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, February 7, 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
$\hat{x}, \underline{\hat{x}}$	Mittelwert/-vektor
$x^*, \underline{x}^*$	Optimaler Wert/Vektor
$x_{0:k}, \underline{x}_{0:k}$	Folge von Werten $(x_0, x_1, \ldots, x_k)$ / Vektoren $(\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_k)$
$\mathbf{A}$	Matrizen
$\mathcal{A}$	Mengen
$\preceq, \prec$	schwache/strenge Präferenzrelation
$\mathbb{R}$	Reelle Zahlen
$\mathbb{N}$	Natürliche Zahlen
■	Ende eines Beispiels
□	Ende eines Beweises

Operatoren

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

Spezielle Funktionen

$\mathrm{Pr}(\mathcal{E})$	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 scripts: 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].
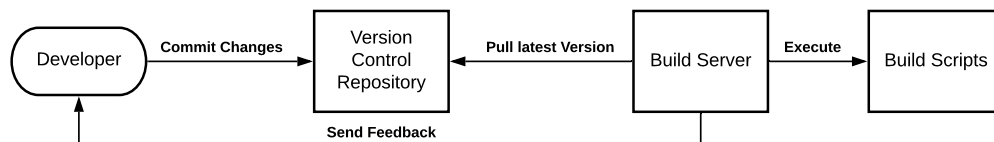
Figure 2.1: Continuous Integration Scenario - Source: Authors own model, based on [DMA07].

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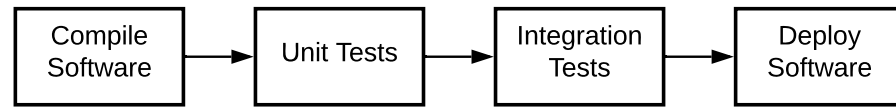
Figure 2.2: An example of a logical build script order for a CI process- Source: Authors own model, based on [DMA07].

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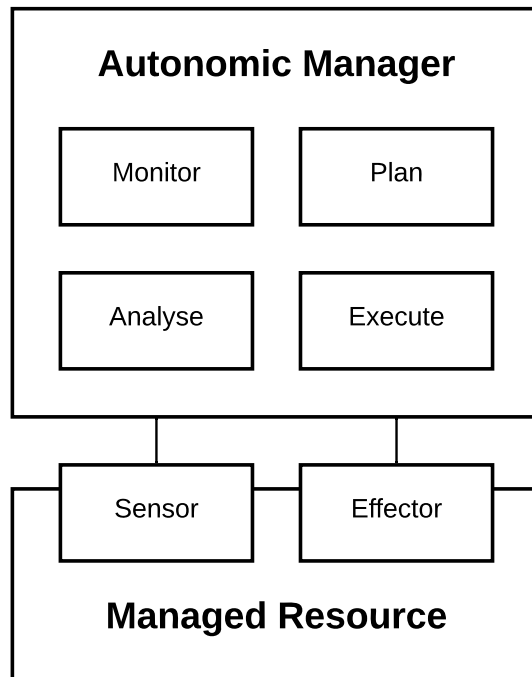
Figure 2.3: Autonomic computing concept - Source: Authors own model, based on [JSAP04].

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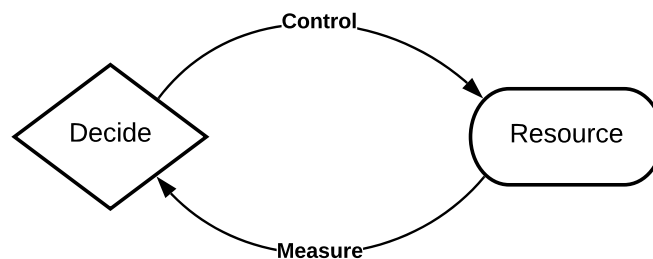
Figure 2.4: The control-loop concept - Source: Authors own model, based on [Mur04].

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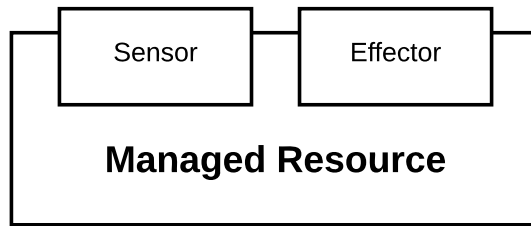
Figure 2.5: Managed resource - Source: Authors own model, based on [JSAP04].

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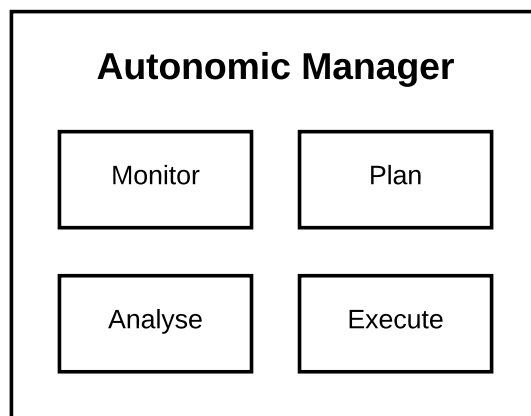
Figure 2.6: Autonomic manager - Source: Authors own model, based on [JSAP04].

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].
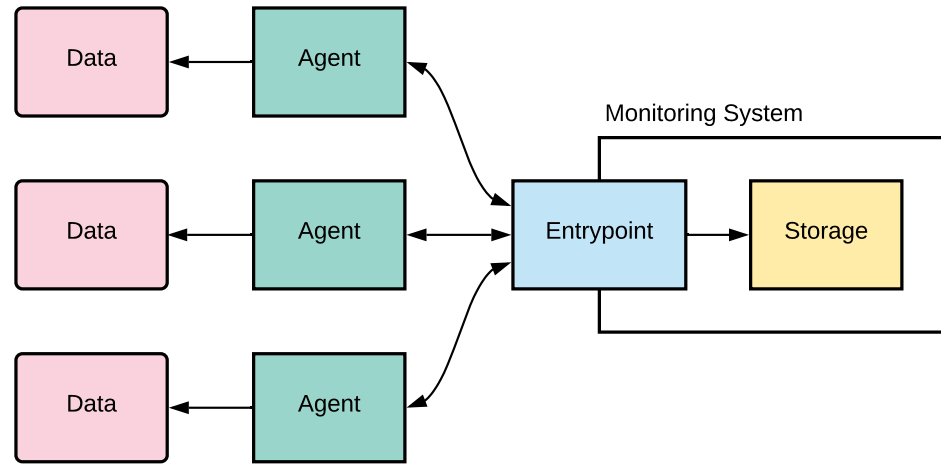
Figure 2.7: The monitoring process

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].
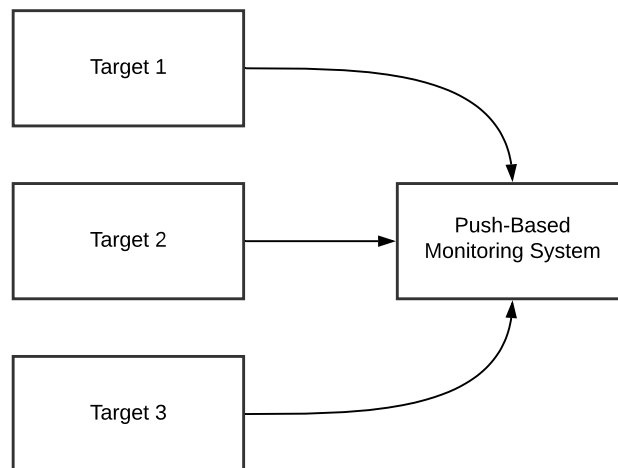
Figure 2.8: Push-based monitoring approach

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
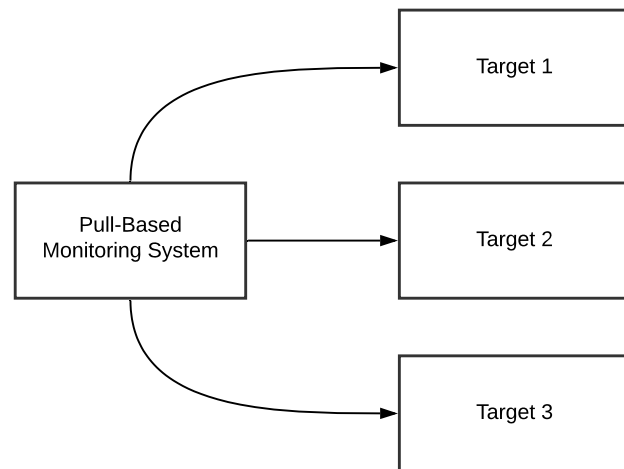
Figure 2.9: Pull-based monitoring approach

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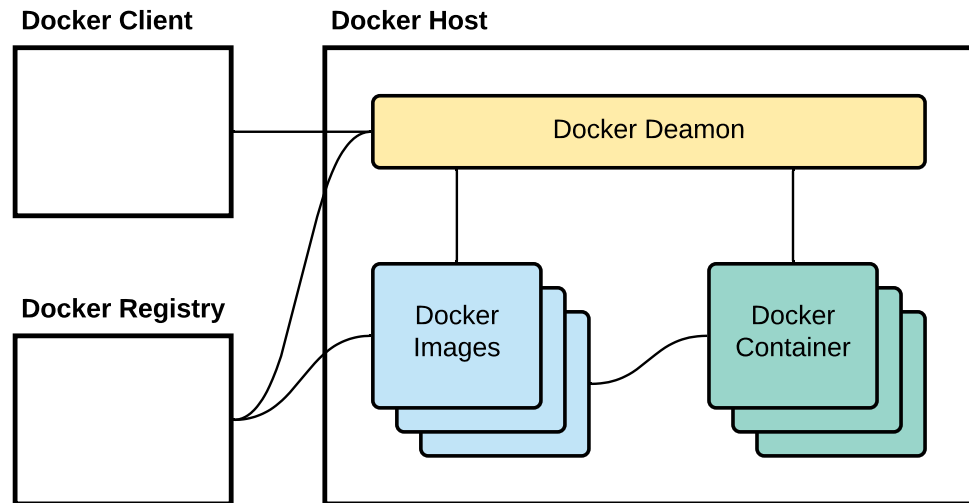
Figure 4.1: Docker architecture - Source: Authors own model, based on [Doc].

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.

```
1 FROM ubuntu
2
3 RUN apt-get update -qy && \
4     apt-get install -y nginx
5
6 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.
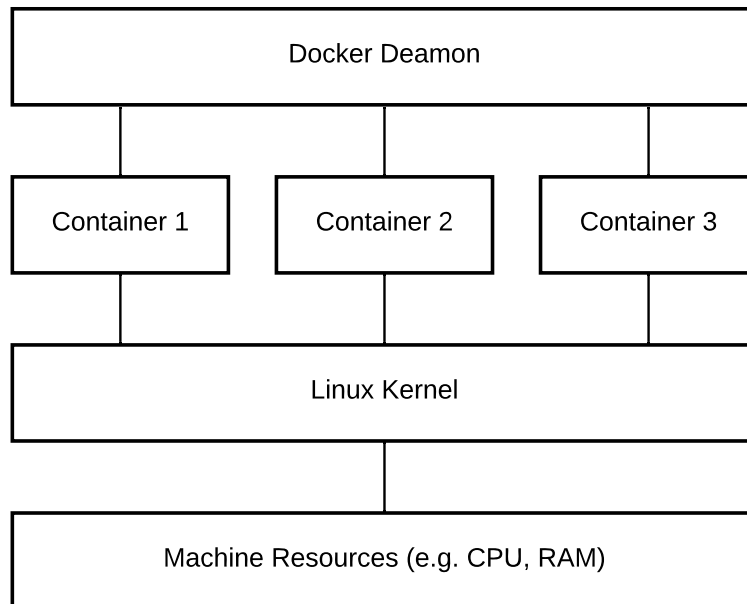
Figure 4.2: Docker basic container structure - Source: Authors own model, based on [BMDM20].

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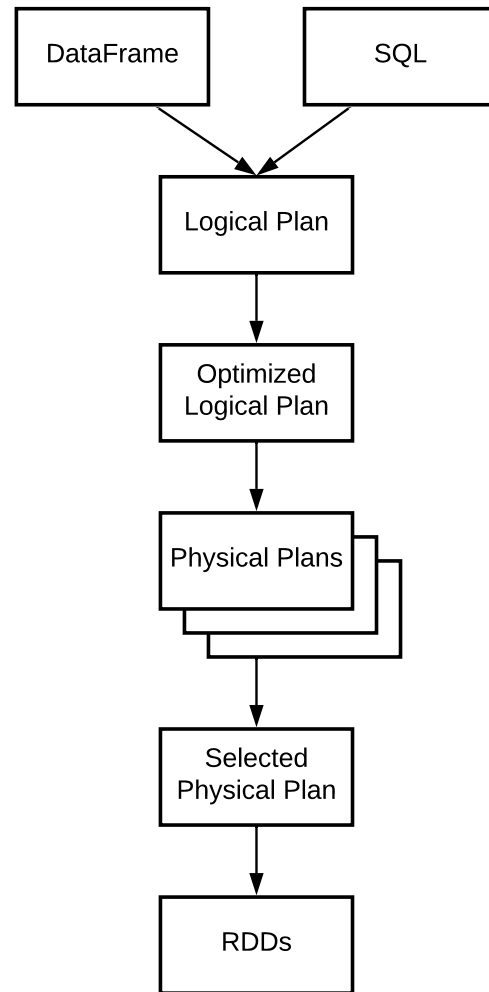
Figure 4.3: Optimization process of the Spark Catalyst - Source: Authors own model, based on [Luu18].

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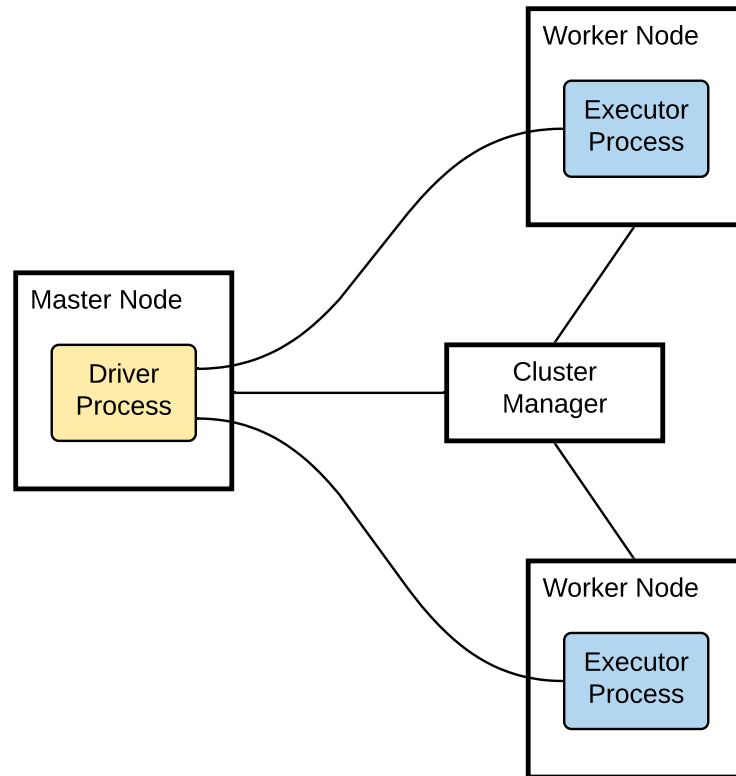
Figure 4.4: Overview of a Spark cluster architecture - Source: Authors own model, based on [Theb].

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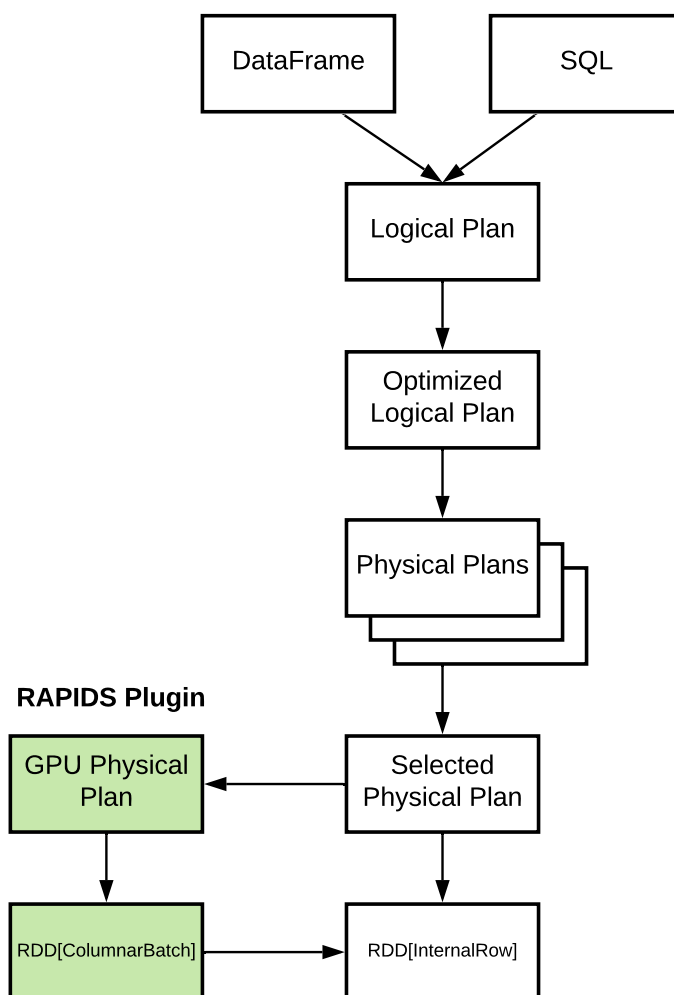
Figure 4.5: Catalyst optimization with RAPIDS accelerator for Apache Spark - Source: Authors own model, based on [McD20].

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

- XGBoost4j Java library

- XGBoost4j-Spark Java library

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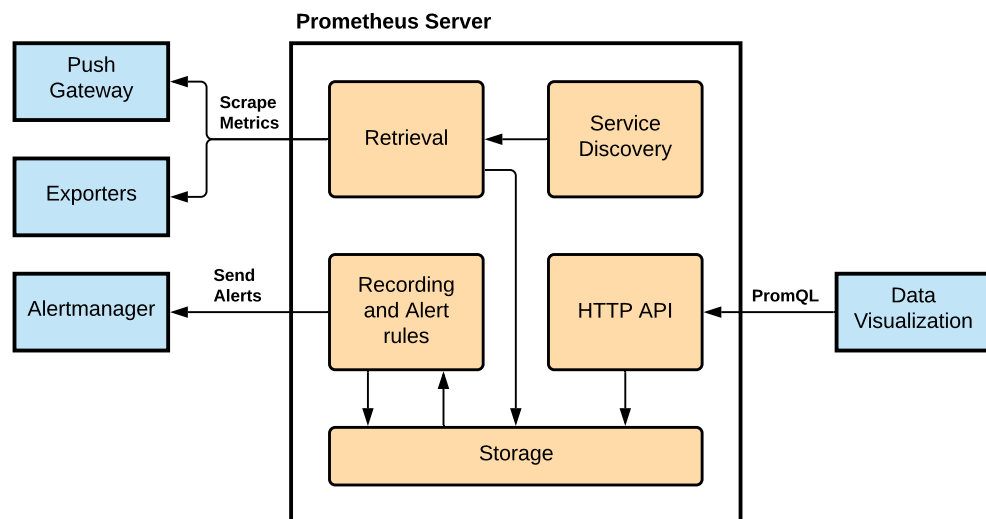
Figure 4.6: Prometheus high-level architecture - Source: Authors own model, based on [Thed, Bra18].

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.

