Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Architecture and Deployment of a Scalable Machine Learning Pipeline in a Distributed Cloud Environment

Master thesis

Name of the study programme

Wirtschaftsinformatik

Fachbereich 4

submitted by

Lorenz Wackenhut

Date:

Berlin, 12.09.2021

First examiner: Prof. Dr.-Ing. Ingo Claßen

Second examiner: Prof. Dr. Martin Spott

Abstract

The architecture and deployment of complex machine learning (ML) pipelines for big data use cases pose many technical design challenges. NET CHECK, the company this thesis was conducted in cooperation with, faces several issues with their existing data architecture and therefore seeks to re-engineer their data and Machine Learning pipelines. This thesis investigates tools, platforms, and frameworks best suited to build scalable, reliable, and maintainable batch prediction pipelines in distributed cloud environments. Specifically, Apache Spark is used for data ingestion, data transformation, and training of the ML models, while Apache Airflow is utilized to orchestrate the individual components of the pipeline. The resulting application is then deployed on a Kubernetes cluster in a Microsoft Azure cloud instance. First, the identified tools and methods are described in detail before an application architecture build according to the research findings will be conceptualized. Subsequently, the implementation of a proof-of-concept application follows the steps dictated by the CRISP-ML(Q) process model. In conclusion, the deployed pipelines are evaluated according to their scalability, reliability, and maintainability, confirming that the re-development outperforms the monolithic legacy application in all categories.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACR	Azure Container Registry
ADLS	Azure Data Lake Store
AKS	Azure Kubernetes Service
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continious Delivery
CI	Continious Integration
CPU	Central Processing Unit
CRISP-DM	Cross-industry Standard Process for Data Mining
CRISP-ML(Q)	Cross-Industry Standard Process Model for the Development of Machine Learning Applications with Quality Assurance Methodology
DAG	Directed Acyclic Graph
DRY	Do not repeat yourself
IDC	International Data Cooperation
IP	Internet Protocoll
KPI	Key Performance Metric
ML	Machine Learning
MLLib	Machine Learning Library
NoSQL	Not only SQL
OS	Operating System
RAM	Random Access Memory
RDD	Resilient Distributed Data sets
SQL	Structured Query Language
VM	Virtual Machine
YARN	Yet Another Ressource Allocator
YAML	YAML Ain't Markup Language

1 Introduction

The most recent report of the International Data Cooperation (IDC) on the global data sphere shows that in 2020 more than 64 zettabytes of data was either created or replicated, which is an increase of 56% compared to 2019, exceeding even the most ambitious forecasts (Reinsel et al., 2021). Moreover, the trend of an annual rise in produced data is nowhere close to an end, as "the amount of digital data created over the next five years will be greater than twice the amount of data created since the advent of digital storage" (Reinsel et al., 2021). The majority of the data is created and stored in the public cloud, as these platforms provide many benefits, such as high availability, elastic scaling, and resource pooling (Assefi et al., 2017, p. 3492). The sheer volume and complexity of this continuously produced data poses serious technical challenges for the extraction of insights from the data sets. Conventional data mining techniques on standard hardware are not sufficient when it comes to big data, which is why sophisticated Machine Learning (ML) frameworks and large-scale distributed infrastructure platforms must be utilized (Zhou et al., 2019, p. 1). Additionally, advanced methods for automated knowledge extraction in the form of orchestrated ML pipelines are necessary to efficiently apply statistical learning algorithms on extensive data sets. An ML pipeline represents an automated workflow that includes all steps from the data ingestion over feature engineering and pre-processing until the training and prediction on unseen data points (Hapke & Nelson, 2020, 1. Chapter).

1.1 Research Problem and Status Quo

NET CHECK GmbH, the industry partner with whom this thesis was conducted in cooperation, faces the same issues discussed earlier. In the past, the company used to benchmark the coverage of telecommunication service providers by manually testing the signal strength across different regions. Nowadays, the firm expanded its services by analysing phone usage data from millions of users, providing insights to mobile carriers, government agencies, and other customers. The crowdsourcing department of the company is challenged with an ever-increasing stream of user data, which must be processed and stored in an efficient and performant manner. Until recently, the data infrastructure was built on top of a semi-on-premises server architecture that could not sufficiently satisfy the data science and engineering teams' requirements, as it was not scalable and performant

enough. Thus, the decision was made to migrate the operations into the Microsoft Azure cloud ecosystem in order to future-proof the data-driven business model with a scalable infrastructure. A big part of this migration process is transferring all existing data - and Machine Learning (ML) pipelines into the new environment. In the course of porting the pipelines to the new cloud infrastructure, they were also to be developed from the ground up with cloud-native technologies to make them horizontally scalable, reliable, fault tolerant, and maintainable for the use case of big data. Therefore, suitable tools, frameworks, and best-practice methods for building such applications must be researched and tested in a proof-of-concept evaluation so that future projects can use it as a starting point for the development process.

Table 1: Overview of a selection of data and ML pipeline deployed at NET CHECK

Pipeline	**Used Technologies**	**Problems**
Detection of the in-door/outdoor state of a user	• Type: Batch • Scheduling: Bash scripts • Processing: Python Pandas • ML: Python Scikit-Learn	• Scalability • Maintainability + Monitoring • Not cloud-ready • Reliability • No retraining with new data • No hyperparameter tuning
Detection of the activity state of a user (e.g., 'in vehicle')	• Type: Micro-Batch • Scheduling: Bash script • Processing: Java • ML: Java Tensorflow	• Scalability • Maintainability + Monitoring • Not cloud-ready • No retraining with new data
Postprocessing pipelines for data aggregation and analysis (e.g., the calculation of network coverage in a specific geographic location)	• Type: Batch • Scheduling: Bash scripts • Processing: SQL Query, Java • ML: N/A	• Scalability • Maintainability + Monitoring • Not cloud-ready

Table 1 gives an overview of currently deployed data – and ML pipelines at NET CHECK and states which technologies were used for the development and which problems are

experienced at present. Row number three summarises all query-based data pipelines which do not employ ML models. Even though the system architecture is fundamentally different for all of the stated examples, they share the same problems, such as no horizontal scalability, poor monitoring and maintenance capabilities, and no cloud support. Additionally, many more ML projects are planned, such as a pipeline that classifies samples according to radio cell parameters. Therefore, the research conducted in this thesis applies to a broad range of projects conducted at the company. However, only one example could be chosen for a proof-of-concept implementation due to time and resource constraints.

The specific use case that was chosen to demonstrate the capabilities of a scalable ML pipeline in a distributed cloud environment is the detection of a user's indoor/outdoor state according to phone usage data. This information is needed for further downstream analysis of the cell phone data and helps to contextualize the gathered sample points. A batch ML pipeline for this purpose is productionized on the old infrastructure and needs to be ported into the Azure environment. Additionally, the legacy pipeline is not scalable in a horizontal fashion, as it was built using conventional Python Data Science libraries and inherits problems related to fault tolerance and maintainability. Therefore, the new pipeline should be based on a system architecture that solves these issues by leveraging modern big data processing and ML tools, workflow orchestration methods, and distributed cluster management technologies.

1.2 Research Questions and Hypotheses

The circumstances explained in the previous section give rise to the following research questions, which are to be answered within the framework of this written work:

- Q1: Which tools, frameworks, and best-practice methods are suited for developing and deploying distributed big data Machine Learning pipelines for batch prediction?
- Q2: How can the identified tools, frameworks, and methods be used in a scalable, reliable, and maintainable system architecture?

These research questions can be synthesized into a hypothesis, which will be tested and verified by building a proof-of-concept batch ML pipeline with the example use case of an indoor/outdoor classification task:

- H1: A batch Machine Learning pipeline built with the identified tools, frameworks, and methods according to the proposed architecture is scalable, maintainable, and reliable and outperforms a centralized approach in these three characteristics.

1.3 Structure

The present research thesis starts by presenting the theoretical foundations and state-of-the-art tools, methods, and frameworks necessary for the development and deployment of scalable batch ML pipelines in distributed cloud environments. Afterwards the most suited architectures supporting the specifications will be discussed and evaluated, followed by an analysis of the currently employed legacy pipeline. The conception and methodology chapter continues by stating the requirements for the proof-of-concept ML pipeline and elucidates the implementation strategy according to the CRISP-ML(Q) framework. In the subsequent chapter a system and workflow architecture for the proof-of-concept application will be conceptualized before the prototype will be implemented according to the gained insights. Finally, the developed ML pipeline will be evaluated according to characteristics of data-intensive applications coined by Kleppmann, namely scalability, reliability, and maintainability (Kleppmann, 2017, pp. 1–25).

2 Theoretical Framework

The following chapter will cover the necessary theoretical foundations to understand common architectures and best practices for building scalable batch ML pipelines in distributed cloud environments. At first, essential terms will be defined and explained before the characteristics of data-intensive applications will be elucidated.

2.1 Definitions of Essential Terms

2.1.1 Big Data

The term big data refers to structured, semi-structured, and un-structured data sets that cannot be extracted and analysed with conventional tools and techniques because of their high volume, velocity and variety. In order to extract value from such extensive data sets, specific technologies, analytical methods, and high computing power are required. (Mauro et al., 2016, pp. 130–131)

The initial definition of big data dates back to a report of Laney which described it by three characteristic attributes starting with the letter V (Laney, 2001). Later on, several additional attributes were added, with most literature referring to five V's as the defining characteristics of big data (Sebei et al., 2018, p. 5):

Volume: The amount of data transcends the capacities of traditional storage and computing technologies and requires a highly scalable infrastructure (Laney, 2001, pp. 1–3).

Velocity: Describes the frequency and speed at which the data is produced and consumed and reaches from batches over near real-time to streaming (Laney, 2001, pp. 1–3).

Variety: Refers to the different data sources, data types as well as the structure and schema of the data (Laney, 2001, pp. 1–3).

Veracity: Is defined as the quality, trustworthiness, and completeness of the available data sets (Amir Gandomi & Murtaza Haider, 2015, p. 139).

Value: This dimension represents the hidden insight that can be extracted from the data set in order to create a net value for the business (Amir Gandomi & Murtaza Haider, 2015, p. 139).

2.1.2 Distributed Systems

As the computational resources needed to gain insights from big data are very demanding, current projects in this field increasingly utilize cluster- and cloud-computing technologies combined with advanced analytical algorithms to gain a competitive advantage (Gunawardena & Jayasena, 2020, p. 1; Migliorini et al., 2020, p. 1).

A collection of autonomous computing units, both software processes and hardware devices, is called a distributed system if they collaborate in a way that a user regards it as a single system (van Steen & Tanenbaum, 2017, p. 2). A distributed system designed for high availability and fault tolerance is typically called a computing cluster and is most commonly used for demanding calculations. The number of nodes in such a system can vary dramatically, and generally, new nodes can be added or removed dynamically, enabling horizontal scaling. (Kumar & Shilpi Charu, 2015, pp. 42–46)

This architecture approach enables the computation and storage of massive amounts of data on low-cost commodity hardware, making big data applications feasible. Traditionally, a company would have to build its own distributed storage and computation network, which is a time- and capital-intensive endeavour. Nowadays, cloud computing democratises distributed architectures for a wide variety of use cases.

2.1.3 Containerization

The last decade has seen a paradigm shift in the way modern applications are developed and deployed, from a monolithic design on a single server towards a distributed microservice architecture. Additionally, to all the advantages this new design approach introduces, such as increased scalability, improved resource management and better resilience through redundancy, there are also some drawbacks that must be addressed. The individual services in a microservice architecture are decoupled from each other, meaning that they are most likely developed separately with their own dependencies and infrastructure needs. This makes it very complex for the DevOps team to deploy and manage individual components without providing a dedicated virtual machine (VM) for every service, posing a massive financial and organisational problem. A solution to this predicament, which gained a lot of traction in the last years, is the containerization of applications in their own consistent environment. Linux container technologies allow the isolation of individual components, including libraries and Unix distribution, on a host machine without the additional overhead of virtualizing a whole operating system (OS). (Lukša, 2018, pp. 1–11)

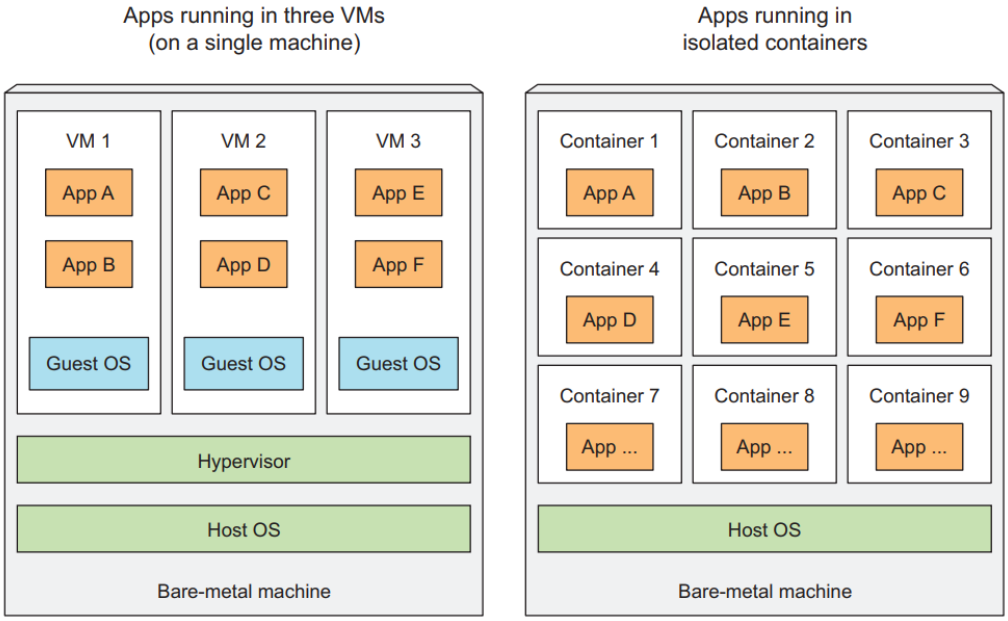


Figure 1: Container technologies versus virtual machines (Lukša, 2018, p. 9)

Figure 1 illustrates how VMs are deployed on a host server, unlike containers running on a bare-metal machine. A VM ships with a fully functional OS including a kernel and relies on a hypervisor to allocate the host resources as well as to perform CPU calls from inside the VM. Therefore, VMs are big in size, slow to start, and take up a considerable portion of the hosts' resources. On the contrary, containers do not need a hypervisor and run as a regular process on the host OS, making them light in nature, fast to boot, and easy to port. (Lukša, 2018, pp. 1–11)

2.1.4 Cloud Computing

Cloud computing enables companies to use IT resources directly via the internet and dynamically scale the required computing power and storage space according to their needs. The limits of CPU performance and storage space are virtually non-existent and thus suitable for even the most complex calculations and analyses. (Thompson & van der Walt, 2010, p. 2)

According to (Balachandran & Prasad, 2017, p. 1114), cloud computing is defined by five characteristics that differentiate it from an on-premises datacentre:

Rapid elasticity: Resources are dynamically provisioned and released to meet the fluctuating needs of users. This process can be automated in order to scale an application both horizontally and vertically, depending on the requirements.

Broad network access: The services are made available via the internet using standard protocols and can be used on a variety of platforms and operating systems. Virtual networks between cloud instances facilitate high-speed data transfers.

Resource pooling: In the case of a public cloud approach, the service-provider's resources, both physical and virtual, are pooled and can thus be used by several users simultaneously by dynamically allocating computing power, storage space, and software resources.

On-demand self-service: End users can automatically claim cloud resources on demand without the need for personal interaction with the service provider. Infrastructure as code approaches enable the automation of resource initiation.

Measured service: Resource consumption is automatically monitored and reported, thus remaining transparent for both the end users and the service provider.

Most service providers differentiate their offering models into a public and private cloud approach which changes the dynamic on how resources are distributed. In a public cloud, resources are allocated anonymously to the users from a shared pool. As the service provider can utilize its hardware efficiently through flexible use, the final price for the end consumers is comparatively low. Since the servers are usually distributed in data centres all over the world, users have access to the cloud within seconds regardless of their geolocation and are therefore stationary independent. In the private cloud model, agreed servers and services are permanently assigned to the contracting company for a specific period. This has the advantage that the customer can decide where the infrastructure will be operated, which can mitigate some legal and security issues. (Winkelhake, 2017, pp. 50–51)

In particular big data applications can benefit from a deployment in a cloud environment, as an on-premises approach poses massive financial hurdles and limits the flexibility of the project. Following the advantages of cloud computing for demanding use cases will be described:

Scalability and elasticity: Computationally intensive transformations and evaluations can be conducted without hardware limitations, as the allocated resources can be dynamically scaled to the individual task profile (Winkelhake, 2017, pp. 88–89)
In addition, the effort of operating a cluster computing network is dramatically reduced, as nodes can dynamically be added and removed.

Availability and reliability: Cloud computing environments offer high reliability and availability defined by the time a service is operational without failure and the probability of a service running functionally correctly, respectively (Mesbahi et al., 2018, p. 2). This is especially important for long-lasting calculations on big data sets, such as the training of machine learning models.

Cost and democratization: In the past, small- and medium-sized companies did not have the financial means to build their own big data architecture, as this involved cost-intensive capital advances for on-premises infrastructure. Cloud providers introduce the possibility to only pay for factually used services in a pay-as-you-go model. (Al-Aqrabi et al., 2015, pp. 88–89)

2.2 Characteristics of Data-Intensive Applications

At first, the necessary attributes of general big data applications will be clarified before deducting these characteristics for ML pipelines in specific.

2.2.1 Scalability

The concept of scalability refers to the ability of a system to handle an increase in load, both in terms of computing intensity as well as data volume. In the case of a significant load surge, hardware resources can be added in a vertical or horizontal manner. Scaling vertically, also called 'scale-up', means adding resources such as memory and CPU cores to a single machine. The pendant to scale up corresponds to 'scale-out', or horizontal scaling, and refers to an addition of computing units in parallel in order to distribute the load. Ideally, the software components of the system dynamically adapt to the added hardware resources without requiring a significant change in the source code. (Kleppmann, 2017, pp. 10–18)

The strategy for achieving scalability in a system highly depends on the requirements of the application and the specific use case. Both the load parameters and the performance metrics for batch processes are profoundly distinct from a web-endpoint, as the first is typically measured by the throughput of data in a specific timeframe and the latter by the latency and thus resulting response time. Therefore, scalability must be included in the design process of an architecture from the beginning on and continuously adapted to the changing requirement patterns as the system matures. (Kleppmann, 2017, pp. 10–18)

2.2.2 Reliability

A reliable system is designed in such a way that it performs well and as planned under the specified data volume and processing load and can cope with anticipated faults, both software and hardware errors, in a resilient manner. (Kleppmann, 2017, pp. 6–10)

An advantage of how cluster computing systems handle fault toleranceis by adding redundant nodes, which help to restore lost data in the case of a machine outage. This strategy helps to prevent a system crash by distributing the responsibility over many units instead of relying on a monolithic architecture that acts as a single point of failure. (Kleppmann, 2017, pp. 6–10)

In the case of data pipelines, reliability is of most importance, as downstream data consumers rely on the integrity of the produced output. An ML pipeline that produces wrong predictions can ultimately be more harmful than no pipeline, as business decisions on corrupted information can lead to serious consequences. Therefore, testing the individual processing steps with unit tests and the overall pipeline with integration tests is a critical step in developing reliable data flows.

2.2.3 Maintainability

Modern software applications are constantly enriched with new features, adapted to new use cases, or ported to new environments and, therefore, an ever-changing and evolving product. In addition, many different people are using, maintaining, and expanding the code over the application's lifecycle. As a result, it is critical to prioritize maintainability when developing applications that are meant to last for an extended period.

A modular architecture not only helps to minimise failures by removing centralized weak points but also facilitates increased maintainability, as associated functionality is grouped in modules and not spread over the entire code base. Software systems that realise a high cohesion within classes and modules and low coupling across different functionalities lead to more reliable and maintainable systems (M. Hitz & Behzad Montazeri, 1995, p. 1).

2.2.4 Batch Processing versus Streaming

Batch processes are designed to transform a set amount of input data in scheduled intervals to produce a specific output. The input data is assumed to be bounded, meaning that it is of a finite and know size. The runtime for systems of this kind usually ranges from several minutes to days. The individual components of such a pipeline can run independently and asynchronously, as long as enough data from the upstream task is available.

