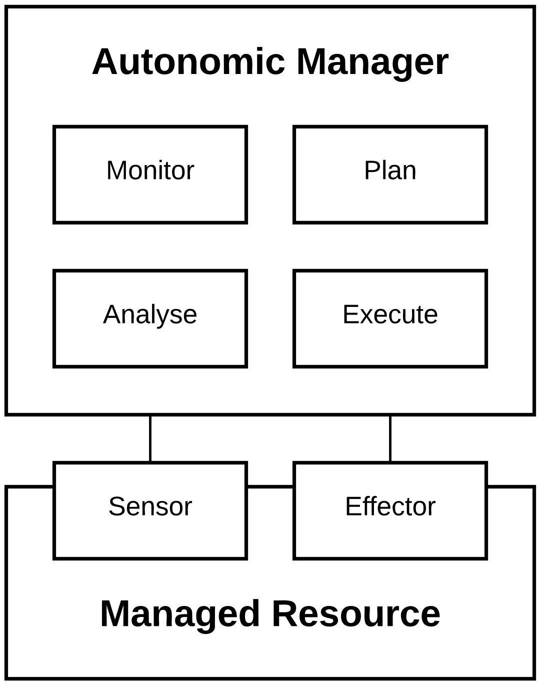

Figure 2.3: Autonomic computing concept - Source: Authors own model, based on [JSAP04].

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].

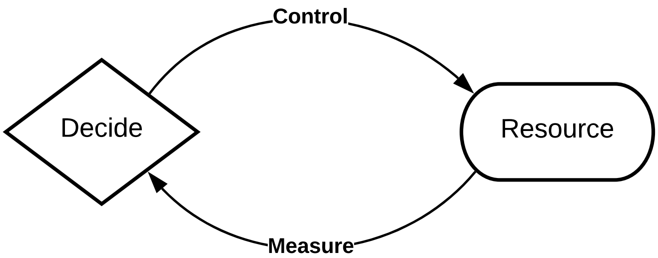

Figure 2.4: The control-loop concept - Source: Authors own model, based on [Mur04].

The core element of the autonomic architecture is the control-loop. Figure 2.4 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].

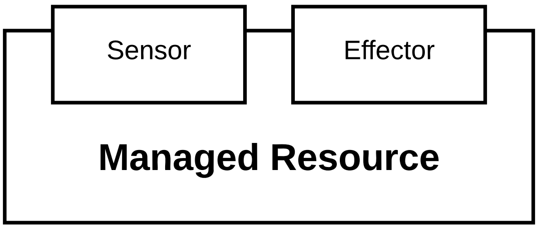

Figure 2.5: Managed resource - Source: Authors own model, based on [JSAP04].

2.3.2 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 2.5. 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 makes them highly scalable [Mur04].

2.3.3 Autonomic Manager

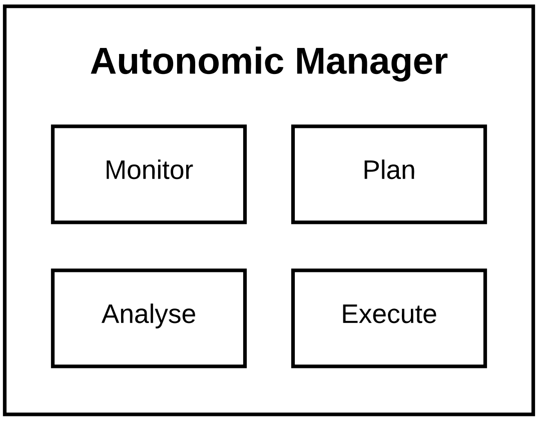

Figure 2.6: Autonomic manager - Source: Authors own model, based on [JSAP04].

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 its 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 analysing the managed resources [Mur04]. The autonomic computing concept digests the MAPE model to implement an autonomic manager, as illustrated in Figure 2.6 [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 phases. For example, an autonomic manager which is responsible to monitor and analyse the system and an autonomic manager to plan and execute. To create a complete and closed control-loop, multiple autonomic manager can be composed together [Sin06].

2.4 Performance Metrics

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

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

It is important to note that measuring performance metrics can cause an overhead. To gather and store performance metrics, additional CPU cycles must be spent. This can have a negative affect on the target performance [Gre20].

Utilization is a performance metric 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].

The performance of a hardware resource can degrade significantly if the utilization approaches 100%. Hardware which is able to perform work in parallel, might not have a performance degrade at 100%. Those hardware is able to accept additional work at a high utilization at a later time [Gre20].

2.5 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 a trivial process [Lig12]. A monitoring system consists of a set of multiple which are responsible to perform measurements on components in the computing environment and collect, store and interpret the monitored data [Lig12].

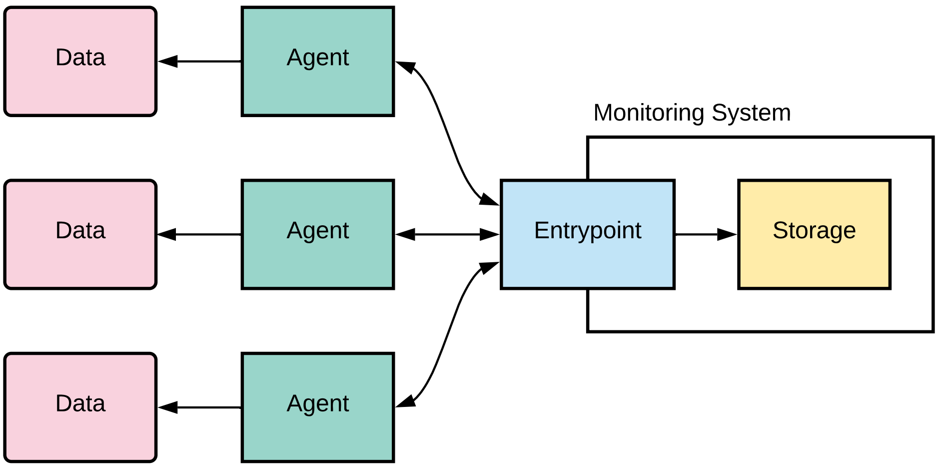

Figure 2.7: The monitoring process

In the monitoring process, illustrated in Figure 2.7, data is continuously collected by agents. An agent is a process that continuously gathers data from a target. The data can be device statistics, logs or system measurements. A pull-based monitoring system pulls the data from all specified agents. In contrast, a push-based monitoring expects data to be pushed from agents. These two approaches are described in Section 2.5.2. After the monitoring system has received the data, it groups the data together into metrics and stores 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 [Far17]
- A push or pull based way of gathering metrics [Far17]
- A multi-dimensional data-model [Far17]
- A powerful query language [Far17]

2.5.1 Database

Continuous data needs to be stored in the most efficient way. Time-series databases (TSDB) are optimized to store and retrieve time-series data. In a time-series database, metrics will be stored in a compact and optimized format. This allows the database to store a massive amount of time-series data on a single machine.

2.5.2 Push- and Pull-Based Monitoring Systems

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. 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].

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

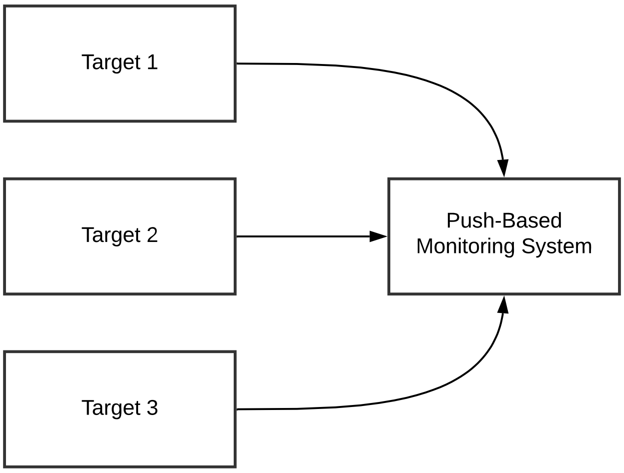

Figure 2.8: Push-based monitoring approach

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

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

2.5.3 Multi-Dimensional Data Model

Metrics are store as time-series data, where a time-series is a combination of a name and a set of optional key-value pairs called labels. The name of a time-series identifies the metric which is measured. Labels provide a multi-dimensional data-model to the stored data. Each combination of labels represent a specific dimensional instantiation of a metric [Thed]. In a dynamic environment, services are dynamically added and removed. Therefore, a dynamic environment needs a multi-dimensional data model to represent all dimension in the environment [Far18]. In addition to a

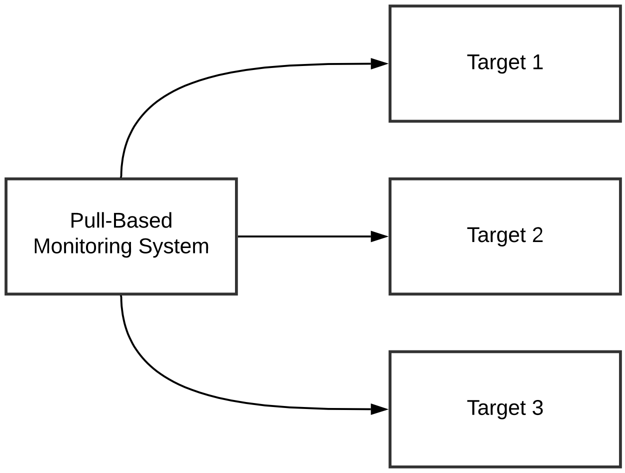

Figure 2.9: Pull-based monitoring approach

multi-dimensional data model, a powerful query language that provides capabilities to perform aggregations and filtering on dimensions is needed.

```
container_cpu_user_seconds_total
```

Listing 2.1: Example of a dimensionless-metric

```
container_cpu_user_seconds_total{image="spark-worker:3.0.1-
    hadoop2.7"}
```

Listing 2.2: Example of a metric with dimensions

Listing 2.1 provides of a metric without labels and Listing 2.2 shows an example of a metric with a label. As the examples show, The metric with a label provides more efficient querying to gather specific informations about a metric.

Chapter 3

Related Work

This chapter provides an overview of related literature for this thesis. Furthermore, the surveyed literature is build on the theoretical foundation introduced in Chapter 2. This chapter introduces work about auto-scaling computing environments, GPU accelerated Apache Spark cluster and the implementation of an automated deployment pipeline. These topics are related to the choice of technologies (Chapter 4), the proposed conceptual design of this thesis (Chapter 5), and the resulting implementation (Chapter 6).

3.1 Auto-Scaling Computing Environments

In recent years, container technologies have been used efficiently in complex computing environments. Dynamic scaling of containerized applications is an active area of research. To accommodate this thesis research objective, the literature research according to auto-scaling environments was focused on two topics: Concepts of *Auto-Scalers* and auto-scaling algorithms.

3.1.1 Auto-Scaler Concepts

Lorido-Botrán et al. [LBMAL14] reviewed state-of-the-art literatures about auto-scaling and explain proposals of an auto-scaling process in a cloud environment. It is mentioned that an *Auto-Scaler* is responsible to find a trade-off between meeting the Service Level Agreement (SLA) and keeping the cost of renting resources low. Two types of SLA exists while maintaining an acceptable trade-off: The application SLA and the resource SLA. The former is a contract between the application owner and the end users (e.g. a certain response time). The resource SLA is agreed by the infrastructure provider and the application owner (e.g. 99.9% availability). They introduced three problems an *Auto-Scaler* faces while scaling an environment, and meeting the SLA:

1. Under-provisioning: An application is under-provisioned if it needs more resources to process the incoming workload. To make resources

available and return the application to its normal state may take some time which causes SLA violations.