```
1 global:
2   scrape_interval: 5s
3
4 scrape_configs:
5   - job_name: cadvisor
6     static_configs:
7       - targets: ["cadvisor:8080"]
8         labels:
9           group: "cadvisor"
10
11 rule_files:
12   - "/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)

```
1  groups:
2    - name: http_requests
3        - record: job:http_inprogress_requests:sum
4          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
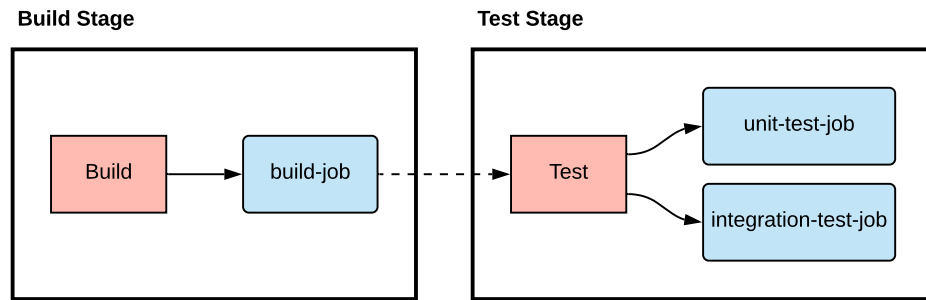
Figure 4.7: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

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.

```
1  stages:
2    - build
3    - test
4
5  build-job:
6    stage: build
7    script:
8      - build_software.sh
9
10 unit-test-job:
11   stage: test
12   script:
13     - run_unit_tests.sh
14
15 integration-test-job:
16   stage: test
17   script:
18     - 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

1 **if** *utilization* $\geq$ *upper_threshold* **then**
        // cluster overload
2 | **if** *heat* < 0 **then**
        | | // reset heat for removal
3 | | *heat* $\leftarrow 0$;
4 | *heat* $\leftarrow heat + 1$;
5 **else if** *utilization* $\leq$ *lower_threshold* **then**
        // cluster overload
6 | **if** *heat* > 0 **then**
        | | // reset heat for adding
7 | | *heat* $\leftarrow 0$;
8 | *heat* $\leftarrow heat - 1$;
9 **else**
        // utilization is within threshold range
        // move heat towards 0
10 | **if** *heat* > 0 **then**
11 | | *heat* $\leftarrow heat - 1$;
12 | **else if** *heat* < 0 **then**
13 | | *heat* $\leftarrow heat + 1$;
14 **end**
15 **if** *heat* $= n$ **then**
16 | Perform a scale-out action;
17 | *heat* $\leftarrow 0$;
18 **else if** *heat* $= -n$ **then**
19 | Perform a scale-in action;
20 | *heat* $\leftarrow 0$;
21 **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

In this chapter, the conceptual design of the implementation is introduced. This chapter's concept is based on the theory of Chapter 2 and uses the technologies introduced in Chapter 4.

5.1 Choice of Technologies

The following technologies are used to create the conceptual design of the implementation:

- Python programming language: Python is used as the main programming language for Apache Spark applications.

- Docker: Docker is used to deploying components in the computing environment as a container. This enables to create homogeneous nodes in the environment. Furthermore, Docker Swarm is used as an orchestration tool and takes care of each component's health status.

- GitLab: The source code repository is hosted on the GitLab platform. Furthermore, the automated deployment pipeline is designed using the CI/CD feature of GitLab. Therefore, using GitLab fulfils the version control and build server requirements mentioned in Section 2.2.2.

- Apache Spark: Apache Spark is used to distribute the workload of training machine learning applications across multiple workers. The goal is to scale the replicas of Apache Spark workers to increase the performance of the cluster.

- NVIDIA RAPIDS: In addition to scaling Apache Spark worker node replicas, the NVIDIA RAPIDS plugin suite is used to enable GPU acceleration on the Apache Spark cluster. Enabling Apache Spark to leverage GPUs increases the computational power as well.

- Prometheus: Prometheus fulfils all requirements of a monitoring system introduced in Section 2.5. It provides a powerful multi-dimensional

data-model and a query language for aggregations and filtering of multi-dimensional time-series data. Additionally, Prometheus is a pull-based monitoring system and includes a time-series database.

- cAdvisor: cAdvisor serves as a monitoring agent. It scrapes performance metrics from Docker containers. Prometheus pulls the data from cAdvisor to save it in its time-series database.

- dcgm-exporter: The dcgm-exporter is used as a monitoring agent as well. It is used to monitor the utilization of the available GPUs.

5.2 Identification of Suitable Metrics for Scaling

Suitable metrics are needed to measure the system performance while the Apache Spark cluster is actively performing work. With the RAPIDS accelerator for Apache Spark, the cluster utilizes GPUs' computing power and CPUs to enable parallelization. Therefore, suitable metrics to measure the system performance are the CPU utilization and the GPU utilization when the cluster is actively performing computations.

5.2.1 CPU Utilization

All Apache Spark workers run on the same machine. Therefore, all available CPU cores on the machine are shared across each Apache Spark worker. To get a value that indicates the CPU utilization between 0% and 100%, a metric is needed that represents the percentage of time all performing applications occupy CPU cycles. cAdvisor provides a performance metric called `container_cpu_usage_seconds_total`[1]. This metric provides the total amount of CPU seconds consumed by core of a container. To calculate the overall CPU utilization for all Apache Spark worker, the value of the performance metric for each worker container over a specific rate is summed up. In this case, the CPU utilization (U_{CPU}) is defined by (5.1) with $n \in \mathbb{N}$ the number of active workers.

$$U_{CPU} = \sum_{n=1}^{\mathbb{N}} container\_cpu\_usage\_seconds\_total_n \qquad (5.1)$$

5.2.2 GPU Utilization

In addition to the CPU utilization, Apache Spark workers can utilize GPUs to accelerate the computation work. The dcgm_exporter agent provides the `dcgm_fi_dev_gpu_util` performance metric. This metric returns the percentual utilization per GPU. Therefore, the GPU utilization (U_{GPU}) is defined by (5.2) with $n \in \mathbb{N}$ the number of available GPUs.

---

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

$$U_{GPU} = \frac{\sum_{n=1}^{\mathbb{N}} dcgm\_fi\_dev\_gpu\_util_n}{\mathbb{N}} \qquad (5.2)$$

5.3 Computing Environment Architecture

The computing environment is deployed on a single machine. The goal is to create a self-optimizing autonomic computing environment (described in Section 2.3). Furthermore, to manage the environment's components and resources, an autonomic manager is designed according to the MAPE architecture (introduced in Section 2.3.3). For simplicity, each node in the computing environment is deployed as a Docker service in a Docker swarm (introduced in Section 4.1.4). This allows to define the state of each service, which includes the number of replicas. The number of replicas can be updated during runtime.
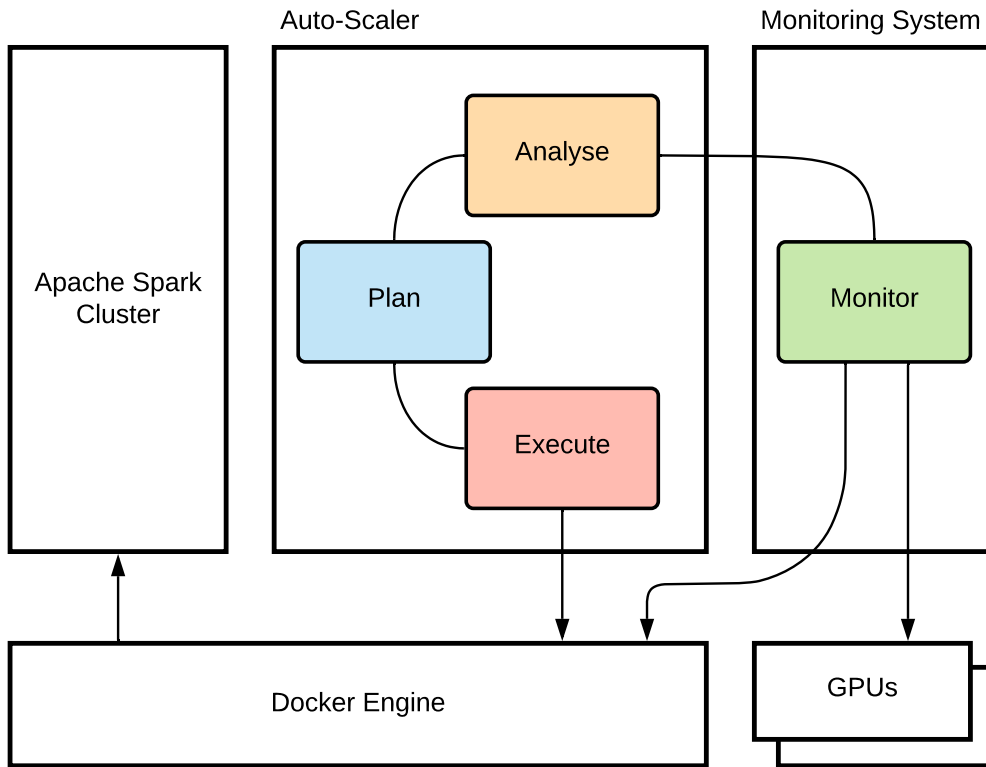
Figure 5.1: Full MAPE control loop architecture

Figure 5.1 provides an overview of the computing environment concept. The managed resources (introduced Section 2.3.2) in this environment are the Apache Spark cluster and the GPUs. These resources are managed by the autonomic manager, which consists of the *Auto-Scaler*, and the monitoring system. Furthermore, the autonomic manager implements all four phases of the MAPE architecture and executes the control-loop. As mentioned

previously, all components are deployed using Docker. Therefore, to manage components in the computing environment, the autonomic manager needs to interact with the overlaying Docker engine.
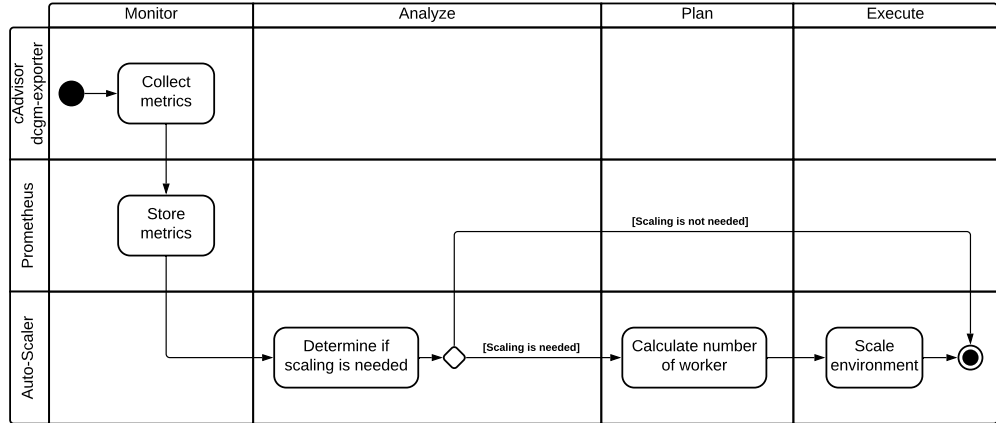
Figure 5.2: UML activity model of the autonomic manager process

Control-loop workflow: The control-loop workflow is illustrated in Figure 5.2. 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 iteration ends. 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 Docker service's replicas.

5.4 Apache Spark Cluster

The Apache Spark cluster is the computing unit of the computing environment. It is responsible for distributing the workload of training machine learning models. Figure 5.3 illustrates the architecture of the Apache Spark cluster. It consists of a master node, multiple worker nodes, and spark-submit nodes.

- Master: The master is responsible for distributing the workload of an application, submitted by a *spark-submit* node, across all available worker nodes. The Apache Spark application architecture is explained in Section 4.2.2.

- Worker: A worker node is responsible for performing the workload given by the master node. Furthermore, the replicas of active worker nodes in the cluster are dynamically adapted by the *Auto-Scaler* at runtime.

Apache Spark Cluster
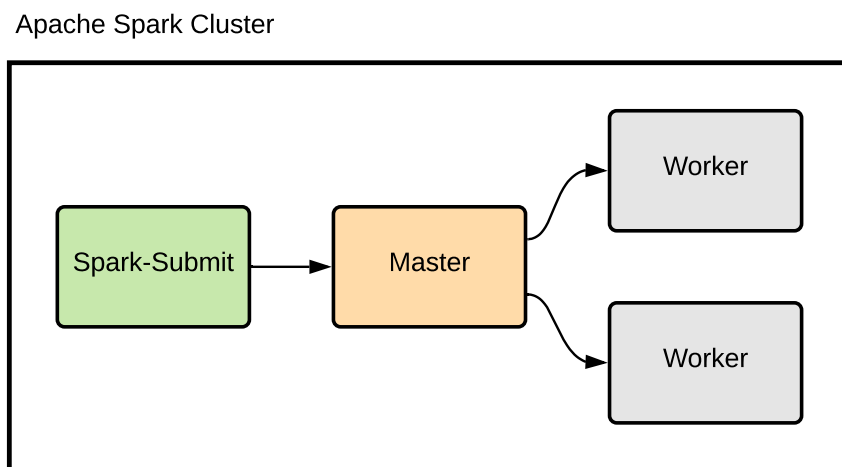
Figure 5.3: Apache Spark cluster architecture

- Spark-Submit: A *spark-submit* node is deployed whenever an application is submitted to the cluster.

The cluster is deployed in standalone mode (Section 4.2.3). Because Docker Swarm is used as the orchestration tool that controls nodes' state, there is no need to use another orchestration tool like Apache Mesos or Kubernetes as the cluster manager. Additionally, this simple approach allows running each node of the cluster in a Docker container without additional configuration.

5.4.1 Homogeneous Apache Spark Worker Nodes

The *Auto-Scaler* scales the replicas of Apache Spark worker nodes. Therefore, the cluster is scaled horizontally. As explained in Section 2.1.1, the horizontal scaling approach is more efficient when scaling homogeneous nodes Each node adds the same amount of computational power to the cluster. To ensure that the worker nodes are homogeneous, the worker Docker service uses the same Docker image for each worker node. The Docker image is created from a custom Dockerfile. This guarantees that each worker uses the same software and has the same amount of computational resources available as each other worker. Additionally, all requirements mentioned in Section 4.3.3 to enable GPU acceleration with RAPIDS are installed in a worker image.

5.4.2 Deploying an Application with spark-submit

Whenever the CI pipeline submits an application to the Apache Spark cluster, it deploys a *spark-submit* Docker container. The *spark-submit* container's purpose is to submit an application with the spark-submit executable (described in Section 4.2.3) to the cluster. Apache Spark cluster's standalone

mode does not support submitting a Python application with the spark-submit executable from outside of the cluster. Therefore, the spark-submit executable must be executed on the host machine with access to the master node [Theb]. To submit an application to the master node, the *spark-submit* container 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 container is deployed with a different setting depending on the configuration and application from the CI pipeline. This configuration can be set in the CI pipeline configuration. After the application has been submitted, the spark-submit node automatically exits. Additionally, the *spark-submit* container sets the number of resources for the executors running on the worker nodes.