The output of one task is typically designed to be the input of the next, which enables chaining of processing steps. Distributed batch processing frameworks operate stateless until the final output is generated, making it possible to retry failed stages and discard corrupted intermediate results safely. This has the effect that the output of the process is the same, even if multiple stages had to be retried, making the framework highly tolerant against faults. (Kleppmann, 2017, pp. 389–430; Singh, 2019, pp. 50–51)

The underlying assumption of finite data for batch processes is not an accurate abstraction of reality, as many systems continuously produce data without a pause. Therefore, data sets used for batch processing must be separated into segments of fixed size. A typical time frame for this separation is once per day or every hour, which is certainly too slow for time-sensitive use cases like fraud detection. Stream processing abandons the concept of time intervals altogether and instead handles data events as soon as they become available. Instead of a file system, stream processes handle events provided by message brokers, which store, validate and route the data between the source system and the streaming pipeline. (Kleppmann, 2017, pp. 439–481; Singh, 2019, pp. 50–51)

2.3 Machine Learning

Machine Learning (ML) is a subcategory of the broader field of artificial intelligence (AI) and studies methods and approaches to designing intelligent systems capable of learning from data. As stated by Wang et al., an ML system generally tries to mimic human learning capabilities and should be able to both extrapolate existing knowledge and acquire new information and skills in a self-improving manner (Wang et al., 2009, p. 1). According to T. M. Mitchell a system has the ability to learn if its performance of achieving a specific task improves with an increase in experience for the given problem (T. M. Mitchell, 1997).

The general goal of ML is to develop a statistical model which is capable of predicting an output variable Y with a function ƒ and the input variables $X_1, X_2, \ldots, X_n$. The relation of Y towards the feature vector X can be formalized by the term $Y = f(X) + \epsilon$, where ϵ represents a random error. Typically, the model will be trained with a set of observed data points in order to estimate f. (James et al., 2017, pp. 15–24)

In the case of supervised ML algorithms, the training data set must include a response variable y_i, also called label, which is used to fit the model. The model learns to relate the response variable to the feature vector and then performs predictions on unseen data

points. There are two types of problems that can be solved with supervised learning algorithms: classification and regression tasks. (James et al., 2017, pp. 26–28)
Contrary, unsupervised ML algorithms can learn on data sets that lack a response variable. In this case, the model learns to relate the input variables or observations to each other instead of a given label (James et al., 2017, pp. 26–28). Examples of unsupervised statistical learning are clustering, anomaly detection, dimensionality reduction, and association rule learning (Géron, 2017, 1. Chapter).

The third category of ML is defined as reinforcement learning, which uses a substantially different strategy to train the model, called 'the agent' in this context. The agent gets rewarded or penalized according to his actions, which he chooses and performs by inspecting his environment. Depending on the feedback, the policy, which defines the strategy for possible actions, is continuously updated to maximize the future reward. (Géron, 2017, 2. Chapter)

Analogous to how batch processing can only transform a set amount of data in defined intervals, batch learning algorithms cannot learn continuously on new data but are trained once and then deployed in production. If enough new data justifies the retraining of the model, the algorithm must be trained from scratch with the union of old and new data. The new model's training, evaluation, and deployment can be automated using an ML pipeline, which enables the scheduled retraining in fixed intervals or as soon as a certain threshold in data size is reached. Contrary to batch learning, and similar to stream processing, online learning systems are feed a steady flow of new data on which they can continuously improve. This type of learning system is helpful if the application needs to be adapted to a constant change in data patterns or if computing resources are scarce. (Géron, 2017, 2. Chapter)

2.3.1 Machine Learning Pipelines

For a long time, the domain of ML and AI has been primarily dominated by Mathematicians and Data Scientists, which created elaborated statistical models in a manual and time-intensive effort. Due to the absence of appropriate tools and information, experiments and developments with ML algorithms were often conducted with insufficient consideration towards the production environment (Kakarla et al., 2021, p. 299). These models were then typically deployed in production by Software Engineers or a DevOps team. However, as

software applications cannot be deployed and forgotten, ML models should not be regarded as one-time projects but must be continuously monitored, maintained, and updated. Otherwise, the predictive power of the algorithm will degrade over time with a change in data patterns. Additionally, models and pre-processing steps, which have been developed locally in a notebook environment, often lack the ability to cope with large amounts of data and tend to be less robust and maintainable compared to well-tested software applications. (Hapke & Nelson, 2020, 1. Chapter)

Hapke and Nelson draw a comparison to the car industry, which was revolutionized by the invention of the assembly line in 1913, which enabled the automation of a highly manual process and therefore made the production of the automobile time and cost-effective (Hapke & Nelson, 2020, 1. Chapter). Analogous to an assembly line, an ML pipeline is a concept that enables the automated orchestration, management, monitoring, and deployment of ML models. Unfortunately, tools and methods used to accelerate and standardize the software development and deployment of traditional applications are not well suited for the domain of ML, as the requirements for this field tend to be considerably different, and a variety of steps are involved (Kakarla et al., 2021, p. 299).

However, given the increasing popularity of ML over the past decade, a wide variety of new tools and best practices around the development and deployment of intelligent algorithms has become available. Considering that the expectations in prediction performance and the amount of data available are constantly rising, the implementation of scalable, reliable, and maintainable data pipelines for ML tasks is an essential success factor for data-driven projects. Well implemented ML pipelines help accelerate, automate, and standardize the development of ML workflows considerably and enable the achievement of reliable results faster. (Migliorini et al., 2020, p. 2)

According to Hapke and Nelson, a data pipeline for ML use cases should include steps that allow the following tasks (Hapke & Nelson, 2020, 1. Chapter):

1. Version control and validity check for new data, allowing the training process to rerun with unseen data.
2. Pre-processing and feature engineering in order to inject domain-specific knowledge and to clean and structure data into a viable format.

3. Effective and performant model training including hyperparameter tuning and progress tracking.
4. Evaluation of the trained and tuned models according to appropriate metrics.
5. Deployment of the validated model into the production environment.

Processing Steps in ML Pipelines

Following, the processing steps in an ML pipeline are presented, allowing the execution of the before-mentioned tasks. Naturally, the specific components of an ML pipeline can vary depending on the use case and the deployed technologies, which is why some of the following steps might be neglected, and others can be added. Figure 2 shows how the individual parts of an ML workflow can be arranged in a pipeline.

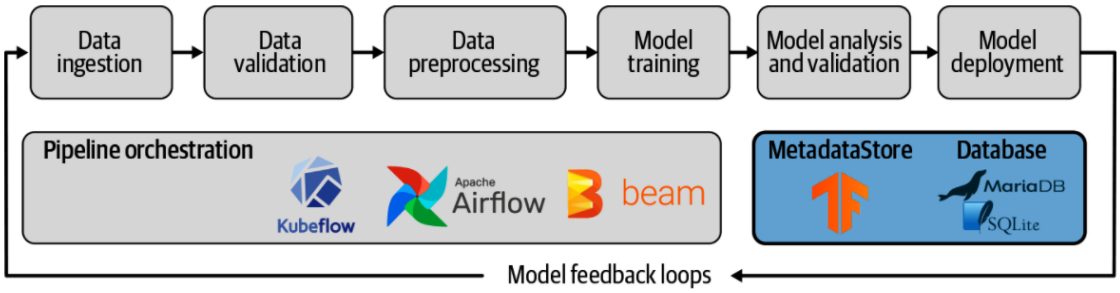


Figure 2: Visualization of the processing steps in an ML pipeline (Hapke & Nelson, 2020)

Ingestion and Validation

The ingestion marks the first step in any data pipeline and allows the loading and processing of the data into a digestible format by the downstream components. If the raw data originates from different source systems, formats, or tables, it must be consolidated into a unified schema (Hapke & Nelson, 2020, 1. Chapter). The ingestion strategies for different data types can diverge substantially from each other, as structured tables are treated very differently from text for natural language processing or images for computer vision. Another important consideration for the ingestion design is the frequency of the processing interval. The ingestion requirements of batch processes differ substantially from streaming pipelines, as the latter typically uses message brokers and event logs instead of conventional file systems as a tap for incoming data (Kleppmann, 2017, pp. 479–481).

After the data has been ingested, it is crucial to validate its schema, potential anomalies, and statistical drift from previous versions, as a reliable and robust model can only be trained on high-quality data. This step also ensures that the data can be passed on to the

feature engineering seamlessly. An automated validity check with implemented warning can decrease the risk of training a model with imbalanced or otherwise unfit data and prevent the loss of valuable time and resources. (Hapke & Nelson, 2020, 4. Chapter)

Pre-Processing

The data pre-processing encompasses all transformation steps after the ingestion, which convert the raw data into a format that the ML algorithm can receive. Firstly, the data must be cleaned by adopting a strategy for handling outliers, null-values, and other anomalies that could either degrade the performance of the trained model or even cause exceptions during the learning process (Singh, 2019, p. 17). Following, new features can be engineered by combining, calculating, and deriving attributes from original columns and injecting domain-specific knowledge into the data set (Migliorini et al., 2020, p. 8). Finally, the features must be converted into a suitable format which varies depending on the used ML algorithm. Typical steps include scaling and normalization, one-hot-encoding, and dimensionality reduction techniques like Principal Component Analysis or Linear Discriminant Analysis. The last step is the feature selection, in which a subset of the most promising attributes is selected in order to reduce the dimensionality of the optimization problem. (Zhou et al., 2019, p. 2)

Model Training and Tuning

The model training is the most prominent component of the ML pipeline and uses the data set which was transformed and curated in the previous steps to fit a statistical learning algorithm. The advantage of a scalable pipeline in a cluster computing environment is the possibility of training multiple models in parallel and letting them compete against each other for the best overall performance (Hapke & Nelson, 2020, 1. Chapter). Hyperparameter tuning refers to optimizing the learning parameters of the algorithm that control the training process. A popular technique for choosing the best combination of hyperparameters is the so-called grid search method, in which a predefined set of parameters is evaluated in an extensive search. Every combination of parameters is assessed by cross-validation utilizing defined performance metrics. (I. Syarif et al., 2016, pp. 1502–1503)

As this process is very resource-intensive, it can take a considerable amount of time to run it on a single machine, so distribution on multiple nodes can dramatically speed up the runtime (Migliorini et al., 2020, p. 4).

Model Evaluation

After the models have been trained, an elaborated evaluation of the prediction quality must be performed to testify the validity of the estimator. The models can be tested against each other on performance metrics such as ROC (receiver operating characteristic curve) and AUC (area under the ROC curve), as well as other indicators such as F1 score or accuracy depending on the characteristics of the ML use case (Migliorini et al., 2020, p. 5). It might also be helpful to verify the final model on a more extensive validation set than the one used during the training phase to check the behaviour on previously unseen data (Hapke & Nelson, 2020, 1. Chapter).

Model Deployment

The last step in the ML pipeline life cycle is the deployment of the trained, tuned, and evaluated model into the production environment. Instead of making this process a one-time effort that must be manually repeated every time a new model version is trained, a strategy for the continuous integration (CI) and delivery (CD) of future releases should be implemented. Unfortunately, DevOps strategies used in Software Engineering are only partially applicable for the delivery workflow of ML models, as not only the pipeline but also the model itself, the input as well as the output data have to be versioned. Therefore, the subdisciplines of DataOps and MLOps emerged, specifically tackling the problems associated with the life cycle of data and ML models. (Sato et al., 2019)

There are many different strategies on how to deploy learning algorithms in production, of which the most prominent ones are depicted in the following:

Competing Models: Hosting multiple models simultaneously and letting them perform the same task in parallel opens up the possibility of running A/B tests on real-life data and further evaluating the performance (Hapke & Nelson, 2020, 1. Chapter). As the data for predictions has to be routed to the appropriate models, considerable amounts of statistics have to be gathered for conclusive results, and additional computing resources must be provided. This approach can increase the complexity of the infrastructure significantly (Sato et al., 2019).

Shadow Models: Similar to the previous approach, two models are hosted in parallel, of which the newer one is deployed in shadow mode. This way, the new iteration of the

algorithm can be tested in-depth for the last time before it replaces the old one in the application. (Sato et al., 2019)

Online learning: As mentioned in chapter 2.3, an online learning model can continuously be improved by feeding it new observations. This strategy requires a substantially different infrastructure than offline learning and poses additional challenges as the model cannot be versioned in the way a static model would be (Sato et al., 2019). This approach is particularly interesting for stream processing use cases with fast-changing data patterns and, therefore, a high adaptation need.

After the model has been deployed successfully, it is crucial to monitor the system's performance precisely and implement warning systems that alarm product owners in the case of anomalies. A logging system that helps to persist statistics on the input, output, and key performance indicators (KPI's) can help detect problems like over- or underfitting, which were missed with training instances. (Sato et al., 2019)

Machine Learning Pipelines for Big Data

Even without large data sets, ML pipelines tend to be computationally intensive, as the preprocessing of the data and the training and tuning of the models can get fairly complex rather quickly. In combination with big data, the operation of a scalable pipeline on a single machine becomes inefficient and impractical as the costs for vertical scaling increase disproportionally to the gained performance. Horizontal scaling in a cluster environment provides a better solution for most use cases, as the load distribution over a large number of commodity computing units keeps the operating costs down, while providing a significant speedup. Even though it is possible to build a custom cluster computing setup on-premises, modern cloud computing environments offer highly flexible infrastructures that enable large-scale processing with little capital expense and fast deployment times. (Weber, 2020, pp. 6–8)

Furthermore, conventional software frameworks and tools, which are used for the processing, training, and predicting normal-sized data sets, are not equipped for the use case in cluster computing systems, as they lack the ability for distribution. Therefore, a machine learning pipeline that is supposed to scale on a cluster needs to combine large-scale parallel computing capabilities with state-of-the-art machine learning algorithms to achieve high data throughput (Zhou et al., 2019, pp. 1–2). Additionally, the individual steps in such a

pipeline must be as independent as possible to enable partial reruns in case of a stage failure. To achieve this kind of fault tolerance coupled with monitoring of the data lineage, workflow management solutions must be implemented (R. Mitchell et al., 2019, p. 4537).

3 State of the Art Approaches

The following chapter introduces concepts and frameworks essential for building a scalable, maintainable, and reliable batch ML pipeline in a distributed cloud environment. The presented methods and tools are adapted to the requirements of the outlined research objectives, which is why the tooling selection is limited to horizontally scalable cloud-native technologies providing batch prediction capabilities. After the necessary components are elucidated, possible architectures supporting the specifications will be discussed and evaluated.

3.1 Big Data Processing with Machine Learning

Building a scalable ML pipeline in a distributed environment poses many challenges, as sophisticated learning algorithms are necessary, and a framework that supports large-scale processing, training, and prediction based on massive data sets in a performant manner is required (Zhou et al., 2019, p. 1). Unfortunately, not many frameworks used to combine these data engineering and ML capabilities in one coherent system, but recently a big influx of open source and proprietary software solutions became available that promise to deliver just that. One of the most popular big data analytics engines is Apache Spark, which offers a powerful distributed processing engine, support for most cluster management systems, and integrated ML libraries, which include all common regression and classification algorithms (Migliorini et al., 2020, pp. 1–2). The following section introduces Spark's architecture, characteristics, and capabilities in the context of data-intensive ML pipelines.

Spark was first introduced in 2009 as a research project at UC Berkley's AMPlab and designed to tackle the problems developers faced with MapReduce when implementing distributed ML algorithms. The MapReduce programming model is based on acyclic graphs, allowing the underlying compute engine to schedule the data flow and tolerate faults without interference. A problem arising with this programming paradigm is the fact that ML algorithms are iteratively trained, which requires a repeated scanning of the whole data set. In the case of MapReduce, this meant that a separate job had to be written for each

training cycle, which loads the entire data set all over again. The initial release of Spark addressed this shortcoming by providing a functional programming API that supported the representation of iterative workflows efficiently and a new processing engine enabling parallel computing. In their first paper introducing Spark, named 'Spark: Cluster Computing with Working Sets', the authors presented how the introduced changes to the compute engine led to a speedup by the factor ten in ML tasks compared to the at that time highly popular Hadoop platform. (Zaharia et al., 2010, p. 1)

Nowadays, Spark is part of the Apache Foundation and licensed as open-source software with thousands of monthly contributors all around the world. The project calls itself a unified analytics engine and strives to combine a broad range of computing tasks in one platform ranging from streaming to batch processes over ML and many more. (Apache Software Foundation, 2021g)

3.1.1 Components of Apache Spark

The Spark platform consists of five main layers illustrated in Figure 3: Storage, resource management, compute engine, ecosystem, and APIs. A powerful feature of Spark is the execution optimization across different layers and components through lazy evaluation. Before any code execution, a computational graph is built, including all previous transformations to optimize the runtime (Chambers & Zaharia, 2018, pp. 4–5; Dugre et al., 112019, p. 41).

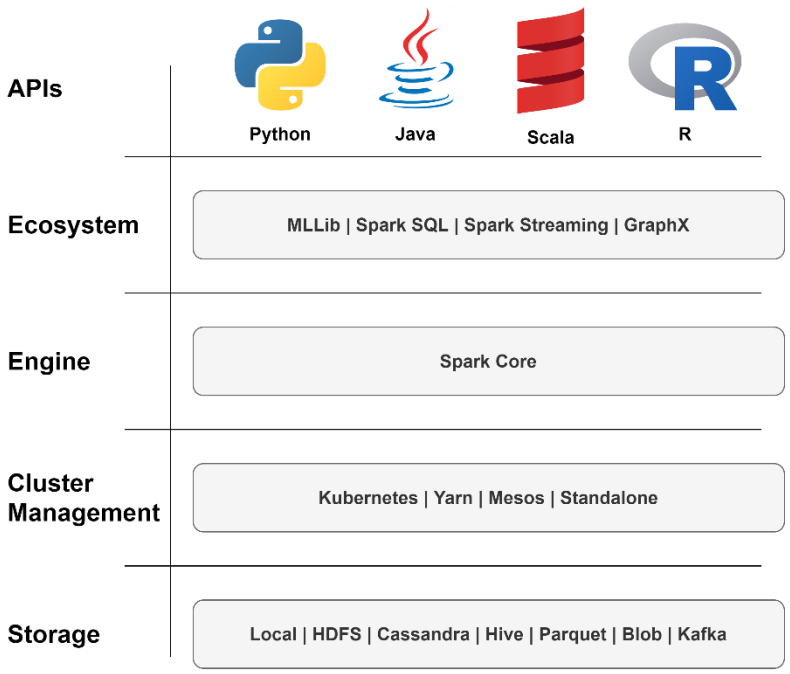


Figure 3: Components of Apache Spark – based on (Singh, 2019, p. 5)

Contrary to the Hadoop ecosystem, Spark does not provide any permanent storage option itself. Instead, it offers a wide variety of connectors to virtually any storage system, ranging from cloud providers such as AWS and Azure over NoSQL key-value stores such as Apache Cassandra to message brokers, including Apache Kafka (Apache Software Foundation, 2021a, 2021d; Chambers & Zaharia, 2018, p. 5).

Spark enfolds its true potential in a cluster environment, where the processing tasks can be distributed over potentially thousands of nodes. The actual application architecture resembles a conventional master-slave design and consists of executor processes, which handle the computation on the data set, and a driver process that assigns tasks to the executors and maintains meta-information about the execution. Every Spark deployment can have an arbitrary number of executors but only one driver. Both the driver and the executors run java processes distinct from each other, which allows them to operate self-sufficient but in a coordinated manner. If sufficient RAM is available, all computations are performed in-memory and eventually written to an external data storage such as Hadoop Distributed File System (HDFS). In order to manage the job distribution, cluster management systems are necessary, which allocate resources to the executors and spin up the individual worker machines. (Chambers & Zaharia, 2018, p. 14; Zhu et al., 2020, pp. 117–118)

Figure 4 shows how the Spark application is distributed over worker nodes by the cluster manager. Spark provides support for resource managers such as YARN (Yet Another

Resource Negotiator) and Mesos as well as Kubernetes since it has been officially declared as production safe by version 3.1 (Apache Software Foundation, 2021f).

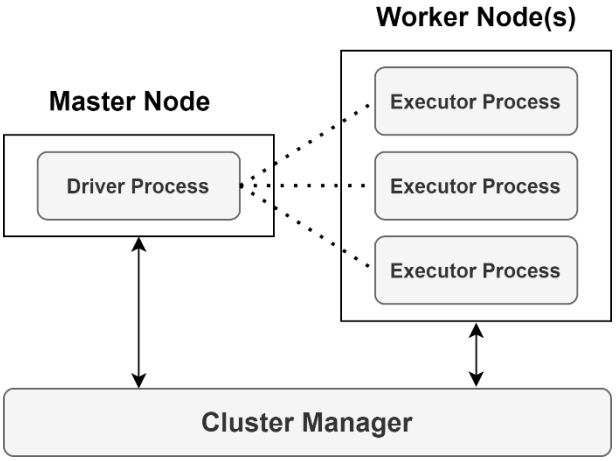


Figure 4: Spark application architecture – based on (Chambers & Zaharia, 2018, p. 14; Zhu et al., 2020, p. 118)

The Spark engine is built around the abstraction of immutable and fault-tolerant data sets called RDDs (Resilient Distributed Data set) which set the foundation for high-level data structures such as data frames and sets (Dugre et al., 112019, p. 41). Additionally, many APIs are provided, enabling the development of additional libraries, of which four are included in the official release: GraphX, Structured Streaming, Spark SQL, and MLLib (Singh, 2019, pp. 8–9).