2. Over-provisioning: Applications which have more resources available than needed, will lead to unnecessary costs for the client.

3. Oscillation: If scaling-actions are executed too quickly before the impact is available, a combination of over-provisioned and under-provisioned applications can occur. A cooldown period after a scaling-action allows to prevent oscillation.

To prevent the mentioned problems from occurring, the authors introduced and explained the MAPE architecture (cf. Section 2.3). MAPE consists of four different phases: Monitor, Analyse, Plan, and Execute. There exist Auto-Scaler proposals which only focus on the Analyse, and Planning phase architecture of the MAPE architecture. Several techniques for the analyse phase are being introduced: Queuing theory, and time-series analysis. As well as for the planning phase: Threshold-based rules, reinforcement learning, and control theory. There exist Auto-Scaler which uses techniques to predict the future state of the environment (e.g. reinforcement learning). These are called reactive Auto-Scalers. Proactive Auto-Scalers use techniques to respond to the current status of the environment (e.g. threshold-based rules).

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 is to balance the overall resource utilization across microservices in the environment. The proposed auto-scaling strategy performed better than state-of-the-art algorithms in processing time, processing cost, and resource utilization. The processing cost of microservices was reduced by 12-20% and the CPU and memory utilization of cloud-servers were maximized by 9-15% and 10-18% respectively.

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 (e.g. *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 system management. 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%.

3.1.2 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. The Scaling Heat algorithm will be used for decision making in this thesis and is explained in detail in Section 4.7.2.

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, as well as Grafana for visualizing metrics. Absolute metrics are more appropriate when it comes to efficient resource allocation. Therefore, the KHPA-A algorithm is more efficient in vertical scaling of resources. In this thesis, the focus for scaling strategies is based on the horizontal scaling approach. Therefore, the KHPA algorithm will be used throughout this thesis and is explained in detail in Section 4.8.

3.2 GPU accelerated Apache Spark Cluster

This thesis aims to enable GPU acceleration for Apache Spark. In research, many solutions have been proposed which try to solve the problem in similar ways. In the following, three different approaches are introduced.

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 Apache Spark applications and invokes the GPU kernel for acceleration. For communication between CPU and GPU, HeteroSpark implements a CPU-GPU communication layer for each worker node using the Java Remote Method Invocation (RMI) API. To execute operations on the GPU, the CPU Java Virtual Machine (JVM) will send serialized data to the GPU JVM using the RMI communication interface. The GPU JVM will deserialize the received data 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 modification of Apache Spark. HetSpark extends Apache Spark with two executors, a GPU accelerated executor and a commodity class. The accelerated executor uses VineTalk[MPK+17] for GPU acceleration. VineTalk contributes as a transport layer between the application and accelerator devices (CPU or GPU). To detect suitable tasks for GPU acceleration, HetSpark uses the ASM[1] framework to analyse the byte code of Java binaries. 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* a CPU-GPU hybrid system build on top of Apache Spark. The goal of SparkGPU is to utilize GPUs to achieve high performance and throughput. SparkGPU tries to solve the following problems statements:

1. The iterator model Apache Spark uses, executes one element at a time. This approach does not match the GPU architecture and underutilizes GPU resources.

2. Apache Spark runs on the JVM and therefore stores its data on the heap memory. GPU programs are usually implemented with GPU programming models like CUDA which cannot access data on the heap. Therefore, data must be copied between the heap and native memory frequently. These copy operations are expensive.

3. Existing cluster manager of Apache Spark manage GPUs in a coarse grained fashion. This can lead to crashes because of insufficient memory when multiple programs run on the GPU concurrently.

To solve the mentioned problem statements, SparkGPU extends Apache Spark in the following ways:

- Enable block processing on GPUs by extending Apache Sparks iterator model. Therefore Apache Spark can better utilize GPUs to accelerate application performance.

- To offload queries to the GPU, SparkGPU extends the query optimizer. The query optimizer will create query plans with both CPU and GPU operators.

- To manage GPUs efficiently, SparkGPU extends the cluster manager and the task scheduler.

1 ASM - `https://asm.ow2.io/` (Accessed: 2021-01-11)

To extend the programming API, SparkGPU provides a new RDD type called GPU-RDD. A GPU-RDD is optimized to utilize the GPU. SparkGPU utilizes native memory on the GPU instead of the Java heap to buffer data in GPU-RDDs. Operations performed on a GPU-RDD can be performed on the GPU. Several built-in operators on the GPU-RDD are provided which support data-parallelism.

SparkGPU is able to execute SQL queries on both CPU and GPU. By adding a set of GPU rules and strategies, SparkGPU extends the query optimizer to find the best execution plan for GPU scheduling.

To manage GPU memory on shared GPUs, SparkGPU provides a user-level GPU-management library. The library will ensure, when memory contention happens, that SparkGPU will stop scheduling new tasks to the Apache Spark cluster.

SparkGPU accomplished to improve the performance of machine learning algorithms up to 16.13x and SQL query execution performance up to 4.83x.

3.3 Implementation of an Automated Deployment Pipeline

Implementing an automated deployment pipeline is a more applied topic and well described in many literature. In this section the main literature used throughout the implementation of this thesis work is being introduced.

The conceptual design and implementation of an automated deployment pipeline in this thesis was mostly inspired by the proposed solution of the book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Humble et al. [FH10]. The authors explain the theoretical idea behind an automated deployment pipeline and explaining an example implementation. The proposed implementation covers the software lifecycle from compiling source code to delivering the software to a production environment. The commit stage which covers the build and test part of the software can be applied in parts for this thesis work.

Chapter 4

Technical Background

This chapter provides information about the technologies and algorithms used for this thesis work. It explains the fundamental concepts of Docker, Apache Spark, RAPIDS accelerator plugin, Prometheus, cAdvisor, and GitLab CI/CD. Additionally the Scaling Heat and the Kubernetes Horizontal Pod Autoscaler algorithms are introduced.

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].

4.1.1 Docker Architecture

Figure 4.1 illustrates the client server architecture of Docker. It consists of a Docker Client, the Docker Deamon, and a Docker 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].

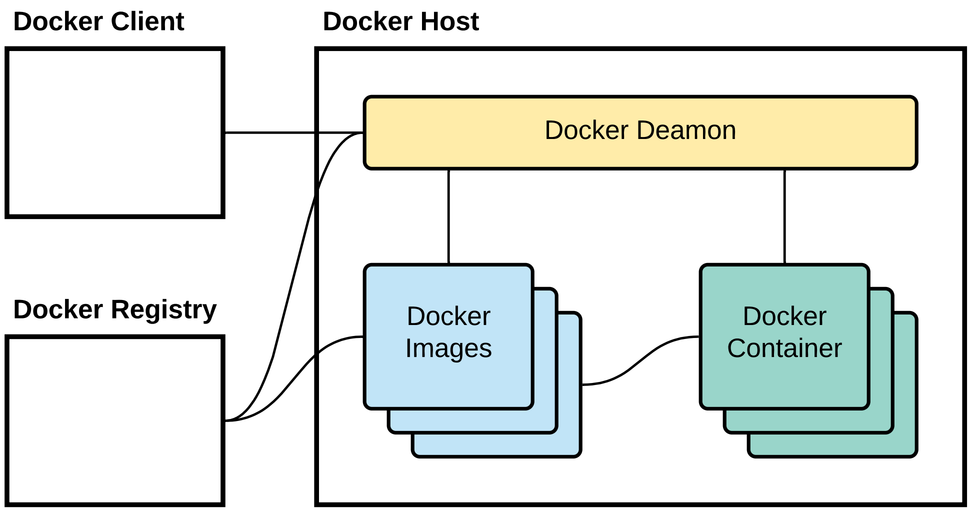

Figure 4.1: Docker architecture - Source: Authors own model, based on [Doc].

4.1.2 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 [Doc, NK19]. Images can be created from existing containers or from executing a build script called *Dockerfile* [NK19].

Images can be build automatically by executing instructions defined in a Dockerfile. A Dockerfile is a text documents that contains instructions. Instructions are commands which will be executed in order to assemble an image. They are defined in the `INSTRUCTION argument` format. A Dockerfile must begin with a `FROM` instruction. The `FROM` instruction defines the parent image from which the image is build [Doc].

Listing 4.1 provides a basic example of a Dockerfile. In the example, *ubuntu* is defined as the parent image. Next, the Ubuntu package list is updated and the *nginx* package is installed using the `apt-get` command. Finally, the Docker image is instructed to listen to the port 80 at runtime with the `EXPOSE` instruction.

```
FROM ubuntu

RUN apt-get update -qy && \
    apt-get install -y nginx

EXPOSE 80
```

Listing 4.1: Basic example of a Dockerfile

4.1.3 Docker Container

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

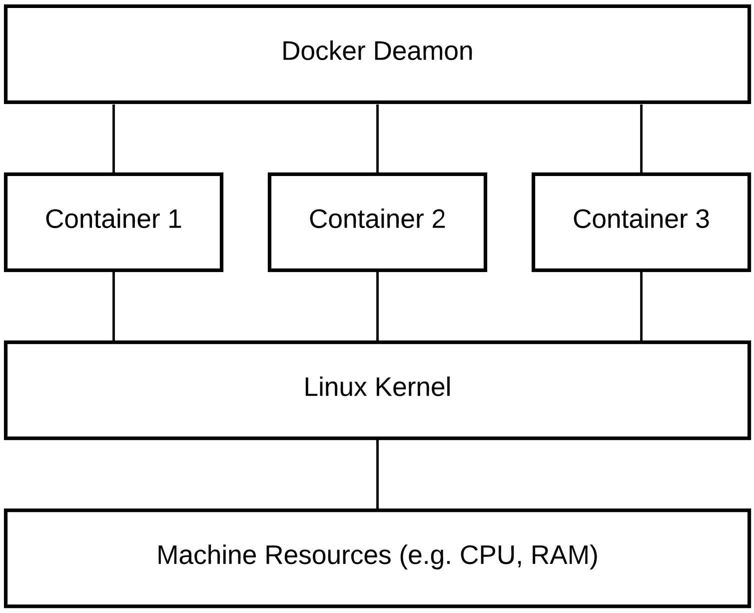

Figure 4.2: Docker basic container structure - Source: Authors own model, based on [BMDM20].

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

Many properties of Docker Swarm mode contribute the fact that it is an ideal tool 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 within 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. Apache Spark manages the available resources and distributes computation tasks across a cluster to perform big-data processing operations at large scale [CZ18]. Before Apache 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]. Additionally, Apache Spark includes an interactive SQL shell and libraries to implement Machine Learning and streaming applications [CZ18].

4.2.1 Spark Programming Model

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

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 Apache Spark and serve as the underlying data structure for higher level APIs (Spark structured API) [ZCD+12]. RDDs are a immutable, partitioned collection of records and can only be initiated through transformations (e.g. map, filter) on data or other RDDs. 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 Apache Spark automatically optimizes the execution for structured API operations [CZ18].

Apache Spark Structured API

Apache 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

1 The Scala programming language - `https://www.scala-lang.org/` (Accessed: 2020-12-18)
2 Java Software - `https://www.oracle.com/java/` (Accessed: 2020-12-18)
3 Python programming language - `https://www.python.org/` (Accessed: 2020-12-18)
4 The R Project for Statistical Computing - `https://www.r-project.org/` (Accessed: 2020-12-18)

for JVM based languages. DataFrames are Datasets of type `Row`, which is the optimized format for computations [CZ18].

Apache Spark Catalyst

Apache Spark also provides a query optimizer engine called Apache Spark Catalyst. Figure 4.3 illustrates how the Spark Catalyst optimizer automatically optimizes Apache Spark applications to run quickly and efficient. Before executing the users 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 plan will be generated into Java bytecode for execution across the cluster [Luu18].