5.5 Autonomic Manager

The autonomic manager is one of the main modules of the computing environment. It is responsible for monitoring all Apache Spark worker nodes' performance metrics and automatically scales 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 2.3.3. To create a complete control-loop, the autonomic manager comprises multiple components, as illustrated in Figure 5.4. It consists
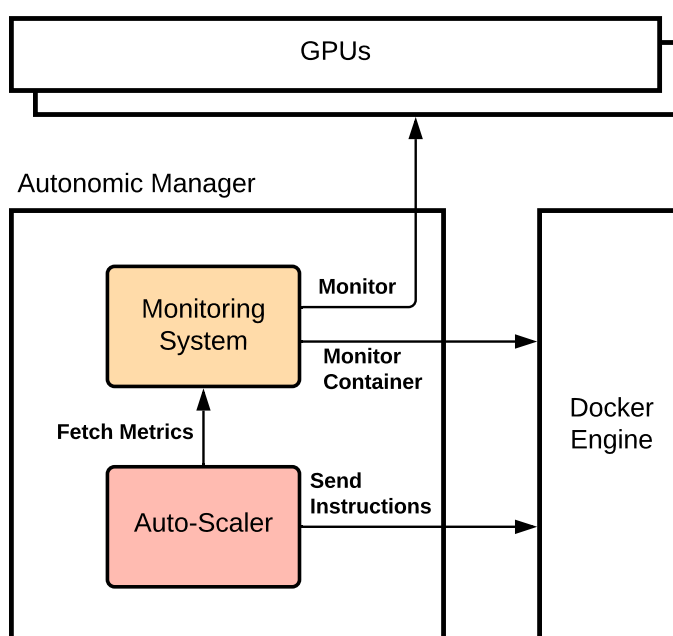
Figure 5.4: Autonomic manager component design

of a monitoring system (the theory is explained in Section 2.5) and an *Auto-Scaler* module. Each component in the computing environment is deployed as a Docker service. Therefore, Docker swarm takes care of the

health status of each component. To monitor Docker container's performance in the computing environment, the autonomic manager needs access to the overlaying Docker engine. Additionally, to monitor GPU performance, a monitoring agent is needed to scrape metrics from the GPUs. Furthermore, the autonomic manager is responsible for scaling the replicas of Apache Spark worker nodes. As being introduced in Section 5.4, a worker service takes care of all worker nodes. Therefore, the autonomic manager needs access to the host machine's Docker engine to send scaling instructions.

5.5.1 Monitoring System

The monitoring system is responsible for performing the Monitor phase. Therefore it monitors the performance of components in the computing environment and makes them available for the *Auto-Scaler*. In this environment, the monitoring system collects the Apache Spark worker Docker container's performance metrics and the GPU performance. It is essential to mention that the number of worker nodes varies over time because the *Auto-Scaler* scales the worker nodes' replicas according to the system performance. Fig-
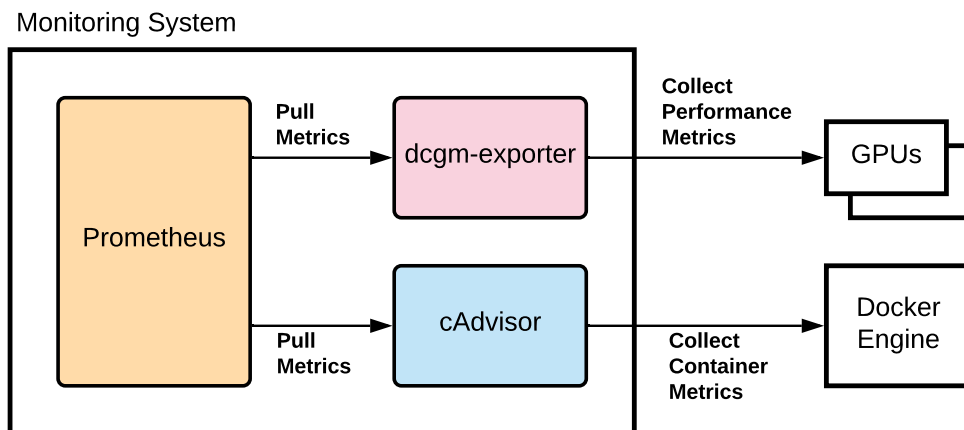
Figure 5.5: Monitoring system conceptual design

ure 5.5 illustrates the architecture of the monitoring system. It consists of three components:

- dcgm-exporter: A monitoring agent that is responsible for collecting GPU performance metrics.

- cAdvisor: cAdvisor is a monitoring agent that collects performance metrics of Docker containers 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.

5.5.2 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. Together with the monitoring system, it creates a complete autonomic manager implementing all phases of the MAPE architecture. The *Auto-Scaler* is designed as a reactive auto-scaler and uses a threshold-based algorithm to adapt worker nodes. To define the thresholds, it can be configured using a configuration file.

As illustrated in Figure 5.4, the *Auto-Scaler* fetches performance metrics from the Prometheus HTTP API. 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.

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: During each period, the *Auto-Scaler* fetches the performance metrics defined in the configuration from the Prometheus HTTP API. After the metrics are received, the *Auto-Scaler* determines if a scaling action is needed using the Scaling Heat algorithm (introduced in Section 4.7.2). If scaling is not necessary, the *Auto-Scaler* continues to collect and analyse performance metrics.

Plan: If a scaling-action is necessary, the *Auto-Scaler* is responsible for determining the number of Apache Spark worker replicas needed to reach the defined utilization goal. To calculate the number of worker nodes, the *Auto-Scaler* uses the Kubernetes Horizontal Pod Auto-Scaling algorithm (introduced in Section 4.8). The number of needed replicas may violate the upper or lower thresholds of active Apache Spark worker nodes. Then, the Auto-Scaler uses the maximum or the minimum number of worker nodes as the desired replicas.

Execute: After the number of needed replicas have been calculated, the *Auto-Scaler* sends the instruction to scale the Apache Spark worker service to the desired number of replicas. Afterward, a cooldown period gets activated. This brings the effect that the Apache Spark cluster needs time to distribute the workload across all new worker nodes efficiently. During the cooldown period, no scaling actions are executed.

Configuration

The *Auto-Scaler* needs specific configuration properties to collect the correct
metrics from Prometheus and scale the Apache Spark worker service replicas.
The following are properties that have to be defined to ensure that the
Auto-Scaler can collect meaningful metrics and scale Apache Spark workers
as expected.

- General properties:
  - Interval seconds: The number of seconds after the *Auto-Scaler*
    starts a new process.
  - Cooldown period: The duration in seconds the *Auto-Scaler* has
    to wait after a scaling action is performed.
  - Recurrence factor: To prevent too many scaling actions, the
    autonomic manager should only execute a scaling action, if the
    utilization thresholds are violated n times.
  - Prometheus URL: The URL of the Prometheus HTTP API.

- Metrics: It should be possible to define a list of multiple metrics. Each
  metric needs to have a variety of properties configured.
  - Target utilization: The target utilization of a performance metric
    is needed by the KHPA algorithm to calculate the number of
    desired replicas.
  - Utilization thresholds: To determine if a scaling action is needed,
    the scaling heat algorithm needs the minimum and maximum
    utilization of the performance metric.
  - Query: A PromQL query defines a performance metric.

- Apache Spark worker properties:
  - Worker service name: The Docker worker's service name is needed
    to update the number of replicas.
  - Worker thresholds: The maximum and the minimum number of
    concurrent Spark workers should be defined. To avoid the cold
    start effect, the minimum amount of workers should be at least 1.
  - Apache Spark master URI: To distribute the Spark Worker work-
    load, all Spark Worker need to communicate with the Spark
    master.

5.6 Automated Deployment Pipeline

This thesis aims to train Machine Learning models by automatically sub-
mitting Apache Spark applications to an Apache Spark cluster. Therefore,
the training process of a model has to be integrated into the application
development lifecycle. The application development lifecycle is automated

using a deployment pipeline (introduced in Section 2.2). In this conceptual design, a CI pipeline is demonstrated to automate the training phase of an application and submits the application to an Apache Spark cluster after the testing was successful.
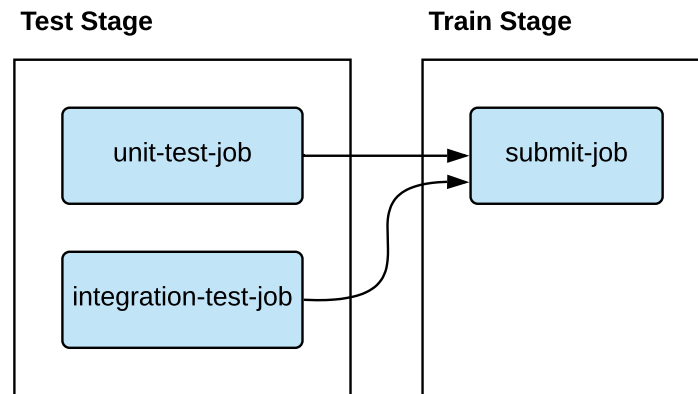
Figure 5.6: Automated Deployment Pipeline concept

Figure 5.6 illustrates the conceptual design of the CI pipeline, which consists of two stages:

1. Test stage: Responsible for performing all tests to validate the application source code.

2. Train stage: If the test stage has been successful, the application is submitted to the Apache Spark cluster to train the model.

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 does not need to be compiled for execution.

5.6.1 Test Stage

Before training the machine learning model, the application source must be validated by a series of tests. Tests can include:

- Unit tests

- Integration tests

- End-to-end tests

Each test is being performed in its own job. All jobs perform in parallel after the test stage has been triggered. If a job fails, the whole test stage is marked as a failure. All participating developers will get a notification.

5.6.2 Train Stage

The train stage is responsible for submitting the Apache Spark application to the Apache cluster after the test state was successful. A *spark-submit* Docker container has to be deployed to the same Docker network to submit an application to the Apache Spark cluster. How a *spark-submit* container performs, an application is explained in Section 5.4.2.

Deploying *spark-submit* Docker containers in the Apache Spark cluster network requires access to the overlying Docker engine. To access the Docker engine within a job of the train stage, the job has to be executed by a GitLab runner on the same host machine. Figure 5.7 illustrates the steps to
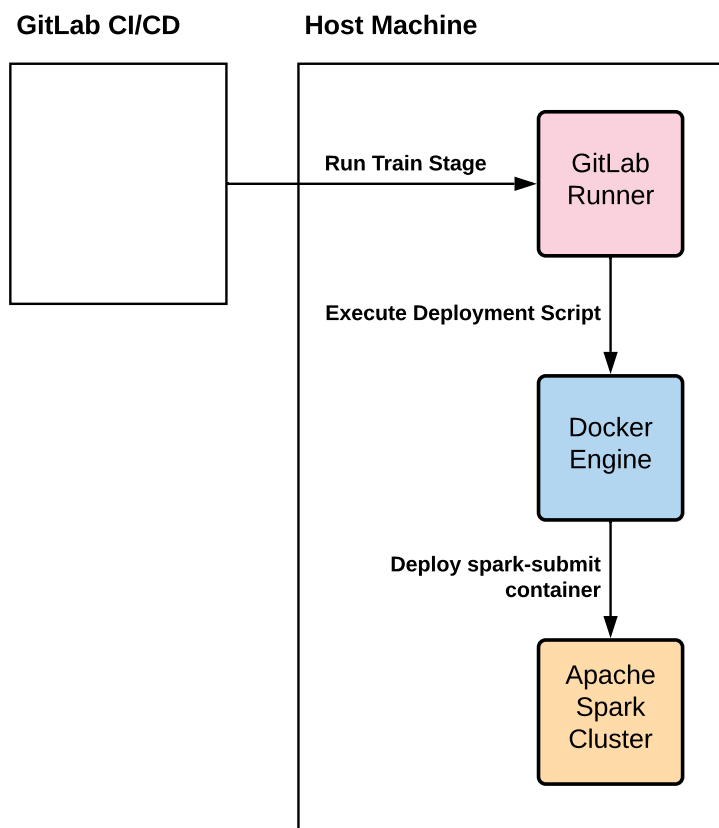
Figure 5.7: Deployment of a spark-submit container

deploy a *spark-submit* container in the Apache Spark cluster swarm network. The GitLab CI/CD server connects to a GitLab on the host machine and instructs it to execute the train stage. The runner then performs the scripts that instruct the Docker engine to deploy a spark-submit container in the Apache Spark cluster.

Chapter 6

Implementation

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

6.1 Implementation Environment

For the implementation of the conceptual design this thesis, a NVIDIA DGX[1] machine is available. This machine is a shared live-system. Therefore, multiple application are performing simultaneously on this machine which share the same resources.

6.1.1 Technical Details

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

---

1 The Universal System for AI Infrastructure - `https://www.nvidia.com/en-us/data-center/dgx-a100/` (Accessed: 2021-01-30)

6.1.2 NVIDIA Docker Runtime

To enable GPU support for Docker containers, the NVIDIA Container Toolkit[2] has to be installed on the host machine. This toolkit provides a runtime library which automatically enables Docker containers to leverage NVIDIA GPUs. It is possible to define the runtime of a Docker container with the docker `run` command. However, this is not supported for Docker services. To deploy Docker services with the NVIDIA runtime enabled, the NVIDIA runtime has to be set as the default runtime for Docker.

Problem statement: On the host machine, the NVIDIA Container Toolkit is installed, however the NVIDIA runtime is not set as 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 Docker containers on the machine. 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 instead of Docker services. Given this problem statement, the *Auto-Scaler* has to be implemented to use the Docker container API instead of the Docker service API to create Apache Spark worker nodes in the environment. Additionally, the dcgm-exporter Docker container has to be created manually and cannot be created with the Docker stack command.

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 is implemented as a custom module in Python 3.8 and deployed as a Docker image using a custom Dockerfile to deploy the *Auto-Scaler* running in a Docker container.

The *Auto-Scaler* integrates the Analyse, Plan, and Execute phase of the MAPE architecture. Furthermore it scales the Apache Worker horizontally by adding and removing Apache Spark worker Docker containers. The Auto-Scaler runs as a background service that periodically performs each phase in order. A file which defines runtime configuration can be set. The Auto-Scaler can be started with the following command:

```
$ python3 run.py --config=config.yml
```

Listing 6.1: Auto-Scaler start command

---

2 NVIDIA Cloud Native technologies documentation - `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html` (Accessed: 2021-01-28)

6.2.1 Technical Background

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

- aiohttp[3]

- APScheduler[4]

- docker[5]

- PyYAML[6]

6.2.2 Configuration

The configuration parameter for the *Auto-Scaler* have been introduced in Section 5.5.2. All parameter are defined in the YAML file format Listing 6.2 provides an example of a configuration. Overall, a configuration file is structured in three sections: General, metrics, and worker.

- 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. 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.

- 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.

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.

```
1 general:
2   interval_seconds: 5
3   cooldown_period_seconds: 180
4   recurrence_factor: 3
5   prometheus_url: "http://localhost:9090"
6
7 metrics:
8   cpu:
```

TODO: Update weil kein DOcker swarm mehr

---

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

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

```
 9    query: 'sum(rate(container_cpu_user_seconds_total{image
      ="spark-worker:3.0.1-hadoop2.7"}[30s]))'
10    target_utilization: 0.5
11    thresholds:
12      min: 0.2
13      max: 0.6
14
15  gpu:
16    query: 'sum(rate(container_cpu_user_seconds_total{image
      ="spark-worker:3.0.1-hadoop2.7"}[30s]))'
17    target_utilization: 0.3
18    thresholds:
19      min: 0.2
20      max: 0.6
21
22 worker:
23   service_name: "computing_spark-worker"
24   thresholds:
25     min: 1
26     max: 30
```

Listing 6.2: Auto-Scaler configuration YAML file

6.2.3 Scaling Apache Worker Nodes

The *Auto-Scaler* performs periodically. Each period fetches performance metrics, analyses the metrics, plans scaling actions, and executes them if

necessary. To perform periodically, it uses the *APScheduler*[7] Python library. The *APScheduler* provides to schedule jobs periodically. In each job, the Analyse, Plan, and Execute phase is performed in order.

Estimation of Necessary Scaling Actions

To estimate of a scaling actions is necessary, the Auto-Scaler uses the Scaling Heat algorithm (introduced in Section 1). The algorithm takes the utilization of a performance metric, the lower- and upper-threshold, and the calculated heat of the last iteration as input parameters. The utilization is received from the Prometheus HTTP API. Furthermore, the *Auto-Scaler* fetches the utilization of all given performance metrics in each iteration. It calculates the heat value for each single performance metric. If the upper-threshold value gets violated, the heat value is increased by 1. Otherwise, if the lower-threshold is violated, the heat is reduced by one. Until the heat value reaches positive or negative of the recurrence-factor a scaling action will be executed. This phase will not be performed if a cooldown period has been activated in the previous iteration.

Calculating the Number of Needed Worker Nodes

To calculate the number of needed Apache Spark worker, the *Auto-Scaler* uses the KHPA algorithm (introduced in Section 4.8). As input parameters, the algorithm takes the current number of active worker nodes, the utilization of the performance metric, and target utilization.

Listing 6.3 shows the implementation of the KHPA algorithm in Python. To receive the number of active worker nodes, the Apache Spark master node has to be defined as a Prometheus target (described in SECTION). This enables to fetch the `metrics_master_aliveWorkers_Value` metric from the Prometheus HTTP API. This metrics returns the number of alive Apache Spark worker nodes in the cluster.

The utilization of the performance metric, which is used for the KHPA algorithm, has already been received in the previous phase. Lastly, the target utilization for the performance metric is defined in the *Auto-Scaler* configuration.