3.1.2 Submitting Applications to Spark

Spark ships with a built-in script that simplifies the launching process of applications in a cluster environment called spark-submit. Spark-submit standardizes and abstracts the deployment for all supported cluster management systems and therefore eliminates the need for separate configuration efforts with different schedulers. The submitted application is sent to the driver process and gets distributed across the executors for parallelisation. As most applications depend on multiple modules and libraries, all dependencies must be packaged with the main application prior to submission. In the case of Scala or Java, the package is bundled into a jar file, contrary to Python, where a zip or egg package must be supplied. (Apache Software Foundation, 2021h)

Following the structure and the most commonly used arguments for spark-submit will be explained. First, the spark-submit script must be called from `./bin/spark-submit`, after

which a list of arguments can be specified. Finally, the actual application is stated (Apache Software Foundation, 2021h):

`--master`: The URL for the master node in the cluster is stated, where the application is submitted to. A possible example for Kubernetes in the Microsoft Azure cloud could be the following URL:
`Kubernetes://https://dns-05nclorenzaks.hcp.westeurope.azmKubernetes.io:443`

`--deploy-mode`: This argument states if the Spark driver process should be deployed on the local machine (`client`) or the master node (`cluster`). It is also possible to deploy the whole spark cluster locally using the computing cores as nodes (`local`). The latter approach is only recommended for development and testing purposes.

`--conf`: Many different configurations can be provisioned while submitting an application depending on the infrastructure and type of program. A common example would be to declare the numbers of Spark executor processes:
`spark.executor.instances=n`

application-file: The path to the packaged application with its dependent libraries and modules must be visible for all nodes inside the cluster. It is also possible to submit an URL to a cloud storage provider such as Microsoft Azure or Amazon AWS.

application-arguments: Finally, any additional application arguments can be stated, which will be parsed by the application's main class.

```
./bin/spark-submit \
    --argument-1 <parameter_1> \
    --argument-2 <parameter_2> \
    --argument-n <parameter_n> \
    <application-file> [application-arguments]
```

3.1.3 Machine Learning with Spark MLLib

Conventional ML packages such as Scikit-Learn (Scikit-Learn Developers, 2021) for Python or Caret (Kuhn, 2019) for R offer great performance and a rich feature set for single

machine use but are very limited when it comes to distribution in a cluster setting. Until Spark introduced MLLib (Machine Learning Library) in 2013, ML tasks on big data sets increasingly started to become a problem, as the available software landscape at that time was not well suited for large-scale distribution (Singh, 2019, p. 8). The release of Spark MLLib changed that by offering a package that includes methods for data loading and cleaning, feature engineering as well as training and evaluation of sophisticated statistical learning algorithms in a distributed fashion. The pipeline API is a fundamental principle for orchestrating processing, training, and prediction steps in MLLib. It enables the combination of multiple ML tasks in a unified pipeline and therefore standardizes the workflow for training and prediction jobs in the Spark ecosystem. (Chambers & Zaharia, 2018, pp. 408–410)

3.2 Cluster Management

Apache Spark ships with a build-in standalone scheduler and supports other third-party resource management systems such as Mesos and YARN. The task of the cluster manager is to maintain the physical nodes in a cluster environment and allocate resources to the workers. Both Mesos and YARN, as well as Sparks' standalone scheduler, run jobs in fundamentally the same way, with some advantages, trade-offs, and intricacies between those options. (Chambers & Zaharia, 2018, p. 281)

Standalone: Spark ships with a built-in scheduler sufficient for smaller jobs but lacks some features third-party cluster management systems offer. (Raju et al., 2019)

Apache Mesos: A once highly popular cluster manager for different big data use cases. According to Raju et al. it features the worst out-of-the-box performance compared to the other scheduling options and is therefore declining in usage (Raju et al., 2019, pp. 1–4).

Hadoop YARN: The most popular option for large-scale Spark cluster environments and still widely used. YARN performs very well for compute-intensive jobs but comes with a heavy installation package, as the complete Hadoop stack must be present on all nodes. (Raju et al., 2019, pp. 1–4)

Kubernetes: The newest addition to the official list of supported Spark cluster managers, which uses Linux container technologies to decouple services and offers good performance for large-sized jobs. (Raju et al., 2019, pp. 1–4)

Kubernetes offers a new way of managing distributed Spark applications by providing a container-based infrastructure that isolates the driver and executer processes in decoupled environments and therefore allows more scalable, reliable, and maintainable deployments. The Kubernetes project started as an internal system at Google, as the company recognised the need for an orchestrator that simplifies and standardizes the deployment of large-scale distributed applications over hundreds of servers. Kubernetes helped Google to design, monitor, and deploy distributed systems for over a decade before it was open-sourced in 2014 and since then transformed into one of the most popular open-source projects globally. Nowadays, Kubernetes represents the de-facto standard API for developing distributed cloud-native applications and is included in every major cloud provider ecosystem. (Apache Software Foundation, 2021f; Burns et al., 2019, p. 1; Lukša, 2018, p. 16)

Kubernetes abstracts away the complexity of a cluster computing environment and makes it seem like one unified computer, enabling programmers to focus on developing the applications instead of the intricacies of the infrastructure. Kubernetes takes care of load-balancing, self-healing, and scaling of the nodes in a cluster and hence acts in a comparable way to an OS but for distributed computing. A fundamental characteristic of Kubernetes is the fact that every application deployed and managed through it is containerized in its own environment and therefore does not affect other processes running on the same server. Services are distributed on the nodes in a cluster to optimize resource utilization and can be shifted from one worker machine to another in case of a hardware failure, making the system fault-tolerant. (Lukša, 2018, pp. 16–18)

3.2.1 Components of Kubernetes

The architecture of Kubernetes follows a master-slave approach and consists of a control plane that manages the communication between the nodes, stores meta-information about the cluster, and distributes resources and applications to the workers. The control plane contains multiple components such as the etcd distributed key-value database, storing all relevant cluster information, the scheduler, which identifies which worker node will host a specific container, and the controller manager, which makes sure that the specified number of nodes and containers is running at all times. All of these services are exposed to the client command tool kubectl and the worker nodes through the Kubernetes API server,

which handles every communication between components and the outside world. (Lukša, 2018, pp. 17–19; Zhu et al., 2020, p. 118)

The worker nodes on the other side run the actual containerized applications in the so-called pods, which are the fundamental deployable unit in a Kubernetes cluster. A pod inherits a unique IP address and can contain one or multiple containers sharing the same virtual network (Zhu et al., 2020, p. 118). In order to run containerized applications, a container runtime is necessary, such as Docker (Docker, 2021). Kubernetes does not prefer nor discriminate against any containerization system as long as it is based on the Linux container technology. The management of the resources, load-balancing, and network traffic is taken care of by the kube-proxy, and finally, the management of the node itself, including the communication with the API server, is handled by the kubelet, which represents the primary node agent (Lukša, 2018, pp. 18–19). In order to facilitate the rearrangement of containers on physical nodes, a Kubernetes pod usually requests and runs on up to one CPU core so that the pods can be swapped on the worker nodes for optimized resource utilization or in case of a fault (Zhu et al., 2020, p. 118). The complete control of deploying and maintaining the containerized services is handed to Kubernetes, which decides on a dynamic basis on which worker machine a pod should be placed. This abstraction layer helps the developers focus on the actual development, while the scheduling is automatically performed by Kubernetes, which is more resource-efficient than a manual orchestration effort (Lukša, 2018, p. 18).

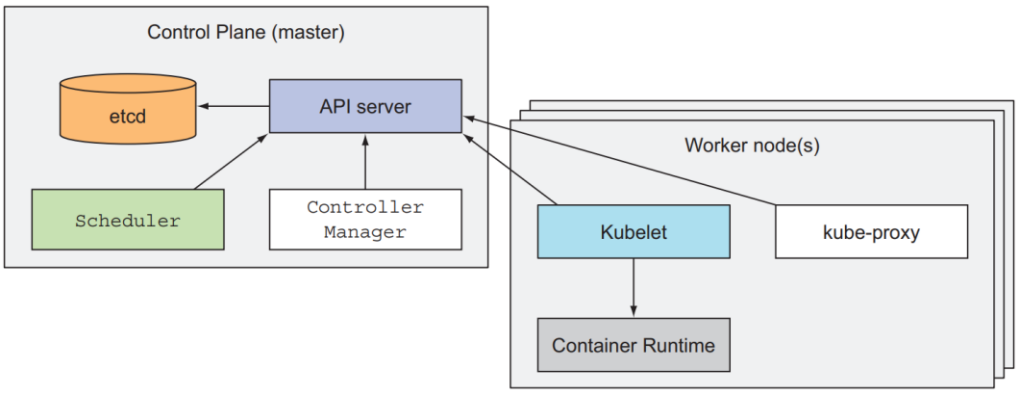


Figure 5: Architecture of Kubernetes (Lukša, 2018, p. 18)

Figure 5 depicts the single Kubernetes components and their relationship with each other in a cluster setting. The graphic also emphasizes the communication process between the

system's individual parts and shows how every interaction is routed and managed by the API server.

3.2.2 Spark on Kubernetes

A new development makes it possible to use Kubernetes as a cluster manager for Spark applications and therefore enables the containerization of the driver and executor processes. When a job is deployed on a Kubernetes cluster through spark-submit, the driver process is instantiated inside a pod and can then request the spin-up of executors within other pods through the Kubernetes API server. As soon as all Spark services are started, the application can be run on the cluster with Kubernetes managing the load-balancing and resource management. In the case of an unforeseen termination of an executor, Kubernetes will spin up a new pod as fast as possible and therefore increase the robustness and fault toleranceof the application. As soon as the job is completed, the executor pods are terminated, and the resources will be released. The driver pod will stay available, making it possible to collect logs through the kubectl command-line tool after the application is completed. It can be removed manually or will be garbage collected after some time. (Zhu et al., 2020, p. 118)

Figure 6 highlights the way a Spark application can be distributed through Kubernetes with the spark-submit script acting as an entry point to the cluster environment.

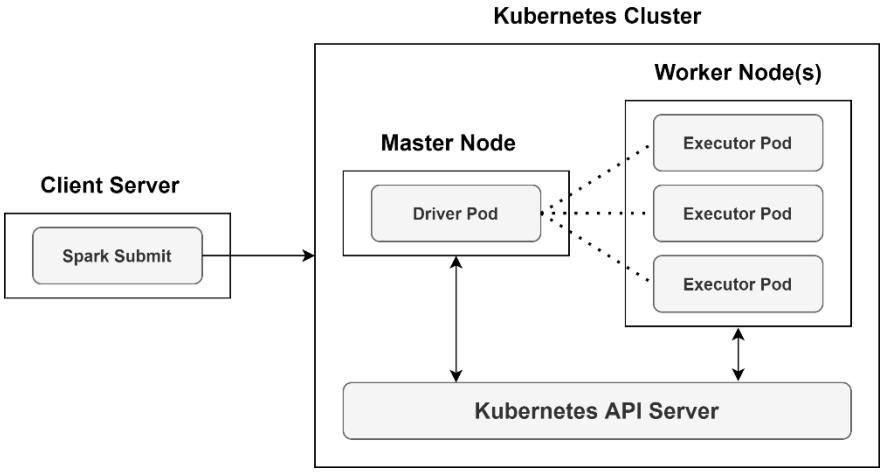


Figure 6: Spark scheduled with Kubernetes – based on (Apache Software Foundation, 2021f)

In addition to the already mentioned aspects, running Spark on Kubernetes can have some further advantages, compared to the more conventional cluster management systems, which are explained in the following section:

Immutability: Until recently, computer systems and software applications have been treated as mutable artefacts, which are instantiated and afterwards continuously updated and modified. Therefore, the system's current state must be regarded as the sum of all modifications over its life cycle. Contrary to this approach, immutable infrastructure is built from scratch every time a new change is introduced, and the old version will be replaced. This methodology has the advantage that the infrastructure declaration always represents the most recent state and therefore acts as a changelog. Old system versions are usually versioned in a repository and can easily be re-instantiated in case of a problem. Immutability is a key concept of images for containerized applications and, therefore, at the core of Kubernetes itself. (Burns et al., 2019, pp. 3–4)

Declarative Configuration: Every deployment in Kubernetes is based on a set of declarative configurations, also known as infrastructure as code. The infrastructure is not only built according to the configuration, but Kubernetes also continuously monitors the state of the system and makes sure that it resembles the definition at all times. This has the advantage that the number of Spark executors is always precisely as it was defined in the submission. (Burns et al., 2019, pp. 4–5)
Another advantage of the declarative configurations for the Spark container images is the fact that only the environment dependencies for this specific task must be included. This makes it possible to create very lightweight container images which run more reliably and performant than bloated imperative systems.

Abstracted Infrastructure: In the past, companies built their own dedicated on-premises computing clusters for Hadoop or Spark applications that were highly specific to the use case at hand and therefore inflexible. The recent development towards public cloud offerings provides scalable, self-service distributed environments, which massively increased business agility. However, a drawback of many cloud APIs is the trend towards offering proprietary infrastructure services designed to simplify the usage of complex systems like Spark and lock consumers into an ecosystem and make a later switch to a different provider increasingly difficult. The usage of container technologies with Kubernetes helps separate developers from specific machines and abstracts the infrastructure to a high level, making it indifferent on which public cloud the cluster is deployed. Transitioning a Kubernetes deployment to a different cloud provider becomes a matter of transferring all of the

declarative configuration artefacts to the new environment. Furthermore, Kubernetes allows the separation from proprietary cloud storage using PersistentVolumeMounts and support for open-source systems such as Apache Cassandra or PostgreSQL. (Apache Software Foundation, 2021a; Burns et al., 2019, pp. 9–10; PostgreSQL, 2021)

3.3 Pipeline Orchestration

An ML pipeline for a complex use case usually consists of many steps and tasks, as described in chapter 2.3.1, which must be executed in a specific order with upstream and downstream dependencies taken into account. This type of scheduled execution is called orchestration and poses specific difficulties addressed in the following section.

Any given component in a pipeline, apart from the initialisation, most likely has multiple requirements and previous tasks that have to be fulfilled before it can be run. Furthermore, big data and advanced analytics workflows tend to increase the complexity of coordinated execution schedules, as special features such as data reproducibility, data provenance, and performant data ingestion are needed (R. Mitchell et al., 2019, p. 4537). Researchers at Google concluded in a paper named ‘Hidden Technical Debt in Machine Learning Systems’ that one of the main reasons for unsuccessful ML projects is the usage of vast amounts of supportive glue code in order to ingest, prepare and route the data through a pipeline. This glue code tends to make the application prone to errors, hard to maintain, and difficult to test, leading to costly long-term problems. According to the report's authors, it is not unusual for an ML system to consist of more than 95% supportive glue code, which is highly problematic. (D. Sculley et al., 2015, p. 5)

Workflow orchestration tools help standardize pipeline executions by abstracting the data and control flow between the system's individual components and making the application more robust, flexible, and maintainable. The responsibility of the workflow management system is to run the individual tasks when certain requirements are met, the monitoring and logging of the runtime and the scheduling of potential reruns, as well as the provisioning of resources (Weber, 2020, pp. 109–110). Most commonly, the definition of the workflow is represented by a directed acyclic graph (DAG) and governed by task-based scheduling algorithms, which run a component as soon as all of the required dependencies have succeeded. A DAG consists of nodes representing the computational tasks and

edges, which visualize the dependencies between the nodes. (R. Mitchell et al., 2019, pp. 4537–4539)

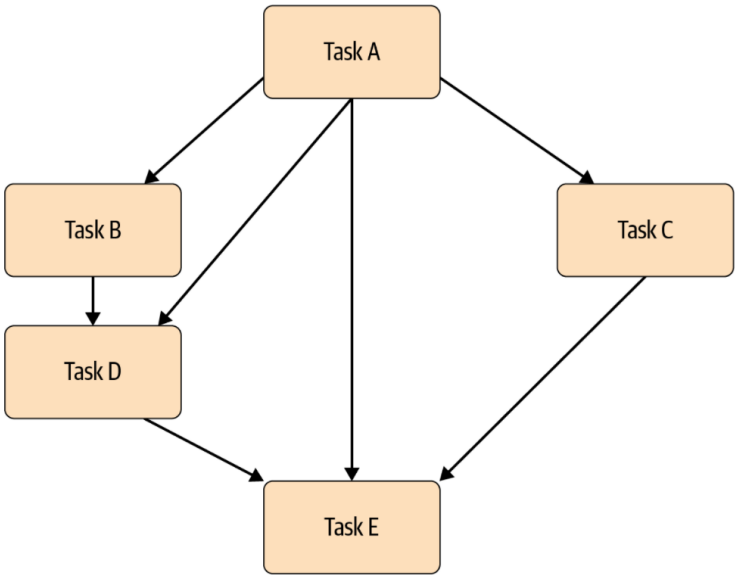


Figure 7: Visual representation of a directed acyclic graph (Hapke & Nelson, 2020)

Figure 7 shows a DAG and visualizes how the tasks are executed in a specific order, which makes the graph directed, and how no task is linked to a previously completed one, making the workflow acyclic. These characteristics are essential, as a cyclic graph could possibly run indefinitely, and an undirected execution would not ensure proper dependency management (Hapke & Nelson, 2020, 1. Chapter). The scientific community optimized this kind of task-centered approach to workflow orchestration for many years and is therefore very efficient in terms of data management and scheduling strategies. Therefore, many open source solutions for workflow management are mature in their development life cycle and ready to be deployed in production. (R. Mitchell et al., 2019, p. 4539)

3.3.1 Apache Airflow for Workflow Management

Many commercial technology companies have recognized the need for flexible, robust, and performant workflow management systems to orchestrate their batch processing pipelines and engineered their own solutions. One of the most famous examples is Apache Airflow developed in-house by AirBnB engineers and later open-sourced in 2016. Since then, Airflow was adopted by countless companies worldwide for data-loading, transformation, and ML tasks, which is why oftentimes, an installation of the system is already present at many workplaces and available through all major cloud providers (Hapke & Nelson, 2020, 11.

Chapter). Airflow is a framework for distributing, scheduling, and monitoring computational tasks that can be either interrelated or independent from each other. Each Airflow task requires a DAG definition file that includes all operations that should be executed in relation to their upstream and downstream dependencies and are written as a declarative configuration in Python. (Singh, 2019, p. 67)

3.3.2 Scheduling Spark Workflows with Airflow

All computational tasks in Airflow are defined by so-called operators inside the DAG declaration file and can be fundamentally different. There are many included operators to choose from, such as the BashOperator for shell scripts, SQL-Operator for relational queries, DockerOperator for launching containerized applications, and many more from third-party contributors (Singh, 2019, p. 73). Airflow also has the ability to schedule Spark jobs via the SparkSubmitOperator, which makes it possible to orchestrate distributed processing and ML pipelines with advanced monitoring and logging capabilities without the need for brittle glue code, increasing both reliability and maintainability. Moreover, Airflow allows defining specific retry strategies for failed pipeline components, making it possible to decouple the individual tasks and only rerun an individual step instead of the whole application. Additionally, the execution of multiple tasks in parallel is supported when using a production-grade database for metadata such as PostgreSQL, which further increases the scalability of Spark workflows (Apache Software Foundation, 2021b). Finally, when submitting a Spark job via Airflow, the same configuration parameters can be applied as with the regular spark-submit script, as explained in chapter 3.1.2, enabling the possibility to run Spark applications orchestrated with Airflow on a Kubernetes cluster. This powerful combination of cloud-native applications provides the means to deploy scalable, reliable, and maintainable ML pipelines in distributed environments.

3.4 Model Serving Architectures

The previous chapters introduced frameworks and systems for building scalable data pipelines, model training, and orchestration. The following section introduces possible serving architectures for the prediction pipeline and analyses which use case should be chosen. As the scope of the thesis is limited to batch ML pipelines in cloud environments, only server-side deployment strategies will be covered.

