

Institute of Parallel and Distributed Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Masterarbeit

# **Comprehensive Support of the Lifecycle of Machine Learning Models in Model Management Systems**

Matthias Popp

**Course of Study:** Softwaretechnik  
**Examiner:** PD Dr. rer. nat. habil. Holger Schwarz  
**Supervisor:** Christian Weber, M.Sc.

**Commenced:** May 4, 2019  
**Completed:** November 4, 2019



## **Abstract**

Today, Machine Learning (ML) is entering many economic and scientific fields. The lifecycle of ML models includes data pre-processing to transform raw data into features, training a model with the features, and providing the model to answer predictive queries. The challenge is to ensure accurate predictions by continuously updating the model with automatic or manual retraining. To be aware of all changes, e.g. datasets and parameters, it is required to store metadata over the entire ML lifecycle.

In this thesis we present a concept and system for comprehensive support of the ML lifecycle. The concept includes a metadata schema, as well as a solution to collect and enrich the metadata. The metadata schema contains information about the experiment, runs, executions, executables and common artifacts in ML such as datasets, models, and metrics. The stored information can be used for comparisons, re-iterations, and backtracking of ML experiments. We achieve this by tracking the lineage of ML pipeline steps and collecting metadata such as hyperparameters. Furthermore, a prototype is implemented to demonstrate and evaluate the concept. A case study, based on a selected scenario, serves as the basis for a qualitative assessment. The case study shows that the concept meets all the requirements and is therefore a suitable approach to comprehensively support ML model lifecycle.



## Kurzfassung

Heutzutage dringt Machine Learning (ML) in viele wirtschaftliche und wissenschaftliche Bereiche vor. Der Lebenszyklus von ML Modellen umfasst im Wesentlichen die Datenvorverarbeitung für das Training des Modells, das Training selbst und die Bereitstellung des Modells zur Beantwortung von Vorhersageanfragen. Da sich Daten mit der Zeit ändern, können Modelle an Genauigkeit verlieren. Deswegen werden ML Pipelines eingesetzt, die das Modell regelmäßig neu trainieren damit es kontinuierlich genaue Vorhersagen liefert. Trotzdem treten immer wieder Fehler in den Pipelines auf. Oft fehlen jedoch die benötigten Metadaten, um diese zu erkennen. Die Herausforderung besteht darin, die benötigten Metadaten in geeigneter Form zu speichern und für Datenwissenschaftler aufzubereiten.

Im Rahmen dieser Arbeit wird ein Konzept zur Metadatenerfassung für den kompletten Lebenszyklus von ML Modellen vorgestellt und an einem Prototyp verprobt. Hierfür werden verschiedene Metadaten und Artefakte während der Nutzungs- und Wartungsphase des Modells gesammelt und gespeichert. Dazu gehören alle deskriptiven und strukturellen Metadaten über die Schritte und deren Reihenfolge sowie die verwendeten Ein- und Ausgangsartefakte wie Datensätze, Modelle und Metriken. Es wird ein Metadatenschema entworfen, das innerhalb des Prototyps umgesetzt wird. Abschließend erfolgt eine Bewertung des Konzepts mittels einer Fallstudie. Abschließend wird demonstriert wie der geschaffene Prototyp Datenwissenschaftler bei einem konkreten Szenario der Root-Cause-Analyse von fehlerhaften ML Modellen unterstützt. Die Fallstudie zeigt, dass das Konzept alle Anforderungen erfüllt und somit ein geeigneter Ansatz ist, um den ML Modelllebenszyklus umfassend zu unterstützen.



# Contents

<b>1</b>	<b>Introduction</b>	<b>17</b>
1.1	Scope and Contributions . . . . .	18
1.2	Thesis Outline . . . . .	19
<b>2</b>	<b>Background</b>	<b>21</b>
2.1	Machine Learning Model Lifecycle . . . . .	22
2.2	Model Management Systems . . . . .	24
<b>3</b>	<b>Related Work</b>	<b>27</b>
3.1	Metadata Management for Machine Learning . . . . .	27
3.2	Management of Workflows in Machine Learning . . . . .	28
<b>4</b>	<b>Scenarios</b>	<b>31</b>
4.1	Scenario 1: Root cause analysis . . . . .	31
4.2	Scenario 2: Collaboration multiple data scientists . . . . .	31
4.3	Scenario 3: Automated workflows . . . . .	32
4.4	Scenario 4: Metadata Collection . . . . .	32
4.5	Scenario 5: Custom Artifact Schemas . . . . .	32
<b>5</b>	<b>Existing Technologies</b>	<b>33</b>
5.1	Commonly used Libraries for Machine Learning . . . . .	35
5.2	Storage and Data Preprocessing . . . . .	36
5.3	Deployment Services . . . . .	37
5.4	Model Lifecycle Management Frameworks . . . . .	37
<b>6</b>	<b>Concept</b>	<b>39</b>
6.1	Requirements . . . . .	39
6.2	Overview . . . . .	41
6.3	Metadata Schema . . . . .	41
6.4	Architecture . . . . .	49
<b>7</b>	<b>Implementation</b>	<b>51</b>
7.1	Infrastructure . . . . .	51
7.2	Components . . . . .	52
7.3	Integration . . . . .	55
<b>8</b>	<b>Case Study</b>	<b>57</b>
8.1	Root Cause Analysis . . . . .	57
8.2	Discussion of further Cases and Requirements . . . . .	61

<b>9 Conclusion and Outlook</b>	<b>63</b>
<b>Bibliography</b>	<b>65</b>
<b>A Appendix</b>	<b>69</b>



# List of Figures

2.1	Model Lifecycle with all steps [WHRS19]	23
2.2	Machine Learning Lifecycle [ACP19]	24
3.1	The ML Schema core vocabulary. The diagram depicts Information Entities as yellow boxes, Processes as blue boxes, and Qualities as green boxes [PEŁ+18].	28
3.2	ML metadata schema to store artifact metadata and lineage information [SBK+17].	29
3.3	ML-based workflow Abstraction.	30
6.1	Concept overview	41
6.2	Proposed Metadata Schema	43
6.3	Architecture Overview	49
7.1	Trace Structure	53
7.2	Metadata are listed in a table for an overview.	55
8.1	Sample pipeline comprising three steps: 1) preprocessing the MNIST dataset [LBBH98], 2) training the model, and 3) deploying the model. Details of the generated steps and artifacts are displayed on the right.	58
8.2	Pipeline trace of an Convolutional Neural Network (CNN) model	59
8.3	A detail view of the validation artifact	60
8.4	An example predict call	60
8.5	Pipeline trace visualization to detect the root cause of an incorrect model	61



## List of Tables

5.1	Existing Machine Learning Frameworks Overview . . . . .	34
6.1	Metadata types [Zen04] . . . . .	42
6.2	Metadata Experiment . . . . .	44
6.3	Metadata Run . . . . .	45
6.4	Metadata Pipeline . . . . .	45
6.5	Metadata Executable . . . . .	46
6.6	Metadata Execution . . . . .	46
6.7	Metadata Artifact . . . . .	47
6.8	Metadata Dataset . . . . .	47
6.9	Metadata Model . . . . .	48
6.10	Metadata Metrics . . . . .	48
6.11	Metadata Event . . . . .	48



## List of Listings

7.1	Example usage of the Python client library. . . . .	54
7.2	Example usage of the TypeScript client library. . . . .	54



# Acronyms

**BAM** business activity monitoring. 32

**CNN** Convolutional Neural Network. 9

**ML** Machine Learning. 3, 5

**MMP** Model Management Platform. 17

**MMS** Model Management System. 17

**PMML** Predictive Model Markup Language. 23

**SVM** Support Vector Machine. 35





# 1 Introduction

Today, many industrial application scenarios require the use of machine learning models. Well-known examples are Amazon or Google which analyze purchasing patterns of customers to offer individualized product recommendations. However, building and operating machine learning models involves high efforts for data scientists. Many different versions of a model are build, updated, and retired at the same time, resulting in a high management effort for a single model. Thus, the lifecycle of machine learning models requires model management across all lifecycle steps [WHRS19].

Unlike in the traditional software lifecycle, machine learning models face different challenges as they are put into production. For example, due to changes in data, models can become stale which leads to false inference and prediction results [BCN+17]. Consequently, a ML model needs to be regularly updated or even re-developed. This leads to regressions to previous phases, whereby many phases are run through several times until the ML model is finally retired. When a model is put into production, many of these steps to use and update the model are implemented in *Machine Learning Pipelines*, which are covered in scripts that are used to execute the pre-processing, training and deployment steps. Weber et al. [WHRS19] call this process the *update-loop* in the life cycle of an ML model. Every time the pipeline is re-run, it produces a new model version that is deployed into its target environment. This makes it easier to update a model when it is stale. However, model staleness is sometimes not detected in time and errors have already been introduced in previous executions of the ML Pipeline. For example, a data scientist could have replaced the training dataset with a wrong one or changed the pipeline configuration. In order to find out why this error occurred and in which previous life cycle step it was introduced, comprehensive metadata must be collected. This includes all descriptive and structural metadata about the steps and their sequence as well as the input and output artifacts used, such as data sets, models and metrics. However, these metadata are often not captured at all and root cause analysis can quickly become a burden for data scientists. A reason for missing metadata is that capturing these metadata is complex. They are spread across different data mining tools, frameworks, and ML serving systems that are used within the model's lifecycle. This imposes a great challenge for comprehensive metadata tracking.