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

Listing 6.3: KHPA implementation using Python 3.8

---

7 https://pypi.org/project/APScheduler/ (Accessed: 2021-01-26)

Performing a Scaling Action

After the number of needed Apache Spark worker nodes are calculated, the *Auto-Scaler* is responsible to scale the replicas of the worker service to reach the desired performance goal. To scale a Docker service using Python, Docker provides a Python library for the Docker Engine[8]. After a scaling action has been executed successfully by the Docker engine, a cooldown period is activated. The duration of the cooldown period can be set in the configuration. During this cooldown no scaling actions are executed.

Removing Apache Spark worker

As being introduced in SECTION RDD, Apache Spark is build on top of a data structure called RDD. A RDD can be compose of bla bla bla. Executors are launched on workers to process data. Therefore, a worker node stores temporary data of an executor containing cache data and shuffle-data. If a worker node store important shuffle data, even if the overall performance is underutilized, the worker cannot be removed. Therefore, worker will not be removed if applications are actively performed on the Apache Spark cluster. Prometheus provides a metrics called `metrics_master_apps_Value` which returns the number of running applications on the cluster. To fetch this metrics, the Apache Spark master node has to be set as Prometheus target. The Auto-Scaler only reduces the number of worker replicas, if this metric returns 0.

6.2.4 Docker Image

A Docker image is needed to deploy the *Auto-Scaler* as a Docker service in the computing environment. Therefore a custom Dockerfile is created to build an *Auto-Scaler* Docker image. Listing 6.4 shows the Auto-Scaler Dockerfile implementation. The python:3 Docker image is set as the parent image. Next, the *Auto-Scaler* source code is copied to image. Afterwards, a Python virtual environment is created with all dependencies installed. As entrypoint, the Docker image starts the *Auto-Scaler* process.

```
1  FROM python:3
2
3  WORKDIR /usr/src/auto_scaler
4
5  # Copy the python module
6  COPY setup.py .
7  COPY src src/
8
9  # Update and install packages
10 RUN pip3 install -e .
11
12 ENTRYPOINT [ "python3", "src/run.py" ]
```

---

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

Listing 6.4: Auto-Scaler Dockerfile

To build the *Auto-Scaler* Docker image, a build script was implemented. Listing 6.5 shows the build script implementation. The script takes the version of the *Auto-Scaler* module as input attribute and uses the version to tag the Docker image. It builds two versions of the *Auto-Scaler*, one tagged with the current *Auto-Scaler* version (e.g. *auto-scaler:1.0*) and one tagged as *latest* (e.g. *auto-scaler:latest*).

```bash
#!/bin/bash

if [ $# -ge 1 ]
  then
    VERSION=$1

    PWD=$(pwd)

    CI_REGISTRY=${CI_REGISTRY-local}
    IMAGE_TAG_PATH="${CI_REGISTRY}/cci/distributed-computing-framework"

    docker build \
      -t $IMAGE_TAG_PATH/auto-scaler:$VERSION \
      $PWD

    docker build \
      -t $IMAGE_TAG_PATH/auto-scaler:latest \
      $PWD
  else
    echo "No arguments supplied\n"
      echo "Use the script as follows:"
      echo "build-image.sh <VERSION>"
      exit 1
fi
```

Listing 6.5: Auto-Scaler build script

The script can be used from the command line as `sh build-image.sh "1.0"`.

6.3 Apache Spark Cluster with GPU Acceleration

The Apache Spark cluster consists of a single master node, a dynamic number of worker nodes, and *spark-submit* nodes. A *spark-submit* is deployed, whenever an application is submitted to cluster. The master worker nodes are deployed in standalone mode.

6.3.1 Apache Spark Base Image

All services in the Apache Spark cluster have the same dependencies but perform different services. To simplify the creation of the Docker images, a base image is created. This base image serves as the parent image for all other images. Additionally, this contributes to the homogeneity of running services in cluster.

To install a specific Apache Spark version on the Docker image, two arguments are provides. The arguments can be set in the Docker build command with the `-build-arg` flag. LISTING BLA provides The base image is build from the `nvidia/cuda:11.0-devel-ubuntu16.04` Docker image. This image is provided by NVIDIA and has already installed all needed dependencies to leverage GPUs to Docker images. The Dockerfile installs all required dependencies to run a standalone Apache Spark node with GPU acceleration. This includes the Apache Spark, JRE 1.8, Python3, GPU discovery script, RAPIDS Java binary, cuDF Java binary, XGBoost4j binary, and XGBoost4j-spark binary.

6.3.2 Apache Spark Master Image

The Apache Spark master node Docker image is created from a custom Dockerfile (see Listing B.2). This image is build on top the *spark-base* image and therefore has already all dependencies installed. It configures the Apache Spark master port to 7077 and the port for the web user interface to 4040. Additionally, the `start-master.sh` script is set as entrypoint. Therefore, when a service is build from this image, it will automatically start an Apache Spark master node in standalone mode.

6.3.3 Apache Spark Worker Image

The Apache Spark worker image is created from a custom Dockerfile as well. This Dockerfile uses the *spark-base* image as parent. As being mentioned in SECTION THEO, the horizontal scaling approach is more effective with homogeneous nodes. Each Apache Spark worker node is being created from the same Docker image given the same resources. Therefore, no additional dependencies have to be installed. To configure the worker nodes resources, the `spark-env.sh` is copied to the Apache Spark `conf/` folder. Listing 6.6 shows the configuration file. It sets the number of GPUs for each executor and the path to the GPU discovery script on the node host. The entrypoint is set to `start-slave.sh` to start an Apache Spark worker in standalone mode. To connect to Apache Spark master node, the URI has to be set as an environment variable.

```
SPARK_WORKER_OPTS="-Dspark.worker.resource.gpu.amount=1 -
    Dspark.worker.resource.gpu.discoveryScript=/opt/
    sparkRapidsPlugin/getGpusResources.sh"
```

Listing 6.6: Environment configuration for all worker nodes

6.3.4 Apache Spark Submit Image

An Apache Spark submit container is deployed by the CI pipeline whenever an application is submitted to the cluster. This container is created from a *spark-submit* Docker image. The spark-submit Dockerfile uses the spark-base image as parent. It copies a custom submit script to the image which is used as the image entrypoint. When it a spark-submit container is started, it executes the custom submit script with all given arguments. The custom submit script is provided at Listing B.5. It provides a simplified interface for the spark-submit executable. Furthermore, it includes all necessary libraries and configuration parameters to perform an application with GPU acceleration enabled. Additionally, if the `CPU_ONLY` environment variable is set to *true*, the script will disable GPU acceleration for the given application. This feature is useful when algorithms optimized for task-parallelism are being executed. Listing **??** demonstrates an usage example of the the custom submit script. In the example, the `train.py` application is executed with a `-max-depth=5` runtime argument given. All provided arguments are forwarded to the spark-submit executable (explain in SECTION XY).

```
1  $ sh submit.sh train.py --max-depth=5
```

Listing 6.7: Usage of the submit script

The script is part of the training state of the CI pipeline which is described in detail in SECTION CI.

TODO: List mit allen envs variablen

6.4 Computing Environment

The computing environment is deployed as a Docker swarm (described in SECTION AB). It consists of several components which are all deployed as Docker services. The conceptual design is explained in SECTION XY. As mentioned in SECTION XY, components which require GPUs are not created as Docker services. These components are deployed as Docker containers in the same swarm network. Overall, the computing environment consists of the following components:

- Autonomic Manager
  - Auto-Scaler
  - Prometheus
  - cAdvisor
  - dcgm-exporter
- Apache Spark Cluster

6.4.1 Deployment of the Computing Environment

To simplify the deployment of a stack of services, Docker proved to define all services in a docker-compose file. The computing environment docker-compose file is defined in Listing A.1. It defines all services, except the

Apache Spark worker and the dcgm-export because these services require the NVIDIA Docker runtime. To deploy all services defined in the docker-compose file, Docker provides the docker stack command. After the Docker stack has been deployed, the dcgm-exporter container has to be deployed and added to the same network. Listing 6.8 provides the process how to deploy the stack and the dcgm-exporter container.

```
$ docker stack deploy -c docker-compose.yml computing
...
$ docker run -d --rm -p "9400:9400" \
    --runtime=nvidia \
    --name dcgm-exporter \
    --network computing_net \
    nvidia/dcgm-exporter:2.0.13-2.1.1-ubuntu18.04
```

Listing 6.8: Auto-Scaler start command

6.4.2 Autonomic Manager

The autonomic manager is responsible to monitor the computing environment and scale the number of Apache Spark worker. It is composed of a monitoring system and the previously mentioned *Auto-Scaler*. Furthermore, the monitoring system is build of a Prometheus, cadvisor, and dcgm-exporter. Together, all components build an autonomic manager according to the MAPE architecture. The monitoring system monitors the components in the computing environment while the *Auto-Scaler* analyses the performance metrics and adapt the number of Apache Spark worker.

Prometheus Target Configuration

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

Listing 6.9 specifies the scrape configuration of the Prometheus system. The `scrape_interval` parameter is set to *5s*. This means, that Prometheus scrapes all defined targets every 5 seconds. Additionally, a the To scrape all needed performance metrics three different targets are set: cAdvisor, dcgm-exporter, and the Apache Spark master node. The configuration is explained in SECTION TECHNICAL PROM. All container related performance metrics are received from the cAdvisor target. The GPU performance metrics are provided by the dcgm-exporter target. As mentioned in SECTION AS, the *Auto-Scaler* needs to know how many Apache Spark worker nodes are active at runtime and how man applications are being executed. This metrics are scraped from the Apache Spark master node which has native support for Prometheus. Each target is given a specific label which to identify the source.

```
global:
  scrape_interval: 5s

rule_files:
```

```
5     - "/etc/prometheus/recording_rules.yml"
6
7   scrape_configs:
8     - job_name: cadvisor
9       static_configs:
10          - targets: ["cadvisor:8080"]
11            labels:
12                group: "cadvisor"
13
14     - job_name: dcgm-exporter
15       static_configs:
16          - targets: ["dcgm-exporter:9400"]
17            labels:
18                group: "dcgm-exporter"
19
20     - job_name: spark-master
21       metrics_path: /metrics/master/prometheus/
22       static_configs:
23          - targets: ["spark-master:4040"]
24            labels:
25              group: "spark"
```

Listing 6.9: Prometheus target configuration in YAML syntax

Prometheus Recording Rules Configuration

In SECTION YB, suitable metrics for monitoring the target performance have been introduced. These two metrics are created as recording rules in Listing 6.10 (see SECTION XY for details about recording rules).

```
1   groups:
2     - name: performance_usage_percent
3       rules:
4         - record: instance:worker_nodes:cpu_usage:percent
5           expr: SUM(RATE(container_cpu_usage_seconds_total{
    image="spark-worker:3.0.1-hadoop2.7"}[30s])) BY (image)
6         - record: instance:gpu:usage:percent
7           expr: SUM(DCGM_FI_DEV_GPU_UTIL{device=~"(?:nvidia4
    |nvidia5)"}) / 2
```

Listing 6.10: Prometheus target configuration in YAML syntax

6.5 GitLab CI/CD Pipeline for Automated Deployment of Machine Learning Applications

The CI pipeline is responsible to test, and submit the application to the Apache Spark cluster. Important to notice, that Python is an interpreted programming language. It is possible to package the Python app as ZIP document or as an egg file. However, to keep the implementation simple, only raw Python files are submitted to the Apache Spark cluster. Therefore,

there is no build stage in this CI pipeline architecture. To be able to deploy a *spark-submit* container on the host machine, a GitLab runner (introduced in SECTION XY) instance has to run on the same machine.

6.5.1 GitLab Runner

As being explained in SECTION THEO, GitLab CI/CD jobs are being executed by a GitLab runner. To deploy a spark-submit container in the swarm network of the Apache Spark cluster, the GitLab needs to run on the same host to get access to the Docker engine.

As mentioned before, to deploy a spark-submit node on the host machine, a GitLab Runner instance is needed. To be able to create Docker container in the same swarm network, the GitLab runner needs access to the Docker engine of the host machine.

In this implementation, the GitLab Runner is deployed as a Docker container. The Docker container is build from the `gitlab/gitlab-runner:latest` image. This image provides an installation of the GitLab runner service and command-line-interface. To enable Docker support in the context of the GitLab Runner Docker container, the Docker socket has to be mounted to the container. This *socket-binding* approach is explained in detail on the GitLab online documentation[9].

The `gitlab-runner` command is set as image entrypoint. It initialises a new GitLab runner on the container. To enable Docker access to the GitLab runner, various settings have to be set. First the docker excutor has to be activated. This enables the runner to perform jobs with user defined Docker images, which is required to use the *spark-submit* image. Second, the runner needs access to the Docker socket as well. Therefore, the mounted Docker socket has to be set as volume for the runner. Lastly, the `docker:latest` image is used for the container where jobs are performed on. This gives jobs the ability to use the Docker CLI. This ability is needed to run the `docker run` command from a job.

The `/builds` directory of a runner is the location where the source code of a repository is located. By default, this directory is only available to the job container deployed by the runner. To share this container with container deployed by a job, it has to be shared with the host system. Docker container cannot shaire their filesystem with other containers, because Docker run in isolation. This is described in detail in SECTION BLA NEXT. Listing 6.11 provides the command to create a GitLab runner in a Docker container. In addition to the previously mentioned configuration settings, a GitLab Runner requires a URL to a GitLab server and a registration token. The registraktion token identifies a specific repository or a GitLab group. Then, the runner can only used from the This GitLab runner is reposnsible to

---

9   Building Docker images with GitLab CI/CD - `https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-socket-binding` (Accessed: 2021-01-31)

perform submit Apache Spark application to the cluster. Therefore the spark-submit tag was set, so it only performs CI jobs tagged with this.

```
1  $ docker run -d \
2      --name gitlab-runner \
3      --restart always \
4      -v /var/run/docker.sock:/var/run/docker.sock \
5      -v /builds:/builds \
6      gitlab/gitlab-runner:latest \
7      gitlab-runner register \
8      -n \
9      --name "Spark-Cluster Runner" \
10     --executor docker \
11     --docker-image docker:latest \
12     --docker-volumes /var/run/docker.sock:/var/run/docker.
       sock \
13     --docker-volumes /builds:/builds \
14     --docker-privileged=true
15     --url https://gitlab.com/ \
16     --registration-token mpCBWzZzhaaJrdqjXYZq \
17     --tag-list spark-submit
```

Listing 6.11: CLI command to start a GitLab runner in a Docker container

6.5.2 Pipeline Architecture

As explained in SECTION AB, a GitLab CI/CD pipeline is configured by a `.gitlab-ci.yml` file in the root directory of the source code repository. LST XY shows the .gitlab-ci.yml configuration file. The pipeline consists of a *test* and a *train* stage.

Test Stage

The first stage is the test stage which consists of a single job called *unit-tests* to perform unit tests. It uses the `python:3.8-slim` Docker image, which has already Python 3.8 installed. To perform the unit tests, first a Python virtual environment with all dependencies is created. Lastly, all tests in the `tests/` directory are performed using the pytest[10] library.

It is important to mention, that the train stage depends on the application. This implementation focuses on simplicity and provides a general concept of a train stage which can be used and extended for other projects.

Train Stage

After the test stage has been performed successfully, the train stage is responsible to perform the Apache Spark application on the Apache Spark cluster. The train stage consists of a single job called *train-model*. This jobs

---

10 pytest - `https://docs.pytest.org/en/latest/` (Accessed: 2021-02-01)

executes the `submit.sh` build script, which is located in the repository root
directory. The script executes the `docker run` command to deploy a *spark-
submit* Docker container in the Apache Spark cluster. It takes the path to
the application Python file and all needed parameters as input and forwards
them to the `docker run` command. To be able to execute the applications
Python file, it has to be mounted to the host filesystem, which is explained
in this section. To configure the *spark-submit* container, all configuration
parameters listed in TABLE CONFIG PARAM OF SPARK-SUBMIT can
be set as environment variables in the `variables` section of the job. The
job has to be tagged with *spark-submit* to get executed by the runner on
the host machine.

Creating a mount-point: As mentioned before in SECTION OBEN, Docker
container run in isolation and do not share the same filesystem with each
other. Each runner registered on the GitLab Runner Docker container runs
its jobs on separate Docker container on the host system. All commands
are being performed in the context of its container. When the *train-job*
deploys a *spark-submit* Docker container with the `docker run` command, it
tries to mount the `/builds` directory from its own filesystem to the *spark-
submit* container. To share the source code between the job container and
the *spark-submit* container, the `/builds` directory is mounted to the host
system, as being explained in SECTION OBEN. To overcome this problem,
the *train-model* job creates a new folder inside the `/builds` directory as
mount-point and moves the application source code files to this location.
This mount-point is available to the *spark-submit* container when given as a
volume. The approach of a mount-point is mentioned in a discussion on the
GitLab website[11].
   - Screenshot von webui output

---

11 Docker volumes not mounted when using docker:dind - `https://gitlab.com/
   gitlab-org/gitlab-foss/-/issues/41227` (Accessed: 2021-02-01)

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

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 Experiments

The performance of the computing environment is measured on two widely used machine learning algorithms. In both experiments, a machine learning model is trained on the computing environment using Apache Spark. Furthermore, each benchmark is evaluated using three different configurations:

1. Using a static number of Apache Spark worker to evaluate the performance of using only CPUs

2. Using a static number of Apache Spark worker with GPU acceleration enabled

3. Dynamically scaling the number of CPU-only Apache Spark worker nodes using the *Auto-Scaler*

The performance difference between GPU accelerated worker nodes and CPU-only worker nodes is explored using two different configurations of the implementation.