3.4.1 Model as a Web Endpoint

An often-used possibility to make the trained model available across different applications is to wrap the predictive algorithm in a service and deploy it separately from the other components of the inference pipeline (Sato et al., 2019). This way, the model can be consumed by any service or application in the company, making it available for a much broader audience. A popular way to achieve this objective is to deploy the ML algorithm as a web endpoint, which other applications can then call through a RESTful API. Typically, the input parameters for such an API call would be packaged as a JSON payload, including a feature vector, an image, or any other input type supported by the model. The result of the inference would then be sent back to the requestor with a short delay. (Weber, 2020, p. 33)

This approach allows it to update the model independently from the rest of the inference pipeline and, therefore, increases the architecture's flexibility (Sato et al., 2019). Due to the availability of model invocations in near-real-time, this serving strategy is mainly used for streaming use cases that require an instant response. As the number of parallel inference calls grows, however, the latency for each prediction can increase quickly, as nodes have to be constantly added and removed from the prediction cluster. Moreover, the API invocation always comes with an overhead due to network connectivity, which is negligible for an individual call but sums up significantly over a large number of prediction requests. Therefore, this architecture should be avoided for batch prediction use cases, which do not require an immediate result but rather process large-scale data in frequent intervals. (Ameisen, 2020, pp. 184–185)

3.4.2 Batch Prediction Pipeline

When the input feature data set is available in advance, serving an ML model as a batch inference pipeline is possible before the prediction is needed for other services. The pipeline can be run on a vast amount of data points and, therefore, pre-predict labels, which are stored and used later. This approach is appropriate if the prediction latency is of no concern and the batch process can be run in set intervals or as soon as a specific amount of raw data is available for ingestion. (Ameisen, 2020, pp. 186–187)

Contrary to the previous approach where a model is wrapped as an individual service, the ML algorithm in a batch prediction pipeline is usually embedded in the application and perceived as a dependency packaged into the source code at built time. This has the negative effect, that the pipeline and the model have to be treated as a single application instance

and cannot be updated independently. In order to combat this side effect, which has an adverse influence on the maintainability of the pipeline, the model can alternatively be published as a data artefact that is ingested at runtime from a repository. Using this strategy, the newest version of the model can be used immediately without the need for rebuilding the application and, therefore, decouple the two components from each other. (Sato et al., 2019)

A batch prediction approach needs the same amount of prediction calls as a model served through a web endpoint but saves the networking overhead for each inference. Additionally, as a batch process is executed at a known time, with a defined amount of input data, the provisioning of the cluster environment and the allocation of resources can be planned more time and resource efficiency. Furthermore, since the results have already been precomputed and only need to be retrieved by a downstream service for consumption, a batch method can be faster and act comparable to caching. (Ameisen, 2020, p. 187)

4 Conception and Methodology

The following chapter begins by analysing the architecture of the existing ML pipeline and the resulting shortcomings with respect to the characteristics of data-intensive applications outlined in chapter 2.2, describing the application's workflow and the interaction of its software components. In the next step, the requirements for the new pipeline will be elaborated, which will address the deficiencies of the previous version. The final part introduces the process model used to develop the proof-of-concept ML pipeline and outlines the individual steps that should be taken to implement such an application successfully.

4.1 Architecture and Deficiencies of the Legacy Pipeline

Figure 8 shows a graphical representation of the legacy pipeline application architecture deployed on a dedicated server. The pre-processing of the training data and the modeling of the ML algorithm is accomplished in two Python notebooks with conventional data science tools such as Pandas and Scikit-Learn, after which the resulting model is saved in the joblib (Joblib developers, 2021) file format on the server.

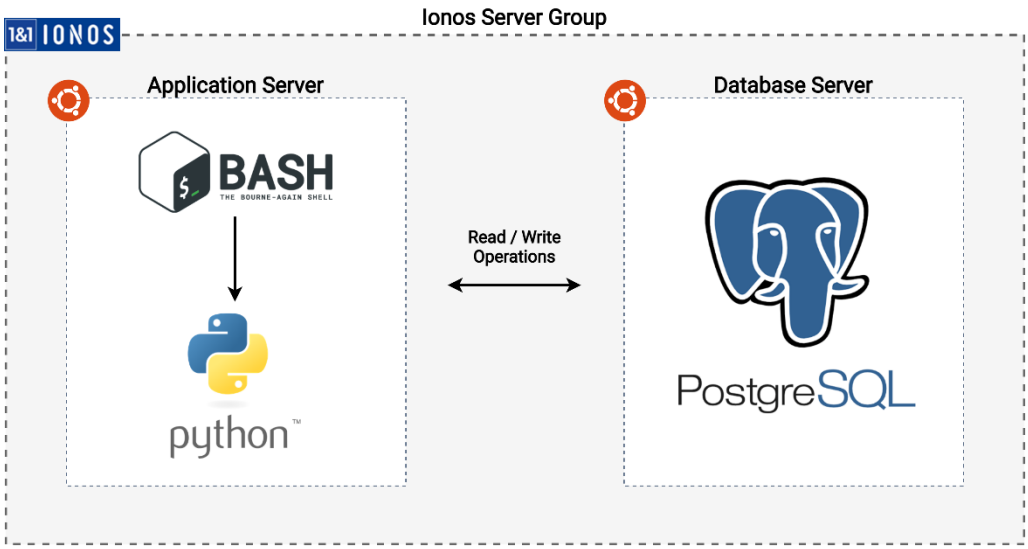


Figure 8: System architecture of the legacy pipeline

The workflow architecture in Figure 9 shows how a bash script is used to schedule the pipeline execution at inference time and iterates through a predefined data range. Then, the unlabelled data sets are ingested via a SQL script, which queries a PostgreSQL database, and pre-processed in a Python script. Finally, the prediction of the indoor/outdoor sample state is performed in the same script, after which the result set is written back into the

database. In case of a failure, the index of the currently handled observation is stored in a temporary table, so the process can be restarted at this exact point.

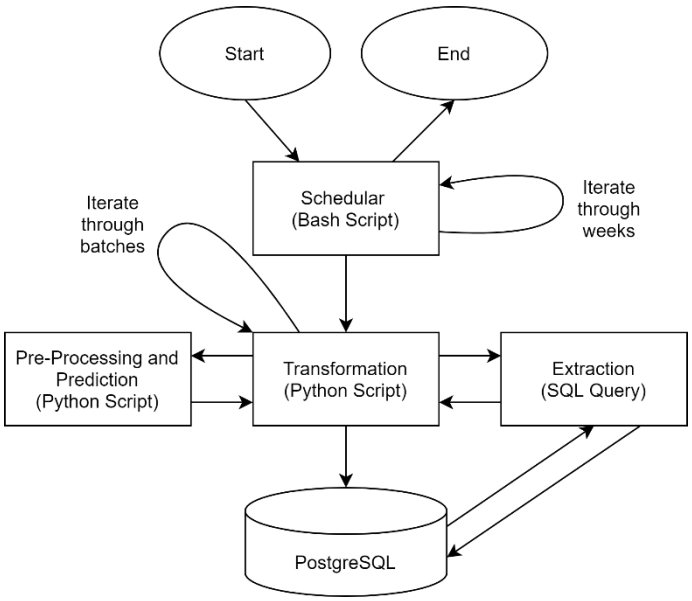


Figure 9: Workflow architecture of the legacy pipeline

As the application was designed to run within the existing infrastructure on a single machine, the pipeline is not natively scalable in a horizontal fashion and therefore struggles to keep up with the ever-increasing amount of data. Unfortunately, a later update to increase the scalability would be complicated, as the employed technologies do not support distribution, which is why a new development is necessary. Another problem is the scheduling strategy of the pipeline, which uses a significant amount of glue code in the form of bash and python scripts to route the data and control flow through the system. As described in chapter 3.3, this anti-pattern can introduce difficulties in terms of the reliability and maintainability of the system. Moreover, the application components are tightly coupled, leading to a crash of the whole pipeline in case of a failure in one part of the system. Finally, a monitoring and maintenance strategy has not been implemented, making the assessment of the system's current status difficult and requiring a manual effort to keep the application running without failure.

4.2 Requirements Engineering

The following requirements for the new implementation of a batch prediction pipeline were synthesized by critically assessing the shortcomings of the old application. Meetings with the Data Science and the Data Engineering department members were held to

understand the constraints and demands of the deployment environment and refine the requirements according to the user needs.

4.2.1 Functional Requirements

Table 2: Description of the functional application requirements

Requirement Name	Description
Data Ingestion	The raw data should be ingested from the company-wide Azure Data Lake Store and combined into one unified table. Before the pre-processing of the data set starts, the new schema must be validated.
Data Preparation	The feature-engineering and pre-processing steps present in the old pipeline must be translated to the new application as closely as possible.
Modeling	There should be the ability to train and compare multiple models in one pipeline run. The training of the models should include hyperparameter tuning and k-fold cross-validation. The final model must be exported as a data artefact to be loaded at inference time.
Deployment	The system must be deployed in the Azure cloud because the whole company's IT infrastructure will also be migrated there. Furthermore, the application should be able to run in a cluster environment comprised of commodity hardware.
Orchestration	The pipeline execution should be orchestrated in a standardized way to facilitate the data and control flow between the individual tasks.
Monitoring	There should be the ability to monitor the runtime of the application in order to detect possible problems.
Configurability	The orchestration, deployment, and execution parameters should be configurable.

4.2.2 Non-Functional Requirements

Table 3: Description of the non-functional application requirements

Requirement Name	Description
Scalability	The applications should be able to scale to multiple nodes in a cluster environment while maintaining a near-linear speedup for every consecutive node. In addition, the resource utilization should be over 95%, both for CPU time and random-access-memory, respectively.

Reliability	The system should be able to handle the failure of a node and recover in a self-healing fashion. Partial reruns of the pipeline should be possible if a part of the pipeline fails. As the pipeline predicts in batches, the availability and latency of the system are of secondary nature.
Maintainability	The pipeline components should be decoupled from each other to maximize expandability and flexibility. As much as possible of the configuration should be maintained as infrastructure-as-code to facilitate reproducibility and migratability.
Model Performance	The accuracy of the prediction and the ROC and AUC metrics should be over 95%.
Software Tools	The tools and frameworks used to build the ML pipeline should preferably be open-sourced, so the application can be migrated into a different cloud environment if needed.

4.3 Methodology for the Development of the new Pipeline

The overarching goal of this thesis is to design the architecture of a scalable, reliable, and maintainable ML pipeline according to the previously identified tools, frameworks, and methods, as well as the development and deployment of a proof-of-concept. Firstly, the application's architecture will be conceptualized using the elucidated state-of-the-art software systems for big data processing and modeling, pipeline orchestration, and cluster management. Both the data and control flow between the individual computational tasks of the training and prediction pipeline will be visualized, as well as the interaction between the software systems on an infrastructure level.

Afterwards, the development and deployment process will be described according to the 'Cross-Industry Standard Process model for the development of Machine Learning applications with Quality assurance methodology (CRISP-ML(Q))' (Stefan Studer et al., 2020, p. 3), which was used for the iterative implementation of the application. CRISP-ML(Q) is a process model for developing ML-powered applications and is derived from the widely used data mining methodology 'CRISP-DM' (Wirth & Hipp, 2000), which was introduced by Wirth and Hipp. CRISP-ML(Q) expands on the previous de-facto standard CRISP-DM by adding an additional step at the end of the process, which covers the monitoring and maintenance of the ML application and therefore prevents the degradation of the predictive power in a

dynamic environment. Additionally, a quality assurance procedure is integrated into every phase of the methodology, which helps identify errors in an early stage and contributes to the reliability of the process. The initialization phase of the CRISP-ML(Q) approach merges the first two phases of the predecessor, as the business objectives and the data understanding typically have to be analysed in parallel because these two activities are strongly correlated to each other. Table 4 compares the individual process phases of CRISP-DM against CRISP-ML(Q), showing how quality assurance is integrated into every step. (Stefan Studer et al., 2020, pp. 1–4)

Table 4: Comparison between the process phases of CRISP-DM and CRISP-ML(Q)

CRISP-DM	**CRISP-ML(Q)**	
1. Business Understanding	1. Business and Data Understanding	Quality Assurance
2. Data Understanding		
3. Data Preparation	2. Data Preparation	
4. Modeling	3. Modeling	
5. Evaluation	4. Evaluation	
6. Deployment	5. Deployment	
-	6. Monitoring and Maintenance	

Following the content and approach of the individual process phases are outlined:

Quality Assurance: Identifying and mitigating potential risks that could affect the success of the ML application is part of every process phase and an essential part of the CRISP-ML(Q) process framework (Stefan Studer et al., 2020, pp. 4–5).

Business and data understanding: The first phase includes activities such as collecting and reviewing data quality, assessing project feasibility and risks, and defining business goals to be translated into software engineering objectives (Stefan Studer et al., 2020, p. 5). This section will also explain how the project initialization and the continuous communication with stakeholders were pursued, how the legacy pipeline was analysed, and how the available data sets were examined.

Data preparation: This phase covers all steps that are necessary in order to obtain a data set that can be used to train an ML model, as well as the preparation steps for the inference pipeline (Stefan Studer et al., 2020, pp. 7–9). This specific project includes data ingestion,

data cleaning, feature engineering, and data pre-processing methods such as one-hot-encoding and data imputation.

Modeling: The goal of the modeling phase is to train an ML algorithm capable of delivering reliable predictions for the domain-specific use case. The modeling approaches used are determined by the previously defined objectives, underlying data, and project constraints (Stefan Studer et al., 2020, pp. 9–12). In this specific case, the modeling phase includes selecting an appropriate algorithm, hyperparameter tuning, and k-fold cross-validation. Several models will be trained in parallel and compete against each other.

Evaluation: The trained model must be evaluated according to the criteria determined in the initial process phase and compared across a wide variety of suitable performance metrics. As multiple models will be trained in the context of this thesis, only the most performant shall be deployed into the prediction pipeline.

Deployment: The deployment phase describes the underlying infrastructure of the environment, the model serving strategy, and the cluster management approach. As this specific project also emphasises pipeline orchestration to a high degree, the workflow management design decisions will be described as part of this section.

Monitoring and Maintenance: ML models tend to have a long life cycle, during which the training and prediction data can drift apart from each other. This problem has to be addressed by maintenance and monitoring efforts, which can trigger an update to the algorithm with fresh training instances (Stefan Studer et al., 2020, pp. 13–14). Furthermore, statistics about the cluster configuration, resource utilization, and metrics from past and current pipeline runs are crucial for improving the application and continuously detecting errors.

Subsequently, the implemented ML pipeline will be evaluated on a Kubernetes cluster by measuring the scalability, resource requirements, and throughput and comparing it to the legacy application. Eventually, the reliability and maintainability of the proposed solution will be assessed.

5 System Architecture

This chapter starts by describing the development environment and its constraints, as this information is crucial for the design process of application architecture. In the second part, a proposal for a system architecture will be presented and described in detail.

5.1 Environment Conditions

As NET CHECK transitioned from a dedicated server infrastructure into a Microsoft Azure cloud environment, the ML pipeline is also implemented in the Azure cloud. Nevertheless, as stated in the project requirements, open-source software should always be favoured over proprietary Azure offerings whenever it is feasible to do so. Therefore, infrastructure components such as the company-wide Data Lake will be used, as they are already implemented.

The development environment is comprised of an Azure VM with 8 GiB of RAM and two vCPUs inside a Standard 'D2as_v4' machine, running an initially clean installation of Ubuntu 18.04. A fresh installation of Ubuntu was chosen to customize the environment to the project's requirements. As NET CHECK already utilizes the Azure Data Lake Store Gen2 (ADLS) as their primary storage system, all project-relevant data was loaded and stored in a separate blob storage container inside the company data lake. For the purpose of creating a flexible Kubernetes compute cluster, the Azure Kubernetes Service (AKS) is utilized, where every node consists of a 'D16as_v4' machine. Docker images necessary for the deployment of Spark on the Kubernetes cluster are pushed to the Azure Container Registry (ACR), from which the images can be pulled during runtime.

Apache Spark 3.1.1 with Hadoop 3.2.2 were installed directly onto the Azure VM, for development purposes and for the fact that the spark binaries have to be present on the machine from which the jobs are submitted. The same applies to Apache Airflow 2.0.1 and Kubernetes 1.20.5, which must be present on the VM, as this machine acts as a control unit for the cluster environment.

5.2 The Architecture of the Pipeline

Chapter 3 covered state-of-the-art software tools and frameworks that are best suited for implementing a batch training and prediction pipeline, according to the outlined project requirements. These systems will be used to implement a proof-of-concept which can later

be used as a blueprint for similar projects. Before the application development can be started, the pipeline's infrastructure architecture and the computational workflow must be designed. This approach ensures that a viable system design is in place before individual components are programmed, reducing the risk of an erroneous implementation. Furthermore, the interactions between all application components, both on a data and control flow level, must be carefully considered to ensure the final system's high scalability, reliability, and maintainability.

5.2.1 Workflow Architecture

The ML application is separated into two distinct workflows, which are executed independently from each other. The training pipeline handles the ingestion of the labelled data set, performs pre-processing steps such as the computation of new features or data imputation, and trains multiple learning algorithms on the curated data. The evaluation of the models is performed in the next step, which determines which classifier is deployed as a data artefact into the inference pipeline according to selected performance metrics.

The prediction pipeline is separate from the training workflow, but since the data has to be processed in a similar fashion, the ingestion and pre-processing modules should have the ability to be re-used in a configurable manner at inference time. This approach limits the amount of repeated code according to the DRY (do not repeat yourself) principle of software development. After the data set has been processed, the newest version of the previously deployed model can be loaded from a repository and used to predict the indoor/outdoor state of the data points. The last step is to load the result set back into the ADLS, where other services can consume it.

Every computational task in the workflow is decoupled from its predecessor and successor and can be executed independently, as intermediate results are stored in the data lake. This design approach increases the system's reliability, as individual pipeline components can be rerun in case of failure. Additionally, the maintainability is increased as well, since every task is represented by a distinct module, which can be updated independently.

Figure 10 illustrates the workflows of both pipelines and shows how the ingestion and the pre-processing task can be used at training and inference time.

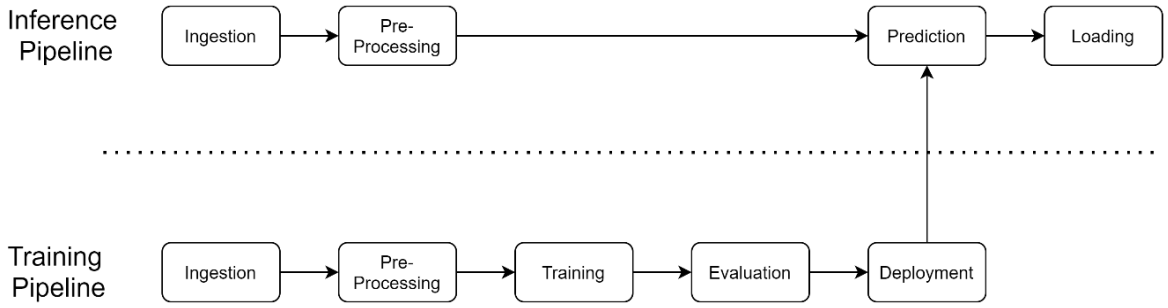


Figure 10: Workflow architecture of the training and prediction pipeline

5.2.2 Application Architecture

The application architecture is designed according to the identified best-practice approaches and software tools described in chapter 3. Therefore, it uses Apache Spark to implement the data ingestion from the ADLS, pre-processing of the raw observations, and modelling of the learning algorithms with MLLib. The pipeline orchestration is managed by Apache Airflow, which runs on an Azure VM separate from the Kubernetes cluster and inherits the DAG description outlined in the previous section 5.2.1. Kubernetes runs inside the AKS and can be accessed through the Airflow VM, as networking is enabled between these two instances. The training and inference data sets are loaded from the ADLS, and all result sets, final and intermediary, are written into the same blob storage container.

A workflow task is triggered through Airflow via the SparkSubmitOperator, which hands over job information such as the Kubernetes master URL, the Spark container image's location, and the application's file-path to the application spark-submit script. Additionally, a YAML (YAML Ain't Markup Language) configuration file is submitted and provides access information for the ADLS, initialization parameters for the data transformations, and directory paths for data loading and writing. The configuration file also states which workflow tasks should be run consecutively and which steps are executed separately. For example, the ingestion and pre-processing of the data set could be performed in one task, while the modeling is run as a separate node in the DAG. This high level of configurability makes the application more flexible and maintainable, as it can be adapted to the specific needs of the use case.

The spark-submit script acts as an entry point to the AKS cluster and sends the job description to the Kubernetes master node, where the API server spins up a pod dedicated to the

Spark driver process. As Spark is not present on the cluster nodes, a pre-build Docker image must be pulled from the ACR. Now that the Spark driver is instantiated, it can request the defined number of executors from the Kubernetes API server and load the job from the specified location, either from a remote URL or inside the Docker container. After the task is completed, the result data set, or the model artefact, can be saved in the ADLS and later ingested by a downstream task or service.

The AKS cluster definition is provided as infrastructure as code and provisioned with Terraform (HashiCorp, 2021). This declarative approach makes it possible to quickly adjust the type of machines and number of nodes in the Kubernetes cluster to the job requirements.

Figure 11 graphically represents the application architecture and highlights the interaction between the individual software components.