State-of-the-Art technologies and scientific concepts target primarily the steps before a model is put into operation and do not cover the ML lifecycle comprehensively. Examples for concepts and according technologies are Model Management System (MMS) such as ModelDB [VM] and MLflow [ZCD+18]. These allow data scientists to track metadata about the history of their experiments so that they can reproduce and share their results. In the future, however, all phases of a model's lifecycle must be supported through metadata tracking to enable efficient root cause analysis for use cases in which a model becomes stale.

In order to address the aforementioned issues and taking up the challenge of comprehensive metadata tracking, the Chair of Application Software (AS) at the Institute for Parallel and Distributed Systems (IPVS) develops and introduces various concepts for model management to the scientific community via a PhD project. The concepts are prototypically implemented and demonstrated in a

Model Management Platform (MMP) to show the applicability for selected use cases. The current prototype contains a repository to store, structure, version and describe models with corresponding metadata. Training and serving of the ML models are not part of the MMP and are performed by distributed execution platforms for parallel data processing such as Apache Spark [ZCF+10]. Thus, the prototype has a similar range of functions compared to existing model management systems.

The goal of the thesis is to develop a concept to extend the MMP by functionalities for comprehensive metadata tracking to support the remaining steps in the life cycle.

### 1.1 Scope and Contributions

This thesis is conducted in the scientific area of *Data Management for End-to-End Machine Learning* [SPVS19]. Existing concepts address the experimentation steps of the ML Lifecycle by tracking, storing, and indexing machine learning models. However, current challenges are to extend the support of metadata tracking to the operational steps of the machine learning lifecycle. For this reason, this work focuses on the operational steps and presents a system that enables comprehensive support in the ML lifecycle. This thesis makes the following five contributions:

1. Creation of exemplary use cases that require backtracking functionalities to demonstrate the need for the concept.
2. An evaluation of existing technologies that provide model management support for data scientists. Criteria for the evaluation are derived from related work and typical activities of data scientists that require model management.
3. A metadata schema to store metadata and artifacts across the whole lifecycle of ML models. Its design eases the interpretation and analysis of such metadata, i.e. lineage information.
4. A prototype of the system that implements the metamodel to provide comprehensive backtracking functionalities. Some core functionalities include the extraction of metadata from ML pipelines and lineage tracking. To keep pace with the rapid changes in the industry, new tools and frameworks can be easily plugged in. In addition, the prototype is cloud native and can therefore be used by both large companies and small ML teams. An appropriate graphical presentation of the metadata helps to meet different stakeholder requirements.
5. Finally, a case study shows how the prototype supports data scientists in diagnosing ML pipelines for a selected use case. Thereby, the functionalities of the system are demonstrated.

## 1.2 Thesis Outline

The remainder of this thesis is structured as follows:

**Section 2 – Background:** This chapter introduces the scientific area in which the thesis is conducted. We describe the areas of data management for machine learning and the Model Management Platform (MMP) which was developed in a previous project.

**Section 3 – Related Work:** The next chapter describes scientific work in the same topic area regarding the lifecycle of a machine learning model, the systems that support it, and according metamodels that are required to store different kinds of metadata.

**Section 4 – Scenarios:** This chapter illustrates five scenarios that illustrate the need for our concept, namely root cause analysis, collaboration between multiple data scientists, automated workflows, metadata collection, and custom artifact schemas.

**Section 5 – Existing Technologies:** This chapter discusses and evaluates existing technologies by defining criteria that are relevant for integration into our concept. We show how our functionality differs from that of existing technologies.

**Section 6 – Concept:** The concept chapter describes the created metadata model as well as the architecture and the functional components of the prototype.

**Section 7 – Implementation:** This chapter describes the prototype that was created to show the usability of the created metamodel. Technologies and software frameworks used within the prototype will be discussed.

**Section 8 – Case Study:** This chapter describes the exemplary application of our prototype to the root cause analysis scenario using a case study. That is, we explicitly highlight how the prototype solves the problems in this scenario.

**Section 9 – Conclusion and Outlook:** Finally, this chapter summarizes the contents of the thesis and gives an outlook on improvements and possible future work.



## 2 Background

Due to the current flood of data, ML is becoming more and more important. Various industries are trying to use ML to gain knowledge from data across a broad spectrum of use cases, ranging from predictive maintenance to recommendation system and even for medical diagnosis. All these use cases have one thing in common, each developed model goes through more or less the same steps, which are called the lifecycle of a ML model.

ML development creates multiple new challenges that are not present in a traditional software development lifecycle. Traditional software developers have a defined set of product features to build or maintained, data scientists will constantly experiment with new libraries, hyperparameters, datasets, etc. to reach the maximal business value. Keeping track of all these adjustment possibilities is extremely difficult.

The understanding of the machine learning lifecycle is constantly evolving. At the beginning the lifecycle was more a cycle, the emphasis was on more on the steps data preparation and modeling. Steps like deployment and delivery were considered less. Data scientists often stuck in their notebooks without taking care of the steps before and after the modeling step. Similar symptoms have been found in software development and an action was to introduce DevOps and DevOps tools. For this reason, MLOps is often referred to when describing the collaboration and communication between data scientists and operations professionals to support the complete lifecycle of ML models. Few approaches try to support all aspects, but none of them meets all requirements. There are no standard systems and interfaces for managing the entire lifecycle. Existing frameworks offer for example training and serving interfaces, but they cannot be used with another ML library. To research this field, the University of Stuttgart developed a MMS called MMP for Industry 4.0 (for more details see Selection 2.2). The focus of the project was already to support the complete ML model lifecycle, but in a more abstract and static approach. Individual steps were inspected separately and transitions were also not well supported. Thus it was difficult to find and detect root causes. For example a ML model was training with a wrong or outdated dataset, it is difficult to reproduce the process of the model training with all the parameters and inputs.

Therefore, this thesis develops a concept to comprehensive support of the lifecycle of ML models, which can be integrated into the existing MMS. Existing approaches are often limited to a subset of steps in a complete ML pipeline. Moreover, the traceability of the pipeline is often not provided to identify the causes and if then mostly not to the beginning. The beginning is not the first step in the pipeline, but the trigger which starts the pipeline. It is often helpful to know why a pipeline was started, either because the dataset has changed or because hyperparameters have been updated.

Major cloud providers are also racing to provide a comprehensive stack that delivers ML as a service (e.g., Amazon, IBM, Microsoft, and Google) by offering MMS. Furthermore, they provide clusters of GPU-backed VMs and containers through a pay-as-you-go pricing model. Data scientists can therefore leverage the cloud computing infrastructure to train machine learning models and take advantage of additional compute resources. Additionally, the cloud is also the first place to go to start experimenting with ML without buying expensive infrastructure. Therefore, the concept is based on the cloud native principle.

In the following, we describe two important topics, which are required to understand the related work, scenarios, concept, and prototype. Section 2.1 explain the ML model lifecycle and Section 2.2 deals with model management systems.

### 2.1 Machine Learning Model Lifecycle

Related work provides several descriptions of the ML model lifecycle. These include generic descriptions such as the model lifecycle of Weber et al. [WHRS19] and Ashmore et. al. [ACP19], and also descriptions, e.g. for deep learning such as the description of Miao et. al. [MLDD17]. These descriptions are discussed in the following.

Weber et. al. [WHRS19] describe the Model Lifecycle as a five step process: *plan model*, *build & test model*, *deploy model*, *use & monitor model*, and *retire model*, shown in Figure 2.1.

**Plan Model.** This is the first step and is used to define requirements and goals. Equivalent to software engineering, stakeholders are involved to verify and evaluate these requirements.

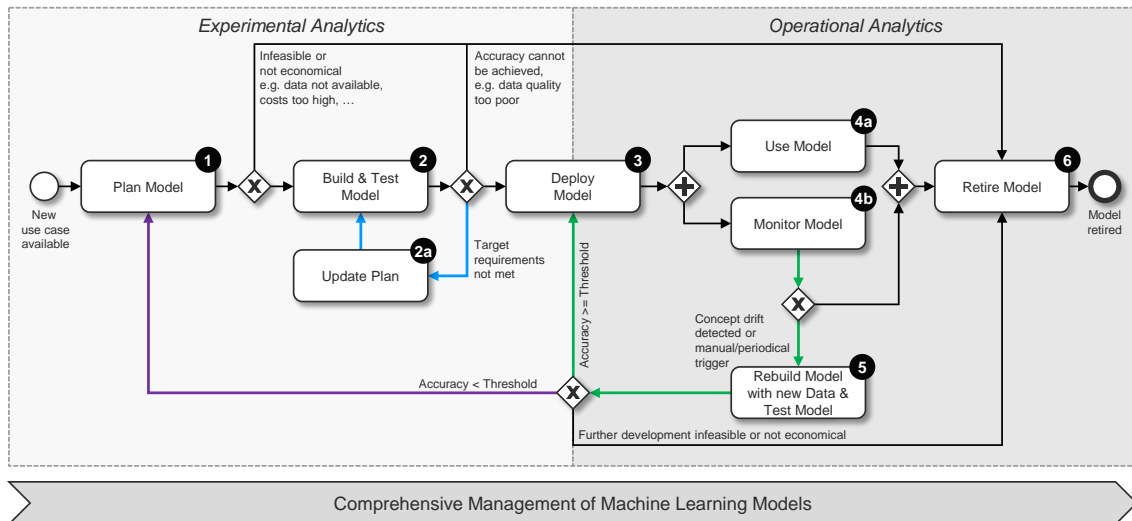
**Build & Test Model.** This step is made up of several substeps. These include data exploration, preprocessing, training, tuning, and test operations. For data exploration, several descriptive summaries of data are generated and possible features are selected to build the model. The preprocessing substep includes data cleansing, feature extraction, and the transformation to the target data format as required input into the algorithm. In the training step, an algorithm is selected, applied to the data, and a model is being generated. These steps are repeated iteratively until the model has reached a desired performance and quality threshold. For tuning the model's performance, several techniques, e.g. hyperparameter tuning are used. The goal is to prevent the model from overfitting by reducing the generalization error of the model. Therefore, Weber et. al. created an intermediate step, the update plan step. In this step, the problems are worked out why the model did not reach the goals and what has to be changed.