4.2.2 Application Architecture

Figure 4.4 illustrates the main architecture of an Apache Spark cluster running an application. The architecture follows the master-worker model. Each running application has one driver process (master) and multiple executor processes (worker) exclusively assigned by the cluster manager. Furthermore, the cluster manager decides on which nodes the processes will be executed [Luu18].

Driver Process

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

Executor Process

The executor process is a JVM process, that runs through the whole duration of an application [Luu18, Theb]. It is responsible to perform all tasks (units of work) assigned by the driver process [CZ18]. After the executor process finish, it reports back to the driver process [CZ18]. Each task will be performed on a separate CPU core to enable parallel processing [Luu18].

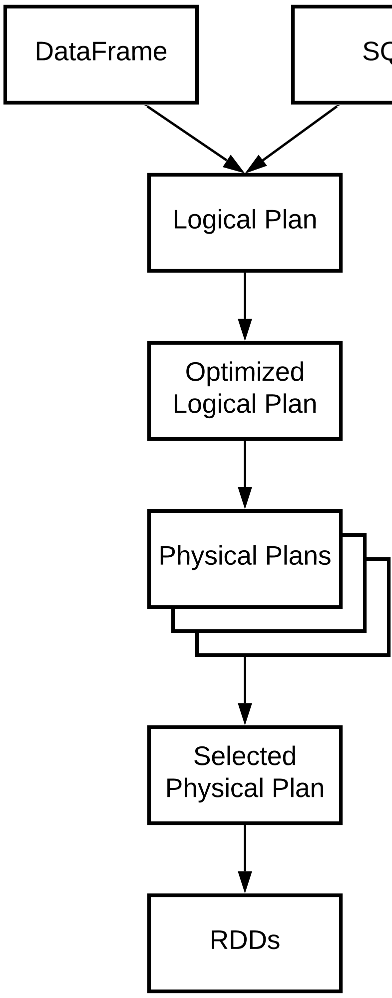

Figure 4.3: Optimization process of the Spark Catalyst - Source: Authors own model, based on [Luu18].

Cluster Manager

The cluster manager is an external service that orchestrates the work between available machines in the cluster [Luu18, Theb]. It decides on which nodes in the cluster the driver process and the executor processes will be launched. Additionally, the cluster manager manages the resources of each node in the cluster [Luu18, CZ18]. Apache Spark supports different services that can run as cluster manager: Standalone mode (introduced in Section 4.2.3), 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.

- Cluster mode: To run an application in cluster mode, the user has to

5 Apache Mesos - `https://mesos.apache.org/` (Accessed: 2021-01-02)
6 Kubernetes - `https://kubernetes.io/` (Accessed: 2021-01-02)

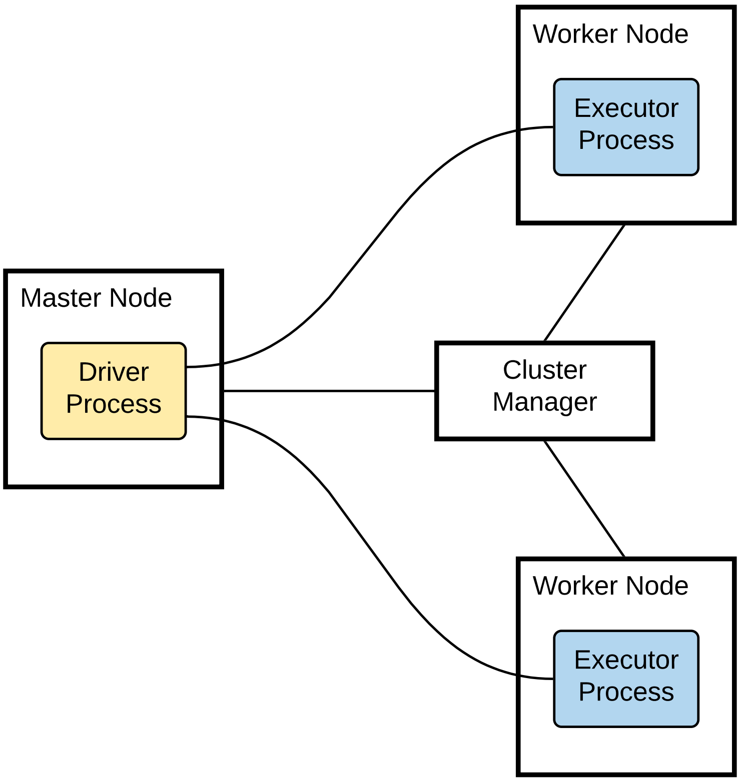

Figure 4.4: Overview of a Spark cluster architecture - Source: Authors own model, based on [Theb].

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 Apache Spark application on machines inside the cluster [CZ18, Luu18].

- Client mode: 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].
- Local mode: The local mode starts an Apache Spark application on a single computer [CZ18]. It is important to mention, that the local mode is not recommended to use in production. Instead it should be used for testing Apache Spark applications [CZ18].

4.2.3 Standalone Cluster Deployment

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

Starting Master and Worker Nodes

Apache Spark provides executable launch scripts to start master and worker nodes in standalone mode. The executables can be found at `sbin/start-master.sh` to start a master node and at `/start-slave.sh` to start a worker node. The worker launch executable requires the master node URI as parameter [Theb].

```
$ ./sbin/start-master.sh
```

Listing 4.2: Usage of master launch script

```
$ ./sbin/start-slave.sh spark://spark-master:7077
```

Listing 4.3: Usage of worker launch script

Listing 4.2 and Listing 4.3 provide an example of how to use both executables to start a master and a worker node. The URI `spark://spark-master:7077` in Listing 4.3 is an example of a master node URI. The master node launch script will print out the master URI after being executed successfully [Theb].

Resource Allocation

In standalone mode, worker need a set of resources configured. Therefore, a worker can assign resources to executors. To specify how a worker discovers resources, a discovery script has to be available [Theb].

Submitting Applications with spark-submit

To submit an application to a standalone cluster, Apache Spark provides the `spark-submit` executable. The executable file is available at `bin/spark-submit` in the installation folder of Apache Spark. In cluster mode the driver of an Apache Spark application (see Section 4.2.2) will be launched from one of the worker processes inside the cluster. The submit process will finish after it has submitted the application. It does not wait for the submitted application to finish [Theb].

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

Listing 4.4: Example usage of the spark-submit executable

Listing 4.4 shows how the spark-submit executable can be used to submit a Python application to a standalone Apache Spark cluster. Spark-submit requires the master node URI and the path to the desired Spark application file. With the spark-submit executable it is possible to submit Python, Java and R applications [Theb].

4.3 RAPIDS Accelerator for Apache Spark

RAPIDS accelerator for Apache Spark is a plugin suite that aims to accelerate computing operations for Apache Spark by executing pipelines entirely on GPUs. It is available for Apache Spark 3 [NVI]. The plugin uses the RAPIDS AI cuDF[7] library to extend the Apache Spark programming model, introduced in Section 4.2.1 [NVI, McD20, APW19].

4.3.1 Extension of the Spark programming model

The plugin suite extends the Apache Spark programming model with a new DataFrame based on Apache Arrow[8] data structures. The new DataFrame API aims to accelerate loading, filtering, and manipulation operations on large datasets. In addition, the Catalyst optimizer (described in Section 4.2.1) is extended to generate GPU-aware query plans [McD20, APW19]. Apache arrow is a data platform to build high performance applications that work with large datasets 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].

Figure 4.5 illustrates how the RAPIDS plugin suite extends the Catalyst optimization process illustrated previously in Figure 4.3. The Spark Catalyst optimizer identifies operators in a query plan that are supported by the RAPIDS API. 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].

4.3.2 GPU Accelerated Machine Learning with XGBoost

RAPIDS accelerates SparkSQL operations, and operations on a DataFrame. Additionally RAPIDS aims to accelerate the training process of machine learning models. Currently, RAPIDS only supports GPU-acceleration for Extreme Gradient Boosting (XGBoost) in SparkML [McD20].

XGBoost is a scalable, distributed gradient-boosted machine learning library. It trys to solve many data science problems, by implementing machine learning algorithms using the gradient boosting technique. With the XGBoost4j-Spark[9] library, XGBoost integrates into the Apache Spark ML library [xgb].

7 Open GPU Data Science - `https://rapids.ai/` (Accessed: 2021-01-01)

8 Arrow. A cross-language development platform for in-memory data - `https://arrow.apache.org/` (Accessed: 2020-12-03)

9 XGBoost Documentation - `https://xgboost.readthedocs.io/en/latest/jvm/xgboost4j_spark_tutorial.html` (Accessed: 2021-01-28)

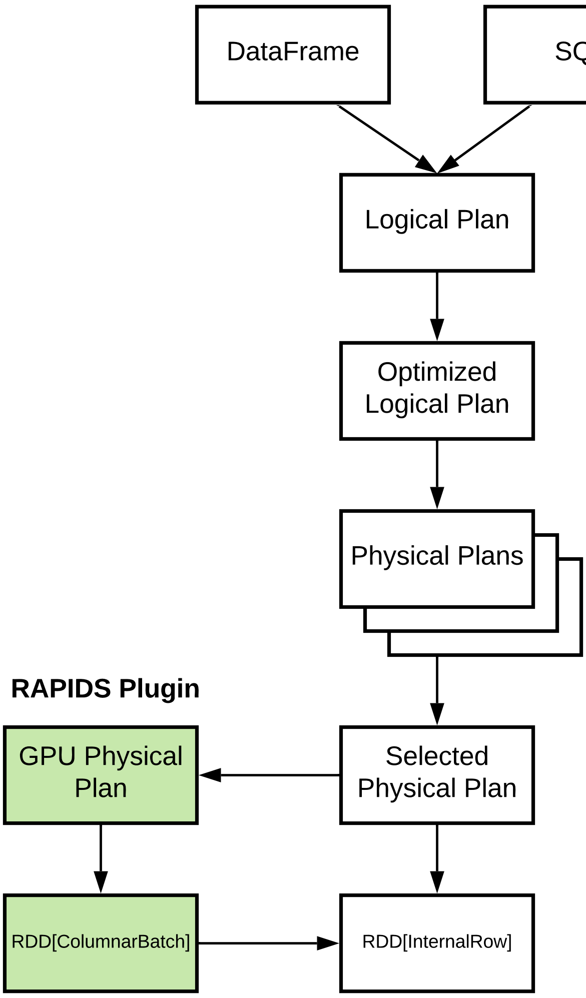

Figure 4.5: Catalyst optimization with RAPIDS accelerator for Apache Spark - Source: Authors own model, based on [McD20].

4.3.3 Installation Requirements for Apache Spark Standalone Mode

The RAPIDS accelerator for Apache Spark is available for a standalone mode Apache Spark cluster. To operate effectively, the following requirements need to be installed on all Apache Spark nodes in the cluster [NVI]:

- Java Runtime Environment (JRE)
- NVIDIA GPU driver
- CUDA Toolkit[10]

10 CUDA Toolkit - `https://developer.nvidia.com/cuda-toolkit` (Accessed: 2021-01-01)

- RAPIDS accelerator for Apache Spark Java library
- cudf Java library which is supported by the RAPIDS accelerator Java library and the installed CUDA toolkit
- GPU resource discovery script

4.4 Prometheus

Prometheus is an open-source monitoring and alerting system [Thed]. To collect and store data, Prometheus supports a multi-dimensional key-value pair based data model, according to Section **??**, which can be analyzed in using the PromQL query language [SP20]. PromQL is a functional query language for selecting and aggregating time-series data in real-time [Thed]. Prometheus follows the pull-based approach, as described in detail in Section 2.5.2, to scrape metrics from hosts and services [BP19].