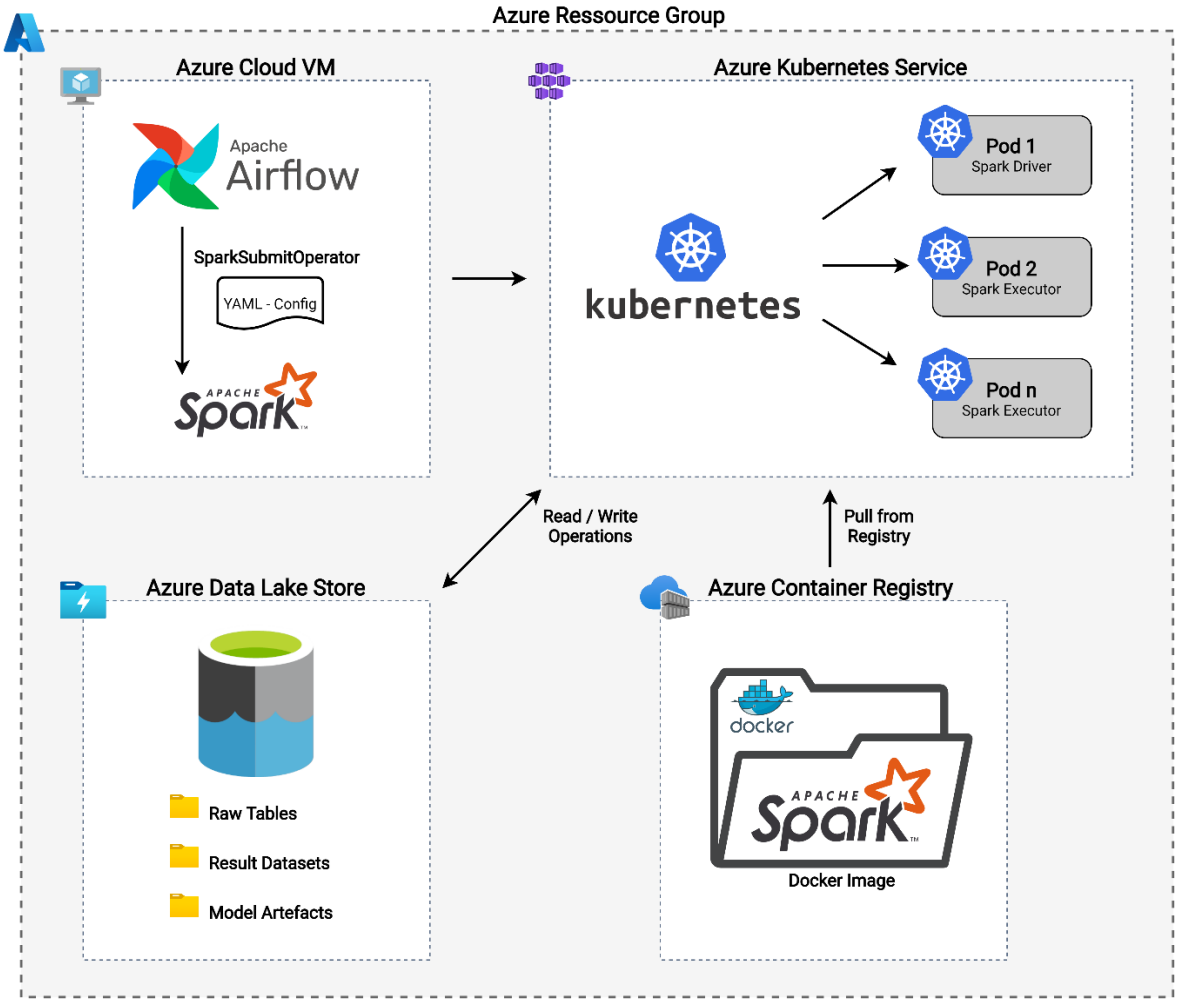


Figure 11: Application architecture of the ML pipeline

5.2.3 Class Design

As stated in chapter 2.2.3, in a maintainable system, associated functionality is grouped in a modular manner to achieve low coupling and high cohesion. A node in the workflow architecture described in section 5.2.1 represents many data manipulations and transformations grouped into one module, forming one execution step in the pipeline. The class design of the application follows this modularity by defining a class for every pipeline step, which can be executed decoupled from the other modules. In order to minimize code repetitions, methods and other functionalities that can be used for both the training and the batch prediction pipeline are elevated into an abstract parent class extended by both child classes.

The main method defines the application's entry point, which loads the YAML configuration file, creates a logger object, and initializes the defined transformation steps of the pipeline. Every transformation class exposes the same `runTransformation()` method, taking a dictionary with input parameters and executes all class transformations. This approach makes it possible to run any transformation class from the main method, without hard coding the exact order or initialization but by using the parameters from the configuration file. All loading or writing processes from or to the ADLS are outsourced into a loader class, which contains specific methods for Parquet tables, model artefacts, and other data types.

6 Development and Deployment

The previous chapters outlined the theoretical basis, described state-of-the-art software tools and methodologies, introduced a process framework, and elucidated the requirements for the architecture and deployment of scalable batch ML pipelines in distributed cloud environments. Chapter 5 presented a possible architecture for such an application, and the following section will combine all of these insights by describing the development of a proof-of-concept. The development steps are modelled according to the process phases outlined by the CRISP-ML(Q) framework.

6.1 Business and Data Understanding

The project was initialized by investigating the potential need for a data pipeline migration from NET CHECK's legacy IT infrastructure towards the newly instantiated Azure cloud architecture. Therefore, clarifying meetings were held with both representatives from the Data Science and ML department as well as the Data Engineering team. As a result, quickly, the need for the migration of an existing ML pipeline was determined, which predicts if a given observation stems from a person located outdoors or in a building. The implemented solution is based on a batch prediction approach and utilizes a Random Forrest classification algorithm to predict the state of a given data point. Additional to the fact that the pipeline had to be migrated, it also featured a list of shortcomings, which are explained in section 4.1. Therefore, the task for this thesis was to build a proof-of-concept ML pipeline that natively runs on the new infrastructure and mitigates the problems of the old application by featuring high scalability, reliability, and maintainability.

After the project scope and requirements were set (listed in chapter 4.2), a thorough examination of the current solution and the available data was started for training and inference purposes. In order to replicate the transformation steps of the old pipeline, an in-depth code analysis had to be conducted, which included a step-by-step execution of every processing task with a test data set.

Table 5: Description of the available data sets

Data Set	Initial Format	Source System	Row Count	Column Count
Training Data	CSV	Legacy Server	544.511	37
Dev. Inference Data	Apache Parquet	ADLS	805.312	N/A
Inference Data	Apache Parquet	ADLS	37.976.624	N/A

The available data for the training pipeline comprises four individual CSV files collected by the Data Science and ML team using different labelling strategies and stored on the same server as the legacy pipeline. Unfortunately, the methods also included manual labelling efforts, which significantly constrain the availability of new training data. In addition, the data sets feature attributes collected from mobile devices and include columns such as the GPS location, cellular signal strength, Android activity mode (e.g., “on bike” or “by foot”), DateTime, and many more. Table 5 describes the available data sets, where the individual training sets were combined into one. The labelled training data is a small, derived subset of the totality of the collected user data stored on the newly established company data lake, which is why a row count for the inference data is not possible as it is many terabytes large. The same holds true for the column count, as the available data spans many tables and dozens of columns. An ER diagram of the available tables can be found under appendix 9.1. Therefore, an inference set was created, containing approximately 37 million rows, which corresponds to half a day's worth of user data. For development purposes, a smaller data set with around 800.000 rows was chosen.

6.2 Data Preparation

After the project scope and requirements were set, the legacy solution and the available data were examined, and the environment setup was established, the development of the ML pipeline could begin. The data preparation phase covers the data ingestion, feature engineering, and pre-processing and therefore acts as the foundation for the downstream components.

6.2.1 Data Ingestion

In order to obtain a data set that can be used to train an ML algorithm that is able to classify the indoor/outdoor state of a given observation, first, the raw tables must be ingested into the pipeline from the source system. The same is true for the unlabelled data at inference

time. As the training data was not yet available in the ADLS and formatted as a CSV file, it had to be moved into the data lake and was transformed into the Apache Parquet file format at the same time. Parquet is a column-oriented data format, which allows efficient and performant compression and loading of large data sets and is primarily used in the Hadoop and Spark ecosystem (Vohra, 2016). As soon as all data sets were available in the data lake, the ingestion into the pipeline was started. Spark makes it possible to connect to the ADLS by modifying the underlying Hadoop configuration in the spark-submit script or inside the application code after the Spark session object was created. In this concrete example, the ADLS account name and key are supplied through the YAML configuration file, which is loaded into the application via the SparkSubmitOperator in Airflow. Inside the main method, the YAML configuration is loaded, and the ADLS account information can be accessed with code under appendix 9.2.

The supplied configuration file holds the ADLS account information needed to connect to the data lake and all file paths, initialization parameters, and saving options for every transformation step. An example configuration for the ingestion of the inference data would be the following:

```
---
adls_account_name: "<account_name>"
adls_account_key: "<account_key>"
processing_tasks:
    - classname: "ExtractorPrediction"
      initialization_parameters:
        container_name: 'data'
        path_raw_tables: '/masterdata/'
        schema_path: '/schema/schema_prediction.pickle'
      transformation_parameters:
        mode: 'save'
        path_df_extract: '/extraction/'
```

After the specified class is initialized with the stated parameters, in this case, the “ExtractorPredictor” class, the transformation steps can be called by executing the `runTransformation()` method.

Table 6: Transformation methods for the data ingestion

	Training Pipeline	Inference Pipeline
Transformation Methods	• `_getTables()` • `_combineTables()` • `_validateSchema()`	• `_getTables()` • `_setJoinConditions()` • `_combineTables()` • `_validateSchema()`

The first transformation that gets called is the `_getTables()` method for both the training and inference pipeline. This method loads the tables from the ADLS into Spark data frames, which enables data manipulation on a high abstraction level. The biggest difference between the labelled and unlabelled data sets is the fact that the training data is already in a unified schema and does not have to be joined but only unionized. On the other hand, the data for the inference pipeline has to be joined and unified from seven different tables, which is why the method `_setJoinCondition()` is used to supply the necessary joining conditions. In both cases, the `_combineTables()` method unifies the individual data frames into a single one. Finally, the now merged data frame schema is assessed by loading a reference schema from the data lake and comparing the columns and data types with the `_validateSchema()` method. This ensures that the data set is compatible with the downstream transformations and ML models. At this point, the inference data inherits the same schema as the training set.

6.2.2 Feature Engineering and Pre-Processing

After the data has been ingested and unified into one validated schema, the injection of domain-specific knowledge in the form of feature engineering and the preparation for the ML model with pre-processing techniques can be started. Spark MLlib inherits a powerful high-level API for the arrangement of multiple transformation steps into a pipeline object. The advantage of the approach is that every execution step within the pipeline is immutable and assigned a unique identifier, which increases traceability and testability. Additionally, the maintainability of the code profits as well, as all pipeline stages are visually organized into a list, making it easy to replace and rearrange the execution steps. (Apache Software Foundation, 2021e)

By design, Spark includes only a handful of Transformer classes that can be used in combination with a pipeline object, but it is possible to extend the parent transformer class and

create custom pipeline tasks. All feature engineering and pre-processing tasks have been encapsulated into standard and custom transformer classes built into a pipeline object with the `_buildStages()` method. An example of a custom transformer class that calculates the distance between two coordinates is found in appendix 9.3.

Table 7: Description of the used transformation classes

	Training Pipeline	**Inference Pipeline**
Transformer classes	• `WeekendExtractor` • `TimeExtractor` • `LevelCalculator` • `CategoricalFilter` • `MedianImputer` • `NumericalImputer` • `StringIndexer` • `OneHotEncoder` • `VectorAssembler`	• `LagCalculator` • `LeadCalculator` • `DistanceCalculator` • `GpsDelayCalculator` • `WeekendExtractor` • `TimeExtractor` • `LevelCalculator` • `CategoricalFilter` • `NumericalImputer` • `StringIndexer` • `OneHotEncoder` • `VectorAssembler`

Table 7 shows the individual transformation classes in order of usage for the feature engineering and pre-processing of the data frames for both pipelines. The creation of lag and lead features is of high importance for this specific use case, as the previous and next state of an observation adds important context information, which is helpful for the indoor/outdoor classification. `DistanceCalculator` and `GpsDelayCalculator` extract additional information from geographic features, whereas `WeekendExtractor` and `TimeExtractor` create extra time columns. Both pipelines include the `CategoricalFilter`, which limits the possible values for a specific column to a predefined set of inputs. This is important, as all string-based columns are later transformed by the `StringIndexer`, which encodes strings with an assigned integer index. Additionally, these encoded strings are then one-hot-encoded. Finally, an ML-specific Spark transformation is performed by the `VectorAssembler`, which combines all feature columns into one vector column, that can be passed into a Spark ML estimator object.

6.3 Modeling

Now that the data frame is fully prepared for the ML algorithm, it can be passed to the training module (or loaded from the ADLS). The legacy solution uses a Random Forest

classification algorithm based on the scikit-learn library, which is why tree-based ensemble methods are also used for this implementation.

A Random Forest algorithm uses an ensemble of decision trees trained by using a bagging method. Instead of relying on one perfectly trained decision tree, a multitude of trees will be built, but only a random subset of features will be considered for every node split. This added randomness results in a greater variety of trees and, in return, trades a lower variance for a higher bias. The tree models are built in parallel, and at the end, a class vote is curated from the combination of all models. (Géron, 2017, 7. Chapter)

In addition to a Random Forest classifier, a Gradient Boosted Tree algorithm will be used as a competing model. Contrary to a Random Forest, the boosting method sequentially trains the tree models, and every successor tree tries to correct the residual errors made by the predecessor. With this method, many weak learners are combined into one strong one. (Géron, 2017, 7. Chapter)

Two modules were implemented, one for a Random Forest algorithm and one for a Gradient Boosted Trees classifier, both realised with the Spark MLlib library. As a result, the two models can be trained in parallel and thus compete against each other, as explained in chapter 2.3.1, enabling the deployment of the best ML algorithm. Both modules contain a method called `train()`, which initializes the model with the feature vector and a label column and calls another method that performs hyperparameter tuning and k-fold cross-validation if stated in the configuration file. The hyperparameter tuning is performed as a grid search, a widely used technique in which a range of possible input parameters is supplied as a parameter grid. The method then trains a model with all possible combinations of hyperparameters in k-fold cross-validation. The model with the best performance metrics, which are defined in advance, will be returned.

Table 8: Grid search values for the hyperparameter tuning

	Random Forest Classifier		
Grid Search Parameters	Parameter	Values	Default Value
	• `maxDepth` • `numTrees` • `minInstancesPerNode` • `bootstrap` • `impurity`	• `[1, 3, 5]` • `[15, 20, 25]` • `[1, 2, 3]` • `[True, False]` • `['gini', 'entropy']`	• `5` • `20` • `1` • `True` • `‘gini’`

	Gradient Boosted Trees Classifier		
Grid Search Parameters	Parameters	Values	Default Value
	• maxDepth • stepSize • minInstancesPerNode	• [1, 3, 5] • [0.05, 0.1, 0.2] • [1, 2, 3]	• 5 • 0.1 • 1

Table 8 shows the chosen values for the hyperparameter tuning. Additional tuning parameters for the Random Forest classifier and the Gradient Boosted Trees classifier can be found under (Apache Software Foundation, 2021d) and (Apache Software Foundation, 2021b), respectively.

6.4 Model Evaluation

After the models have been trained, they must be evaluated in order to determine the algorithm with the best fit for the use case at hand. The evaluation is performed in a separate module which takes one or multiple models as input parameters and calculates several performance metrics, such as accuracy, F1-score, and AUC. One of these metrics can be chosen as a criterion for deploying the best-suited model, which will be loaded into the inference pipeline as a model artefact.

Table 9: Optimal hyperparameter values after a grid search

	Random Forest Classifier		
Grid Search Parameters	Parameter	Default Value	Optimal Value
	• maxDepth • numTrees • minInstancesPerNode • bootstrap • impurity	• 5 • 20 • 1 • True • ‘gini’	• 5 • 20 • 3 • False • ‘gini’
	Gradient Boosted Trees Classifier		
Grid Search Parameters	Parameters	Default Value	Optimal Value
	• maxDepth • stepSize • minInstancesPerNode	• 5 • 0.1 • 1	• 5 • 0.2 • 2

The hyperparameter tuning with the grid search technique yields the optimal values shown in Table 9 after a 10-fold cross-validation process. However, since the optimal value for

several parameters, such as `maxDepth`, is at the upper limit of the specified range, this suggests that the result could be improved by increasing the range for these parameters in a subsequent grid search.

Table 10: Performance metrics for the trained ML models
(TN: true negatives, FP: false positives, FN: false negatives, TP: true positives)

	Random Forest Classifier		**Gradient Boosted Trees Classifier**	
Confusion Matrix	TN=53965	FP=532	TN=54105	FP=392
	FN=979	TP=53729	FN=543	TP=54165
Weighted Precision	0.98619		0.99144	
Weighted Recall	0.98616		0.99143	
F1-Score	0.98616		0.99143	
Accuracy	0.98616		0.99143	

Table 10 contains the calculated performance metrics for both classifiers after the evaluation with the test data set. Both models perform exceptionally well due to the sophisticated feature engineering process and the injected domain-specific knowledge. However, even though both classifiers reach an accuracy beyond 95%, the Gradient Boosted Trees model performs slightly better than the Random Forest. This becomes most apparent in the number of misclassified false negatives, which are 44% higher for the bagging method algorithm.

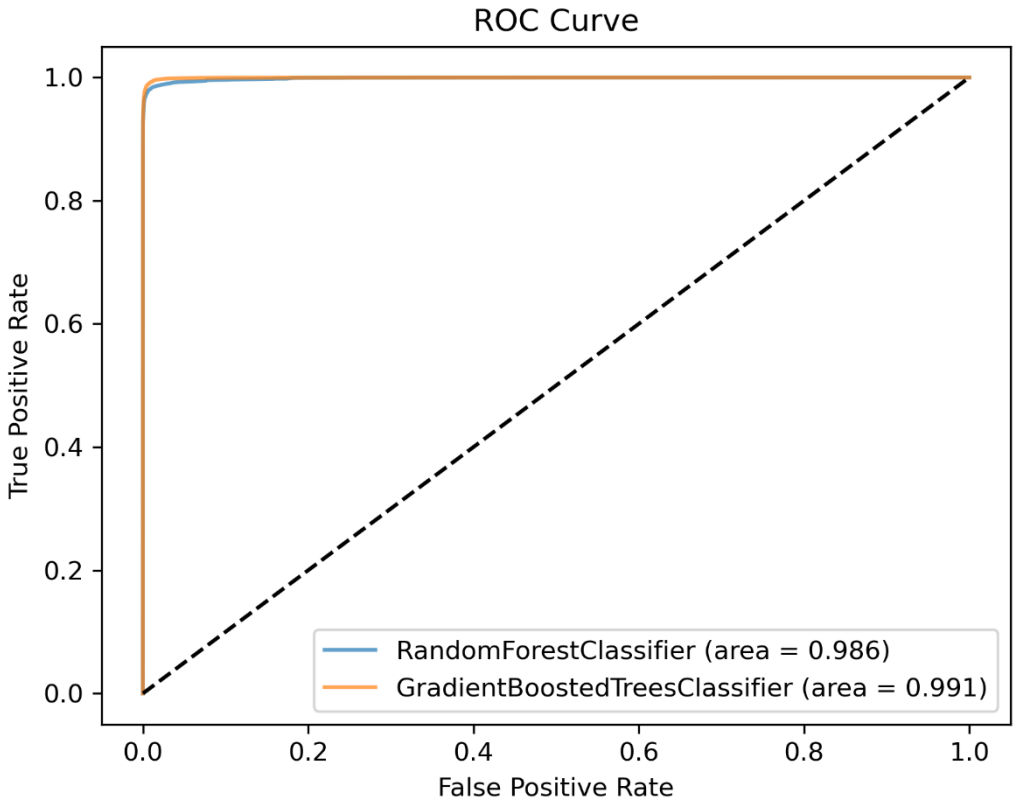


Figure 12: ROC curve for both classifiers

The result is further confirmed by the ROC curve plot in Figure 12, which shows that the Random Forest model performs inferior measured by the number of true positives and therefore reaches a lower AUC. In consequence, the Gradient Boosted Trees classifier is deployed into the inference pipeline.

6.5 Deployment and Orchestration

This section is split into three parts, as the model serving strategy and the actual pipeline deployment have to be examined separately. The third section will explore the implementation of the workflow architecture with Apache Airflow.