**Deploy Model.** This is the step between the experiment phase and operational phase. The goal is to deploy the model in its target environment so that the model can be integrated with its predefined components. Typically, target environments are ML serving systems, e.g. Apache Spark [ZCF+10] for batch and stream analytics.

**Use & Monitor Model.** After the model is deployed to its target environment, the model can be used by different stakeholders. While the model is being used it can also increase its performance by online or offline training. At the same time, data scientists need to monitor the model. The monitoring step is important because models can become stale over time. Not only the usage (e.g. requests per minute) can be monitored, also the input data can be observed to detect concept drifts.

**Retire Model.** This is the last and final step of the ML model lifecycle process. There can be many different reasons why a model needs to be retired. Examples are infeasibility, bad quality, changing circumstances or economical reasons.

Ashmore et. al. [ACP19] divided the ML lifecycle in four stages, shown in Figure 2.2. The first three stages, *Data Management*, *Model Learning*, and *Model Verification* cover the process by which ML models are produced. The fourth stage, *Model Deployment*, comprises the deployment, monitoring, and updating of ML models, like in other areas of software engineering. They assume in their work, that requirements engineering was done before and therefore it is not part of the ML



**Figure 2.1:** Model Lifecycle with all steps [WHRS19]

lifecycle itself.

The first stage *Data Management* consists of four activities, the *collection* of data samples from the real-world or a representation of the real world, *augmentation* methods for adding further data samples, *preprocessing* to reduce the complexity of the data or smoothing of data, and *analysis* used to validate if additional data collection, augmentation and preprocessing is required.

*Model Learning* requires the previously generated data to perform, *model selection* based on the type of problem (e.g. regression or clustering) and the structure of data, *training* of the model, *hyperparameter selection* to increase the performance of the model, and *transfer learning* which can be used if models in related context are available. If the model does not fulfil predefined levels of performance, the process needs to return to the previous stage.

In the *Model Verification* stage, the challenge is to ensure that the previously generated model works as well as it should. In a *test-based verification* process, unseen data from the data management phase is used to test the model and if the error rate violates criteria, the process has to return to the data management or model learning stage. The aforementioned criteria are established by a *requirement encoding* activity. As a last activity before the model is ready for deployment, a *formal verification* (e.g. model checking or mathematical proof) may be used to verify if the model fulfils the requirements for the ML component.

After the model is verified, the model is deployed in the *Model Deployment* stage. The model is integrated with existing system components in the *integration* step. Like other software components, the model will also be *monitored* and updated through offline maintenance or online training.

More specific lifecycle management for deep learning models is described by Miao et. al. [MLDD17]. It has similarities to the already described lifecycles, but they assume that the data samples are already collected and preprocessed. Moreover, they also split the train model stage in two stages: *Create/Update Model* and *Train/Test Model*. The reason for this is that often well-known models for similar tasks in other domains already exist and can therefore be easily reused with minimal adjustments.

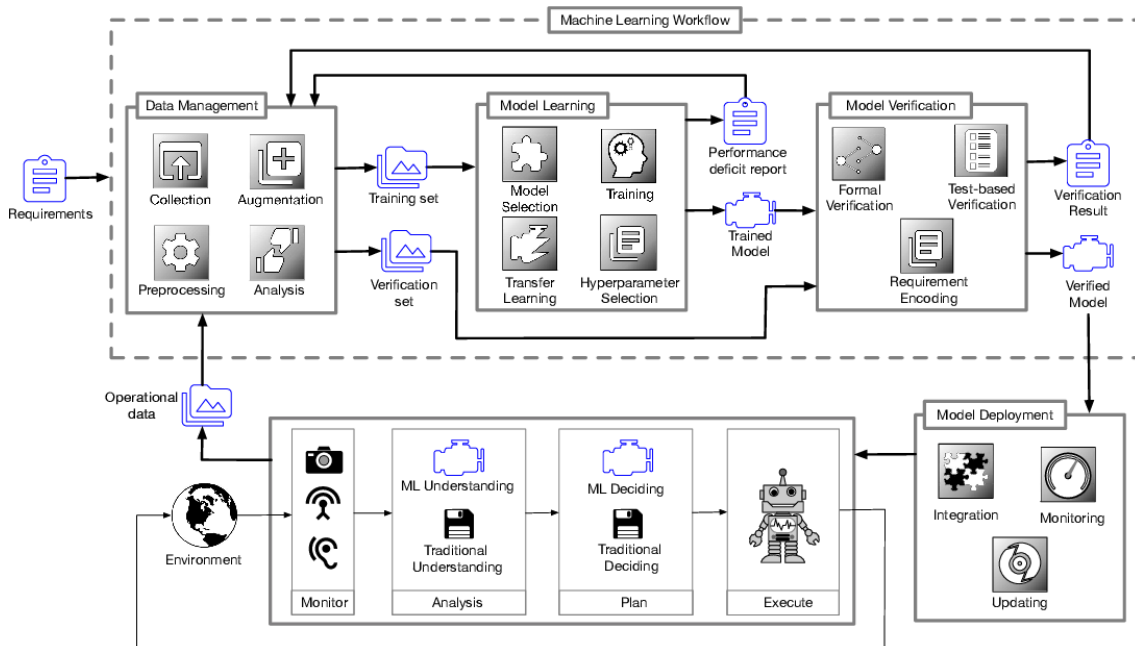


Figure 2.2: Machine Learning Lifecycle [ACP19]

## 2.2 Model Management Systems

Over the lifecycle of a ML model, data scientists build many tens to hundreds of models. In the beginning, a lot of try and error are necessary before a model fulfills some acceptance criteria. Due to changes in the requirements or data sets, a ML model must be updated over its lifetime. To manage all these created ML models, a model management system is needed so that the scientists do not need to remember all created models and can answer questions like "Which model performs best with this kind of data set?". The goal that any ML model management system attempts to achieve is to support the entire ML model lifecycle as described in Chapter 2.1. Therefore, they must also capture and query the semantics, metadata, and lineage of workflow systems [WGLM18], as described in Section 3.2.

ModelDB [VSL+16] was the first open-source system for model management which tries to handle the problem of storing, tracking, and indexing large numbers of ML models. By providing an overview of models created so far, data scientists can recapitulate their results. Additionally, it helps to find trends and features by performing meta-analyses across models. Moreover, with indexing, the scientist can browse the models. Due to the central storage of the automatically tracked models and environments, collaboration between data scientists is improved. The tracking of metadata is realized with provided native client libraries for popular ML environments. The ModelDB backend then stores and indexes the data so that the frontend can visualize it. This visualization provides functionality to explore the models and run meta-analyses.

Machine learning as a service (MLaaS) is a general definition of various cloud-based platforms that cover most ML issues such as data preprocessing, model training, model storage, model evaluation, and prediction [RGC15]. The big four cloud providers Amazon, IBM, Microsoft, and Google developed their own MLaaS solutions. Amazon Machine Learning services, Azure



Machine Learning, Google Cloud AI, and IBM Watson are the leading cloud MLaaS services, but unfortunately, the designs and implementation of these products are not open source. Companies like Uber (Michelangelo) and Facebook (FBLearner) also created their own ML platforms, but these are only created for company intern usage and therefore are not released as open software. The MMP, as mentioned before, is a project developed by students at the University of Stuttgart. Due to the constantly growing amount of ML models in production, the goal of the project was to develop a platform for the efficient administration of these models. The system was designed to support users with different roles. Data scientists, IT architects and other stakeholders are able to store, version and search ML models. In addition, metadata were automatically extracted from Predictive Model Markup Language (PMML) models. A unique feature is the Enterprise Architecture Management for Analytics view. It allows a user to dynamically assign models to specific business processes and organization units.



## 3 Related Work

This chapter describes relevant related work on which this work is based in various areas. It is divided into areas of ML Model Metadata Management 3.1 and ML Model Workflow Management 3.2. Both areas metadata management and workflow management are well established topics in science. However, the specification to ML is a newer research area and are therefore dealt with in this chapter.