4.4.1 Prometheus Architecture

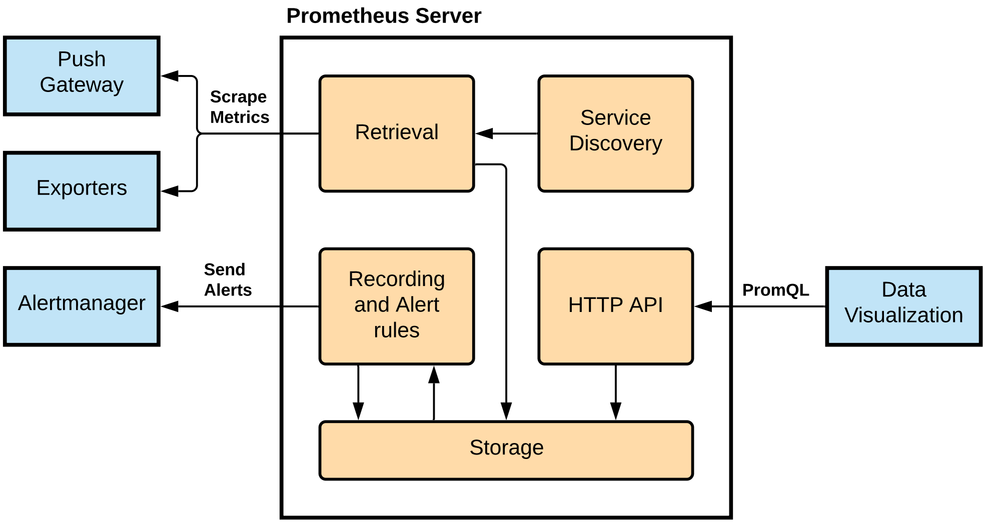

Figure 4.6: Prometheus high-level architecture - Source: Authors own model, based on [Thed, Bra18].

Figure 4.6 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, Push Gateway, and data visualization [Thed].

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 [Thed]. The core components of the Prometheus Server, as illustrated in Figure 4.6, are the following:

- Service Discovery: As being 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]. Therefore, Prometheus is able to notice targets which are not responding [Bra18].

- Retrieval: Prometheus sends a HTTP request to each target to scrape metrics. The request is send each interval, which can be set in the configuration [Bra18].

- HTTP API: Prometheus provides a HTTP API. This API can be used to request raw data and evaluate PromQL queries. Data visualisation tool can use this API to create visualisations of the requested metrics [Bra18].

- Recording and alert rules: Recording rules enable to precompute frequently needed or compute-intensive PromQL expressions. The result will be saved as a set of time-series in the local storage. This enables to query a recording rule at a much faster speed than the original PromQL expression [Bra18, Thed].

  Alert rules define conditions based on PromQL expressions. If a condition becomes true, an alert will be send to an external service [Thed].

- Storage: Received data is stored in a custom highly efficient format on a local on-disk time-series database [Thed]. Prometheus does not offer a solution for distributed storage across a cluster of machines [Bra18].

Optional Components

The Prometheus ecosystem offers a set of components which are optional and can be activated depending on the monitoring needs. The optional components illustrated in Figure 4.6 are the following:

- 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].

- 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 metrics from the host and exports them to the Prometheus server [SP20].
- Push Gateway: If a target is not designed to be scraped, metrics can be pushed against the Push Gateway [Thed]. The Push Gateway converts the data into the Prometheus data format and passes them to the Prometheus server [SP20].
- Data Visualisation: Prometheus supports various tools for virtualization of the scraped data. Grafana[11] is one of the widely used tools for this occasion.

4.4.2 Prometheus Configuration

Configuration related to scraping jobs and rules are configured in configuration files. Configuration are written in the YAML file format [Thed].

Listing 4.5 shows a valid configuration file example. In the `global` configuration section, default values can be set. Targets are defined in the `scrape_configs` section. Each target is defined as a scrape job with a unique name. A target can be defined statically using the `static_configs` parameter or dynamically using the available service discovery mechanisms [Thed]. Rules have to be configured in a separate YAML file. To load rules into Prometheus, the file path has to be set in the `rule_files` parameter.

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: cadvisor
    static_configs:
      - targets: ["cadvisor:8080"]
        labels:
          group: "cadvisor"

rule_files:
  - "/etc/prometheus/recording_rules.yml"
```

Listing 4.5: Prometheus configuration file example

Listing 4.6 is an configuration example of a recording rule. Rules are defined in a rule group. Each rule is defined by a name and a valid PromQL expression [Thed].

11 Grafana: The open observability platform - `https://grafana.com/` (Accessed: 2021-01-19)

```
groups:
  - name: http_requests
    - record: job:http_inprogress_requests:sum
      expr: sum by (job) (http_inprogress_requests)
```

Listing 4.6: Prometheus rules configuration file example

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 [BP19, Goo].

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 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].

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 repository [Git].

4.6.2 Example of a Basic CI/CD Pipeline Architecture

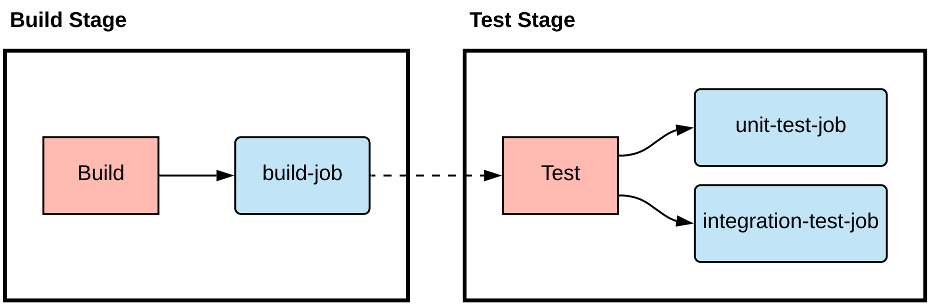

Figure 4.7: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

Figure 4.7 illustrates an architecture of a basic CI/CD pipeline. The pipeline consists of a build and a test stage. Stages will be performed in order. The test stage will only be performed after the build stage was successful. The build stage consists of a single job named *build_job*. This job is executed first, after a change is pushed to the repository. Two jobs exist in the test stage. These jobs are executed in parallel when the test stage is triggered.

Listing 4.7 provides the configuration for the basic pipeline example. Each job executes a shell script to perform the desired actions. The shell scripts have to be located in the source code repository.

```
stages:
  - build
  - test

build-job:
  stage: build
  script:
    - build_software.sh

unit-test-job:
  stage: test
  script:
    - run_unit_tests.sh

integration-test-job:
  stage: test
  script:
    - run_integration_tests.sh
```

Listing 4.7: Example of a `.gitlab-ci.yml` configuration file

4.6.3 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 Scaling Heat

The Scaling Heat algorithm is a decision making algorithm to determine if a scaling action is necessary. It was introduced by Barna et al. [BKFL17] to overcome issues of traditional recurrence factor algorithms [BKFL17].

4.7.1 Recurrence Factor

In an Autonomic Computing environment, a scaling decision is made in each interval after data has been retrieved from the monitoring system (see Section 2.3 for the Autonomic Computing architecture). Sudden performance spikes can occur and can cause the decision algorithm to perform unnecessary scaling actions. These unnecessary scaling actions can have a negative impact on the overall computing performance. To overcome this issue, a recurrence factor needs to be introduced to the decision making algorithm. With a recurrence factor (n), a scaling action will only be performed until a performance threshold has been violated n times [BKFL17].

Traditional recurrence factor algorithm require violations to occur regularly. If a performance violation of the opposite direction occurs, the algorithm can get stuck in the violation process. Therefore, no scaling actions will be performed [BKFL17].

4.7.2 Scaling Heat Algorithm Concept

Algorithm 1: Scaling Heat decision making algorithm[BKFL17]