The two benchmarks are:

- XGBoost classification model using the *Fannie Mae's Single-Family Historical Loan Performance Dataset*[1][Fan]

- XGBoost regression model using a Taxi fare dataset[2]

The source code and the dataset used in these experiments are available on Github on the *spark-xgboost-examples*[3] repository from NVIDIA. The repository provides a *mortgage* and a *taxi* application. Both applications come with a CPU-only implementation and a GPU implementation.

Both classification and regression algorithms are supervised machine learning algorithms. The goal of supervised machine learning algorithms is to train a model by finding patterns in labeled data. Then, the model is used to predict labels on new data based on the learned labels. The classification algorithm identifies the category of a label. A regression algorithm predicts a continuous numeric value [McD20].

7.2.1 Mortgage Experiment

In this experiment a XGBoost classification model is trained. The model is trained to

7.2.2 Taxi Experiment

This experiment train a XGBoost regression model to predict a fare amount of a taxi trip based on the amount of passengers and the trip distance. The dataset used to train the model is provided on the Github repository. FIGURE XY shows the first five lines of the dataset.

7.3 Static Worker CPU only

The first experiment is conducted using a static number of Apache Spark worker using only CPUs to evaluate each benchmark. The goal of this experiment is, to conduct performance results which can be compared to the GPU accelerated experiment and the Auto-Scaler experiment. Each worker was given a set of resources for each executor: 80 CPU core per executor and 16GB of RAM. FIG XY illustrates the observed results. It represents the mean execution time in seconds a spark-job needed with the given Apache Spark worker nodes. For the classification benchmark, the best result was achieved with 10 worker nodes. Running on 15 worker nodes, the spark job time increased about 6 seconds. The regression benchmark achieved the

---

1   Downloaded from: `https://docs.rapids.ai/datasets/mortgage-data` (Accessed: 2021-02-06)

2   The Taxi dataset is available at: `https://github.com/NVIDIA/spark-xgboost-examples/tree/spark-3` (Accessed: 2021-02-06)

3   spark-xgboost-examples - `https://github.com/NVIDIA/spark-xgboost-examples/tree/spark-3` (Accessed: 2021-02-06)
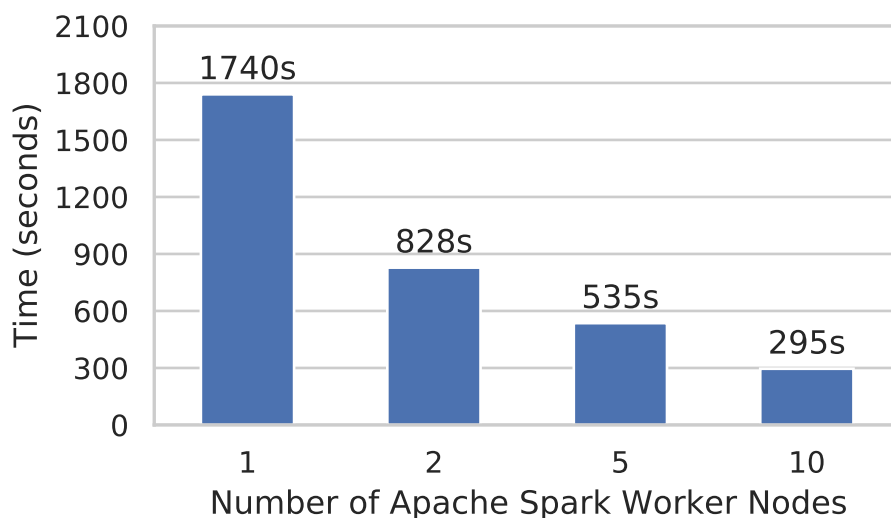
Figure 7.1: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].
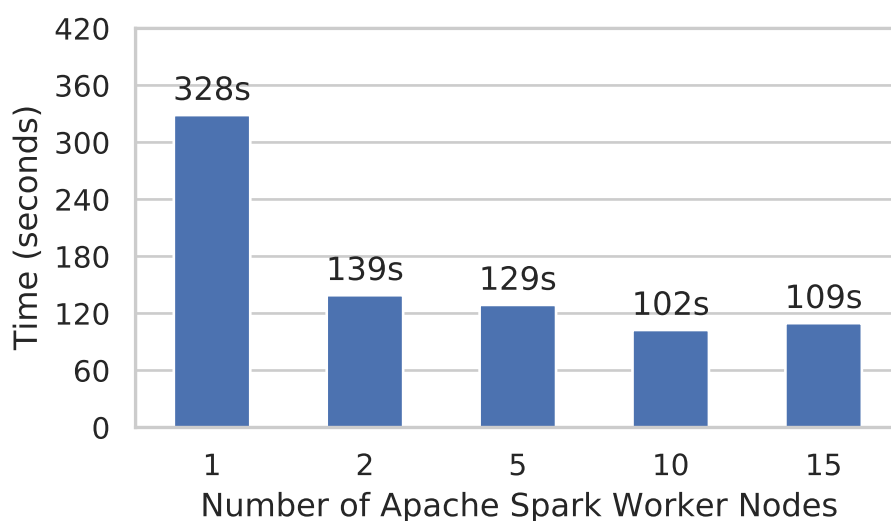

Figure 7.2: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

best result with 10 worker nodes as well. Using 15 workers nodes took the spark job time to 109s increased by 7 seconds. The performance of all 10 iterations for the classification benchmark is available at ANHANG A and for the regression benchmark at ANHANG B.

7.4 GPU Acceleration

The goal of this experiment is to explore the impact of GPU accelerated Apache Spark worker nodes while training machine learning models. For this

experiment, two GPUs have been available. Each benchmark was evaluated 10 times with 1 and 2 GPUs. The results illustrate the mean time of the
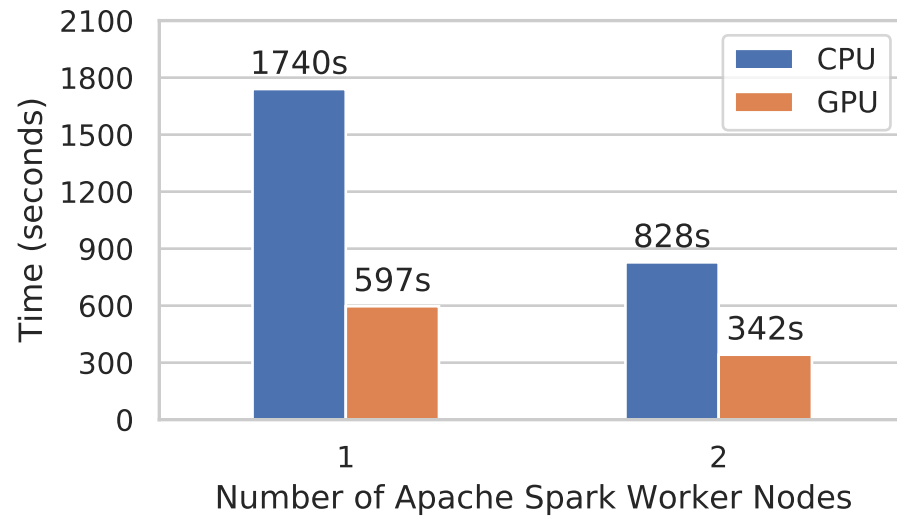

Figure 7.3: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].
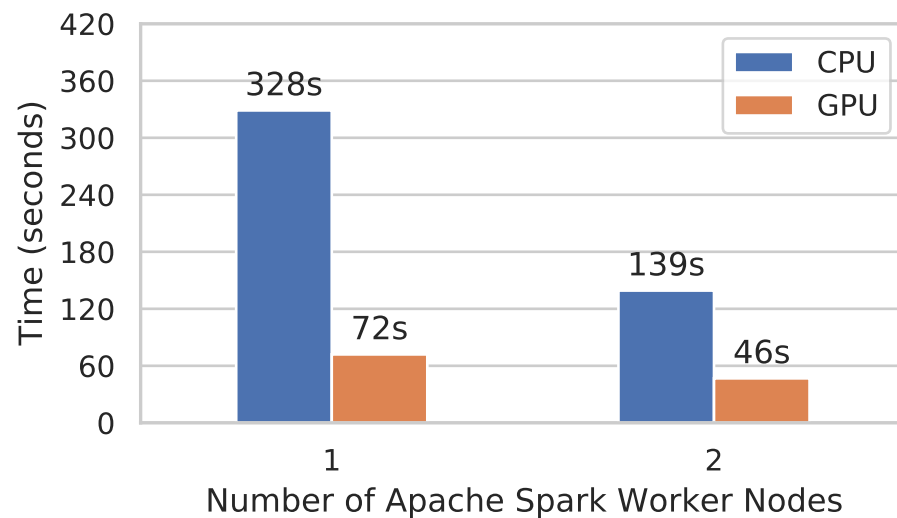

Figure 7.4: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

spark-job execution time in seconds. For the classification algorithm, using 1 GPU accelerated worker was able to decrease the execution time by 1143 seconds, using 2 GPUs a betterness of 486 seconds. Using 1 GPU thats an 2.9x increase and for 2 GPUs an increase of 2.4x.

7.5 Auto-Scaler

lieber Beide algos in einem diagramm und dann auf die verschiedenen einstellungen in einer tabelle eingehen.

To test the impact of the Auto-Scaler while training machine learning applications, both benchmarks have been tested with different configurations. Table 7.1 shows the Auto-Scaler configuration parameters for each benchmark. The configuration parameters are have been choosen in accordance to the results of the static worker experiment. Both benchmarks in the static worker experiment achieved the best overall performance using 10 worker nodes. Therefore, this value is chosen as the maximum number of worker nodes. The recurrence factor for the regression experiment was set to 1, because an iteration using 2 worker nodes took 80 SECONDS in mean. Using a higher recurrence factor for this experiment lead to no scaling actions. The target CPU utilization is the mean utilization of each benchmark using 2 worker nodes. The CPU utilization values have been chosen in accordance to the minimum and maximum values of the static worker experiment.

The results, illustrated in FIG A, showed that scaling the number of Apache Spark worker nodes during the training of a machine learning model increased the overall time of the spark-jobs. In the classification benchmark, the Auto-Scaler caused an increase of 216 seconds in comparison of no auto-scaling. Dynamically scaling worker nodes during the regresssion benchmark increased the mean time by 38 seconds. Given this result,

Parameter	Classification	Regression
Interval	5 seconds	5 seconds
Recurrence factor	3	1
Cooldown period	60 seconds	60 seconds
Target CPU utilization	20%	5%
Minimum CPU utilization	10%	2%
Maximum CPU utilization	25%	10%
Minimum worker nodes	2	10
Maximum worker nodes	2	10

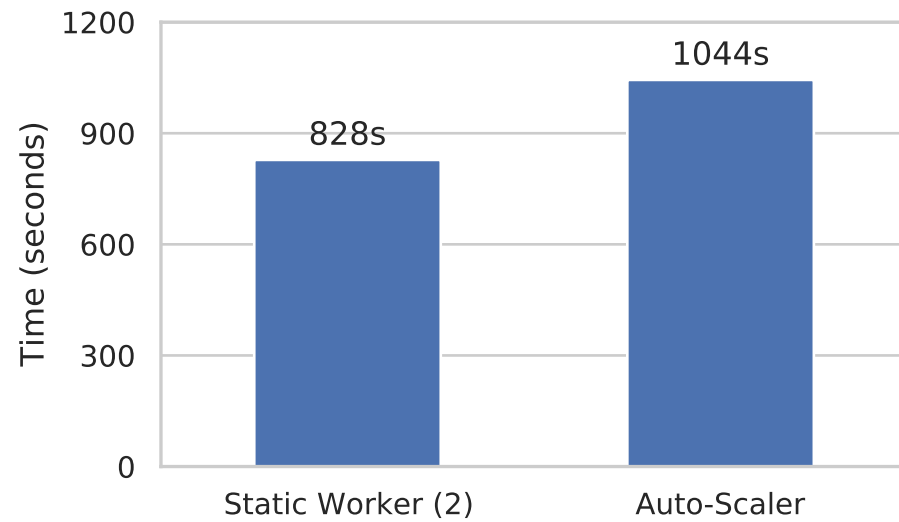**Table 7.1:** Auto-Scaler configuration parameter

Figure 7.5: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].
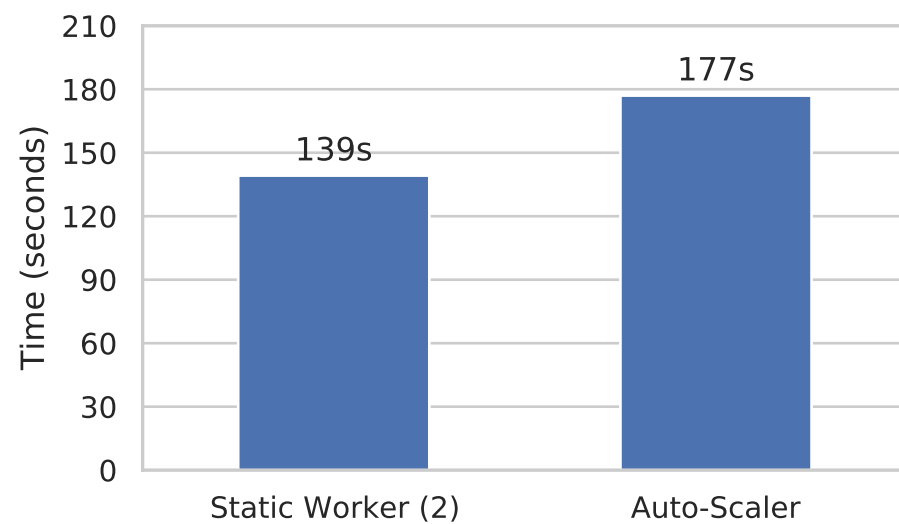

Figure 7.6: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

Chapter 8

Outlook

8.1 Optimizing Scaling

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

8.2 Reinforcement Learning for Auto-Scaling

bla bla

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

[Fan] Fannie Mae: *Fannie Mae Single-Family Loan Performance Data.* `https://capitalmarkets.fanniemae.com/credit-risk-transfer/single-family-credit-risk-transfer/fannie-mae-single-family-loan-performance-data`. – Accessed: 2021-02-06

[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 Envi-
          ronments, 2013

[LBMAL14] Lorido-Botrán, Tania ; Miguel-Alonso, Jose ; Lozano,
          Jose: A Review of Auto-scaling Techniques for Elastic Ap-
          plications 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

[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, Christo-
          foros ; Soudris, Dimitrios ; Bilas, Angelos: VineTalk: Sim-
          plifying 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 Documen-
        tation.* `https://nvidia.github.io/spark-rapids/`. – Ac-
        cessed: 2020-12-04

[PNKM20] POTDAR, Amit ; NARAYAN, DG ; KENGOND, Shivaraj ;
        MULLA, Mohammed: Performance Evaluation of Docker Con-
        tainer 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: Het-
        eroSpark: 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 De-
        ployment.* 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 com-
        puting, 2006

[SP20]   SABHARWAL, Navin ; PANDEY, Piyush: *Monitoring Microser-
        vices and Containerized Applications: Deployment, Configura-
        tion, and Best Practices for Prometheus and Alert Manager.*
        Apress, 2020. – ISBN 9781484262160

[Thea]   THE APACHE FOUNDATION: *Arrow. A cross-language devel-
        opment platform for in-memory data.* `https://arrow.apache.`
        `org/`. – Accessed: 2020-12-03

[Theb]   THE APACHE SOFTWARE FOUNDATION: *Spark 3.0.1 Docu-
        mentation.* `https://spark.apache.org/docs/3.0.1/`. – Ac-
        cessed: 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, Inte-
          gration, 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`. – Ac-
          cessed: 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 Dis-
          tributed 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: Clus-
          ter 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

Computing Environment Implementation

```yaml
1  version: "3.7"
2
3  networks:
4      computing-net:
5          name: computing_net
6          attachable: true
7
8  services:
9      spark-master:
10         image: spark-master:3.0.1-hadoop2.7
11         networks:
12             - computing-net
13         ports:
14             - 4040:4040
15             - 7077:7077
16         volumes:
17             - ./conf/spark-master/metrics.properties:/usr/
   bin/spark/conf/metrics.properties
18             - ./conf/spark-master/spark-defaults.conf:/usr/
   bin/spark/conf/spark-defaults.conf
19
20     prometheus:
21         image: prom/prometheus
22         networks:
23             - computing-net
24         volumes:
25             - ./conf/prometheus/prometheus.yml:/etc/
   prometheus/prometheus.yml
26             - ./conf/prometheus/recording_rules.yml:/etc/
   prometheus/recording_rules.yml
27         command:
28             - "--config.file=/etc/prometheus/prometheus.yml"
29         ports:
30             - "9090:9090"
31         depends_on:
32             - cadvisor
33
34     cadvisor:
```

```
35          image: google/cadvisor
36          networks:
37              - computing-net
38          ports:
39              - "8080:8080"
40          volumes:
41              - "/:/rootfs:ro"
42              - "/var/run:/var/run:ro"
43              - "/sys:/sys:ro"
44              - "/var/lib/docker/:/var/lib/docker:ro"
45              - "/dev/disk/:/dev/disk:ro"
46          command:
47              - "--docker_only=true"
48              - "--logtostderr=true"
49          depends_on:
50              - spark-master
51              - spark-worker
52
53      auto-scaler:
54          image: auto-scaler:latest
55          networks:
56              - computing-net
57          volumes:
58              - "/var/run/docker.sock:/var/run/docker.sock:ro"
59              - ./conf/auto-scaler/config.yml:/etc/autoscaler/
    config.yml
60          command:
61              - '--config=/etc/autoscaler/config.yml'
62          depends_on:
63              - prometheus
```