6.5.1 Model Deployment

The inference pipeline contains a module that groups all functionality for applying the previously deployed model, in this case, the Gradient Boosted Trees classifier, on the transformed, unlabelled batch data. The most recent model artefact is loaded from the data lake and used to predict the indoor/outdoor label of the curated data set by taking the formerly created feature vector column as a parameter input. The final data frame, including the

predicted label, is then saved back to the ADLS in Parquet format and consumed by other services and applications.

6.5.2 Pipeline Deployment

The pipeline application is composed of a main method, a folder called ‘jobs’, which contains all classes for the individual pipeline tasks, and a folder called ‘config’, which inherits the YAML configuration files. The ‘jobs’ folder is compressed into a zip file and supplied to Spark via the `--py-files` flag, enabling the Python interpreter to import the file as a directory of modules. Finally, the application is pushed onto the ADLS, from which it can be loaded into the Spark Docker container and executed.

In order to run the pipeline, several third-party Python libraries such as NumPy (NumPy, 2021) and additional Spark packages such as ‘hadoop-azure‘ (Apache Software Foundation, 2021c) are necessary. These additional resources can either be supplied through spark-submit or included in the Docker container at build time. Unfortunately, at the time of development, a bug in Spark 3.1.1 prevented submitting packages to Kubernetes, which is why all resources had to be defined in the Dockerfile (GitHub, 2021). The custom Dockerfile for the PySpark container image can be reviewed under appendix 9.4. The modified Dockerfile must be built, tagged, and pushed to the ACR via the Azure CLI (command line interface). Now the Kubernetes cluster can be created according to a declarative configuration file using Terraform, which makes it possible to change the cluster parameters such as the underlying type of machines and number of nodes in an instant. At this point, the whole infrastructure is ready to run the actual pipeline, which can be submitted via Airflow or spark-submit directly. An example spark-submit command to run the prediction task on the AKS cluster could be structured like appendix 9.5.

6.5.3 Workflow Orchestration

The application infrastructure is set up, and the pipeline has been deployed, making it possible to execute it via spark-submit over the command line. In this configuration, the pipeline can only be executed with a main method that defines the interaction between the individual modules and acts as glue code between the pipeline steps. As stated in chapter 3.3, this approach is highly problematic, as it reduces the maintainability of the application and makes it prone to errors. Therefore, Apache Airflow is utilized to orchestrate the pipeline execution via the SparkSubmitOperator, making it possible to manage the workflow

architecture of the pipelines. Each workflow DAG is defined by a declarative Python script, which holds all metadata necessary for executing the individual pipeline components. This includes all parameters which would have been otherwise supplied to spark-submit directly, instructions in case of failures, scheduling sequences, and execution policies. The DAG declaration file for the inference pipeline can be viewed under appendix 9.6. Airflow allows taking precise control over the way the pipeline components interact with each other and how reruns in case of failures are handled, making the application much more maintainable and reliable. For example, if the execution of one DAG - node fails, it can be rerun without the need of repeating the upstream components, as partial results are saved to the data lake.

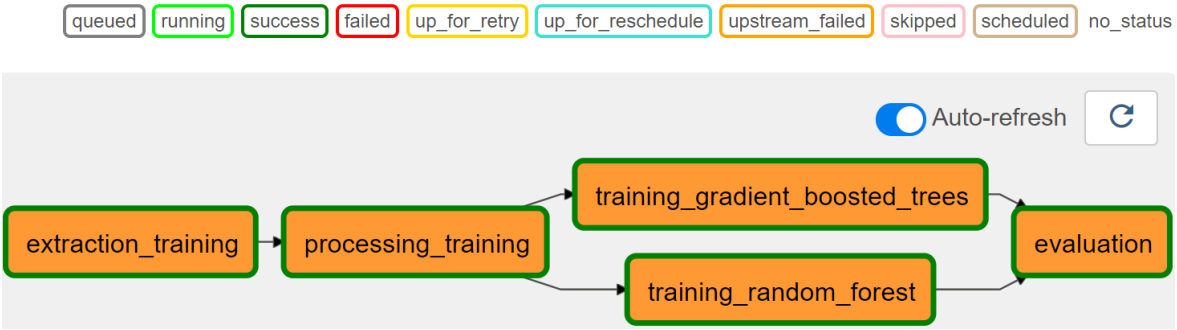


Figure 13: DAG of the training workflow inside the Airflow web UI

Figure 13 shows the DAG of the training pipeline with its individual components. In this configuration, the training of the models can be performed in parallel, which refers to the concept of competing models introduced in chapter 2.3.1. After both models have been trained, they are evaluated in the last computational node, which also deploys the better-performing algorithm.

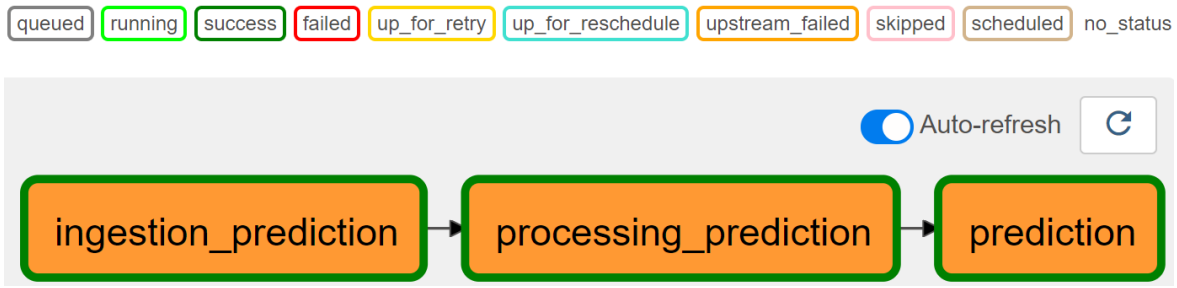


Figure 14: DAG of the inference workflow inside the Airflow web UI

Figure 14 depicts the DAG for the inference pipeline consisting of three computational nodes, where the prediction task also performs the loading of the final labelled data set into the ADLS.

6.6 Monitoring and Maintenance

The system's scalability, reliability, and maintainability cannot be maximised without sufficient monitoring and maintenance capabilities, as these application characteristics require a fast feedback loop to efficiently scale to the load, respond to failures, and add necessary new features. Thus, every architectural component of the application should provide the ability to review execution statistics and performance metrics for historical and active pipeline runs. On the level of the Spark application, a logging system realises the monitoring of the pipeline that records notable events from every executor process and consolidates them into one log file, which can be saved to the data lake for later review. For this purpose, the Apache Log4j API is utilized, which comes included with Spark and is compatible with the ADLS file directory. Additionally, the Spark Web UI (user interface) provides detailed information about the currently running job, ranging from resource utilization over initialization parameters to logical and physical execution plans. It also features an event timeline, as shown in Figure 15, which is very useful in order to get an overview of the job progression and execution times. The Web UI can also be invoked for historical jobs by activating the Spark history-server, which saves all events and statistics to the data lake.

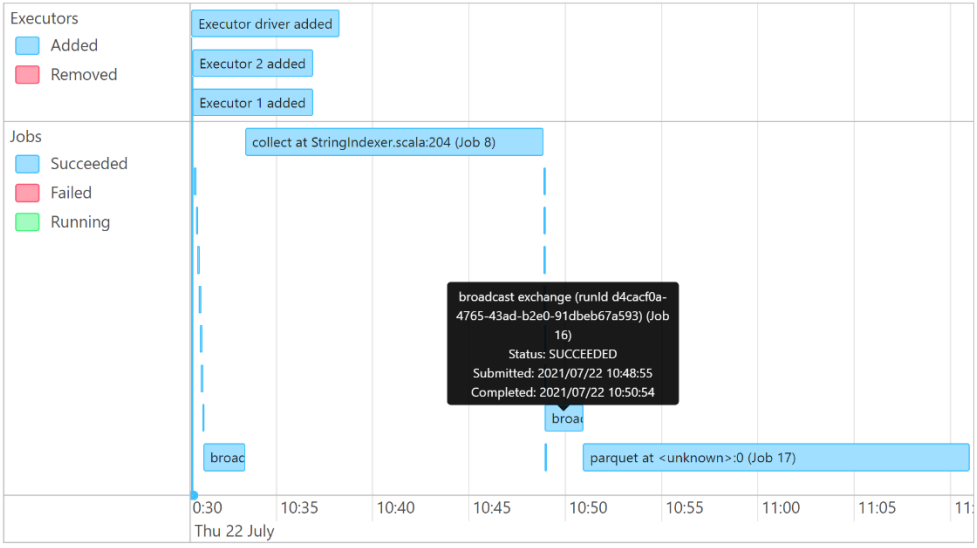


Figure 15: Spark web UI event timeline

Kubernetes allows to closely monitor the cluster environment for resource utilization and components health. During the execution of the pipeline, the status of every pod can be checked, and after completion, the logs for the Spark driver pod remain active until manually deleted. This makes it possible to review potential problems in the cluster settings retrospectively.

All workflow-related metrics, such as execution times, number of reruns, planed schedules, and more, can be accessed via the Airflow webserver, which offers a rich UI with detailed graphs and statistics for every DAG. Airflow also has the ability to send emails in case of failed pipeline tasks or scheduled reruns, which significantly reduces the response time to failures. Finally, the separation into a fully automated training pipeline and a workflow at inference time make it possible to quickly retrain the ML algorithm in specified intervals or in the event of significant data drift. Unfortunately, the collection of new training data turned out to be problematic for this specific use case, which is why this functionality could not be tested sufficiently.

7 Evaluation

The following chapter covers the evaluation of the proof-of-concept ML pipeline presented in the previous section and will address the application's performance in terms of scalability, reliability, and maintainability.

7.1 Scalability

First, the testing setup and the methodology for the scalability evaluation will be elucidated before the results are presented in the last section. The raw data for the scalability tests can be found under appendix 9.7.

7.1.1 Testing setup

All scalability tests are conducted in an AKS cluster environment utilizing general-purpose computing nodes of the ‘D16as_v4’ type, featuring 16 vCPU cores and 64 GB of RAM. This type of machine is chosen because it provides a good balance between raw compute power and fast memory. The test data set for the inference pipeline was increased substantially and now sums up to 37.976.624 rows, which is the equivalent to half a day’s data produced by NET CHECKS crowdsourcing users. As the collection of additional training data is outside of the scope of this work, the training data set remained of the same size as described in chapter 6.1. In order to test the scalability of the application without the overhead created by many read/write operations due to the saving of partial results and scheduling efforts from Airflow, the pipelines are tested as unified, continuous runs.

Additionally, the legacy pipelines will be benchmarked with the same data sets, both for the training and the inference, to establish a baseline metric for comparison. The old pipelines are executed in their production environment, which comprises an application server featuring a 6-core CPU and 32 GB of RAM and a database server running PostgreSQL with ten cores and 24 GB of memory. As the underlying hardware is significantly different from the nodes used for the scalability test of the new ML pipelines, the execution times cannot be compared directly. Nevertheless, the benchmark between the two iterations serves as a good reference point and adds context to the achieved results.

7.1.2 Methodology

First of all, the resource allocation for the cluster environment has to be tuned to prevent idle CPU cores or unused memory. According to Gupta et al., the best processing

performance for Spark jobs can be achieved by delegating five CPU cores to each executor and the equal amount to the driver process (Gupta et al., 2018). In order to verify this claim, each test is run twice, once with three and then with five cores per executor. As one CPU core per cluster node has to be reserved for Kubernetes services, 15 cores can be allocated to the driver and executors. This means that three executors with five cores and five executors with three cores can be deployed per cluster node, except for the master node, which can run one less executor due to the Spark driver process. In the configuration with three cores per executor, each Spark process is given 8GB of RAM, and in the five-core setup, 13GB of memory are assigned. The left-over memory cannot be distributed, as it is needed for system processes. Both pipelines are tested, starting with one cluster node and ending with a total of five computational nodes. In the end, the execution time for the pipeline version scheduled with Airflow is compared to the basic Spark version.

7.1.3 Results

In the first step, the resource allocation is checked with the Kubernetes command `kubectl describe node`, revealing an extensive list of statistics describing running processes and used resources per cluster node. Table 11 shows that the resource utilization per cluster node was over 95% in every case, but the three-core setup manages to slightly outperform the 5-core configuration. Unfortunately, there seems to be no option to allocate resources more fine-grained, e.g., memory in MB, and an increase in RAM for the five-core setup resulted in an overcommitment and eventual failure.

Table 11: Ressource allocation for each cluster node

Cores per Executor	Memory per Executor	Resource Usage: Master Node	Resource Usage: Worker Nodes
3	8 GB	CPU: 98% Memory: 99%	CPU: 96% Memory: 98%
5	13 GB	CPU: 98% Memory: 96%	CPU: 96% Memory: 96%

The second test revolves around the scalability of the inference and training pipelines with an increasing number of cluster nodes and is conducted by saving all Spark events to the ADLS via the history-server. Thus, after all tests are completed, the data for every run can

be analysed retrospectively. Figure 16 shows the execution time for the inference pipeline with increasing node count, both for the 3-core and 5-core executor configuration. As expected, the execution time drops significantly as more and more nodes are added to the cluster. The sharp decrease in runtime from one to two nodes is due to the fact that the master node, which is the only node in the first bar column, also hosts the Spark driver process and therefore runs one less executor.

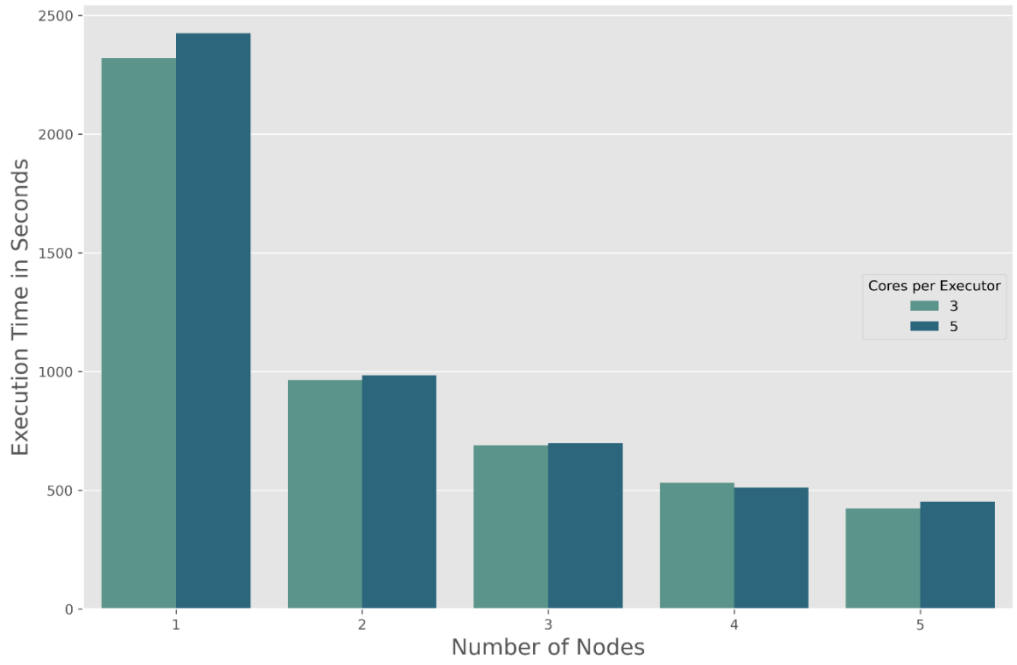


Figure 16: Execution time for the inference pipeline with increasing node count and alternating cores per executor

Contrary to what is stated in the literature, the three-core setup performs slightly better in this scenario, offering a faster execution time in four out of five tests. This is most likely because the three-core configuration utilizes the available resources marginally better. The highest throughput is achieved with five nodes and three cores per executor with around 89.000 records/s. As the tree-core configuration offers better results in most cases, this setup was used for the remaining tests.

Figure 17 shows the execution time for the training pipeline without hyperparameter tuning in relation to an increase in the number of nodes. Contrary to the inference pipeline, the training pipeline does not seem to show any decrease in processing time and, therefore, no horizontal scalability. The most plausible explanation for this behaviour is the insufficient training data set size, which cancels out any performance gain realised through distribution with an increase in overhead. The same result is achieved when executing the

training pipeline, including the hyperparameter tuning and k-fold cross validation – the execution time does not significantly change with an increase in computing resources.

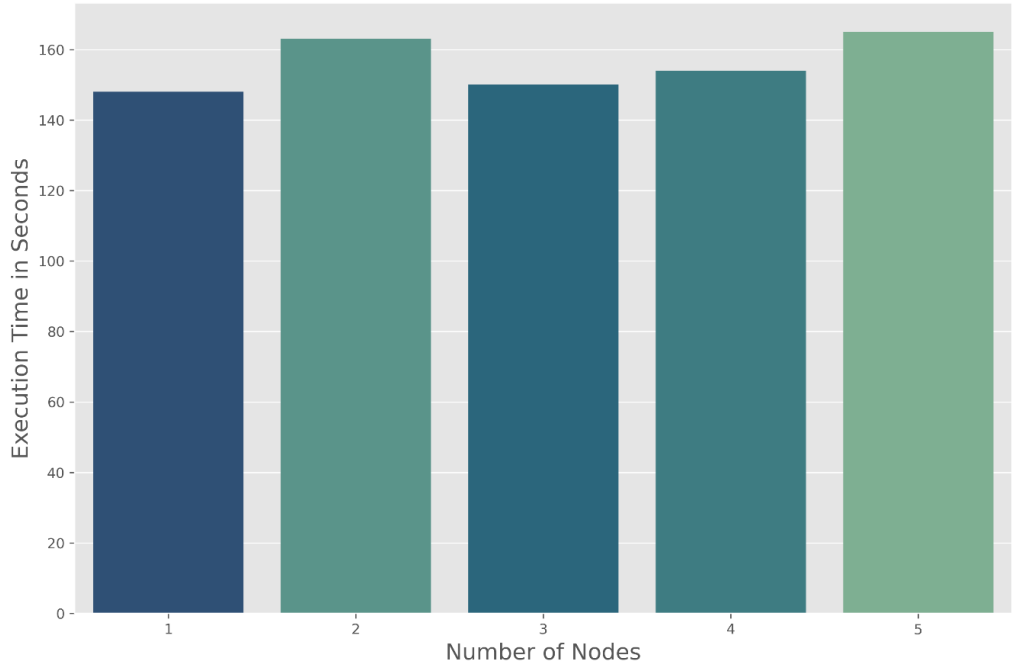


Figure 17: Execution time for the training pipeline with increasing node count

To test the hypothesis whether the lack of scalability is due to the small training data set, the number of rows in the training set is artificially increased by self-union joins to a total of 38.148.040 observations, which is very close to the size of the inference set. In this scenario, the training pipeline significantly reduces the execution time with an increase in the cluster size, which is visually represented by Figure 18. The increased data set for the training pipeline is used for all subsequential tests.

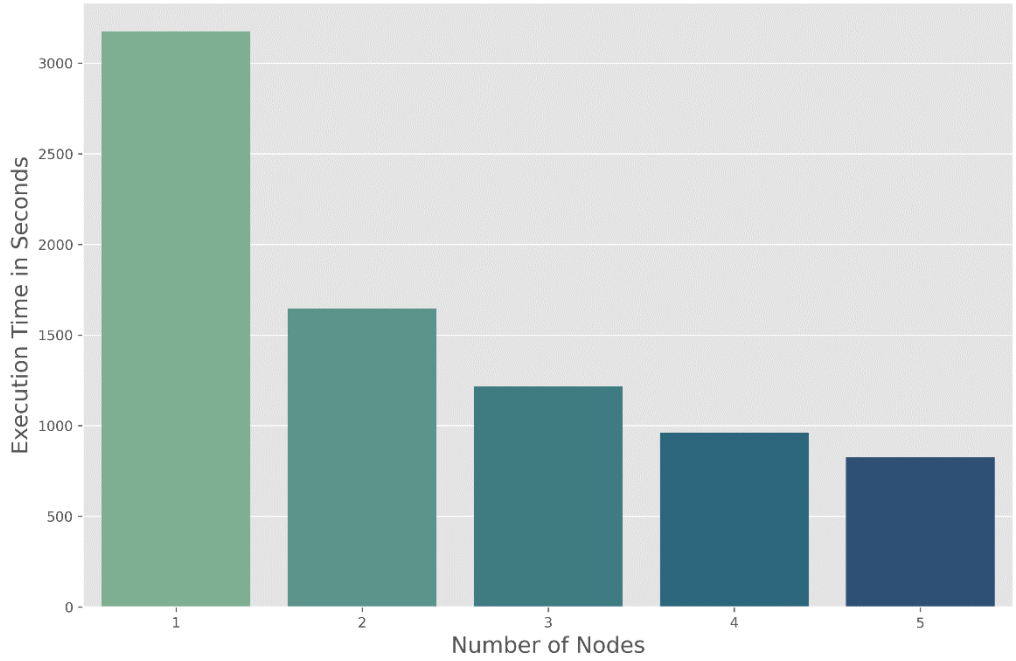


Figure 18: Execution time for the training pipeline with increasing node count after the data increase

Figure 19 shows the measured speedup for the inference and training pipeline with an increasing count in executors compared to an optimal, linear speedup. The expected sublinear scalability of the pipelines is due to the increased overhead with a rising number of executors and parts of the application that do not benefit from parallelisation. As neither of the speedup curves has yet reached its vertex, it can be expected that the speedup would increase if more nodes were added to the cluster.