```
Input: utilization - The retrieved utilization for a performance
       metric
Input: lower_threshold and upper_threshold - Range limit of the
       performance metric
Input: heat - Current heat value of a performance metric. Indicating
       if a scaling action is necessary
Output: heat - New heat value for the next iteration
if utilization ≥ upper_threshold then
    // cluster overload
    if heat < 0 then
        // reset heat for removal
        heat ← 0;
    heat ← heat + 1;
else if utilization ≤ lower_threshold then
    // cluster overload
    if heat > 0 then
        // reset heat for adding
        heat ← 0;
    heat ← heat − 1;
else
    // utilization is within threshold range
    // move heat towards 0
    if heat > 0 then
        heat ← heat − 1;
    else if heat < 0 then
        heat ← heat + 1;
end
if heat = n then
    Perform a scale-out action;
    heat ← 0;
else if heat = −n then
    Perform a scale-in action;
    heat ← 0;
return heat
```

Algorithm 1 introduces the Scaling Heat algorithm. The algorithm is based in a concept called *heat*. The value of heat indicates if a scaling action of removing or adding components is necessary. If the given utilization of a performance metric violates the upper threshold, the heat value will increase. Violations of the lower threshold will cause a decrease respectively. When the heat reaches the recurrence factor n, positive for adding and negative for removing nodes, a scaling action will be executed. After executing a scaling action, the heat value will be set to 0 [BKFL17].

4.8 Kubernetes Horizontal Pod Autoscaler

Kubernetes Horizontal Pod Autoscaler (KHPA) is an auto-scaling algorithm used in Kubernetes. Kubernetes is an orchestration tool that allows to create and deploy units called Pods. A Pod is a running process on a cluster that encapsulates an application. KHPA scales the number of replicas per Pod based on the utilization of performance metrics. The algorithm is based on a control loop. Each n seconds, the algorithm gathers performance metrics and computes the target number of replicas to achieve the desired utilization of a performance metric [CP17].

The algorithm computes the number of replicas for a single performance metric. If a scaling action depends on multiple performance metrics, the number of replicas has to be computed for each performance metric. The largest number of replicas is used as the target number of replicas [Thec].

KHPA takes as input the number of active replicas for a pod ($active\_replicas$), the utilization of the performance metric of each replica ($pod\_utilization$), and the target utilization of the performance metric ($target\_utilization$). The formula to compute the target number of pods P is defined by [Thec]:

$$P = \left\lceil active\_replicas \times \left(\frac{\sum pod\_utilization}{target\_utilization} \right) \right\rceil \tag{4.1}$$

Chapter 5

Conceptual Design

5.1 Design Restrictions

TODO: Describe Chapter

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: To distribute the workload of training ML models, Apache Spark is a requirement.
- Python programming language: Python is 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.
- GitLab: All source code is hosted on a GitLab repository. Additionally, the CI pipeline runs on GitLab CI/CD (introduced in SECTION X).

5.2 Automated Deployment Pipeline

The objective of this thesis is to automatically submit an Apache Spark application to the Apache Spark cluster to train ML models. Therefore, the training process has to be integrated into the application development lifecycle. The source code of an Apache Spark application is hosted on a Git repository. After each committed change on the repository, the pipeline is triggered.

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

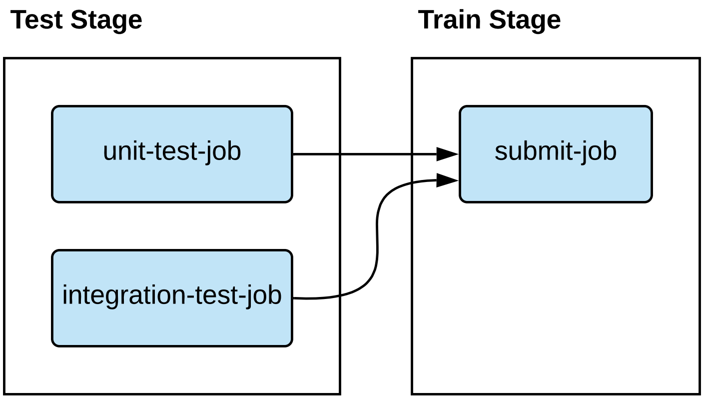

Figure 5.1: Automated Deployment Pipeline concept

Figure 5.1 illustrates the conceptual design of the Automated Deployment Pipeline. The pipeline consists of two different stages:

1. Test stage: After the build stage has succeeded, the test stage will perform tests.
2. Train stage: If the tests have been successful, the application is submitted to the Apache Spark cluster for training.

It is important to mention, that a build stage is missing in this conceptual design. The build stage includes compiling source code into a format that can be executed directly. Python is an interpreted language and therefore no compilation is needed to execute the source code. As being mentioned in Section SPARK, Apache Spark supports different languages than Python. For example, for Java application, a build stage is needed to compile the source code to a .jar binary, which can be submitted to the Apache Spark cluster.

TODO: Quelle

5.2.1 Test Stage

In order to detect error in an early stage, the source code has to be tested. The test stage is responsible to the the source code within a set of various tests. Tests can include:

- Unit tests:
- Integration tests:
- End-to-end tests:

For each different test, a new job in the test stage is being created. All jobs will perform in parallel after the test stage has been triggered. If a job has failed, the whole test stage is marked as failure and all participating developers will get a notification.

5.2.2 Train Stage

The train stage is responsible to submit the Apache Spark application to the Apache cluster after the test state was successful. As being mentioned in SECTION XY, an Apache Spark application will be submitted to the Apache Spark cluster by creating a spark-submit Docker container in the same Docker swarm network. Therefore, after the train stage has been triggered a spark-submit container has to be deployed in the Apache Spark cluster to submit the latest version of the Apache Spark application.

To access the Apache Spark cluster Docker swarm network, the train stage has to be executed on the same machine. Therefore a GitLab runner performing on the machine is needed. Additionally, the GitLab runner needs access to the underlying Docker engine to deploy new container to a given network. Figure 5.2 illustrates the steps to deploy a spark-submit container

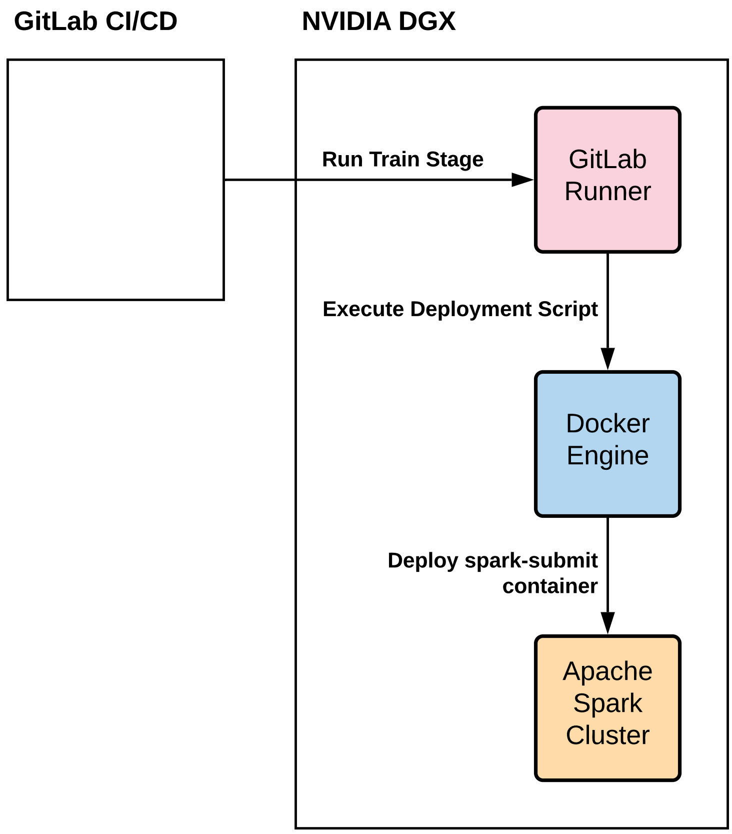

Figure 5.2: Deployment of a spark-submit container

in the Apache Spark cluster swarm network. GitLab CI/CD performs the train stage on the GitLab runner which is running on the NVIDIA DGX machine. The GitLab Runner executes the deployment script defined in the train stage. The deployment script executes a `docker run` command to deploy a spark-submit container in the Apache Spark cluster swarm network.

5.3 Identification of Suitable Metrics for Scaling

To scale the number of Apache Spark worker in accordance to the actively performing workload, suitable metrics to define the cluster performance have to be defined. With the RAPIDS accelerator for Apache Spark enabled, the Apache Spark cluster is able to utilize the computing power of GPUs and CPUs to enable parallization. Therefore, suitable metrics to measure the performance are the overall CPU utilization across all worker and the GPU utilization of all available GPUs. These utilization metrics will be based on the time when the Aapche Spark cluster is actively performing computations.

5.3.1 CPU Utilization

All Apache Spark worker run on the same machine. Therefore, all available CPU cores on the machine will be shared across each running Apache Spark worker.

cAdvisor provides a performance called metric `container_cpu_usage_seconds_total`[2]. This metric provides the total amount of CPU seconds consumed by core per container. To calculate the overall CPU utilization for all Apache Spark worker, the value of the performance metric for each over a specific rate has to be summed up. Therefore, the CPU utilization (U_{CPU}) is defined by:

$$U_{CPU} = \sum_{n=1}^{ActiveWorker} container\_cpu\_usage\_seconds\_total_n \tag{5.1}$$

5.3.2 GPU Utilization

Two GPUs on the machine are available across all Apache Spark Worker. The dcgm_exporter agent provides the `bla_bla` performance metric. This metric returns the procentual utilization per GPU. Therefore, the overall GPU utilization (U_{GPU}) is defined by:

The system has a fixed number of GPUs to use.

$$U_{GPU} = \frac{\sum bla\_bla}{ActiveGPUs} \tag{5.2}$$

5.4 Computing Environment Architecture

The computing environment will be deployed on a single machine. The goal is to create a self.optimizing environment, according to the autonomic computing environment described in SECTION AB. The be able to scale

2 Monitoring cAdvisor with Prometheus - `https://github.com/google/cadvisor/blob/master/docs/storage/prometheus.md` (Accessed: 2021-01-21)

components in the environment, each components is deployed as a Docker container using Docker Swarm mode (see Section XY). Docker Swarm mode allows to define the number replicas per service an keeps track of it. The number of replicas can be adapted at runtime. This gives the environment an additional self-healing ability. To enable communication, all Docker container run in the same network.

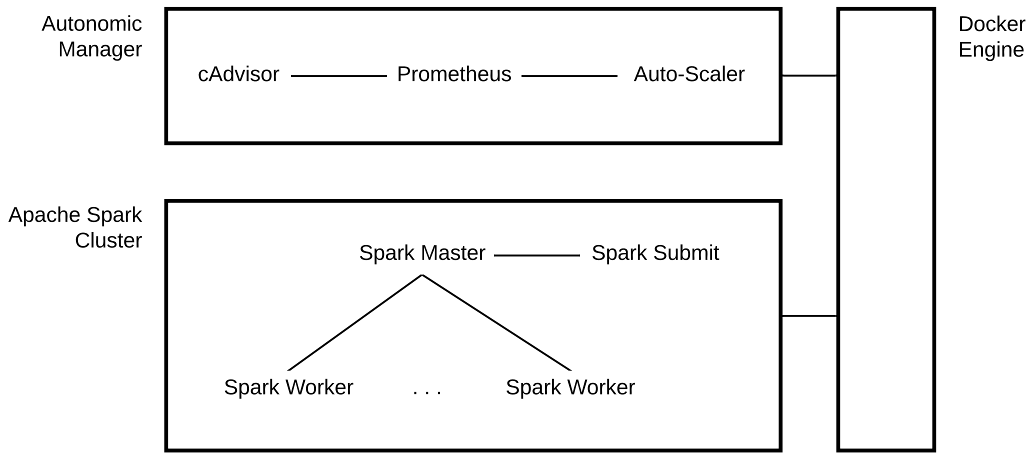

Figure 5.3: Overall cluster architecture - Source: Authors own model.

Figure 5.3 introduces the conceptual architecture of the computing environment. The computing environment consists of two main modules which consist of individual components:

- Autonomic manager: Is responsible to automatically monitor the hardware resource utilization of the environment and adapt the number of Apache Spark worker to reach a defined utilization.
- Apache Spark cluster: An Apache Spark cluster enables to distribute the workload of training ML models. Additionally it enables to perform computation parallel on multiple CPU cores and GPUs.

As being introduced in SECTION XY, an autonomic computing environment consists of an autonomic manager and managed resources. In this environment, all Apache Spark worker container are the managed resources.

5.5 Apache Spark Cluster

The Apache Spark cluster is the computing unit of the computing environment. It is responsible to distribute the workload of ML training applications. The Apache Spark is deployed in standalone mode. Standalone mode is the most simple mode without the need to install and configure an additional service as a cluster manager. An orchestration tool like Kubernetes or Apache Mesos as cluster manager is not needed, because Docker Swarm mode is used as orchestration tool. Each node in the cluster (Master, Worker, and

TODO: Figure hier

Submit) will be run as a Docker container. FIG XY illustrates the Apache Spark cluster architecture. It consists of a single master node, multiple worker nodes, and none or multiple spark-submit nodes. The number of worker nodes will be adapted by the Autonomic Manager in accordance to the utilization of defined performance metrics. A spark-submit node is only deployed until an application is being submit by the CI pipeline described in SECTION XY.

5.5.1 Homogeneous Apache Spark Worker Nodes

As being mentioned in SECTION AB, horizontal scaling is most efficient by scaling homogeneous node. To ensure each worker node is homogeneous, the same Docker image is used for all worker container. This guarantees that each worker has the same resources available and uses the same software as any other worker.

5.5.2 Deploying an Application with spark-submit

A spark-submit node is deployed each time an application is being submitted to the cluster. The purpose of a spark-submit node is to submit an application with the spark-submit executable to the cluster (described in SECTION XY). Standalone mode does not support to submit a Python application with the spark-submit executable from outside of the cluster. Therefore, a node running the spark-submit executable has to be submitted within the cluster [Theb]. To submit and application to the master node, the spark-submit node needs to be in the same Docker swarm network. The node is deployed as a Docker container instead of a Docker service. Each spark-submit node is deployed with a different setting depending on the configuration and application from the CI pipeline. After the application has been submitted, the spark-submit node automatically exits.