Listing A.1: Computing environment docker-compose file

Appendix B

Apache Spark Cluster Implementation

```
1  FROM nvidia/cuda:11.0-devel-ubuntu16.04
2
3  LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"
4
5  ARG SPARK_VERSION
6  ARG HADOOP_VERSION
7
8  # Install all important packages
9  RUN apt-get update -qy && \
10     apt-get install -y openjdk-8-jre-headless procps python3
       python3-pip curl
11
12 # Install Apache Spark
13 RUN mkdir /usr/bin/spark/ && \
14     curl https://ftp-stud.hs-esslingen.de/pub/Mirrors/ftp.
       apache.org/dist/spark/spark-${SPARK_VERSION}/spark-${
       SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz -o spark.
       tgz && \
15     tar -xf spark.tgz && \
16     mv spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}/*
       /usr/bin/spark/ && \
17     rm -rf spark.tgz && \
18     rm -rf spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION
       }/
19
20 # Add GPU discovery script
21 RUN mkdir /opt/sparkRapidsPlugin/
22 COPY getGpusResources.sh /opt/sparkRapidsPlugin/
       getGpusResources.sh
23 ENV SPARK_RAPIDS_DIR=/opt/sparkRapidsPlugin
24
25 # Install cuDF and RAPIDS
26 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
27 RUN curl -o ${SPARK_RAPIDS_DIR}/rapids-4-spark_2.12-0.2.0.
       jar https://repo1.maven.org/maven2/com/nvidia/rapids-4-
       spark_2.12/0.2.0/rapids-4-spark_2.12-0.2.0.jar
28 ENV SPARK_CUDF_JAR=${SPARK_RAPIDS_DIR}/cudf-0.15-cuda11.jar
```

```
29 ENV SPARK_RAPIDS_PLUGIN_JAR=${SPARK_RAPIDS_DIR}/rapids-4-
      spark_2.12-0.2.0.jar
30
31 # Set all environment variables
32 ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
33 ENV SPARK_HOME /usr/bin/spark
34 ENV SPARK_NO_DAEMONIZE true
35 ENV PYSPARK_DRIVER_PYTHON python3
36 ENV PYSPARK_PYTHON python3
37 ENV PATH /usr/bin/spark/bin:/usr/bin/spark/sbin:$PATH
38
39 WORKDIR ${SPARK_HOME}
```

Listing B.1: Apache Spark base image Dockerfile

```
1 ARG SPARK_VERSION
2 ARG HADOOP_VERSION
3
4 FROM spark-base:$SPARK_VERSION-hadoop$HADOOP_VERSION
5
6 LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"
7
8 # Set ports
9 ENV SPARK_MASTER_PORT 7077
10 ENV SPARK_MASTER_WEBUI_PORT 4040
11
12 EXPOSE 4040 7077
13
14 # Start master-node in standalone mode
15 ENTRYPOINT [ "sbin/start-master.sh" ]
```

Listing B.2: Apache Spark master image Dockerfile

```
1 ARG SPARK_VERSION
2 ARG HADOOP_VERSION
3
4 FROM spark-base:$SPARK_VERSION-hadoop$HADOOP_VERSION
5
6 LABEL maintainer="marcel.pascal.stolin@ipa.fraunhofer.de"
7
8 # Add spark-env
9 COPY spark-env.sh ${SPARK_HOME}/conf/spark-env.sh
10
11 # Set port
12 ENV SPARK_WORKER_WEBUI_PORT 4041
13
14 EXPOSE 4041
15
16 # Start worker-node
17 ENTRYPOINT ./sbin/start-slave.sh ${SPARK_MASTER_URI}
```

Listing B.3: Apache Spark worker image Dockerfile

```
1 #!/usr/bin/env bash
2
3 #
```

```
4  # Licensed to the Apache Software Foundation (ASF) under one
       or more
5  # contributor license agreements.  See the NOTICE file
       distributed with
6  # this work for additional information regarding copyright
       ownership.
7  # The ASF licenses this file to You under the Apache License
       , Version 2.0
8  # (the "License"); you may not use this file except in
       compliance with
9  # the License.  You may obtain a copy of the License at
10 #
11 #     http://www.apache.org/licenses/LICENSE-2.0
12 #
13 # Unless required by applicable law or agreed to in writing,
       software
14 # distributed under the License is distributed on an "AS IS"
       BASIS,
15 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
       express or implied.
16 # See the License for the specific language governing
       permissions and
17 # limitations under the License.
18 #
19
20 # This script is a basic example script to get resource
       information about NVIDIA GPUs.
21 # It assumes the drivers are properly installed and the
       nvidia-smi command is available.
22 # It is not guaranteed to work on all setups so please test
       and customize as needed
23 # for your environment. It can be passed into SPARK via the
       config
24 # spark.{driver/executor}.resource.gpu.discoveryScript to
       allow the driver or executor to discover
25 # the GPUs it was allocated. It assumes you are running
       within an isolated container where the
26 # GPUs are allocated exclusively to that driver or executor.
27 # It outputs a JSON formatted string that is expected by the
28 # spark.{driver/executor}.resource.gpu.discoveryScript
       config.
29 #
30 # Example output: {"name": "gpu", "addresses
       ":["0","1","2","3","4","5","6","7"]}
31
32 ADDRS=`nvidia-smi --query-gpu=index --format=csv,noheader |
       sed -e ':a' -e 'N' -e'$!ba' -e 's/\n/","/g'`
33 echo {\"name\": \"gpu\", \"addresses\":[\"$ADDRS\"]}
```

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

```
1  #!/bin/bash
2
3  DRIVER_MEMORY=${DRIVER_MEMORY-4g}
4  EXECUTOR_MEMORY=${EXECUTOR_MEMORY-8g}
```

```
5
6 if [ "$CPU_ONLY" == "true" ]
7     then
8     echo "Submit Spark app with CPU only"
9
10    $SPARK_HOME/bin/spark-submit \
11        --master $SPARK_MASTER_URI \
12        --driver-memory $DRIVER_MEMORY \
13        --executor-memory $EXECUTOR_MEMORY \
14        --conf spark.driver.extraClassPath=${SPARK_CUDF_JAR
    }:${JAR_RAPIDS}:${LIBS_PATH}/xgboost4j_3.0-1.3.0-0.1.0.
    jar:${LIBS_PATH}/xgboost4j-spark_3.0-1.3.0-0.1.0.jar \
15        --conf spark.executor.extraClassPath=${
    SPARK_CUDF_JAR}:${JAR_RAPIDS}:${LIBS_PATH}/xgboost4j_3
    .0-1.3.0-0.1.0.jar:${LIBS_PATH}/xgboost4j-spark_3
    .0-1.3.0-0.1.0.jar \
16        --jars ${SPARK_CUDF_JAR},${LIBS_PATH}/xgboost4j-
    spark_3.0-1.3.0-0.1.0.jar,${LIBS_PATH}/xgboost4j_3
    .0-1.3.0-0.1.0.jar,${JAR_RAPIDS} \
17        --py-files ${LIBS_PATH}/xgboost4j-spark_3
    .0-1.3.0-0.1.0.jar,/tank/data/users/chh-ms/spark-xgboost-
    examples/examples/apps/python/samples.zip \
18        $@
19 else
20    echo "Submit Spark app with GPU acceleration"
21
22    RAPIDS_GPU_ALLOC_FRACTION=${RAPIDS_GPU_ALLOC_FRACTION-1}
23    RAPIDS_INCOMPATIBLE_OPS=${RAPIDS_INCOMPATIBLE_OPS-"false
    "}
24    RAPIDS_DEBUG=${RAPIDS_DEBUG-"NONE"}
25    RAPIDS_GPU_POOL=${RAPIDS_GPU_POOL-"ARENA"}
26
27    $SPARK_HOME/bin/spark-submit \
28        --master $SPARK_MASTER_URI \
29        --driver-memory $DRIVER_MEMORY \
30        --executor-memory $EXECUTOR_MEMORY \
31        --conf spark.plugins=com.nvidia.spark.SQLPlugin \
32        --conf spark.rapids.memory.gpu.pool=$RAPIDS_GPU_POOL
    \
33        --conf spark.rapids.memory.gpu.allocFraction=
    $RAPIDS_GPU_ALLOC_FRACTION \
34        --conf spark.rapids.sql.incompatibleOps.enabled=
    $RAPIDS_INCOMPATIBLE_OPS \
35        --conf spark.rapids.memory.gpu.debug=$RAPIDS_DEBUG \
36        --conf spark.task.resource.gpu.amount=1 \
37        --conf spark.executor.resource.gpu.amount=1 \
38        --conf spark.driver.extraClassPath=${SPARK_CUDF_JAR
    }:${JAR_RAPIDS}:${LIBS_PATH}/xgboost4j_3.0-1.3.0-0.1.0.
    jar:${LIBS_PATH}/xgboost4j-spark_3.0-1.3.0-0.1.0.jar \
39        --conf spark.executor.extraClassPath=${
    SPARK_CUDF_JAR}:${JAR_RAPIDS}:${LIBS_PATH}/xgboost4j_3
    .0-1.3.0-0.1.0.jar:${LIBS_PATH}/xgboost4j-spark_3
    .0-1.3.0-0.1.0.jar \
40        --jars ${SPARK_CUDF_JAR},${LIBS_PATH}/xgboost4j-
    spark_3.0-1.3.0-0.1.0.jar,${LIBS_PATH}/xgboost4j_3
    .0-1.3.0-0.1.0.jar,${JAR_RAPIDS} \
```

```
41          --py-files ${LIBS_PATH}/xgboost4j-spark_3
    .0-1.3.0-0.1.0.jar,/tank/data/users/chh-ms/spark-xgboost-
    examples/examples/apps/python/samples.zip \
42          $@
43 fi
```