### 3.1 Metadata Management for Machine Learning

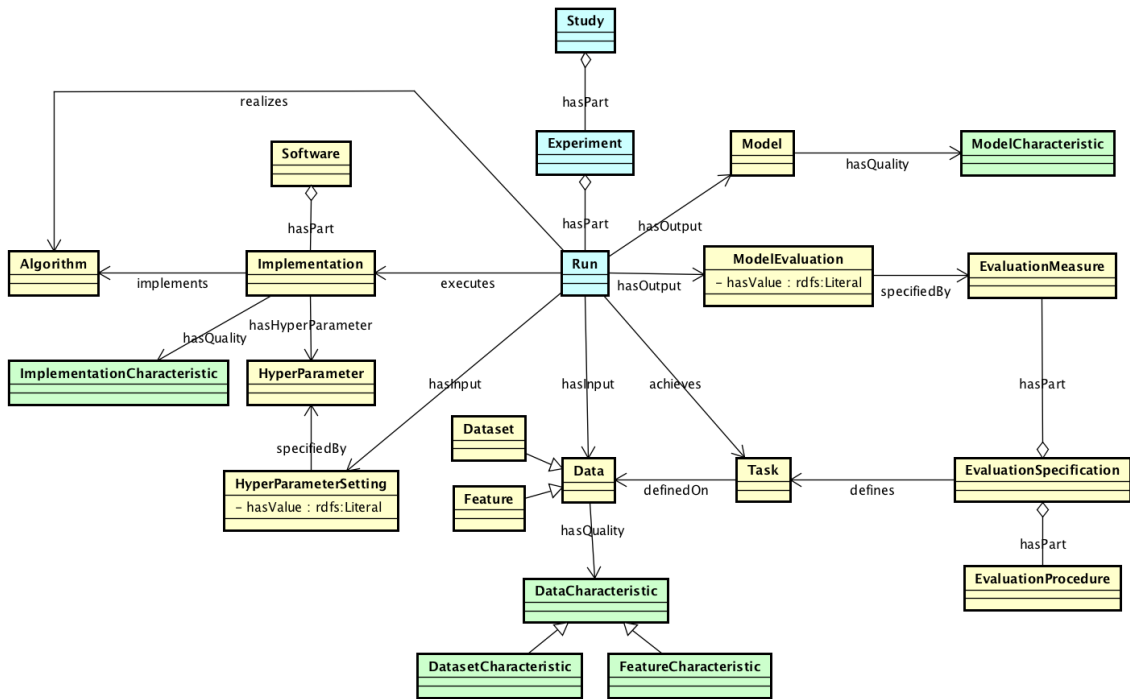
Typically, data scientists conduct this experimentation in their own ad-hoc style without a standardized way of storing and managing the resulting experimentation data and artifacts [WGLM18]. The consequences of this are the experiments are not comparable, searchable or reproducible. For this reason, metadata management of machine learning was introduced, which are described below.

Platforms such as OpenML [VRBT14] are used to promote cooperation between scientists, when developing machine learning models. To ensure this, a ML metadata schema is needed. Publio et. al [PEŁ+18] created one which is currently used in OpenML. Shown in Figure 3.1, the W3C Machine Learning Schema Community Group created an ontology. It provides a set of classes, represented by boxes. Properties are represented by arrows without filled heads, arrows with empty heads represent subclass relations, and arrows with diamonds represent part-of relations. Information about ML algorithms, data sets and experiments can be displayed and exchanged. The schema is designed to be easily specialized and extended, for instance, to represent more complex designs like deep learning problems. The core schema is divided into four parts. It includes representations of:

- data, data sets and their characteristics.
- algorithms, implementations, and parameters of implementations and software.
- models, model characteristics, and model evaluations.
- machine learning experiments with granularity of runs and studies as highest level.

Schelter et. al [SBK+17] go one step further and try to find the trade-off between generality and interpretability of the schema. On the one hand, they created predefined attributes (e.g. learning algorithms of a model) and on the other hand, they are open and allow use cases specific annotations. As shown in Figure 3.2 the schema also provides lineage information, like which evaluation was used to validate the model. They also follow the declarativity principle and store no code or data itself. They limit themselves to only store pointers to the used code or input data sets.

To reduce additional effort of data scientists, they integrated functionality to extract metadata automatically. They limited themselves to the ML frameworks SparkML, scikit-learn, and MXNet.



**Figure 3.1:** The ML Schema core vocabulary. The diagram depicts Information Entities as yellow boxes, Processes as blue boxes, and Qualities as green boxes [PEL+18].

### 3.2 Management of Workflows in Machine Learning

The development that ML models are part of large ML pipelines requires the use of appropriate workflow management systems. Workflow management systems are software systems to support workflow control of administrative processes. There are different types of workflows, such as business workflows and scientific workflows. Business workflows are more focused on control-flow patterns and events, whereas scientific workflows are more dataflow oriented [LAB+06]. Scientific workflow managers, such as Kepler [LAB+06] or Apache Taverna [HWS+06], are specially designed to compose and execute a series of calculations or data manipulation steps. More specific scientific workflows are machine learning based workflows. A ML based workflow is a directed acyclic graph of operations (See Figure 3.3). Each node in the graph represents an operator ( $S_i$ ) and each edge represents the output ( $A_i$ ) of an operation. An operation can perform data transformations or train a ML model. The parameters  $\gamma_i$  influence the outputs of the operations.

There are many examples of coarse-grained provenance systems including a large number of scientific workflow systems described above (each of these systems capture transformations that are applied to data and seek to answer questions about how a dataset came to be). Some systems, such as yesWorkflow [MSK+15], also capture data lineage as part of their provided functions.

In the following, approaches will be presented that are focused on ML based workflows.

Hummer et. al [HMR+] have an approach to providing machine learning models and have developed a cloud-based framework to support the whole life cycle of artificial intelligence applications. They have encountered the following problems, which they try to solve with their framework, *Automation*,

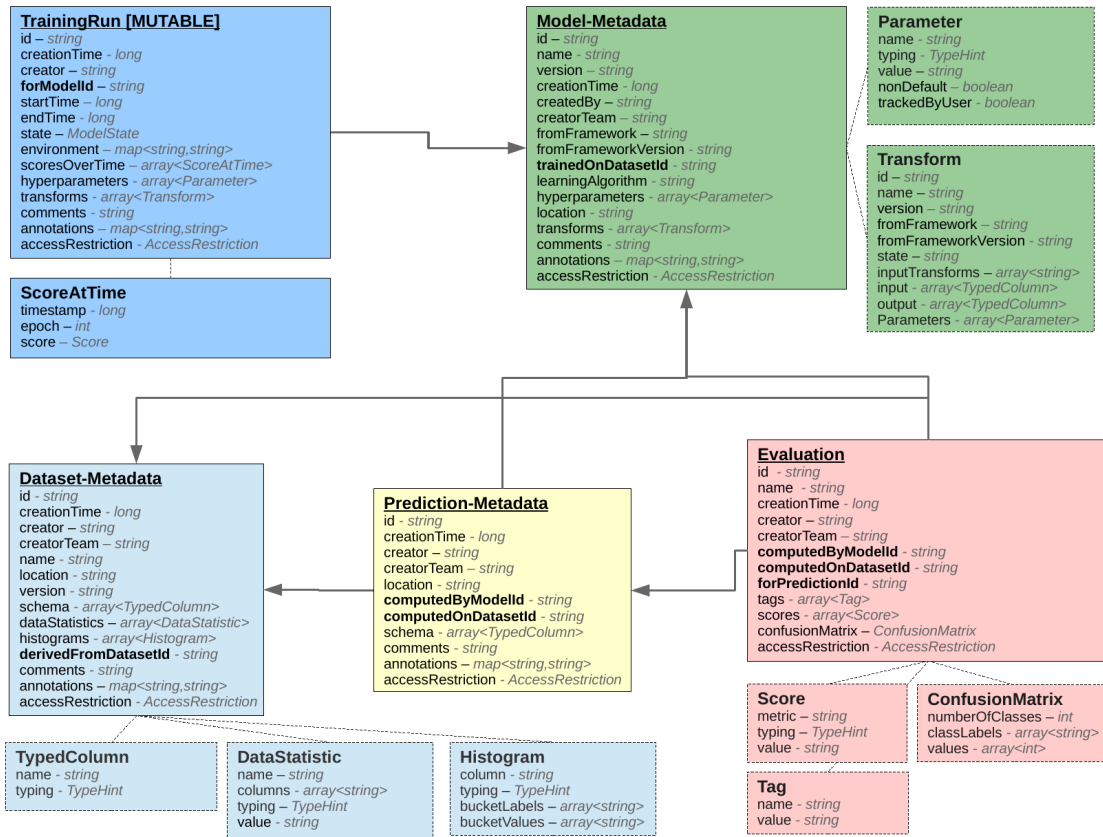
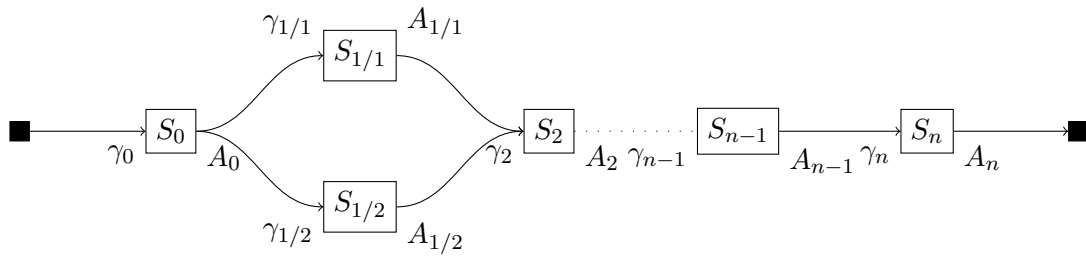


Figure 3.2: ML metadata schema to store artifact metadata and lineage information [SBK+17].