5.5.3 GPU Acceleration with RAPIDS

One objective of this thesis is, to enable GPU acceleration for Apache Spark. To do this, the RAPIDS accelerator plugin for Apache Spark is being used. Each worker becomse the same amount of GPU resources.

5.6 Autonomic Manager

The autonomic manager is a main module of the computing environment. The theoretical concept of an autonomic manager is described in SECTION XY. It is responsible to monitor the performance metrics (introduced in SECTION AB) of all Apache Spark worker nodes and automatically scale the number of worker nodes to adapt to a specified performance goal. The autonomic manager will be implemented according to the MAPE architecture as described in SECTION XY. To create a complete control-loop, the autonomic manager is composed of multiple components as illustrated in Figure 5.4. It consists of a monitoring system (explained in SECTION XY)

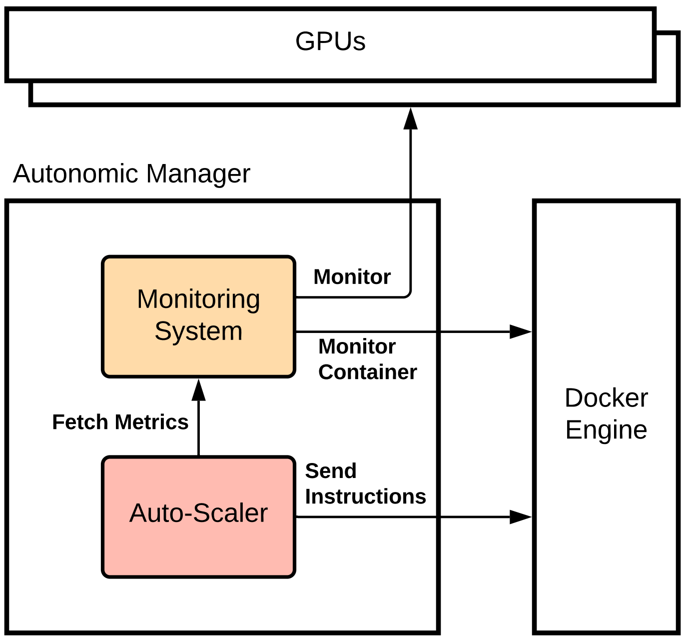

Figure 5.4: Autonomic manager component design - Source: Authors own model.

and an *Auto-Scaler* module. Each component in the computing environment is deployed as a Docker service or container. Therefore, the autonomic manager needs access to the Docker engine to send instructions and monitor performance metrics of Docker container. As being mentioned before, a fixed number of GPUs will be available on the machine. These GPUs need to be monitored on the node level instead of tthe container level. To monitor the GPU utilization, the autonomic manager needs access to the GPUs as well.

5.6.1 Monitoring System

The monitoring system is a main module of the autonomic manager. The tasks of a monitoring system are, to monitor the performance of components in the environment (see SECTION XY). In this environment, the monitoring system collects the performance metrics of the Apache Spark worker Docker container and the GPU performance. It is important to mention that the number of worker nodes varies over time. The Auto-Scaler will scale the replicas of worker nodes according to the system performance. Therefore, it is responsible to perform the Monitor phase of the MAPE architecture. The number of worker node varies over time Figure 5.5 illustrates the architecture of the monitoring system. It consists of three components:

- dcgm-exporter: A monitoring agent which is responsible to collect GPU performance metrics.

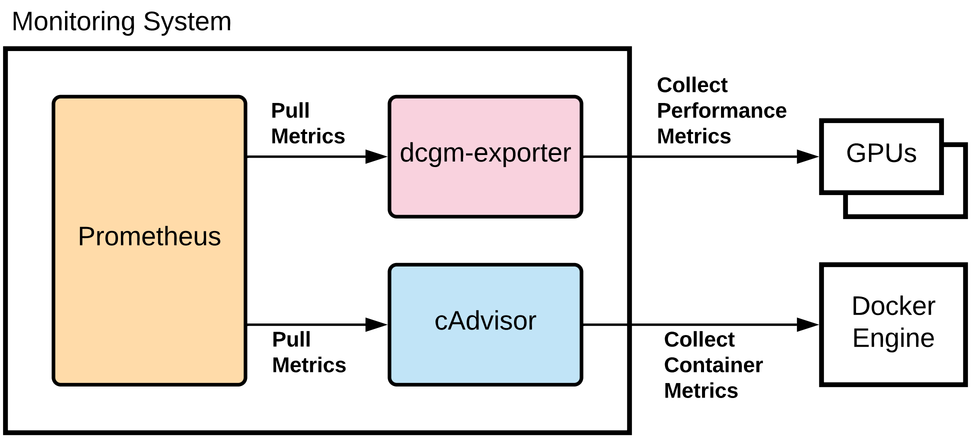

Figure 5.5: Autonomic manager component design - Source: Authors own model.

- cAdvisor: cAdvisor is monitoring agent which collects performance metrics of Docker container in the environment.
- Prometheus: Prometheus collects the performance metrics from all monitoring agents and saves them as time-series data in a time-series database.

Each component will be deployed as a Docker service in the overall Docker swarm.

5.6.2 Auto-Scaler

The *Auto-Scaler* is the second module of the autonomic manager and responsible to dynamically adjust the replicas of Apache Spark worker nodes in the computing environment to accommodate specified performance goals. It implements the Analyse, Plan, and Execute phases of the MAPE architecture. In addition to the monitoring system, the Auto-Scaler creates a complete autonomic manager implementing all four phases of the MAPE architecture. The *Auto-Scaler* is a reactive auto-scaler (described in SECTION XY) and used a threshold-based algorithm to adapt the number of worker nodes. As illustrated in Figure 5.4, the *Auto-Scaler* fetches performance metrics from the monitoring system. To fetch metrics from the monitoring-system, it connects to the HTTP API of Prometheus. After it received the performance metrics, the Auto-Scaler analyses the metrics, plans scaling-actions to adjust the number of worker replicas and sends instructions to the Docker engine. The *Auto-Scaler* can be configured using a configuration file.

MAPE Phases

As being mentioned, the *Auto-Scaler* implements the Analyse, Plan, and Execute phases of the MAPE architecture. Each phase has a different workflow to accommodate its goal.

Analyse: In order to determine if a scaling-action is necessary, the *Auto-Scaler* has fetch and analyse all defined performance metrics. During each period, the *Auto-scaler* fetches performance metrics from the Prometheus HTTP API with specified PromQL queries. 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 and analyse performance metrics from Prometheus.

Plan: If a scaling-action is necessary, the *Auto-Scaler* is responsible to determine the number of Apache Spark worker replicas, needed to reach the utilization goals. A scaling plan consists of instructions to add or remove worker nodes. These instructions are send to the Docker engine. To calculate the number of worker nodes, the *Auto-Scaler* uses the Kubernetes Horizontal Pod Auto-Scaling algorithm (introduced in SECTION XY). In addition, the Auto-Scaler needs to check if the estimated number of worker replicas violate the specified upper- and lower-thresholds of active worker nodes.

Execute: After a scaling plan is created, the Auto-Scaler needs to send the instructions to the Docker engine. After scaling the number of worker replicas, the Apache Spark cluster needs time for changes to take effect. Therefore, a cooldown period is activated after each scaling action (explained in Section AB). During the cooldown period, no scaling actions are executed.

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.

5.6.3 Control Loop

Figure 5.6 provides an overview about the complete control loop The monitoring system monitors performance metrics from Docker containers and the GPUs on the machine. Next, the Auto-Scaler analyses the performance metrics, creates scaling plans and sends instructions to the Docker engine to scale the Apache Spark cluster.

The control-loop workflow is illustrated in Figure 5.7. It starts in the Monitor phase. All monitoring agents (cAdvisor and dcgm-exporter) collect performance metrics from their targets. Next, Prometheus pulls the metrics from all monitoring agents and saves the data in its time-series database. In the analyse phase, the Auto-scaler determines if a scaling action is necessary. If a scaling action is not needed, the workflow ends and starts again in the Monitor phase in the iteration. Otherwise if a scaling action is needed, the Auto-Scaler determines the number of Apache Spark worker replicas in the Plan phase. Lastly, in the Execute phase, the Auto-scaler scales the Apache Spark worker nodes.

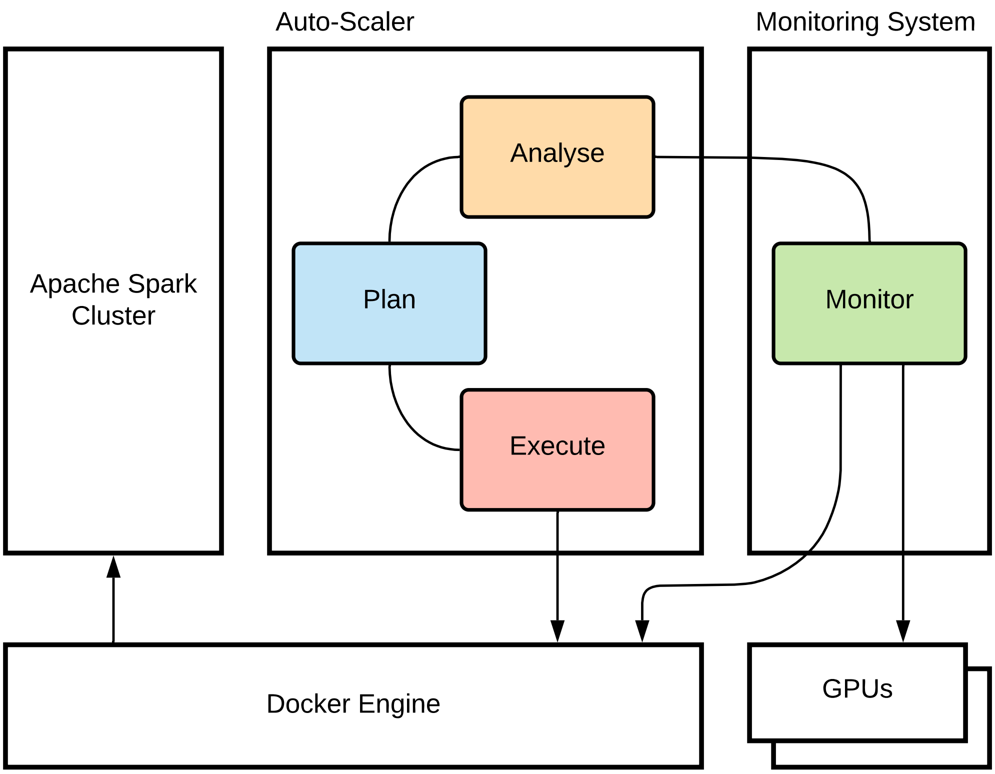

Figure 5.6: Full MAPE control loop architecture

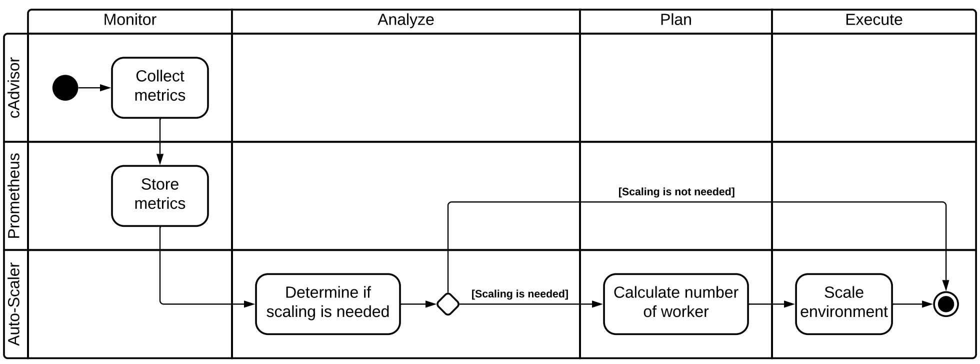

Figure 5.7: UML activity model of the autonomic manager process - Source: Authors own model.

Chapter 6

Implementation

This chapter explains the implementation process of the conceptual detail introduced in Chapter 5.

6.1 General

To implement the introduced conceptual design, several requirement on the host machine have to be provided.

6.1.1 The Host Machine

The host machine is NVIDIA DGX. It is important to mentioned that the machine is a shared live system. Many applications from different departments are performing on the machine. All application are sharing the same resource on the machine.

The hardware specification of the machine is the following:

- 11GB RAM
- 300 TB DIsk space
- 8x NVIDIA Tesla GPU a 32GB
- 500x Intel CPU COre 29876

The following software is installed on the machine:

- Ubuntu 30.5
- Docker 19.5
- NVIDIA Docker runtime 1

- Docker Swarm - Each component as a Docker file to create services - Nvidia runtime to use GPUs

6.1.2 Problem Statement

TODO: besser erklären

To deploy Docker container with access to GPU resources, the NVIDIA runtime for Docker has to be installed. The NVIDIA runtime for Docker allows to access GPU resources from inside a Docker container. The NVIDIA runtime has to be enabled as default runtime for Docker to use it in Docker services. Just install the following and set it in the file XY. The NVIDIA runtime 4.x is already installed on the host machine but is not set as the default runtime. Changing the default runtime requires a restart of the Docker service. Restarting the Docker service is not possible because it requires to quit all running application on the system which require Docker. Therefore, components which require access to GPU resources (Apache Spark worker nodes and the dcgm-exporter) cannot be deployed as Docker services. The solution to this problem is, to deploy these components as Docker container. The docker run command provides a runtime attribute to set the NVIDIA runtime as default runtime for this container.

LST AB provides an example about deploying a Docker container with the NVIDIA runtime enabled.

Given this problem statement, the Auto-Scaler has to be implemented to deploy Apache Spark worker Docker container instead of scaling the replicas of the worker service.

6.2 Auto-Scaler

The Auto-Scaler is a main module of the autonomic manager. It is responsible to analyse performance metrics, plan scaling actions in accordance to the performance metrics and execute scaling actions to adapt the number of Apache Spark worker in the computing environment. It will be implement after the conceptual design described in SECTION XY.

6.2.1 Technical Background

The Auto-Scaler is implement in Python 3.8. It consists of different classes, each having different responsibilities. The following Python libraries have been used for the implementation:

- aiohttp[1]
- APScheduler[2]
- docker[3]
- PyYAML[4]

1 `https://pypi.org/project/aiohttp/` (Accessed: 2021-01-26)
2 `https://pypi.org/project/APScheduler/` (Accessed: 2021-01-26)
3 `https://pypi.org/project/docker/` (Accessed: 2021-01-26)
4 `https://pypi.org/project/PyYAML/` (Accessed: 2021-01-26)

To make asynchronous HTTP calls, the aiohttp library was used. Asynchronous HTTP calls are important because the performance metrics have to analysed at runtime. The APScheduler is needed to periodically tasks. To interact with the Docker engine, Docker provides a Python SDK. The configuration for the Auto-Scaler is defined in YAML files. The PyYAML library is used to parse YAML files.

6.2.2 Configuration

The configuration parameter for the Auto-Scaler have been introduced in Section ??. The configuration for the Auto-Scaler will be specified in a YAML file. Listing 6.1 describes an example of an Auto-Scaler configuration. Overall, a configuration file is structured in three sections: General, metrics and worker. Table 6.1 lists all available configuration parameters. It describes the value type and the default value of each parameter. Some parameters are required to be defined by the administrator and have no default value.

General: The general section defines details about the scaling and heat algorithm and the Prometheus URL.

Metrics: Metrics is a list of performance metrics configuration parameters. The name of a metric (*cpu* and *gpu* in the example Listing 6.1) can be set by the administrator. A performance metric requires a query in the PromQL syntax. Additionally a target utilization is needed and the minimum and maximum utilization of the performance metric.

TODO: Multi-dimensional metrics/queries

Worker: To scale the replicas of the Apache Spark worker service, the name of the Docker service needs to be set. In addition, the minimum and maximum number of concurrent worker nodes needs to be defined to prevent an overhead of running worker nodes.