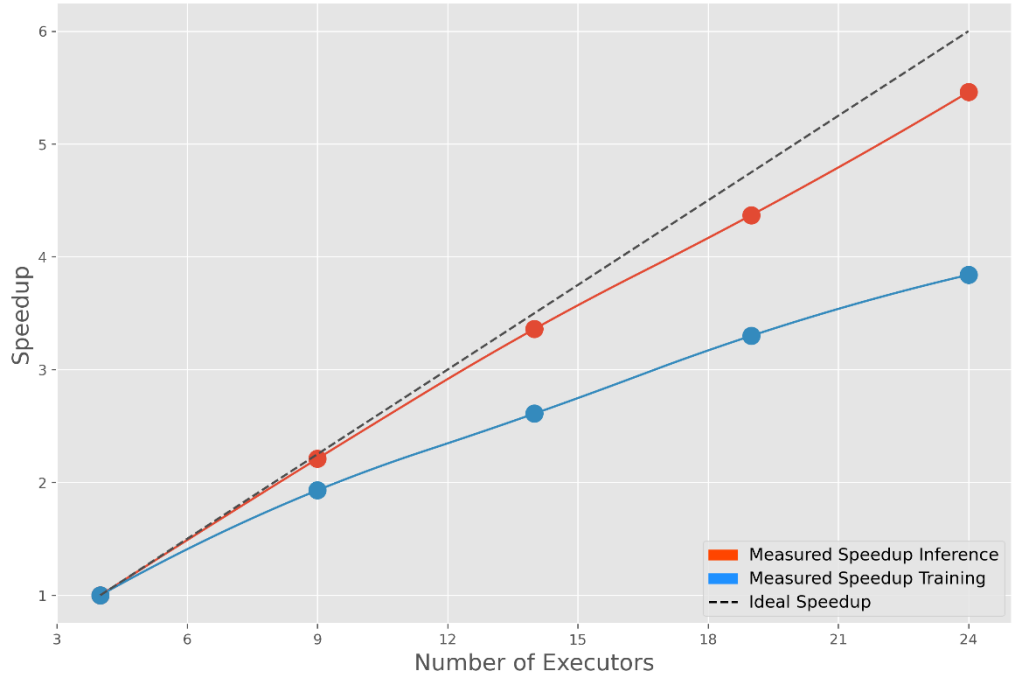
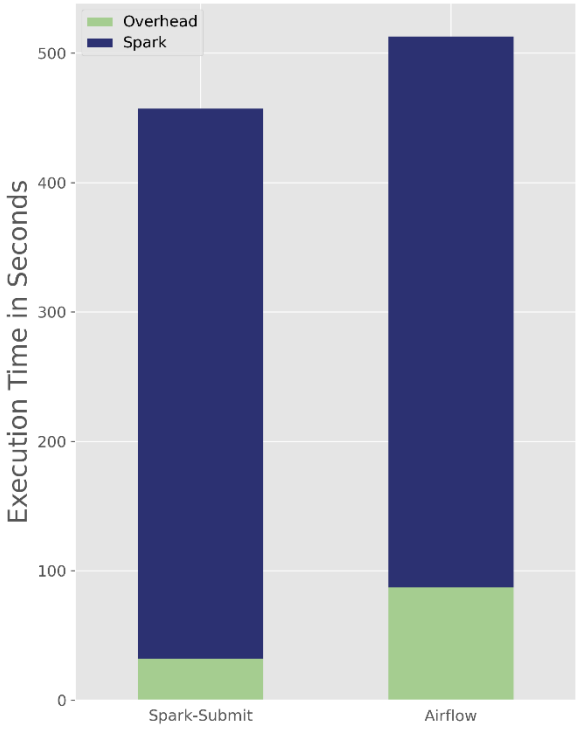


Figure 19: Speedup for the inference pipeline with increasing executor count

In the five-node cluster configuration, the inference pipeline was also executed with Airflow as the workflow manager to test the scheduling overhead's significance. For this scenario, the processing time for the actual Spark application was measured by stopping the spark-submit command's runtime in the bash shell, and the Airflow execution time in the Web UI. Figure 20 compares these two pipeline executions in a stacked bar chart and reveals the difference in overhead for the Airflow scheduling. The overhead in this chart not only includes the time for Airflow and the Spark-submit command but also for Kubernetes.



Whereas the overhead is increased by more than 170% for the Airflow scheduling, the runtime of the actual Spark application is identical. The additional overhead for the

orchestration with Airflow could be drastically reduced by using a production-ready backend for Airflow, such as the Celery Executor with Redis (Airflow, 2021).

Figure 21 shows which part of the training and inference pipeline makes up which proportion of the execution time. Due to the elaborated joins in the data ingestion phase, this step in the inference pipeline makes up most of the processing time. The most significant share of the execution time for the training pipeline is spent on the model training, which is expected behaviour. The evaluation of the ML model takes up more time than expected and is potentially an area of inefficiency, which should be corrected in future developments.

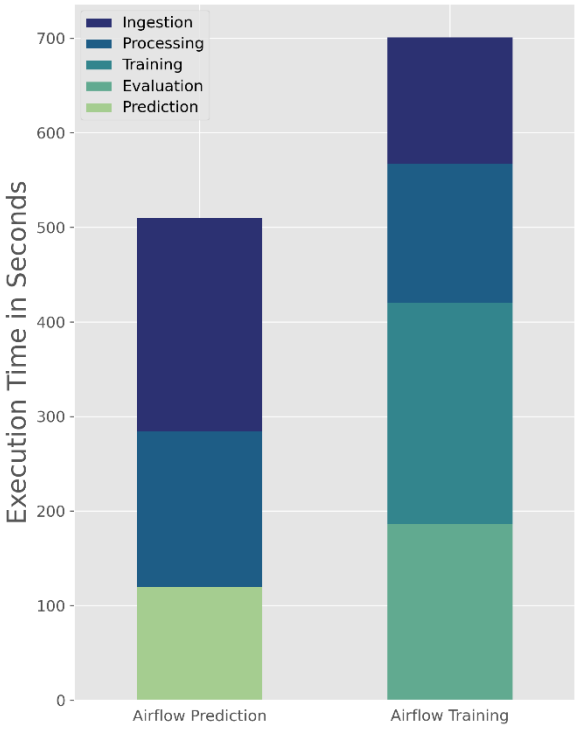


Table 12: Execution times of the old application vs. the new pipelines

Pipeline	Execution Time in Seconds
Training Old	165
Training New (single node)	148
Inference Old	8036
Inference New (single node)	2320

Table 12 shows the measured execution time of the new pipelines compared to the legacy application on the same data sets. The new pipelines are tested with one cluster node and three cores per Spark executor. Even though the training pipeline seems to have a slight performance advantage in this comparison, the small size of the training set does not justify a redevelopment in this case. As discovered in previous tests, the training pipeline only starts to scale as expected with considerably bigger data sets. On the other hand, the comparison shows a significant advantage for the new development for the inference pipelines, tested with significantly larger data sets, as the execution time is nearly 250% faster.

The scalability of the newly developed application meets the defined project requirements, as both pipelines are able to scale out to multiple nodes in a cluster environment comprised of commodity hardware nodes. Additionally, the resource efficiency is on a very high level, as >95% of the available CPU and memory are utilized. Furthermore, the overhead generated by the orchestration with Airflow is minuscule compared to the added benefits of a workflow management system and can be further improved using a production-ready Airflow backend. Finally, the benchmark with the old pipelines shows that the inference pipeline has a clear advantage compared to the old development in execution performance and confirms the hypotheses that the training pipeline can only reach its full capabilities with a significantly larger training set.

7.2 Reliability

The reliability of a data pipeline can be evaluated by the integrity of the produced output and by the system's fault tolerance against failures. The quality of the trained ML model was already analysed in chapter 6.4, which is why this section focuses on the resilience of the application against unforeseen events.

7.2.1 Methodology

In order to simulate the unexpected failure of a Spark component, the pipeline will first be submitted to Kubernetes with a specified number of executor processes. After all pods are created and the application runs, a random executor will be terminated with the command: `kubectl delete pod <pod_name>`. In the second step, the test will be repeated, but this time, all executor pods will be killed at the same time. In the final test, the driver pod will be terminated. During every test, the status of the application will be monitored with: `kubectl get pods --watch`.

7.2.2 Results

In the first test, a random executor is terminated, in this case, `ml-pipeline-exec-1`, and the system's reaction is observed. Figure 22 shows the logs for Kubernetes after the application was submitted. All executors and the driver process are running correctly before the executor is terminated. Immediately after `ml-pipeline-exec-1` is killed, Kubernetes responds to the failure and creates `ml-pipeline-exec-5` to keep the system in the state described by the declarative configuration. The only effect this unforeseen event has on

the pipeline is that the execution time is marginally prolonged, as the state of the executor has to be recreated.

NAME	READY	STATUS	RESTARTS	AGE
ml-pipeline-driver	0/1	Pending	0	0s
ml-pipeline-driver	0/1	ContainerCreating	0	0s
ml-pipeline-driver	1/1	Running	0	1s
ml-pipeline-exec-1	0/1	Pending	0	0s
ml-pipeline-exec-1	0/1	ContainerCreating	0	0s
ml-pipeline-exec-2	0/1	Pending	0	0s
ml-pipeline-exec-2	0/1	ContainerCreating	0	0s
ml-pipeline-exec-3	0/1	Pending	0	0s
ml-pipeline-exec-3	0/1	ContainerCreating	0	0s
ml-pipeline-exec-4	0/1	Pending	0	0s
ml-pipeline-exec-4	0/1	ContainerCreating	0	0s
ml-pipeline-exec-3	1/1	Running	0	1s
ml-pipeline-exec-1	1/1	Running	0	2s
ml-pipeline-exec-4	1/1	Running	0	2s
ml-pipeline-exec-2	1/1	Running	0	2s
ml-pipeline-exec-1	1/1	Terminating	0	42s
ml-pipeline-exec-5	0/1	Pending	0	0s
ml-pipeline-exec-5	0/1	ContainerCreating	0	0s
ml-pipeline-exec-5	1/1	Running	0	2s

Executor pod is terminated

New executor pod is initialized

Figure 22: Kubernetes log for the reliability test of the ML pipeline

The same behaviour can be observed when all executors are killed at once. Kubernetes spins up the same number of pods as were terminated, and the application continues to run without additional problems. The driver process is killed in the last test to see if the system can recover from this worst-case scenario. Unfortunately, the application crashes instantly, and the previous state cannot be recovered. This is because the driver process is the primary agent communicating with the Kubernetes API server, and the system cannot run without it.

Even though the conducted tests only represent a small subset of the possible failures that can occur in a data pipeline, they show that the self-healing properties of Kubernetes are a powerful capability in order to make a system more resilient. In summary, the system's reliability meets the project's requirements in terms of both predictive power and fault tolerance, but further testing is needed.

7.3 Maintainability

The maintainability of a system is difficult to assess quantitatively, so the following section focuses on the qualitative aspects of this characteristic. The decoupled class design, the division into logical workflow phases, and the usage of build-in API abstractions, such as the Spark ML pipeline object, increase the flexibility, expandability, and ultimately

maintainability of the system significantly compared to a monolithic architecture. New features can be added without the need for a whole system refactoring due to the modularity of the application. Testing is simplified, as every transformation step is contained in its own class and can be evaluated separately. Furthermore, all system components use declarative configurations, from the creation of the infrastructure with Terraform to the configuration of the pipeline with YAML files, which makes it easier to adapt the system to specific needs and port it to a different environment. In addition, the extensive monitoring functions described in chapter 6.6 enable the tracking of the execution and possible problems both during runtime and in retrospect, which increases the maintainability of the system considerably.

8 Conclusion

The present thesis aimed to identify adequate tools, frameworks, and methods for developing and deploying scalable batch ML pipelines in distributed cloud environments. The gained insights should then be applied in a system and workflow architecture that allows implementing a scalable, reliable, and maintainable application.

The qualitative research of state-of-the-art ML pipeline techniques revealed that Apache Spark, in combination with its MLLib library, provides the ideal platform for developing distributed data transformation pipelines and training performant ML models. Furthermore, the use of Kubernetes as a cluster management system proved to have significant advantages over other alternatives, as it provides a decoupled and abstracted architecture by utilizing Linux containerization technologies. Additionally, Kubernetes supports declarative configurations with immutable objects and self-healing properties, increasing the system's reliability against unforeseen events. Finally, Apache Airflow has been found to be the optimal workflow management system for a batch ML pipeline in this scenario, as it features rich scheduling and monitoring capabilities and is compatible with Spark and Kubernetes.

The identified tools and best-practice approaches were then combined in a workflow and system architecture based on the Azure cloud platform that supports horizontal scalability, reliability against failures, and maintainability for monitoring and extensibility purposes. The implementation of the actual prototype was conducted according to the CRISP-ML(Q) process model, which includes a description of the data ingestion, preparation, model training, evaluation, deployment, and monitoring phases of the development. The evaluation of the trained models revealed that the Gradient Boosted Trees algorithm has a slight advantage over the Random Forest classifier, which is why the former was deployed into the inference pipeline as a data artefact. For the application deployment on the Kubernetes cluster, custom Docker images were created, which are pulled from the ACR as soon as a pipeline component is submitted to Spark. As a result, all pipeline phases can be run individually in a decoupled manner, allowing for a flexible orchestration with Airflow and enabling partial reruns in case of failure. The methodology of declarative configurations was adopted throughout the project, enabling the management of the application and its infrastructure using infrastructure-as-code.

Finally, the newly developed proof-of-concept ML pipeline was evaluated according to the characteristics of data-intensive applications, namely scalability, reliability, and maintainability. The scalability tests included benchmarks with different cluster configurations that showed an optimal execution performance with three CPU-cores per Spark executor and a good speedup curve for the inference workflow with up to five hardware nodes. As the size of the training data set was insufficient to reap the benefits of computational distribution, it was artificially increased to the same amount of data points as the inference set. After this modification, the training pipeline showed similar scalability properties as the prediction workflow. The system's reliability was tested by randomly terminating Spark executors, starting with one and ending by killing all executor processes at once. Through the self-healing abilities of Kubernetes, the system was able to recover from these incidents as long as the Spark driver process was left untouched. This shows that the system is well equipped for unforeseen events and features high reliability. The same holds true for the maintainability of the developed pipelines, which were assessed qualitatively. Maintenance and monitoring were an integral part of the development process from the beginning. Therefore, they were included in all aspects of the system, from the decoupled class design to the logging capabilities included in the Spark application and the extensive monitoring features of Airflow.

Due to the fact that the standard backend of Airflow does not allow the scheduling of parallel tasks, the ability of the training pipeline to train multiple models at the same time could not be tested sufficiently. Both training tasks are scheduled sequentially in the setup used for this work, but the models are evaluated against each other in the downstream node simultaneously. Therefore, no performance gain is realised compared to a basic, sequential workflow architecture. In future developments, this shortcoming should be addressed by deploying Airflow with a production-ready backend database that supports the celery executor, such as Redis. Another limitation of this thesis is caused by the insufficient amount of available training data and the complicated data collection and labelling process. As a result of this constraint, the automatic retraining of the ML model with new training data could not be implemented, and the scalability of the training pipeline had to be evaluated with an artificially enlarged data set. Implementing automated retraining of the model in case of data shift should have a high priority for further developments based on

the presented proof-of-concept. Even though the architecture and the proof-of-concept conceptualized in this thesis apply to a broad range of batch prediction use cases, the evaluation results must be validated in upcoming projects, as only one example could be implemented. Future research in this field should investigate the consistency of the realized results with other examples, such as the ones mentioned in chapter 1.1.

The hypothesis put forward at the beginning of this work, stating that an ML pipeline built using the identified tools, frameworks, and methods according to the proposed architecture is scalable, maintainable, and reliable, and outperforms a centralized approach, could be verified. The developed proof-of-concept fulfilled all project requirements and exceeded the legacy application in all three characteristics of data-intensive applications. Therefore, this thesis sets the foundation for upcoming developments of data and ML pipelines at NET CHECK and helps pave the way towards a future-proof and data-centric business model that utilizes big data as a competitive advantage.

References

Airflow. (2021). *Celery executor.* Apache Software Foundation. https://airflow.apache.org/docs/apache-airflow/stable/executor/celery.html. Retrieved 27.08.2021.

Al-Aqrabi, H., Liu, L., Hill, R., & Antonopoulos, N. (2015). Cloud bi: Future of business intelligence in the cloud. *Journal of Computer and System Sciences*, *81*(1), 85–96. https://doi.org/10.1016/j.jcss.2014.06.013

Ameisen, E. (2020). *Building machine learning powered applications: Going from idea to product* (First edition). O'Reilly Media, Inc.

Amir Gandomi, & Murtaza Haider (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, *35*(2), 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007

Apache Software Foundation. (2021a). *Cassandra* [Computer software]. Apache Software Foundation. https://cassandra.apache.org/_/index.html.

Apache Software Foundation. (2021b). *Executor*. https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html. Retrieved 17.06.2021.

Apache Software Foundation. (2021c). *hadoop-azure* [Computer software]. Maven. https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-azure.

Apache Software Foundation. (2021d). *Kafka* [Computer software]. Apache Software Foundation. https://kafka.apache.org/.

Apache Software Foundation. (2021e). *Ml pipelines*. https://spark.apache.org/docs/latest/ml-pipeline.html. Retrieved 13.07.2021.

Apache Software Foundation. (2021f). *Running spark on kubernetes*. https://spark.apache.org/docs/latest/running-on-kubernetes.html. Retrieved 28.05.2021.

Apache Software Foundation. (2021g). *Spark* [Computer software]. Apache Software Foundation. https://spark.apache.org/.

Apache Software Foundation. (2021h). *Submitting applications*. https://spark.apache.org/docs/latest/submitting-applications.html. Retrieved 09.06.2021.

Assefi, M., Behravesh, E., Liu, G., & Tafti, A. P. (2017). Big data machine learning using apache spark mllib. In J.-Y. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, & C. Wang (Eds.), *2017 ieee international conference on big data: dec 11-14, 2017, boston, ma, USA : Proceedings* (pp. 3492–3498). IEEE. https://doi.org/10.1109/BigData.2017.8258338

Balachandran, B. M., & Prasad, S. (2017). Challenges and benefits of deploying big data analytics in the cloud for business intelligence. *Procedia Computer Science*, *112*, 1112–1122. https://doi.org/10.1016/j.procs.2017.08.138

Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: Up and Running, 2nd Edition* (2nd edition). O'Reilly Media, Inc.

Chambers, B., & Zaharia, M. (2018). *Spark: The definitive guide: Big data processing made simple*. O'Reilly.

D. Sculley, Gary Holt, D. Golovin, Eugene Davydov, Todd Phillips, D. Ebner, Vinay Chaudhary, M. Young, J. Crespo, & Dan Dennison (2015). Hidden technical debt in machine learning systems. In *Nips*.

Docker. (2021). *Docker* [Computer software]. Docker. https://www.docker.com/.

Dugre, M., Hayot-Sasson, V., & Glatard, T. (112019). A performance comparison of dask and apache spark for data-intensive neuroimaging pipelines. In *2019 ieee/acm workflows in support of large-scale science (works)* (pp. 40–49). IEEE. https://doi.org/10.1109/WORKS49585.2019.00010

Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: Concepts, tools, and techniques to build intelligent systems* (First edition). O'Reilly Media. http://proquest.tech.safaribooksonline.de/9781491962282.

GitHub. (2021) [*spark-35084][core]: spark 3: Supporting "—packages" in k8s cluster mode*. https://github.com/apache/spark/pull/32397. Retrieved 27.08.2021.

Gunawardena, T., & Jayasena, K. (2020). Real-time uber data analysis of popular uber locations in kubernetes environment. In *2020 5th international conference on information technology research (icitr)*.

Gupta, P., Sharma, A., & Jindal, R. (2018). An approach for optimizing the performance for apache spark applications. In *2018 4th international conference on computing communication and automation (iccca)*.

Hapke, H. M., & Nelson, C. (2020). *Building machine learning pipelines: Automating model life cycles with TensorFlow* (First edition). O'Reilly Media, Inc.