*Quality Assurance, Traceability, Risk Management and Feedback Cycle.* Their system design aims to solve these problems. Model training pipelines can be used to ensure *Automation and Quality Assurance*. Metadata management is used to collect data during the complete cycle and can be used later on to determine which data is used to train which model. This will solve the problem of *Traceability*. With the help of monitoring and event triggers, the quality can be kept constantly high, since the automated pipeline is started when the data changes. Each step can be monitored and leads to the fact that *Quality Assurance and Risk Management* be met. Environment Abstractions and Cross-Cloud Model Training Pipelines are used to become independent of the environment. A code generator is used to convert the pipeline definition into specific environments, possible artifacts are Airflow DAG, Jenkins Pipeline, Composer Program or Argo Workflow, etc. This also makes the framework easy to extend and support further environments.

Apache Spark Pipeline Api [MBY+15] is one of the core features of MLlib. MLlib is one of the only ML libraries that provides native support for pipeline construction. The functionalities are described in more detail in Section 5.1. The disadvantage of this is the strong dependency on Apache Spark. It is not designed to use other ML libraries. Therefore, solutions and concepts of workflow management systems, which offer support for a broader range of ML tools (libraries, metadata collection, workflow managements), are investigated.



**Figure 3.3:** ML-based workflow Abstraction.

## 4 Scenarios

In this section, we provide some scenarios that motivate the problem we are going to solve by our concept. They demonstrate the need for a metadata schema and a concept for automatic analysis of stored metadata. Furthermore, the scenarios are later used for the case study and our evaluation.

### 4.1 Scenario 1: Root cause analysis

Alice and Bob are two data scientists who use machine learning techniques to develop a handwriting recognition system. To start they use the MNIST dataset from Lecun et. al [LBBH98] (handwritten digits from zero and nine). They creating three Python scripts, one to download the dataset and transform it, one to train the model, and one to deploy the model. The allocation has therefore been made in such a way that they can work independently, test it independently and run it independently. After the model reaches its desired accuracy, they run the deployment script so that the model can be tested on live data. After a while, Alice discovers an extension of the MNIST dataset, the EMNIST dataset by Cohen et. al [CATV17]. They change the download link parameter in the first script and run all the scripts sequentially. Due to new security requirements, the dataset is now be stored internally. Therefore, Bob downloads and stores the dataset in an internal storage and changes the parameter in the first script. Meanwhile, Alice also optimizes the training script und change some hyperparameters. After the changes have been made, Alice is confused because the deployed model returns digits higher than nine. Unfortunately, Alice thought the cause is the changed training script and not recognized that they used accidentally the letter dataset, an EMNIST dataset with 26 classes instead of 10 like the digits dataset. To sum up, this scenario contains two problems that prevent root cause analysis for ML pipelines that generate faulty ML models. (1) Alice cannot detect the problem because there is no system or tool to store and analyze metadata for ML pipelines that have been executed multiple times. (2) Both do not know which initial triggers the process to run the scripts again.

### 4.2 Scenario 2: Collaboration multiple data scientists

Imagine Bob and Alice are two data scientists who want to predict if a bank account is used for money laundering. They agree to split the task, Bob focuses on the data preparation and Alice on the model training part. Bob gathers data from multiple data sources and sends the enriched data to Alice. Alice creates a python classification script that uses the data as input and true or false as output. After Alice is finished with the fine tuning of the model by trying out several algorithms and hyperparameters, Bob and Alice are impressed by the good results and want to deploy the model to production. They ask Charlie, the IT guy in the company if he can create a web service for the model. He agrees and acquires from Alice. After a while, they detect that the model got stale and

marked to many false positives bank accounts. The following problems occurred when finding the problem and recreating the model. (1) Bob could not remember which dataset he used; (2) Alice and Bob's datasets are not the same; (3) Alice just gave Charlie the best model, but did not store the used hyperparameter.

### **4.3 Scenario 3: Automated workflows**

A company has a recommendation system for their online shop. Due to changing items, pricing and costumers they have a workflow for updating the model used for recommendation. Every step from data collection, to model training, validation and deployment are automated. They defined various triggers to start the workflow; every month, after 100 items are added since the last training or an update of the training script. After a while, their business activity monitoring (BAM) shows that new items are sold less than expected. After some research, they found out that the recommendation system does not recommend the new items. The last workflow run stops after the validation, (1) but they do not know how often the workflow ran since the last deployment, (2) nor they can find out why the workflow was triggered before all runs failed. Problems may be that the number of elements reached a threshold value, for which the model type is not suitable, or the updated training script has an error.

### **4.4 Scenario 4: Metadata Collection**

We imagine a company that trains and deploys several ML models. To save time, they want to reuse parts of previously created ML pipelines. Some steps are customer specific and therefore cannot be reused, but steps such as data transformation, validation, and deployment can be reused. The idea of the company is to reuse not only the mentioned steps but also the outcomes. For example, the same step has the same input and therefore also the same output. That means that the output itself can be reused. They wonder (1) how to check whether steps or several steps are the same and if they are the same (2) is the input the same every time.

### **4.5 Scenario 5: Custom Artifact Schemas**

A university institute is already using the state of art technologies and frameworks to prepare data, train and apply their models. They store a lot of metadata, which they also analyze automatically. They have ML pipelines that both scientists and students can start. Because other experiments and institutes depend on the deployed models, they created a new step that validates the model before it is automatically deployed. For all other steps, the frameworks they use support metadata capture, but not for this specific purpose. (1) They want to create a new metadata schema for the validation step so that they can automatically analyze and visualize this metadata.



## 5 Existing Technologies

This chapter considers existing overall solutions, as well as specialized solutions for specific challenges. The economy has already recognized the potential of ML. Big software companies have implemented their own solutions, which they are also sell as ML as service. In addition, often used solutions have developed from scientific research as well. The chapter is structured in evaluating of existing technologies, commonly used ML libraries (see Section 5.1), sotorage and data preprocessing (see Section 5.2), and deployment services (see Section 5.3). The order of sections is chosen as data scientist would need the technologies to develop a ML model.

In the following, an overview of known complete solutions is presented. These were evaluated under the following basic features and presented in Table 5.1.

1. **Open Source** means that the software code is public available, can be changed and used [LV04]. In additional, it follows the definition of the Open Source Initiative <sup>1</sup>.
2. **Model Storage** describes a feature to store a model in a specific format inside the system or to store the storage location of a model. The model or the model location can be queried at any time.
3. **Metadata Collection** includes the possibility to add metadata to the model and additional artifacts manually or the system creates metadata automatically. The schema of the metadata are for this feature not relevant. To customize or extend the schema are extra features.
4. **Deploy Models** describes the feature to be able to deploy a model to any environments. The service should be accessible and has a defined input and output.
5. **Track Models** means that there is a possibility that a user can track how the model was created. The minimal requirements are to know the used dataset and training script.
6. **Model Lifecycle Support** feature means that the service provides in any kind of steps in the model lifecycle. It has not to support all steps and it can also split the lifecycle in different steps than defined in 2.1.

The feature selection was mostly based on the feature list of the MMP, because the MMP was build to support the complete model lifecycle. As the Table 5.1 shows, the most frameworks are supporting most of the features. Requirements to ensure that the complete lifecycle is supported were identified, considering the existing technologies and described scenarios, and presented in Section 6.1. Unfortunately, these requirements can only be fulfilled by a few frameworks and even then only in part. For example Kubeflow can list all Artifacts of a specific type (see Requirement FR1), but can not identify all artifacts that are related to a given artifact (see Requirement FR5).

---

<sup>1</sup><https://opensource.org/docs/osd>

Technologies	Features					
	Open Source	Model Storage	Metadata Collection	Deploy Models	Track Models	Model lifecycle Support
MLflow [ZCD+18]	●	●	●	●	◐	◐
ModelDB [VM]	●	●	●	○	◐	○
Verta.ai [Var]	◐	●	●	●	◐	◐
ModelOps [HMR+]	◐	●	●	●	●	●
Polyaxon [Pol]	◐	●	●	●	●	●
Kubeflow [Goob]	●	●	◐	●	●	○
MMP	●	●	●	○	◐	●
IBM Watson [IBM]	○	●	●	●	●	●
Azure Machine Learning [Mic]	○	●	●	●	●	●
Google AI Platform [Gooa]	○	●	◐	●	◐	●
Amazon SageMaker [Ama]	○	●	◐	●	○	◐
Michelangelo (Uber) [Ube]	○	●	◐	●	●	●
FBLearner Flow (Facebook) [Fac]	○	●	●	●	●	◐
Bighead (AirBnB) [Air]	○	●	○	●	◐	○

**Table 5.1:** Existing Machine Learning Frameworks Overview

○: not supported, ◐: partly supported, ●: supported

## 5.1 Commonly used Libraries for Machine Learning

The number of libraries is quite large. The common goal is to support data scientist by providing pre-implemented algorithms. Each tool is designed for various purposes and has its own advantages. For the most problems there is no perfect tool, and often only a combination can lead to successful results. The large number often confronts developers of the previously mentioned frameworks with challenges. The consequences are, that only a few support of these libraries is provided.

In the following, some popular ML libraries are briefly described with their basic features [NDB+19]. Due to the continuous development their features may be obsolete.

**TensorFlow**<sup>2</sup>. is an open source ML library for research and production. It supports loading of datasets, building and training of models up to the provision of models. The range of useable environments is huge, TensorFlow can create models for desktop, mobile, web, and cloud. With the extension TensorFlow Extended (TFX), it also supports ML pipelines. The TFX pipeline is a sequence of components, e.g. trainer, evaluator or pusher, which are built upon TFX libraries. For the orchestration, Apache Airflow and Kubeflow can be used. Metadata associated with the ML workflow are recorded with the ML Metadata (MLMD) feature, which is a integral part of TFX, but can also be used independently.