Listing B.5: Custom submit script

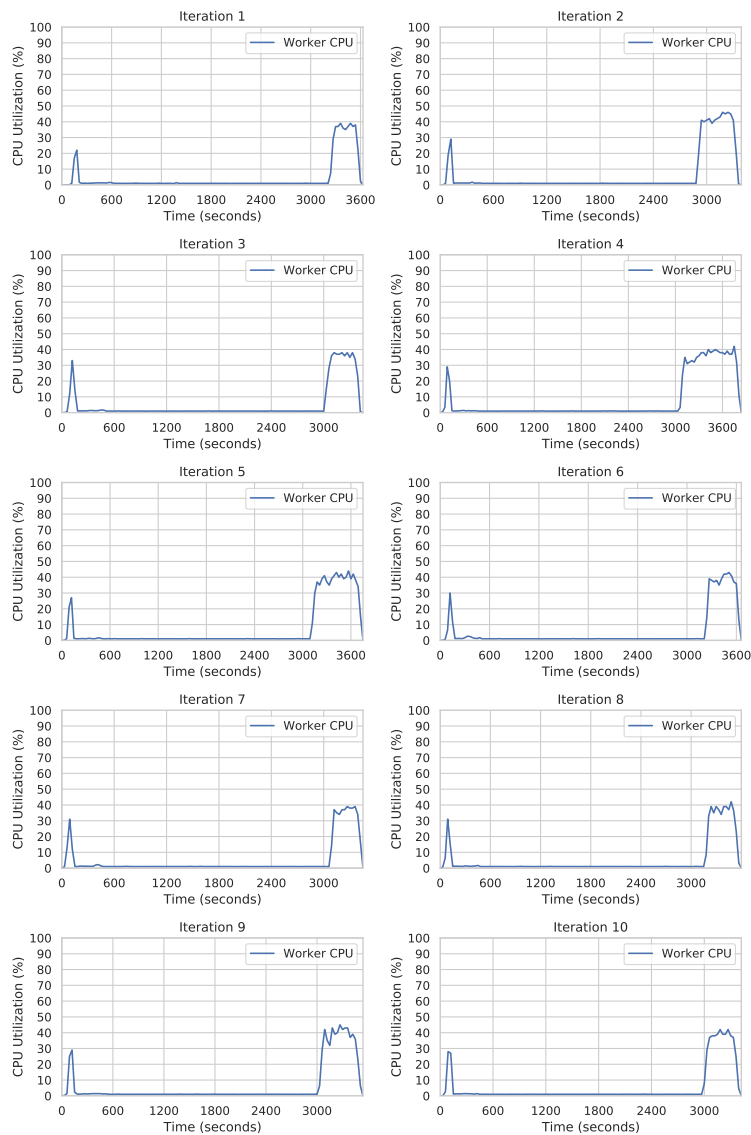# B.1 Evaluation Data

B.1.1 Classification Benchmarks

Figure B.1: Classification algorithm performance evaluation of the computing environment with 1 Apache Spark worker
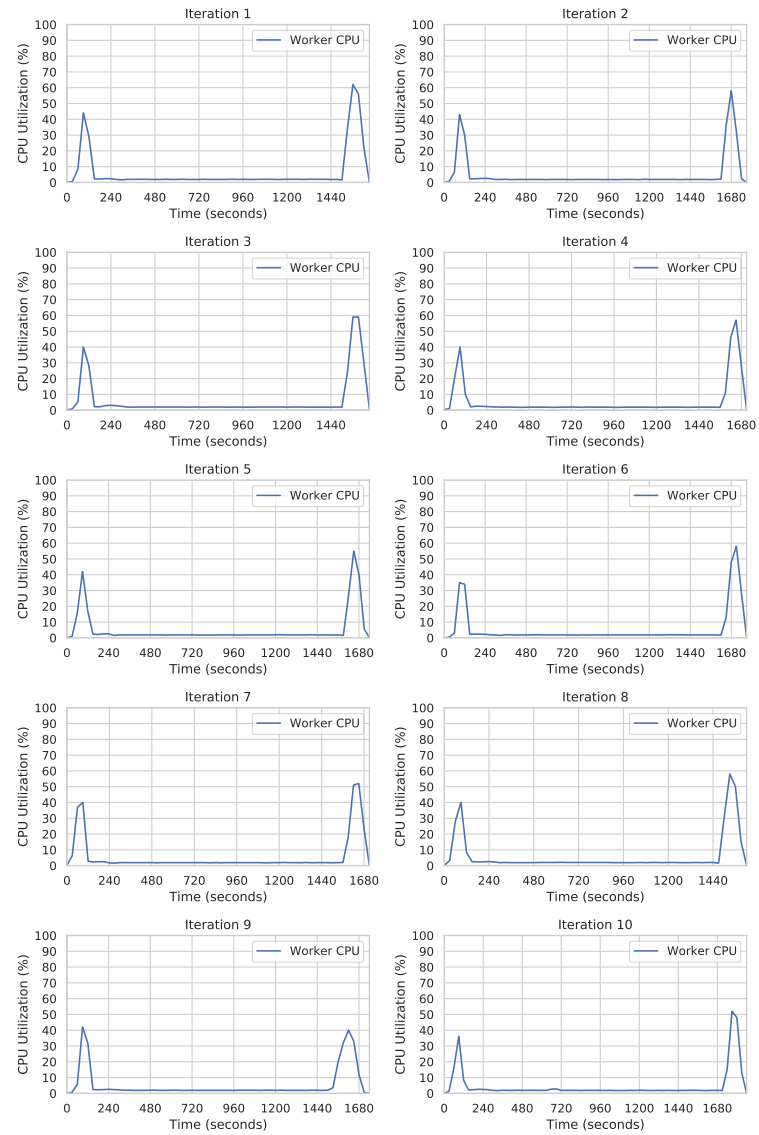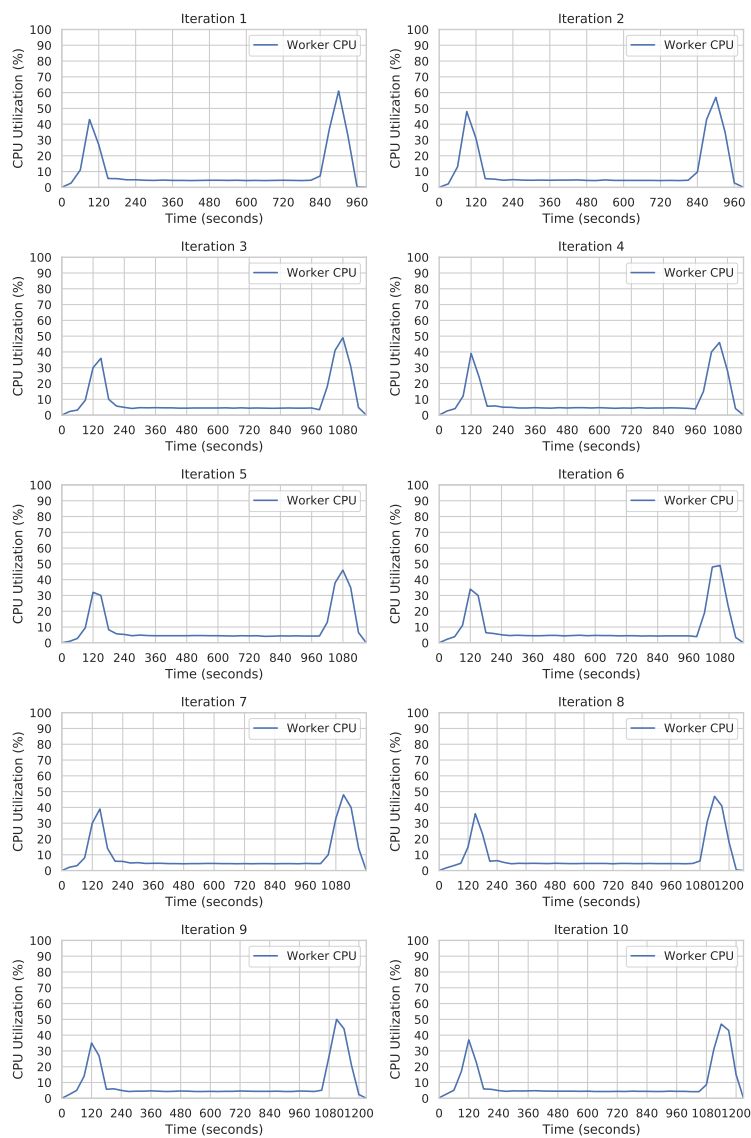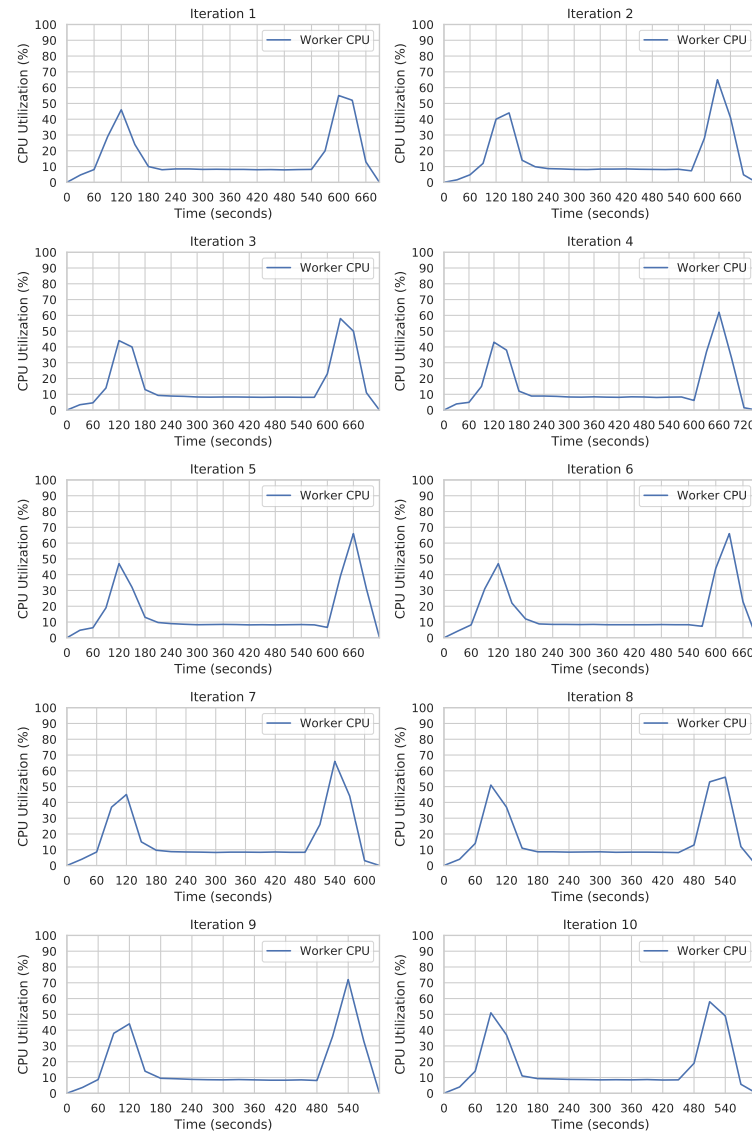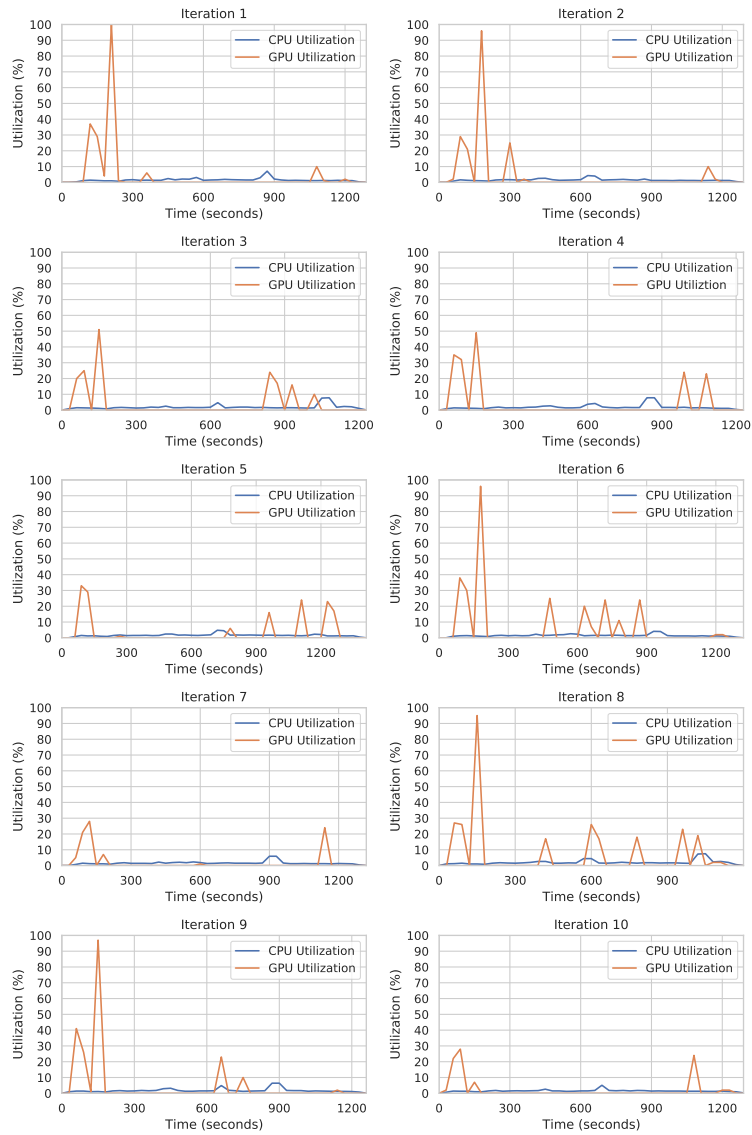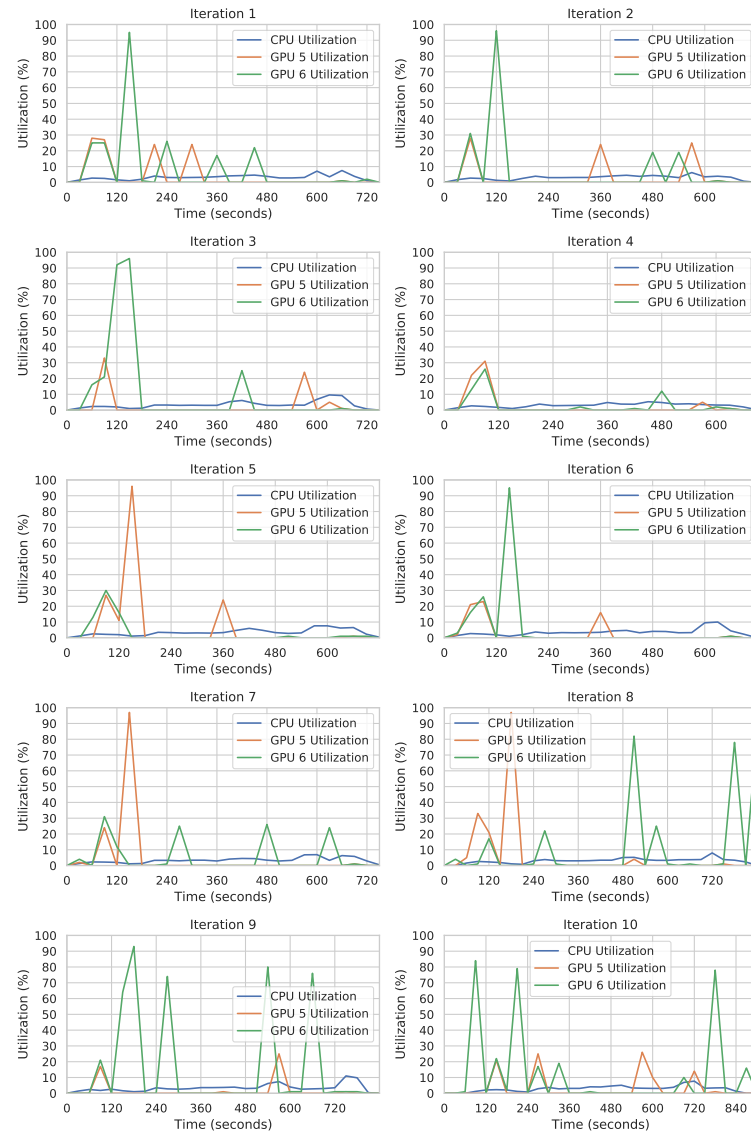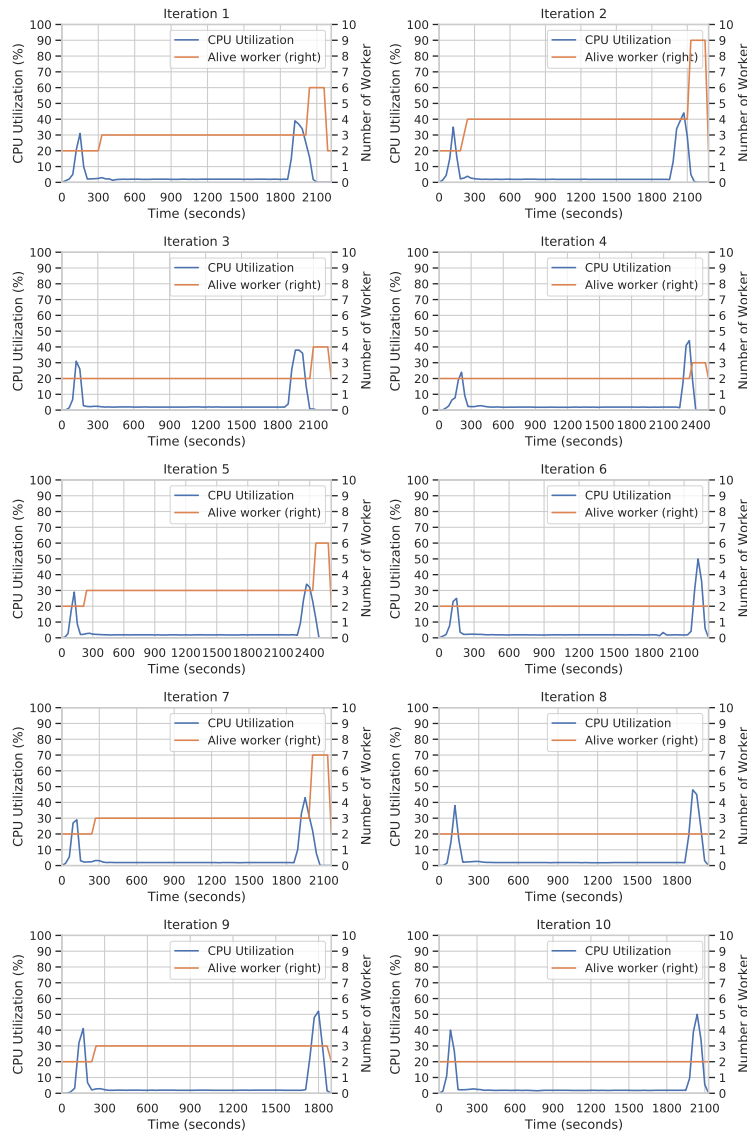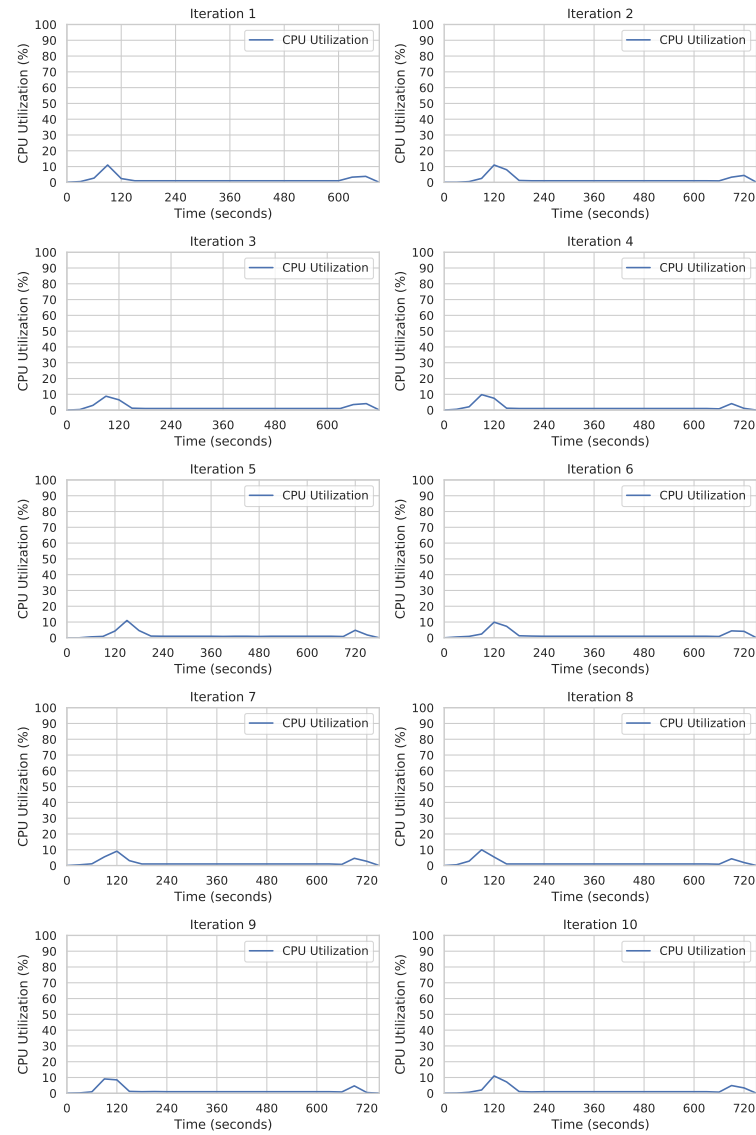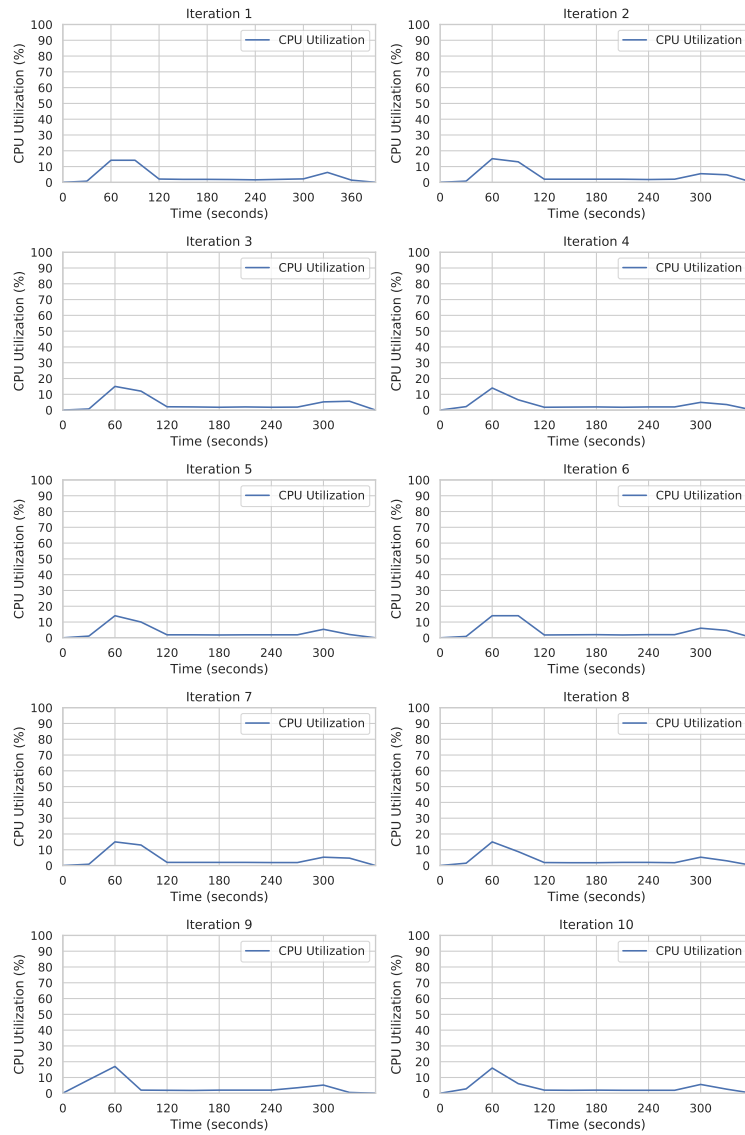
Figure B.2: Classification algorithm performance evaluation of the computing environment with 2 Apache Spark worker

B.1.2 Regression Benchmarks

Figure B.3: Classification algorithm performance evaluation of the computing environment with 5 Apache Spark worker