```
general:
  interval_seconds: 5
  cooldown_period_seconds: 180
  recurrence_factor: 3
  prometheus_url: "http://localhost:9090"

metrics:
  cpu:
    query: 'sum(rate(container_cpu_user_seconds_total{image
     ="spark-worker:3.0.1-hadoop2.7"}[30s]))'
    target_utilization: 0.5
    thresholds:
      min: 0.2
      max: 0.6

  gpu:
    query: 'sum(rate(container_cpu_user_seconds_total{image
     ="spark-worker:3.0.1-hadoop2.7"}[30s]))'
    target_utilization: 0.3
```

```
    thresholds:
      min: 0.2
      max: 0.6

worker:
  service_name: "computing_spark-worker"
  thresholds:
    min: 1
    max: 30
```

Listing 6.1: Auto-Scaler configuration YAML file

6.2.3 Scaling Apache Worker Nodes

The Auto-Scaler performs periodically. Each period it fetches performance metrics, analyses the metrics, plans scaling actions, and executes them if necessary. To perform tasks periodically, it uses the APScheduler library. This allows to perform a tasks each n seconds. In each task, the Analyse, Plan, and Execute phase is performed in order.

Estimation of Necessary Scaling Actions

The Scaling Heat algorithm, introduced in Section 4.7, is being used to estimate if a scaling action is necessary. The algorithm is being used because it will prevent the Auto-Scaler to perform unnecessary scaling actions. During each interval, after performance metrics have been received from

Name	Type	Default
general		
interval_seconds	Integer	1
cooldown_period_seconds	Integer	180
recurrence_factor	Integer	1
prometheus_url	String	*Required*
metrics		
query	String	*Required*
target_utilization	Float	0.5
thresholds		
min	Float	0.5
max	Float	0.5
worker		
service_name	String	*Required*
thresholds		
min	Float	0.5
max	Float	0.5

Table 6.1: Auto-Scaler configuration parameter

the monitoring system, a heat value will be calculated for each performance metric specified in the configuration under *metrics*. The algorithm uses a recurrence factor which has to be defined in design time. The Auto-Scaler configuration provides a parameter called *recurrence_factor* (see Table 6.1 for details).

To store and calculate the heat for each performance metric, a class called `HeatStore` was created. FIGURE XY describes the UML class diagram for the `HeatStore` Python class. The class can be used to retrieve, update and reset the heat for a list of performance metrics.

The Auto-scaler begins with the Analyse phase of the MAPE architecture. First it fetches the performance metrics from the specified Prometheus API. To make HTTP requests, the Auto-Scaler uses the aiohttp library. This library allow to make asynchronous HTTP requests. Performing asynchronous HTTP calls is important, because this does not block the current thread, the application is running on. This will not stop the loop. After receiving the performance metrics, the Auto-Scaler analyses the metric by checking if they violate the defined performance thresholds. If no performance violation has occured, the current task has finished and the Auto-Scaler for the next interval. Otherwise, the Auto-Scaler will perform the Plan phase. To execute the Plan phase, a set of conditions have to be given:

1. The Scaling Heat algorithm has to reach the recurrence factor
2. No cooldown period should be activated

Calculating the Number of Needed Worker Nodes

The KHPA algorithm will be used to calculate how many worker are needed to reach the target utilization (see SECTION XY for algorithm details). In this project, the calculation is done for the CPU and GPU utilization. The highest number of the desired worker node replicas is chosen.

In the Plan phase, the Auto-Scaler calculates the number of needed Apache Spark worker nodes using the KHPA algorithm.

```python
def calculate_number_of_needed_worker(active_worker: int,
    utilization: float,
    target_utilization: float):
  return math.ceil(
    active_worker * (utilization / target_utilization))
```

Listing 6.2: KHPA implementation using Python 3.8

Listing 6.2 shows the implementation of the KHPA algorithm in Python.

Performing a Scaling Action

Docker provides a Python library for the Docker Engine[5]. This library will be used to perform the swarm scaling action.

If worker need to be removed, it is necessary to check if the worker are running any applications. Removing a worker while an application is performing will cause the cancellation of the application. To check if applications

After a scaling action has been performed, a cooldown period will be applied. The cooldown period is needed because the number of desired worker nodes can keep fluctuating due to the dynamic nature of performance metrics.

After the number of desired Apache Spark worker has been calculated, the Auto-Scaler adds or removes worker container. During the execution of Apache Spark applications, it is not possible to remove worker nodes. Because of the bla bla. An alerantive approach is described in SECTION DISCUSSION. In each interval, the Auto-Scaler checks if an application is actively performing. To get this value, the Auto-Scaler uses the Prometheus HTTP API and sends the BBLA BLA PromQL query. Therefore, thr Apache Spark master node has to be defined as Prometheus target (described in SECTION XY). Docker SDK bla bla

6.2.4 Docker Image

- Dockerfile hier erklären - Wie den Auto-Scaler per konsole starten (mit config parameter)

6.3 Deployment of a Docker Swarm

6.3.1 Hardware

6.3.2 Software info

Hier tabelle mit versionen von eingesetzter software

6.3.3 Swarm

Vielleicht euch einfach das ganze kapitel Swarm nennen? - Dockerfile erläutern - GPUs (nur die 2 bestimmten)

5 Docker SDK for Python 4.4.1 Documentation - `https://docker-py.readthedocs.io/en/4.4.1/` (Accessed: 2021-01-05)

6.3.4 Build Script

6.3.5 Apache Spark Cluster with GPU Acceleration

The Apache Spark cluster is created in standalone mode, see Section 4.2.3 for deployment details. The cluster consists of a single Apache Spark master node and a dynamic number of Apache Spark worker nodes. The master and worker container will run as a service in a swarm (see Section 4.1.4). For submitting an application to the cluster, an individual container performing the spark-submit executable will be deployed. Each node runs in an independent Docker container.

Overall four Docker images are needed to create the Apache Spark cluster introduced in Section ??:

- Base image
- Master image
- Worker image
- Submit image

The master, worker and submit image require common packages to be installed and a set of common configuration. Therefore, an additional base image will serve as a base image.

Apache Spark Base Image Installation Details

The base image Dockerfile is available at Listing A.1. As parent, the base image uses the `nvidia/cuda:11.0-devel-ubuntu16.04` Docker image. The parent image runs Ubuntu[6] in version 16.04. Additionally the CUDA Toolkit and the NVIDIA GPU driver are already installed. Docker provides the ability to set build arguments. To be able to install a specific Apache Spark version, two arguments, can be set when building the Docker image. Apache Spark will be installed at `/opts/Spark`. This directory will be set as the working directory for the Docker image as well. Furthermore, required Ubuntu packages will be installed. This includes the Java Runtime Environment Version 8, which is a requirement for Apache Spark. To enable GPU acceleration on all Apache Spark nodes, the base image will install the compiled Java files for the RAPIDS plugin at `/opt/sparkRapidsPlugin` (for RAPIDS installation requirements, see Section 4.3.3). The .jar files can be downloaded in the maven repository. To enable Apache Spark to discover available GPUs, a GPU discovery script is needed (see Section 4.2.3 for details about resource allocation). This discovery script will be placed at `/opt/sparkRapidsPlugin` as well. The discovery script is introduced at Listing A.4.

6 Enterprise Open Source and Linux - `https://www.ubuntu.com/` (Accessed: 2021-01-03)

Standalone Master and Worker Image

The master and worker image are build on top of the Apache Spark base image. Therefore, no additional installation steps are required. The master and worker nodes will be launched in standalone mode (see Section 4.2.3 for standalone mode details).

Master image: Implementation of the master node Dockerfile is available at Listing A.2. The master node image needs two ports configured: The Apache Spark service port and the port for the web user interface. The Apache Spark service port ist set to 7077 and the web user interface port to 4040. To start the master node in standalone mode, the start-master.sh launchable will be set as image entrypoint which requires no additional arguments.

Worker image: Listing A.3 describes the implementation of the worker node Dockerfile. The port for the worker web interface will be exposed at 4041. To start the worker in standalone mode, the start-slave.sh executable will be set as entrypoint for the image. The launch script requires the master node URI as a parameter. To keep the configuration simple, the environment variable HIER DIE VAR will be set in the compose file. Listing 6.3 describes the configuration environment for the worker. As mentioned previously (in Section 5.1), for this project two GPUs are available on the DGX workstation. Furthermore, the worker needs to know where to find the GPU resource discovery script.