HashiCorp. (2021). *Terraform* [Computer software]. HashiCorp. https://www.terraform.io/.

I. Syarif, A. Prügel-Bennett, & G. Wills (2016). Svm parameter optimization using grid search and genetic algorithm to improve classification performance. *TELKOMNIKA Telecommunication Computing Electronics and Control*, *14*, 1502–1509.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2017). *An introduction to statistical learning: With applications in R* (Corrected at 8th printing). *Springer texts in statistics*. Springer.

Joblib developers. (2021). *Joblib* [Computer software]. Joblib developers. https://joblib.readthedocs.io/en/latest/.

Kakarla, R., Krishnan, S., & Alla, S. (2021). *Applied Data Science Using PySpark*. Apress. https://doi.org/10.1007/978-1-4842-6500-0

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems* (First edition). O'Reilly Media.

Kuhn, M. (2019). *Caret* [Computer software]. Caret Development. https://topepo.github.io/caret/.

Kumar, R., & Shilpi Charu. (2015). *Comparison between Cloud Computing, Grid Computing, Cluster Computing and Virtualization.* https://doi.org/10.13140/2.1.1759.7765

Laney, D. (2001). 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, *6*(70), 1.

Lukša, M. (2018). *Kubernetes in action*. Manning Publications. http://proquest.tech.safaribooksonline.de/9781617293726.

M. Hitz, & Behzad Montazeri (1995). Measuring coupling and cohesion in object-oriented systems. In

Mauro, A. de, Greco, M., & Grimaldi, M. (2016). A formal definition of big data based on its essential features. *Library Review*, *65*(3), 122–135. https://doi.org/10.1108/LR-06-2015-0061

Mesbahi, M. R., Rahmani, A. M., & Hosseinzadeh, M. (2018). Reliability and high availability in cloud computing environments: A reference roadmap. *Human-Centric Computing and Information Sciences*, *8*(1). https://doi.org/10.1186/s13673-018-0143-8

Migliorini, M., Castellotti, R., Canali, L., & Zanetti, M. (2020). Machine learning pipelines with modern big data tools for high energy physics. *Computing and Software for Big Science*, *4*(1). https://doi.org/10.1007/s41781-020-00040-0

Mitchell, R., Pottier, L., Jacobs, S., Da Silva, R. F., Rynge, M., Vahi, K., & Deelman, E. (2019). Exploration of workflow management systems emerging features from users perspectives. In *2019 ieee international conference on big data (big data)*.

Mitchell, T. M. (1997). *Machine learning* (International ed. [Reprint.]. *McGraw-Hill series in computer science*. McGraw-Hill.

NumPy. (2021). *NumPy* [Computer software]. NumPy. https://numpy.org/.

PostgreSQL. (2021). *PostgreSQL* [Computer software]. PostgreSQL. https://www.postgresql.org/.

Raju, A., Ramanathan, R., & Hemavathy, R. (2019). A comparative study of spark schedulers' performance. In *2019 4th international conference on computational systems and information technology for sustainable solution (csitss)*.

Reinsel, D., Rydning, J., & Gantz, J. (2021). *Worldwide Global DataSphere Forecast, 2021–2025: The World Keeps Creating More Data — Now, What Do We Do with It All?* International Data Cooperation. https://www.idc.com/getdoc.jsp?containerId=US46410421.

Sato, D., Wider, A., & Windheuser, C. (2019). *Continuous delivery for machine learning*. https://martinfowler.com/articles/cd4ml.html. Retrieved 24.05.2021.

Scikit-Learn Developers. (2021). *Scikit-Learn* [Computer software]. Scikit-Learn Developers. https://scikit-learn.org/stable/.

Sebei, H., Hadj Taieb, M. A., & Ben Aouicha, M. (2018). Review of social media analytics process and big data pipeline. *Social Network Analysis and Mining*, *8*(1). https://doi.org/10.1007/s13278-018-0507-0

Singh, P. (2019). *Learn PySpark*. Apress. https://doi.org/10.1007/978-1-4842-4961-1

Stefan Studer, Thanh Binh Bui, C. Drescher, A. Hanuschkin, Ludwig Winkler, S. Peters, & Klaus-Robert Müller (2020). Towards crisp-ml(q): A machine learning process model with quality assurance methodology. *ArXiv*, *abs/2003.05155*.

Thompson, W. J., & van der Walt, J. S. (2010). Business intelligence in the cloud. *SA Journal of Information Management*, *12*(1). https://doi.org/10.4102/sajim.v12i1.445

van Steen, M., & Tanenbaum, A. S. (2017). *Distributed systems* (Third edition (Version 3.01 (2017))). Pearson Education.

Vohra, D. (2016). Apache parquet. In D. Vohra (Ed.), *Practical hadoop ecosystem* (pp. 325–335). Apress. https://doi.org/10.1007/978-1-4842-2199-0_8

Wang, H., Ma, C., & Zhou, L. (2009). A brief review of machine learning and its application. In W. Hu (Ed.), *2009 international conference on information engineering and computer science: iciecs 2009 ; wuhan, china, 19 - 20 December 2009* (pp. 1–4). IEEE. https://doi.org/10.1109/ICIECS.2009.5362936

Weber, B. G. (2020). *Data science in production: Building scalable model pipelines with Python* [CreateSpace Independent Publishing].

Winkelhake, U. (2017). *Die Digitale Transformation der Automobilindustrie: Treiber - Roadmap - Praxis*. Springer. https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=5191366.

Wirth, R., & Hipp, J. (2000). Crisp-dm: Towards a standard process model for data mining. *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*.

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, *10*, 10.

Zhou, K., Song, M., & E, H. (2019). Spark-based machine learning pipeline construction method. In *2019 international conference on machine learning and data engineering*

(icmlde 2019): taipei, taiwan, 2-4 December 2019 (pp. 1–6). IEEE. https://doi.org/10.1109/iCMLDE49015.2019.00012

Zhu, C., Han, B., & Zhao, Y. (2020). A comparative study of spark on the bare metal and kubernetes. In *2020 6th international conference on big data and information analytics (bigdia)*.

9 Appendix

9.1 Class Diagram of the Data Model

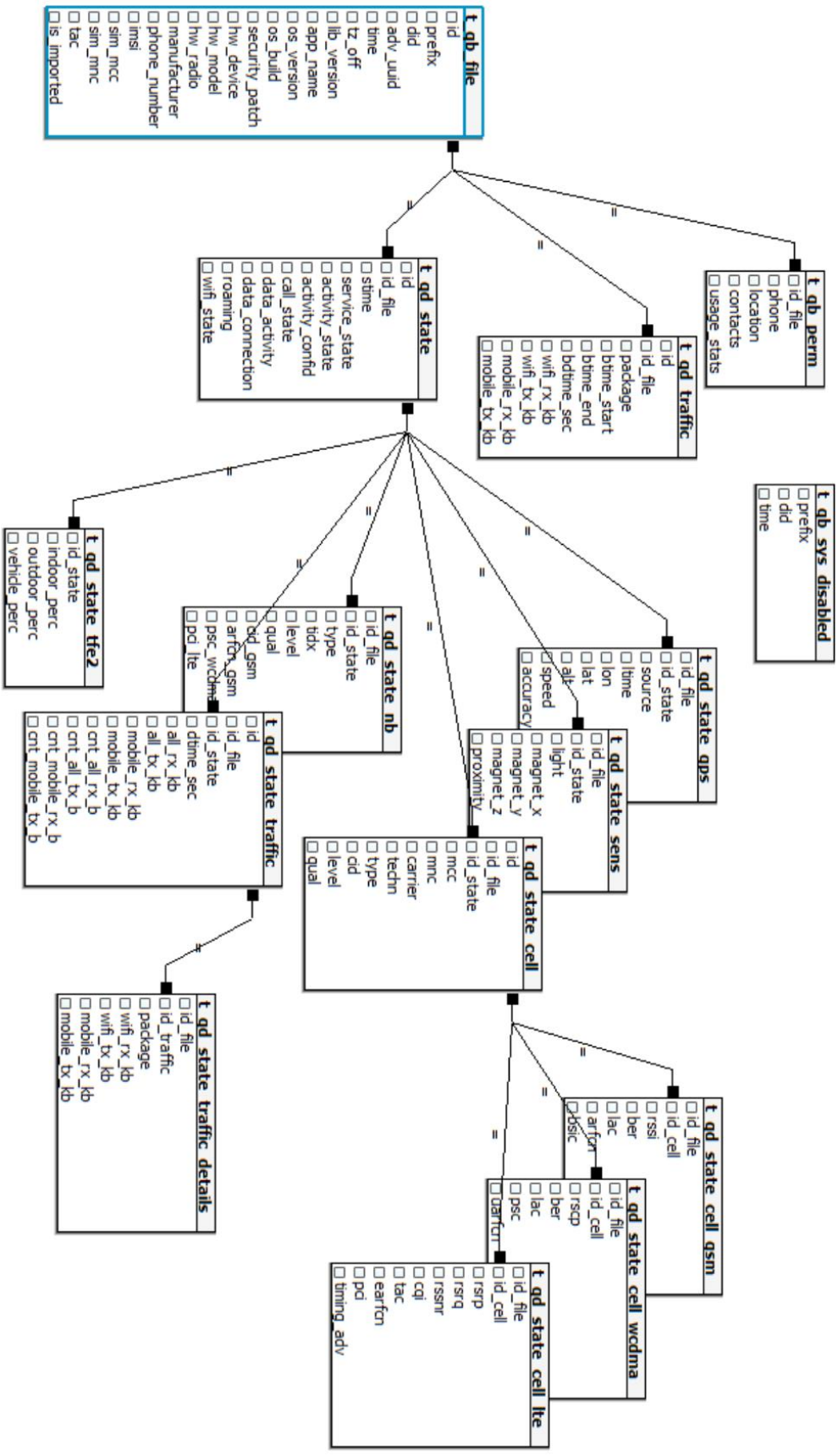


Figure 23: Diagram of the data model (NET CHECK internal documents, 2021)

9.2 Loading of the YAML Configuration

```
from pyspark.sql import SparkSession
import glob
import yaml

def loadConfig(path_config):
    """
    Loads a YAML configuration file from a specified path

    Parameters
    -----
    path_config: str, required
        path to the file
    """
    with open(path_config) as f:
        config = yaml.load(f, Loader=yaml.FullLoader)
    return config

spark = SparkSession.builder.getOrCreate()

config_path = glob.glob(f"{pyspark.SparkFiles.get('')}/*.yml")[0]
config = loadConfig(config_path)

adls_account_name = config['adls_account_name']
adls_account_key = config['adls_account_key']
spark.conf.set(
    f"fs.azure.account.auth.type.{adls_account_name}.dfs.core.windows.net",
    "SharedKey"
     )
spark.conf.set(
    f"fs.azure.account.key.{adls_account_name}.dfs.core.windows.net",
    adls_account_key
    )
```

9.3 Example for a Custom Transformer Class

```
from pyspark.ml.pipeline import Transformer
import pyspark.sql.functions as F
from pyspark.sql.types import *
from math import radians, cos, sin, asin, sqrt

class DistanceCalculator(Transformer):
    # DistanceCalculator herits of property of Transformer
    def __init__(self, lat_a, lon_a, lat_b, lon_b):
        """
        Initiates a DistanceCalculator transformer

        Parameters
        -----
        lat_a: str, required
            latitude coordinat of point a
        lon_a: str, required
            longitude coordinat of point a
        lat_b: str, required
```

```
        latitude coordinat of point b
    lon_b: str, required
        longitude coordinat of point b

    Returns
    -----
    Transformer object

    """
    self.lat_a = lat_a
    self.lon_a = lon_a
    self.lat_b = lat_b
    self.lon_b = lon_b
    self.output_col = 'DISTANCE'

def this():
    # define an unique ID
    this(Identifiable.randomUID("DistanceCalculator"))

def copy(extra):
    defaultCopy(extra)

def _transform(self, df):
    @F.udf(returnType=IntegerType())
    def _calcDistance(lon_a, lat_a, lon_b, lat_b):
        """
        Calculates the distance between two points in meter
        """
        geo_list = [lon_a, lat_a, lon_b, lat_b]
        if any(x is None for x in geo_list):
            return None
        else:
            # Transform to radians
            lon_a, lat_a, lon_b, lat_b = map(radians, geo_list)
            dist_lon = lon_b - lon_a
            dist_lat = lat_b - lat_a
            # Calculate area
            area = sin(dist_lat/2)**2 + cos(lat_a) * \
                cos(lat_b) * sin(dist_lon/2)**2
            # Calculate the central angle
            central_angle = 2 * asin(sqrt(area))
            radius = 6371
            # Calculate distance
            distance = central_angle * radius
            return int(abs(round(distance, 3)) * 1000)

    return (
        df.withColumn(
            self.output_col,
            _calcDistance(
                F.col(self.lon_a),
                F.col(self.lat_a),
                F.col(self.lon_b),
                F.col(self.lat_b))
        )
    )
```

9.4 Custom Dockerfile for the Pyspark Container Image

```
# inherit from base Spark image
ARG base_img=spark-base:latest

FROM $base_img
WORKDIR /

# Reset to root to run installation tasks
USER 0

RUN mkdir ${SPARK_HOME}/python
RUN mkdir /root/.ivy2
RUN apt-get update && \
    apt install -y python3 python3-pip && \
    pip3 install --upgrade pip setuptools && \
    # Install python libraries
    pip3 install numpy pytz pyyaml && \
    # Removed the .cache to save space
    rm -r /root/.cache && rm -rf /var/cache/apt/*

COPY python/pyspark ${SPARK_HOME}/python/pyspark
COPY python/lib ${SPARK_HOME}/python/lib
# include Spark packages such as Hadoop-azure
COPY .ivy2 /root/.ivy2

WORKDIR /opt/spark/work-dir
ENTRYPOINT [ "/opt/entrypoint.sh" ]

# Specify the User that the actual main process will run as
ARG spark_uid=0
USER ${spark_uid}
```

9.5 Example for Spark Submit to Kubernetes

```
$SPARK_HOME/bin/spark-submit \
    --master k8s://<cluster_url> \
    --deploy-mode cluster \
    --name ml-pipeline \
    --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
    --conf spark.executor.cores=5 \
    --conf spark.executor.instances=14 \
    --conf spark.driver.memory=3g \
    --conf spark.executor.memory=3g \
    --proxy-user root \
    --conf spark.kubernetes.container.image=<image_url> \
    --conf spark.hadoop.fs.azure.account.auth.type.
        <adls_account>.dfs.core.windows.net=SharedKey \
    --conf spark.hadoop.fs.azure.account.key.<adls_account>
        .dfs.core.windows.net=<adls_key> \
```

```
    --files "abfss://py-files@<adls_account>/config_prediction.yml" \
    --py-files "abfss://py-files@<adls_account>/jobs.zip" \
        "abfss://py-files@<adls_account>/main_kubernetes.py"
```

9.6 DAG Declaration File for the Prediction Pipeline

```
from airflow import DAG
from datetime import datetime, timedelta
from airflow.contrib.operators.spark_submit_operator
      import SparkSubmitOperator
from airflow.utils.dates import days_ago
from airflow.models import Variable


default_args = {
    'owner': 'lorenz',
    'start_date': days_ago(5),
    'email': ['lorenz.wackenhut@netcheck.de'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
}

default_conf_spark = {
    'spark.kubernetes.authenticate.driver.serviceAccountName': 'spark',
    'spark.executor.cores': 3,
    'spark.executor.instances': 24,
    'spark.executor.memory': '8g',
    'spark.driver.memory': '8g',
    'spark.driver.cores': 3,
    'spark.kubernetes.container.image': '<container_image>',
    'spark.hadoop.fs.azure.account.auth.type.<user>.dfs.core.
            windows.net': 'SharedKey',
    'spark.hadoop.fs.azure.account.key.<user>.dfs.core.
            windows.net': '<adls_key>'
}

spark_home = Variable.get("SPARK_HOME")

dag = DAG('prediction_pipeline_k8',
          schedule_interval='@once',
          default_args=default_args)

with dag:
    ingestion_prediction = SparkSubmitOperator(
        task_id='ingestion_prediction',
        name='ingestion_prediction',
        conn_id='spark_k8s',
        proxy_user='root',
        application='abfss://py-files@<user>.dfs.core.windows.net
            /main_kubernetes.py',
        files='abfss://py-files@<user>.dfs.core.windows.net
            /configuration/config_extraction_prediction.yml',
        py_files='abfss://py-files@<user>.dfs.core.windows.net/jobs.zip',
        conf=default_conf_spark,
```

```
        execution_timeout=timedelta(minutes=120),
        dag=dag)

    processing_prediction = SparkSubmitOperator(
        task_id='processing_prediction',
        name='processing_prediction',
        conn_id='spark_k8s',
        proxy_user='root',
        application='abfss://py-files@<user>.dfs.core.windows.net
            /main_kubernetes.py',
        files='abfss://py-files@<user>.dfs.core.windows.net
            /configuration/config_processing_prediction.yml',
        py_files='abfss://py-files@<user>.dfs.core.windows.net/jobs.zip',
        conf=default_conf_spark,
        execution_timeout=timedelta(minutes=120),
        dag=dag)

    prediction = SparkSubmitOperator(
        task_id='prediction',
        name='prediction',
        conn_id='spark_k8s',
        proxy_user='root',
        application='abfss://py-files@<user>.dfs.core.windows.net
            /main_kubernetes.py',
        files='abfss://py-files@<user>.dfs.core.windows.net
            /configuration/config_prediction.yml',
        py_files='abfss://py-files@<user>.dfs.core.windows.net/jobs.zip',
        conf=default_conf_spark,
        execution_timeout=timedelta(minutes=120),
        dag=dag)

ingestion_prediction >> processing_prediction >> prediction
```

9.7 Raw Data for the Scalability Tests

Table 13: Raw data for the scalability tests

num_nodes	cores_per_executor	pipeline	spark_start	spark_end	time_spark	time_parquet	time_bash
1	3	prediction_gbt	09:48:28	10:27:08	00:38:40		
1	5	prediction_gbt	10:30:34	11:10:58	00:40:24		
1	3	training_gbt	13:39:05	13:41:33	00:02:28		
1	5	training_gbt	13:43:26	13:46:12	00:02:46		
1	3	training_gbt_increased	16:02:04	16:54:58	00:52:54		
1	5	training_gbt_hyper	14:43:50	15:51:52	01:08:02		
2	3	prediction_gbt	13:26:18	13:42:22	00:16:04		
2	3	prediction_gbt	10:23:54	10:41:26	00:17:32		
2	5	prediction_gbt	12:58:35	13:15:00	00:16:25		
2	3	training_gbt	14:18:50	14:21:33	00:02:43		
2	5	training_gbt	13:50:34	13:52:50	00:02:16		
2	3	training_gbt_increased	14:16:30	14:43:54	00:27:24		
3	3	prediction_gbt	08:36:56	08:48:26	00:11:30		
3	5	prediction_gbt	08:52:39	09:04:18	00:11:39		
3	5	training_gbt	09:12:36	09:14:55	00:02:19		
3	5	training_gbt_hyper	10:38:50	11:46:33	01:07:43		
3	3	training_gbt	11:52:54	11:55:24	00:02:30		
3	3	training_gbt_increased	14:49:53	15:10:10	00:20:17		
4	3	prediction_gbt	12:07:06	12:15:57	00:08:51		
4	5	prediction_gbt	12:21:10	12:29:42	00:08:32		
4	3	training_gbt	12:36:33	12:39:07	00:02:34		
4	5	training_gbt	12:49:29	12:51:45	00:02:16		
4	3	training_gbt_increased	15:20:37	15:36:39	00:16:02		
5	3	prediction_gbt	13:06:04	13:13:19	00:07:15		
5	5	prediction_gbt	13:16:29	13:24:02	00:07:33		
5	3	training_gbt	13:26:18	13:29:03	00:02:45		
5	5	training_gbt	13:36:09	13:38:35	00:02:26		
5	5	training_gbt_hyper	14:22:50	15:32:41	01:09:51		
5	3	prediction_gbt	13:41:09	13:48:14	00:07:05	00:03:41	00:07:37
5	3	training_gbt_increased	15:41:52	15:55:38	00:13:46		
5	3	prediction_gbt_airflow			00:07:06	00:05:04	
5	3	training_gbt_airflow			00:00:00	00:00:38	
5	3	training_gbt_airflow_increased					
		prediction_legacy					02:13:56
		training_legacy					00:02:45

9.8 Declaration on Oath

I hereby declare in lieu of an oath that

- I have prepared this academic paper independently and without unauthorised assistance,
- I have not used any sources or aids other than those indicated,
- I have marked as such the passages taken verbatim or in terms of content from the sources used,
- the paper has not yet been submitted in the same or similar form to any other examination authority.

Berlin, 12.09.2021