**Caffe**<sup>3</sup>. is a deep learning framework with expression, speed and modularity in mind. it is mainly used for CNNs. It provides modeling and deploying deep learning models with support for a wide variety of architectures.

**Keras**<sup>4</sup>. is an API for neural networks and runs on top of other frameworks like TensorFlow. The main focus of Keras is to enable fast experimentation, due to user friendliness, modularity and extensibility.

**PyTorch**<sup>5</sup>. PyTorch is a deep learning library written in Python, which supports tensor computation with strong GPU acceleration. Caffe2, a library with different design choices and improvements to Caffe in some fields, are now part of PyTorch.

**Chainer**<sup>6</sup>. is a python based powerful, flexible and intuitive deep learning framework. It follows the *Define-by-Run* approach to ensure the flexibility. That means, that the network ist defined and fixed first, and then the user feeds it with training data.

**Scikit-learn**<sup>7</sup>. is a widely known open-source python library. It features various model types (e.g. classification, regression and clustering) and includes algorithms such as Support Vector Machines (SVMs), random forest, gradient boosting, k-means, and many more. Additionally, it also provides a kind of pipeline support. It is a sequentially list of transformers and a final estimator. A transformer is an algorithm which can transform one dataset into another dataset. An Estimator is an algorithm which creates a transformer from a dataset, e.g. a learning algorithm is an estimator that which creates a ML model.

---

<sup>2</sup><https://www.tensorflow.org/>

<sup>3</sup><https://caffe.berkeleyvision.org/>

<sup>4</sup><https://keras.io/>

<sup>5</sup><https://pytorch.org/>

<sup>6</sup><https://chainer.org/>

<sup>7</sup><https://scikit-learn.org/>

**Apache MXNet<sup>8</sup>** is a flexible and efficient library for deep learning. Dependency scheduling, parallel computation, support for symbolic and imperative programming are some outstanding features. It also supports deployment of trained models in different environments, e.g. to IoT devices.

**Apache Spark MLlib<sup>9</sup>** is a scalable ML library that contains common learning algorithms, e.g. naive bayes, gradient-boosted trees, Gaussian mixtures and many more. Moreover, it supports featurization for transformation, feature extraction and dimensionality reduction. Another feature MLlib provides is ML Pipelines. ML Pipelines is a uniform set of high-level APIs that helps to create ML pipelines. The concept is inspired by the concept of pipelines in scikit-learn. A pipeline chains multiple transformers and estimators together.

It becomes clear that not one library dominates the range of ML. There are several different approaches to make it easier for data scientists to create ML models. The library must be carefully selected for each requirement of the model. Therefore, the business workflows and overall frameworks (listed in Table 5.1) should not restrict the choice.

## 5.2 Storage and Data Preprocessing

In the end-to-end ML model training process, the process twice faces the problem of communicating with a storage solution.

The first time is when the training data is loaded. Often data originates from several data sources and needs to be preprocessed. Additionally, the data volume can reach up to terabytes or even petabytes. Data preprocessing includes data cleaning, transformation, normalization, feature extraction and selection. The product is the final training dataset which is either temporarily stored or used directly for the training [KK06]. The challenge is also known as Big Data and has also spawned new computer architectures for real time data intensive processing [CGM13]. Apache Hadoop and Apache Spark [ZCF+10] are projects that allow processing of large data sets across clusters of computers.

The second time is when the ML model has been trained and should be stored for later deployment. Machine learning models are often serialized objects that are saved to disk. These formats are not human-readable and therefore not interpretable. With the exception of PMML, the serialized models are stored in a binary format [GZLW09]. Therefore, these models are categorized under unstructured data. Therefore a suitable data storage must be chosen. Additionally, a versioning is also recommendable. If the storage solution does not support versioning, the aforementioned frameworks often have versioning integrated. DVC<sup>10</sup> is a good example how models, datasets and intermediate files can be versioned. They use git as backbone for storing and versioning of the metadata of the files, not the large files themselves.

---

<sup>8</sup><https://mxnet.apache.org/>

<sup>9</sup><https://spark.apache.org/mllib/>

<sup>10</sup><https://dvc.org/>

## 5.3 Deployment Services

Mostly the last step of a ML workflow is the deploy step. The challenges of ML deployment are the variety of the aforementioned frameworks, the different languages, the performance aspect as well as the scalability. The format how the model is stored, open standards (e.g PMML, PFA, ...) or the framework custom formats (e.g. pickle, pth, ...). Another problem is the friction between the teams, the data scientists want to use the latest model, the operations team wants stability and minimal change and the business team wants the highest business value.

Moreover, the requirements of the deployed model and the complete system are also relevant. Questions like: service or client side calculation, real time predictions, how many request per second, what is the size of one request must be considered. The most frequently chosen approach is ML as a service, the model is deployed on a server and users can query it with their data via an API. The more secure approach is to deploy the model on the client side, e.g MLCapsule [HZG+18], and therefore the data never leaves the local environment. The handle request in real time with large request bodies, the service need to be scalable. The logical consequence is to use cloud technologies and adapt solutions for existing services in the cloud. Seldon <sup>11</sup> is a good example, they use the container technology and deploying their models on Kubernetes as orchestration. Therefore they can use features, like scalability, monitoring, zero downtime updates, A/B testing, security and many more.

Similar principles are also followed by the aforementioned frameworks. For example the four big cloud provider provide model as a service with pre-trained model and deployment of custom models. Derakhshan et. al chose a different approach and deploying the complete pipeline [DRAV19]. They propose a approach called proactive training, which use utilizes sample of historical data. Instead of a longer training time with regular training, proactive training shortens the training time, but can maintain accuracy.

## 5.4 Model Lifecycle Management Frameworks

In the following, some frameworks are explained and evaluated in more detail. The choice fell on Kubeflow 5.4.1, ModelDB 5.4.2, Polyaxon 5.4.3 and MIFlow 5.4.4. Reason for this are they are open source or have at least a open source version, fulfilling the most requirements (see Section 6.1) and are frequently used by the ML community.

### 5.4.1 Kubeflow

The goal of Kubeflow is not to implement another Model Management Platform, but to focus on the deployment of Machine Learning Workflows on Kubernetes (a open-source container orchestration [Ber14]). Thus, models can be deployed on different infrastructures in a simple, portable and scalable way. For complete end-to-end support for ML workflows, Kubeflow can be used with the help of Kubeflow pipelines. Kubeflow pipelines uses Argo as workflow engine. It includes all of the components in the pipeline and how they are combined. Each component is a self contained set of user code that performs exactly one step. Pipelines can easily be defined using a YAML file, as

---

<sup>11</sup><https://www.seldon.io>

in Argo, or a provided sdk [Goob].

Earlier this year they also encountered the problem of metadata collection. Because each step is independent and uses external components, it is difficult to collect metadata and store it in a consistent format. Therefore they have started to implement an extra Kubeflow metadata service<sup>12</sup>. It is based on the aforementioned MLMD from TensorFlow. They are also created a new metadata schema and are working on a metadata management. Their approaches were analysed in detail and incorporated into our concept.

### 5.4.2 ModelDB

ModelDB is a system to manage machine learning experiments. The aim is to support the data scientist in finding trends and characteristics by performing meta-analyses across models [VM]. For a detailed description of the components, we refer to Section 3.2 Management of Workflows in Machine Learning.

### 5.4.3 Polyaxon

Polyaxon also aims to support the entire life cycle of machine learning models. The tool is based on Kubernetes to make the experiments reproducible, scalable, and portable. The main focus is on the steps train and build models. In addition, Polyaxon stores metadata such as accuracy or loss during model training. It also stores hyperparameters and supports hyperparameter optimization. These functionalities work with the most common ML libraries. In addition, the platform offers possibilities to deploy models. With the help of the orchestration, models of any size can be trained and GPU trainings and parallel trainings are also supported [Pol].

### 5.4.4 MIFlow

MLflow is an open source platform to manage the ML lifecycle, including experimentation, reproducibility and deployment. It currently offers three components: Tracking, Projects, Models. Tracking is an API for recording and querying experiment runs, including the code used, parameters, input data, metrics and arbitrary output files. Projects is a simple format for packaging code into reusable projects. Each project is simply a directory with code and a descriptor file. This file specify the dependencies, the parameters that can be used, and the entry point to run the project programmatically. Models is a generic format for packaging ML models. Diverse tools understand the format at different levels of abstractions and are able to deploy the ML model [ZCD+18].

---

<sup>12</sup><https://github.com/kubeflow/metadata>

## 6 Concept

The previous Chapters related work (3) and existing technologies (5) considered and evaluated approaches to support backtracking functionalities of ML pipelines. As a result, the evaluation showed that these approaches offer a good base, but do not meet all requirements to enable comprehensive backtracking functionalities. To cope with these issues, this chapter presents a metadata schema and a backtracking concept for ML pipelines developed in the context of this work.

By using the concept, it should be possible to implement the scenarios introduced in Chapter 4. This chapter is divided into two sections. First, Section 6.1 presents the requirements to be fulfilled by the concept. Second, Section 6.2 presents the conceptual metadata schema. It serves as a basis for storing metadata about runs of ML pipelines.