```
SPARK_WORKER_OPTS="-Dspark.worker.resource.gpu.amount=2 -
    Dspark.worker.resource.gpu.discoveryScript=/opt/
    sparkRapidsPlugin/getGpusResources.sh"
```

Listing 6.3: Environment configuration for all worker nodes

Submit Image

As mentioned in Section 5.1, a requirements is, that Apache Spark application will be implemented in Python. Accordingly, the pyspark Python module needs to installed on all submit nodes. Apache Spark application will be placed at */opt/spark-apps*. SECTION CI describes how an Apache Spark application will be copied to a submit node. As entrypoint, the image will perform a custom submit script (available at LISTING AB). This script performs the spark-submit executable (usage described in detail in Section 4.2.3).

6.3.6 Autonomic Manager

As mentioned in Section **??**, the autonomic manager will consist of a monitoring system and the Auto-Scaler to create a complete control loop.

The monitoring system conceptual design was introduced in SECTION XY. It consists of a cAdvisor (SECTION XY) service and a Prometheus (SECTION XY) service. All modules will run as individual Docker services in the overall swarm. The following DOcker images will be used for the monitoring system:

- **cAdvisor:** google/cadvisor
- **Prometheus:** prom/prometheus

Prometheus Target Configuration

As mentioned in Section 4.4, Prometheus is a pull-based monitoring tool. It requires a list of targets to pull performance metrics from.

Listing 6.4 specifies the scrape configuration of the Prometheus system. The cAdvisor service is specified as a target. Prometheus will scrape every 5 seconds performance metrics from cAdvisor. All performance metrics will be labeled with the cAdvisor lable. The cAdvisor service is available at cadvisor:8080.

```
scrape_configs:
    - job_name: prometheus
      scrape_interval: 5s
      static_configs:
          - targets: ["localhost:9090"]

    - job_name: cadvisor
      scrape_interval: 5s
      static_configs:
          - targets: ["cadvisor:8080"]
            labels:
                group: "cadvisor"
```

Listing 6.4: Prometheus target configuration in YAML syntax

6.4 Automatic Deployment of Apache Spark Applications

Vielleicht eher das Kapitel so nennen - gitlab-ci.yml erklären - Screenshot von webui output

Chapter 7

Evaluation

Möglich: 1. Effizienz Auto-Scaler - Dynamisch skalieren vs statische Anzahl - > was ist schneller

2. GPU Worker Ab wievielen statischen Worker mit CPU ONLY wird dieselbe Leistung erreicht

TODO: Describe Chapter

7.1 Experimental Environment

The experiments have been conducted on a NVIDIA DGX. Table AB describes the hardware available on the DGX. Two of the eight GPUs have been available to conduct the experiments.

The DGX is a live-system as being mentioned in Section XY. Therefore not all available hardware resource have been exclusively available to conduct these experiments.

7.2 Workload

The performance evaluation of the implementation was conducted using two ML algorithms: K-Means and Naive Bayes. To implement the algorithms, Apache Sparks ML library has been used. The K-Means implementation is available at ANHANG A and the Naive Bayes implementation at ANHANG B.

The *infinite MNIST dataset*[LCB07] has been used to to perform the algorithms. It consists of 8 million datapoints.

7.2.1 K-Means

K-Means is a unsupervised machine learning algorithm for clustering. The algorithm aims to place unlabelled data, that appear to be related, into cluster.

In this section, we introduce perhaps the simplest unsupervised machine learning algorithms—k-means clustering. This algorithm analyzes unlabeled samples and attempts to place them in clusters that appear to be related. The k in "k-means" represents the number of clusters you'd like to see imposed on your data.The algorithm organizes samples into the number of clusters you specify in advance, using distance calculations similar to the k-nearest neighbors clustering algorithm. Each cluster of samples is grouped around a centroid—the cluster's center point. Initially, the algorithm chooses k centroids at random from the dataset's samples. Then the remaining samples are placed in the cluster whose centroid is the closest. The centroids are iteratively recalculated and the samples re-assigned to clusters until, for all clusters, the distances from a given centroid to the samples in its cluster are minimized. The algorithm's results are:

7.3 Efficiency of GPU Acceleration

7.4 Auto-Scaling using CPU Metrics

7.5 Results

Ab wie viele worker ist CPU besser wie GPU

Figure 7.1: A PGF histogram from `matplotlib`.

Chapter 8

Outlook

8.1 Optimizing Scaling

TODO: Describe Chapter

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

8.3 Pro Active Auto-Scaler

Nicht ganz sicher ob Pro Active hier das richtige Wort ist Punkt is: KHPA ist kacke frür Spark -> Reinforcement Learning besser. Noch besser: Gew. Anzahl an Worker bereit stellen und erst bei bedarf aktivieren. Docker up Zeit sparen

Chapter 9

Conclusion

9.1 Cluster architecture

TODO: Describe Chapter

Bibliography

[APW19] AGUERZAME, Abdallah ; PELLETIER, B. ; WAESELYNCK, François: GPU Acceleration of PySpark using RAPIDS AI. In: *DATA*, 2019

[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

[Bra18] BRAZIL, Brian: *Prometheus: Up & Running*. O'Reilly Media, Inc., 2018. – ISBN 9781492034148

[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

[DMA07] Duvall, Paul M. ; Matyas, Steve ; Andrew, Glover: *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007. – ISBN 9780321336385

[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

[FH10] Farley, David ; Humble, Jez: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Video Enhanced Edition.* Addison-Wesley Professional, 2010. – ISBN 9780321670250

[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

[Goo] Google LLC: *cAdvisor Github Repository.* `https://github.com/google/cadvisor`. – Accessed: 2020-12-31

[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

[KBE16] Khattak, Wajid ; Buhler, Paul ; Erl, Thomas: *Big Data Fundamentals: Concepts, Drivers & Techniques.* Pearson, 2016. – ISBN 9780134291185

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

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

[LBMAL14] Lorido-Botrán, Tania ; Miguel-Alonso, Jose ; Lozano, Jose: A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. In: *Journal of Grid Computing* 12 (2014), 12. http://dx.doi.org/10.1007/s10723-014-9314-7. – DOI 10.1007/s10723–014–9314–7

[LCB07] Loosli, Gaëlle ; Canu, Stéphane ; Bottou, Léon: Training Invariant Support Vector Machines using Selective Sampling. Version: 2007. http://leon.bottou.org/papers/loosli-canu-bottou-2006. In: Bottou, Léon (Hrsg.) ; Chapelle, Olivier (Hrsg.) ; DeCoste, Dennis (Hrsg.) ; Weston, Jason (Hrsg.): *Large Scale Kernel Machines*. Cambridge, MA. : MIT Press, 2007, 301-320

[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

[Ros17] Rossel, Sander: *Continuous Integration, Delivery, and Deployment*. Packt Publishing, 2017. – ISBN 9781787286610

[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 Kubernetes Authors: *Kubernetes Documentation*. https://kubernetes.io/docs/. – Accessed: 2021-01-28

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

[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

[xgb] xgboost developers: *XGBoost Documentation.* `https://xgboost.readthedocs.io/en/latest/index.html`. – Accessed: 2021-01-28

[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

Appendix A

Apache Spark Cluster Implementation

```
FROM nvidia/cuda:11.0-devel-ubuntu16.04

LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"

ARG SPARK_VERSION
ARG HADOOP_VERSION

# Install all important packages
RUN apt-get update -qy && \
    apt-get install -y openjdk-8-jre-headless procps python3 python3-pip curl

# Install Apache Spark
RUN mkdir /usr/bin/spark/ && \
    curl https://ftp-stud.hs-esslingen.de/pub/Mirrors/ftp.apache.org/dist/spark/spark-${SPARK_VERSION}/spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz -o spark.tgz && \
    tar -xf spark.tgz && \
    mv spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}/* /usr/bin/spark/ && \
    rm -rf spark.tgz && \
    rm -rf spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}/

# Add GPU discovery script
RUN mkdir /opt/sparkRapidsPlugin/
COPY getGpusResources.sh /opt/sparkRapidsPlugin/getGpusResources.sh
ENV SPARK_RAPIDS_DIR=/opt/sparkRapidsPlugin

# Install cuDF and RAPIDS
RUN curl -o ${SPARK_RAPIDS_DIR}/cudf-0.15-cuda11.jar https://repo1.maven.org/maven2/ai/rapids/cudf/0.15/cudf-0.15-cuda11.jar
RUN curl -o ${SPARK_RAPIDS_DIR}/rapids-4-spark_2.12-0.2.0.jar https://repo1.maven.org/maven2/com/nvidia/rapids-4-spark_2.12/0.2.0/rapids-4-spark_2.12-0.2.0.jar
ENV SPARK_CUDF_JAR=${SPARK_RAPIDS_DIR}/cudf-0.15-cuda11.jar
```

```
ENV SPARK_RAPIDS_PLUGIN_JAR=${SPARK_RAPIDS_DIR}/rapids-4-
    spark_2.12-0.2.0.jar

# Set all environment variables
ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
ENV SPARK_HOME /usr/bin/spark
ENV SPARK_NO_DAEMONIZE true
ENV PYSPARK_DRIVER_PYTHON python3
ENV PYSPARK_PYTHON python3
ENV PATH /usr/bin/spark/bin:/usr/bin/spark/sbin:$PATH

WORKDIR ${SPARK_HOME}
```

Listing A.1: Apache Spark base image Dockerfile

```
ARG SPARK_VERSION
ARG HADOOP_VERSION

FROM spark-base:$SPARK_VERSION-hadoop$HADOOP_VERSION

LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"

# Set ports
ENV SPARK_MASTER_PORT 7077
ENV SPARK_MASTER_WEBUI_PORT 4040

EXPOSE 4040 7077

# Start master-node in standalone mode
ENTRYPOINT [ "sbin/start-master.sh" ]
```

Listing A.2: Apache Spark master image Dockerfile

```
ARG SPARK_VERSION
ARG HADOOP_VERSION

FROM spark-base:$SPARK_VERSION-hadoop$HADOOP_VERSION

LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"

# Add spark-env
COPY spark-env.sh ${SPARK_HOME}/conf/spark-env.sh

# Set port
ENV SPARK_WORKER_WEBUI_PORT 4041

EXPOSE 4041

# Start worker-node
ENTRYPOINT ./sbin/start-slave.sh ${SPARK_MASTER_URI}
```

Listing A.3: Apache Spark worker image Dockerfile

```
#!/usr/bin/env bash

#
```

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

# This script is a basic example script to get resource information about NVIDIA GPUs.
# It assumes the drivers are properly installed and the nvidia-smi command is available.
# It is not guaranteed to work on all setups so please test and customize as needed
# for your environment. It can be passed into SPARK via the config
# spark.{driver/executor}.resource.gpu.discoveryScript to allow the driver or executor to discover
# the GPUs it was allocated. It assumes you are running within an isolated container where the
# GPUs are allocated exclusively to that driver or executor.
# It outputs a JSON formatted string that is expected by the
# spark.{driver/executor}.resource.gpu.discoveryScript config.
#
# Example output: {"name": "gpu", "addresses":["0","1","2","3","4","5","6","7"]}

ADDRS=`nvidia-smi --query-gpu=index --format=csv,noheader | sed -e ':a' -e 'N' -e'$!ba' -e 's/\n/","/g'`
echo {\"name\": \"gpu\", \"addresses\":[\"$ADDRS\"]}
```

Listing A.4: GPU discovery script - Source: `https://github.com/apache/spark/blob/v3.0.1/examples/src/main/scripts/getGpusResources.sh` (Accessed: 2021-01-03)