Figure B.4: Classification algorithm performance evaluation of the computing environment with 10 Apache Spark worker

Figure B.5: Classification algorithm performance evaluation of the computing environment with 1 Apache Spark worker and GPU acceleration enabled
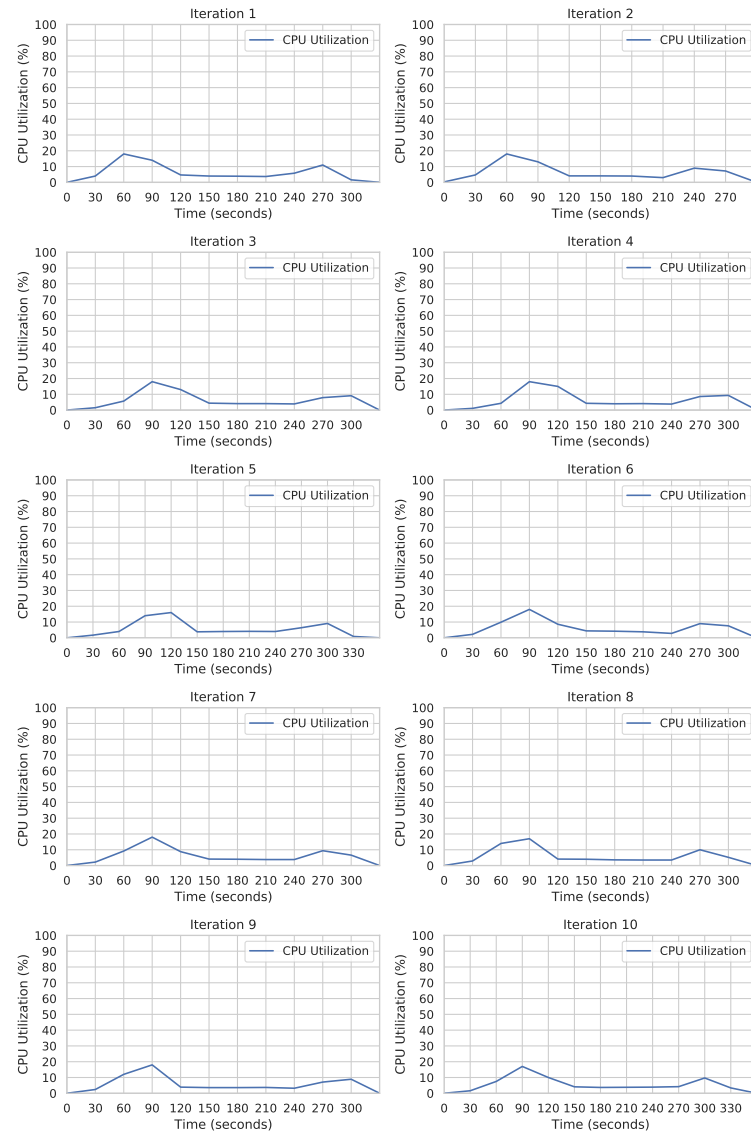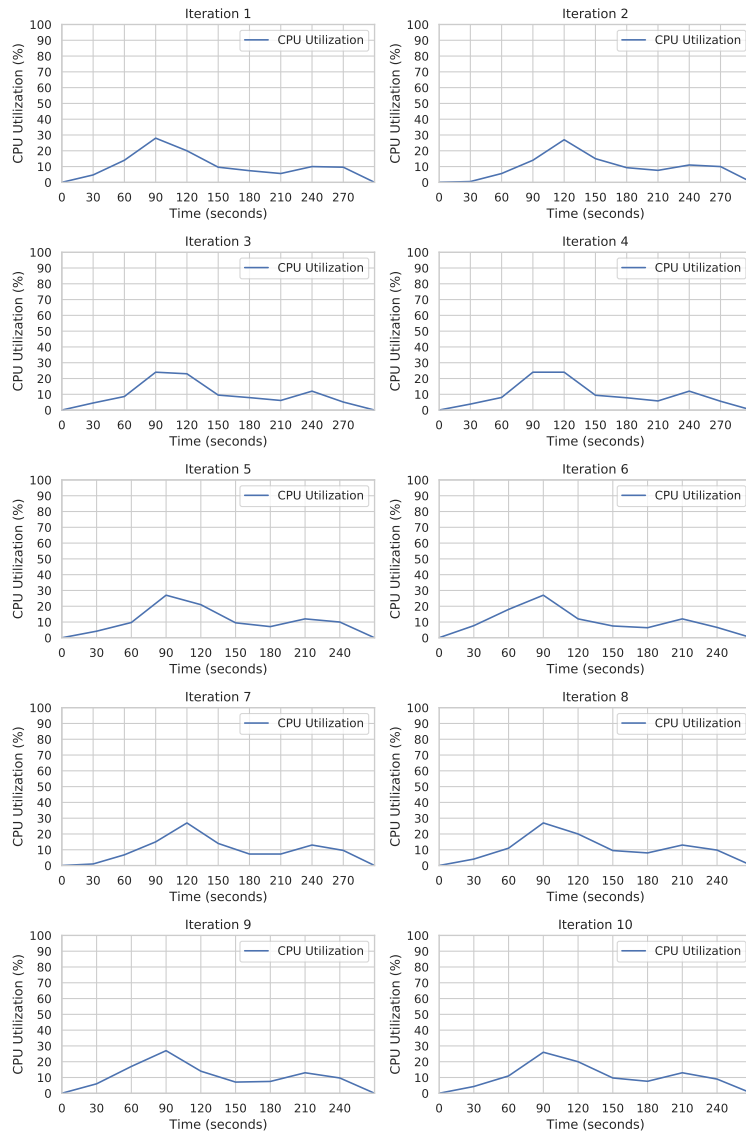
Figure B.6: Classification algorithm performance evaluation of the computing environment with 2 Apache Spark worker and GPU acceleration enabled

Figure B.7: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].

Figure B.8: Classification algorithm performance evaluation of the computing environment with 1 Apache Spark worker

Figure B.9: Classification algorithm performance evaluation of the computing environment with 2 Apache Spark worker
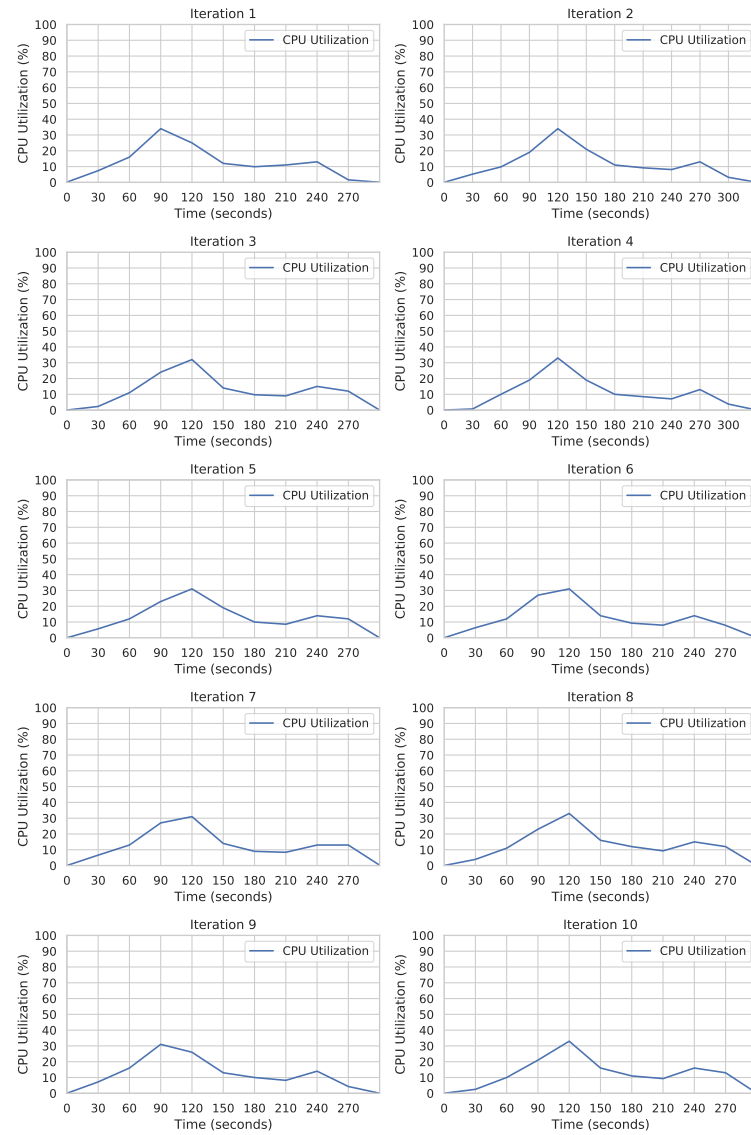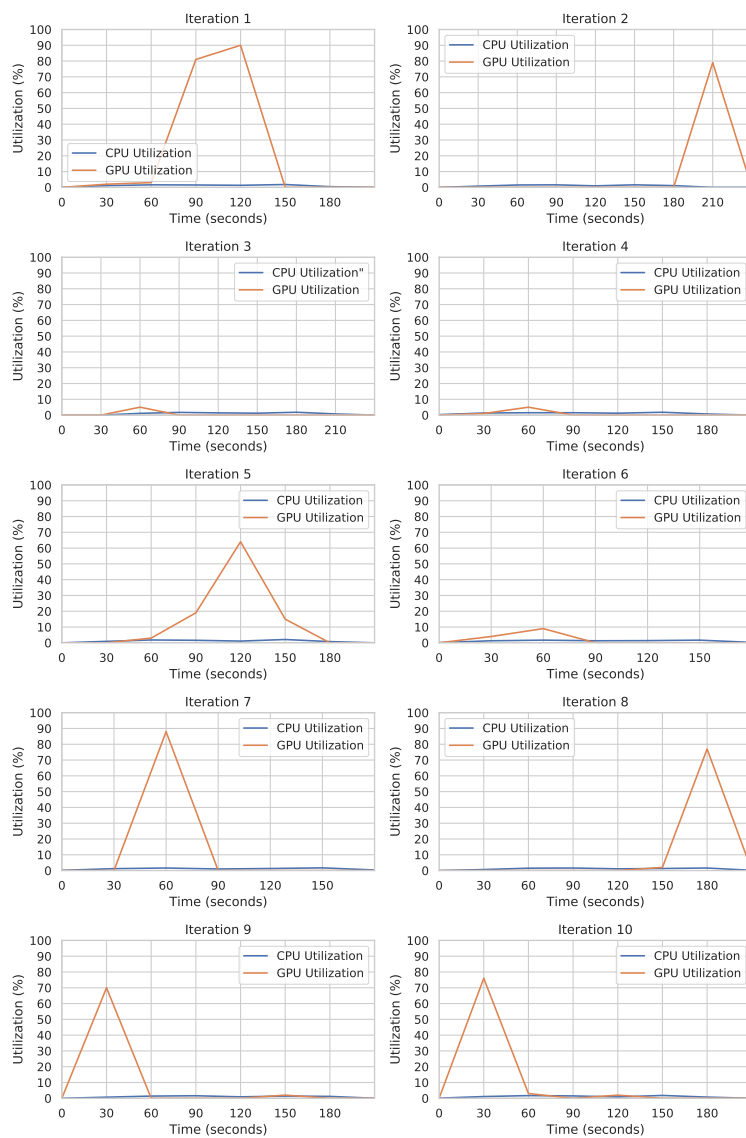
Figure B.10: Classification algorithm performance evaluation of the computing environment with 5 Apache Spark worker

Figure B.11: Classification algorithm performance evaluation of the computing environment with 10 Apache Spark worker

Figure B.12: Classification algorithm performance evaluation of the computing environment with 15 Apache Spark worker

Figure B.13: Classification algorithm performance evaluation of the computing environment with 1 Apache Spark worker and GPU acceleration enabled
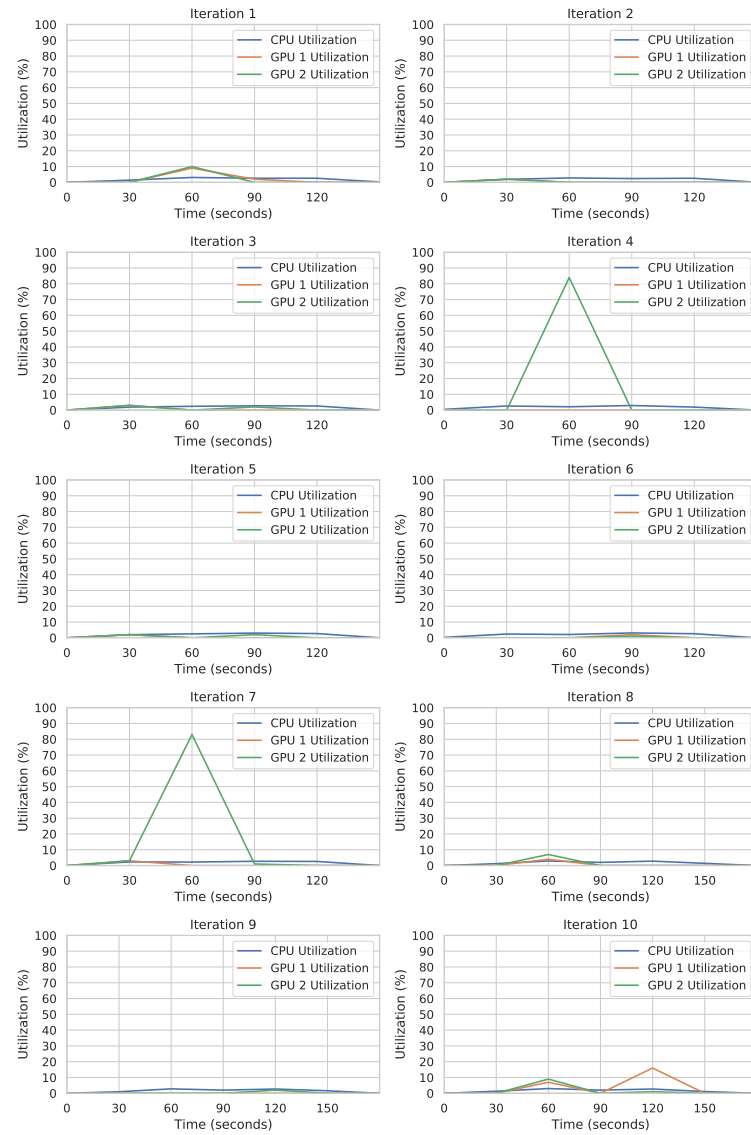
Figure B.14: Classification algorithm performance evaluation of the computing environment with 2 Apache Spark worker and GPU acceleration enabled
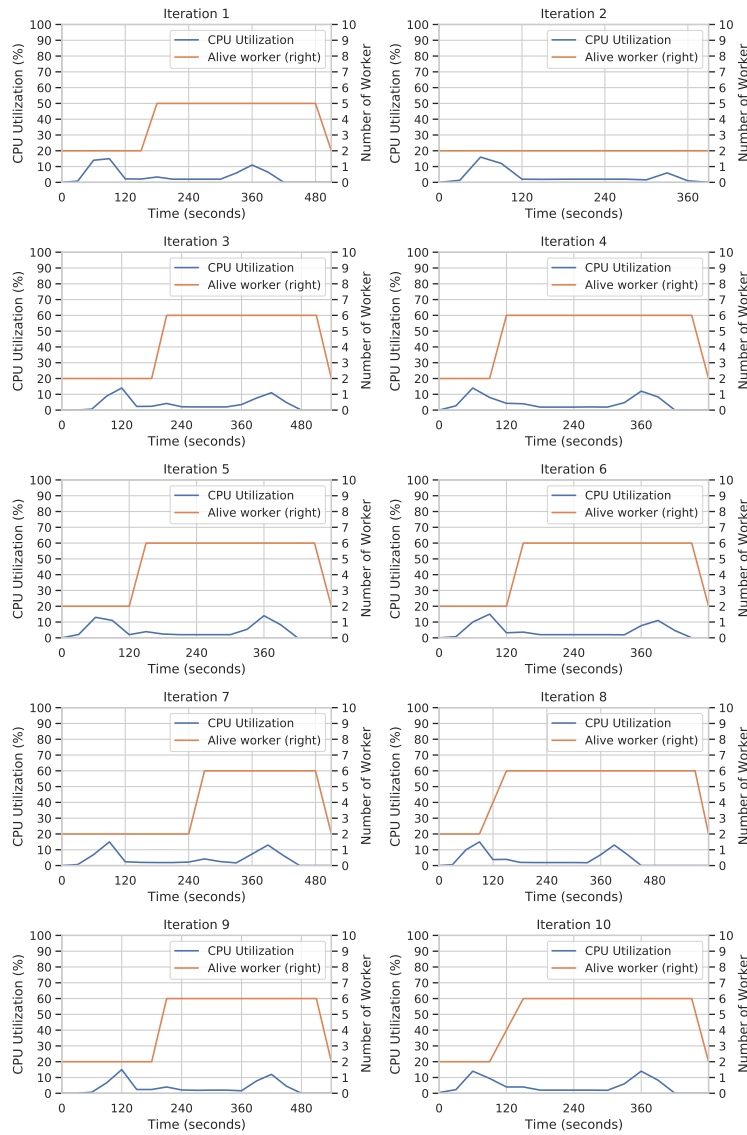
Figure B.15: Basic architecture of a GitLab CI/CD pipeline - Source: Authors own model, based on [Git].