### 6.1 Requirements

In the following, the requirements were specified to achieve the objective described above. From our motivating scenarios (Chapter 4) and analyze of existing technologies (Chapter 5) we derived multiple requirements for our concept. For better understanding, the terms Artifact and Metadata in this context are briefly explained. Artifacts refer here to datasets, models, model metrics, and other constructs that are created in one pipeline step or are inputs of a pipeline step. Metadata are on the one hand describing data and on the other hand additional information about other data (e.g. artifacts).

#### 6.1.1 Functional Requirements

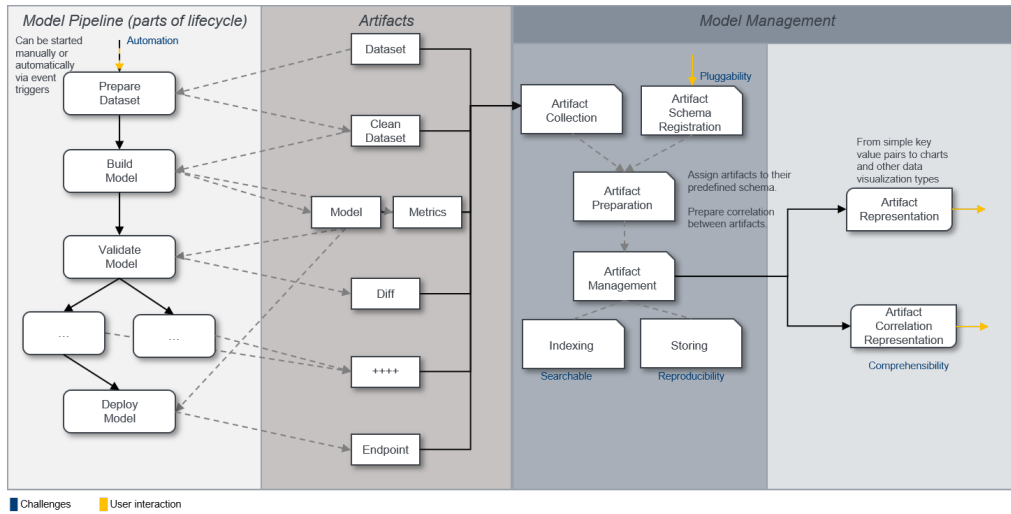
- FR1 List all Artifacts of a specific type, e.g. all Models that have been trained or datasets that were used.
- FR2 Load two Artifacts of the same type for comparison, e.g. to compare results from two experiments.
- FR3 Show a DAG of all executions of a run and their input and output artifacts, e.g. to visualize the pipeline for debugging and discovery.
- FR4 Revert through all events to see how an artifact was created, e.g. to see what data went into a model, or to enforce data retention plans.
- FR5 Identify all artifacts that are related to a given artifact, e.g. to see all Models trained from a specific dataset, to mark models based upon bad data.

- FR6 Determine if a pipeline has been run using the same input artifacts and parameters as in previous runs, e.g. to determine whether a component/step has already completed the same work and the previous output can just be reused.
- FR7 Determine which trigger start the pipeline, e.g. a manual trigger, a periodic trigger or a trigger based on automatically detected changes in data.
- FR8 The collected metadata should be enough to reproduce every artifact, e.g. if a model is accidentally deleted, it should be possible to reproduce it based on the metadata about the executed run of the pipeline. This includes metadata about parameters and input artifacts for each executed step of the pipeline.
- FR9 Metadata should be collected automatically with possible manual interventions, e.g. pipeline runs should be tracked automatically but the user should be able to insert annotations based on his requirements (e.g. client identifier).
- FR10 Show a DAG for all executions in a experiment and their artifacts, e.g. to visualize traces and artifact changes over all runs.
- FR11 Add new artifact types, with custom attributes, e.g. users should be able to create artifacts based on their requirements.

### 6.1.2 Non-functional Requirements

- NFR1 **Pluggability:** Data scientists use many different ML frameworks, libraries and tools because each has its advantages and disadvantages for individual use cases. Therefore, the system should not be restricted to specific technologies.
- NFR2 **Extendability:** The metadata schema should not be restricted to predefined artifact. It should be easily extensible so that users can create their own artifacts for his purpose, they want to keep track on.
- NFR3 **Reusability:** To provide an easy start, connection to existing pipelines should exist and can be reused. They should also find the basic metadata schemas to have a first start point.
- NFR4 **Flexibility:** The system should not only support one use case. Different roles should be able to use the system. AIn addition, the system should support easy integration into existing tools and services.
- NFR5 **Scalability:** The system needs to provide scalability. That is, it should handle parallel pipeline executions and simultaneously store and present metadata to users.
- NFR6 **Hybrid Environments:** The system should be cloud-ready and should be easily useable in private, public or hybrid clouds.





**Figure 6.1:** Concept overview

## 6.2 Overview

In the following, the concept is described. The goal is to cover as much as possible lifecycle steps (Section 2.1). An overview is displayed in Figure 6.1. On the left side an existing ML model pipeline. The pipeline is often managed by existing workflow management systems (Section 3.2). We want to collect and extract metadata from the pipeline, which approximately represents the lifecycle, and the management system to meet all requirements (Section 6.1). To be able to analyze and interpret the metadata we created a metadata schema (Section 6.3). Moreover, the schema is extendable due to a schema registration functionality (Section 6.3). Correlations between metadata are afterward enriched. Possibilities for this would be to communicate with the workflow management system or previously stored data (e.g. pipeline metadata, executable metadata, ...). As just mentioned everything is stored and indexed. Both, the collecting and the output of the metadata is carried out via an API, described in Section 6.4.2. The interface can then also be used to explore and visualize the metadata as well as the relation between them.

## 6.3 Metadata Schema

The discussion of scenarios and related work on metadata schemes showed the necessity for a new metadata schema. In this section, a new metadata schema is presented, which supports the operational phase in the lifecycle of ML models.

For generating the metamodel and prototyping we identified additional approaches of MLFlow, ModelDB, TensorFlow TFX ML Metadata, and Kubeflow Metadata Management.

MLFlow provides tracking as well as experiments, runs and artifact metadata which are well defined. However, the schema for the artifact is too generic and only are output files in any format. The approach to have experiments and runs, which is usually a single execution of a machine learning pipeline is adopted in similarly.

ModelDB metadata schema is too simple to adapt from it. Moreover, it is limited to model and

Name	Captures	Problem to solve
Descriptive Metadata	name, description, annotations	differentiations between, understanding, analytics
Administrative Metadata	version, createTime	preservation
Structural Metadata	connections among executions and artifacts	artifact provenance, lineage of artifacts

**Table 6.1:** Metadata types [Zen04]

dataset metadata. They have no approach for pipelines or executions.

Because Kubeflow Metadata Management is built on top of TensorFlow TFX ML Metadata, only Kubeflow Metadata Management is considered. It should be noted that the project was still under development at the time of writing. The concept of the subdivision of artifacts in models, datasets and metrics would be adopted and enriched.

Schelter et. al metamodel supports also tracing, runs, models and dataset, but missing links to workflows and environments [SBK+17].

The metamodel introduced by Publio et al. [PEŁ+18] was also created with the approach to support traceability between dataset and trained model. The hierarchical structure also meets the requirements. Therefore, parts of this schema were adopted.

Metadata are typically divided into *descriptive*, *structural* and *administrative* metadata [Zen04]. Each type is designed to solve specific problems, which are shown in Table 6.1.

Not only the types have certain functionalities, but in general, metadata have essential functions. They make it possible to find resources according to relevant criteria, to bundle similar resources and to create interdependencies between different subsystems.

The metamodel can be used by any system that needs to track the metadata generated in steps of the update loop of ML models. That is, it spans the deploy, use, and monitor, and rebuild and test model steps.

Although the explicit and comprehensive representation of a metadata schema has been identified as essential, most commercial and research approaches only provide limited solutions. Especially the type *structural* was barely considered. Therefore, we created a new metadata schema, with similarities to existing ones, but with more focus on the *structural* type. The overall concept of the metadata schema is shown in Figure 6.2. The high number of relations between individual schemas is noticeable. Also, a kind of hierarchy is recognizable. In the following, the individual schemas are presented.

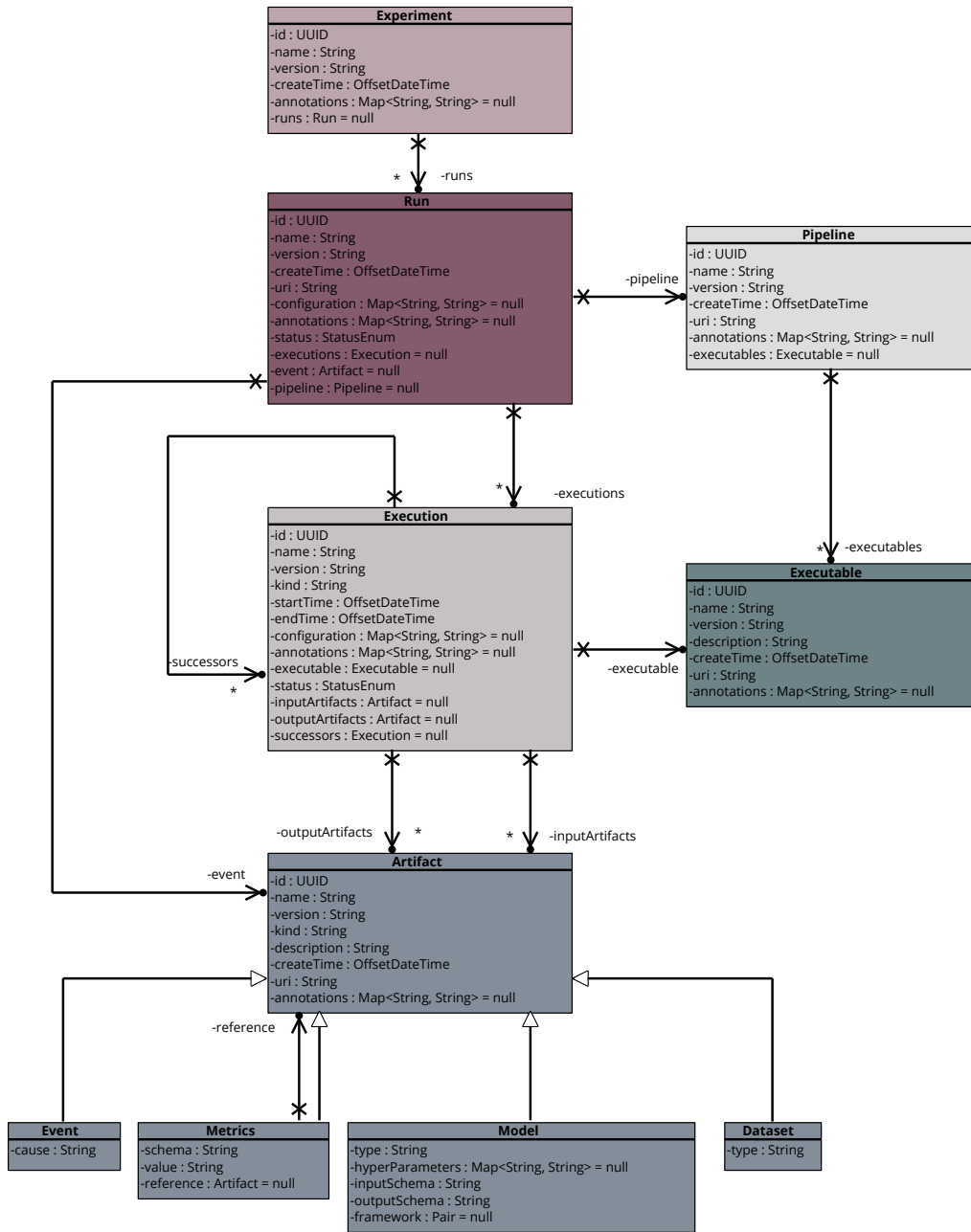


Figure 6.2: Proposed Metadata Schema

## Experiment

*Experiments* are designed to bundle different runs with the same goal or use case. For example, a goal can be to create a ML model which recognizes handwritten letters. Each time the appropriate ML pipeline is executed with different data or hyperparameters, the run is connected to the same experiment. Experiments are good start points for analytics. The mentioned tracing and backtracking feature are designed to analyze artifacts created in runs which a bundled in one experiment. Therefore also runs are separated from each other.

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
createTime	Timestamp	N	time of the object creation
runs	List<Run>	N	all runs created in this experiment
annotations	Map<String, String>	N	custom arbitrarily key value pairs

**Table 6.2:** Metadata Experiment

## Run

*Runs* refer to a sequence of executions. The order of the execution is stored in the execution itself. Often sequences are defined through single pipeline configuration files. Therefore, the run has a reference to a pipeline entity. Moreover, not only single executions have a configurations, but also runs. These are often global configuration ,which apply to all executions, e.g. secrets, running environments or logging levels. A special attribute is the event attribute. It describes how and why the run was started. Examples are manual, time-controlled or event-controlled.

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
createTime	Timestamp	Y	time of the object creation
uri	String	N	uniform resource identifier (e.g. workflow framework run visualization)
status	String	Y	the status of the run, which could be success/failed/running
executions	List<Execution	N	all executions executed in this run
configuration	Map<String, String>	N	the configuration of this run
pipeline	Pipeline	N	which pipeline is used
event	Artifact	N	the event which started the run. Can be every artifact, but recommended is the event artifact
annotations	Map<String, String>	N	custom arbitrarily key value pairs

Table 6.3: Metadata Run

### Pipeline

A *Pipeline* is just a wrapper entity for workflow frameworks (see Section 3.2). Because sequences of executions are often described by configuration files, we can create a connection to them. This ensures the derivation of provenance information.

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
createTime	Timestamp	Y	time of the object creation
uri	String	N	uniform resource identifier (e.g. storage location)
executables	List<Executable	N	
annotations	Map<String, String>	N	custom arbitrarily key value pairs

Table 6.4: Metadata Pipeline

### Executable

An *executable* contains the metadata for one step in a pipeline and can be executed multiple times in different runs. Typical executables are scripts, functions or containers. For example, there could be one executable to preprocess the data, one to train the model, and one to deploy the model to its target environment.

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
createTime	Timestamp	Y	time of the object creation
uri	String	N	uniform resource identifier (e.g. git link or docker image reference)
description	String	N	describes what the executable does
annotations	Map<String, String>	N	custom arbitrarily key value pairs

Table 6.5: Metadata Executable

### Execution

*Executions* are the unique execution of one executable. They represent exactly one step in a pipeline. They are started either after another execution or it is the first step in the pipeline. The attribute successors described here the executions which are started after the current execution is finished. The most important attributes in this entity are the input and output artifacts. These are elementary for the backtracking features. Provenance questions such as how the model was created and which dataset was used can be answered with this information.

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
kind	String	Y	kind of execution (e.g. transformation, train, validation, ...)
uri	String	N	uniform resource identifier (e.g. workflow framework pipeline step reference)
status	String	Y	success/failed/running
startTime	Timestamp	N	start of the execution
endTime	Timestamp	N	end of the execution
executable	Executable	Y	reference to the used executable
inputArtifacts	List<Artifact>	N	used artifact
outputArtifacts	List<Artifact>	N	created artifact
successors	List<Execution>	N	list of ensuing executions
configuration	Map<String, String>	N	explicit configuration for this execution
annotations	Map<String, String>	N	custom arbitrarily key value pairs

Table 6.6: Metadata Execution

## Artifact

*Artifacts* are the most imported entity and describe the outputs of a run. Each run should produce at least one artifact. Each step of the run (execution) produces and consumes artifacts, which are also used by other systems that are not part of the pipeline. Besides connections to runs and experiments, artifacts have also relations between each other. For example, a model artifact often relates to a dataset artifact, because it is trained by using this specific dataset. These relations can be derived by examining information about executions and runs. The artifact entity is a parent entity for the following entities, dataset, model, metrics, and event. These were selected because they are commonly used across different ML tasks. The extendability of the metadata schema is guaranteed by simply creating new artifact types. The four entities are defined due to the common use, but every user can create his or her own artifact types. Possible extensions could be, validation (the result of a cross-validation method) or deployment (describes the running environment and location).

Name	Type	Required	Comment
id	String	Y	a unique identifier
name	String	Y	a custom short descriptive name
version	String	Y	a version, e.g. v1 or 1.0
createTime	Timestamp	Y	time of the object creation
kind	String	Y	dataset, model, metrics, event and all custom extensions
description	String	N	additional information about the artifact
uri	String	N	uniform resource identifier (e.g. model storage location)
annotations	Map<String, String>	N	custom arbitrarily key value pairs

**Table 6.7:** Metadata Artifact

The metadata for a dataset is captured by the dataset entity. The URI attribute from the parent entity artifact is used as a pointer to the actual storage location, e.g. a distributed file system such as HDFS. Further extension and differentiation possibilities result from the different types of data records. For example, when using a data source that supports SQL the SQL query string could be a good attribute to store. To store the SQL query string a new specific artifact entity, e.g. SQL-Dataset, can be created or the annotations attribute can be used.

Name	Type	Required	Comment
All attributes from artifact			
kind	String	Y	dataset
type	String	N	the type of the storage (e.g. Blob, SQL, HDFS, ...)

**Table 6.8:** Metadata Dataset

A common goal of ML pipelines is to train a model. Therefore, the creation of a model artifact is a logical consequence. The model entity describes a trained model. However, this model does not need to be stored in a model file. If a model is not meant to be used because of insufficient quality,

it is often not worth it to keep a model file. Nonetheless, all metadata for its creation should at least be stored. The hyper-parameter and framework attribute can be used to compare models. Storing the input and output schema is also required because models are not self-explanatory when using them in ML serving systems.

Name	Type	Required	Comment
All attributes from artifact			
kind	String	Y	model
type	String	N	e.g. Linear Regression, SVM, CNN, ...
framework	Pair<String, String>	N	used framework and the used version (e.g. <TensorFlow, 1.0>, <Apache Spark, 2.3>, ...)
hyperParameters	Map<String, String>	N	the hyper parameters for the model
inputSchema	String	N	a description of what the model expects as input
outputSchema	String	N	a description of what the model returns as result

**Table 6.9:** Metadata Model

The metrics entity is a kind of extension of the artifact itself. It should have a reference to an existing artifact and can, for example, describe the accuracy of a model or the quality of a dataset.

Name	Type	Required	Comment
All attributes from artifact			
kind	String	Y	metrics
reference	Artifact	N	e.g. model or any other artifact
schema	String	N	describe how to interpret the value attribute (e.g. number,vector, ...)
value	String	N	the value of the metric

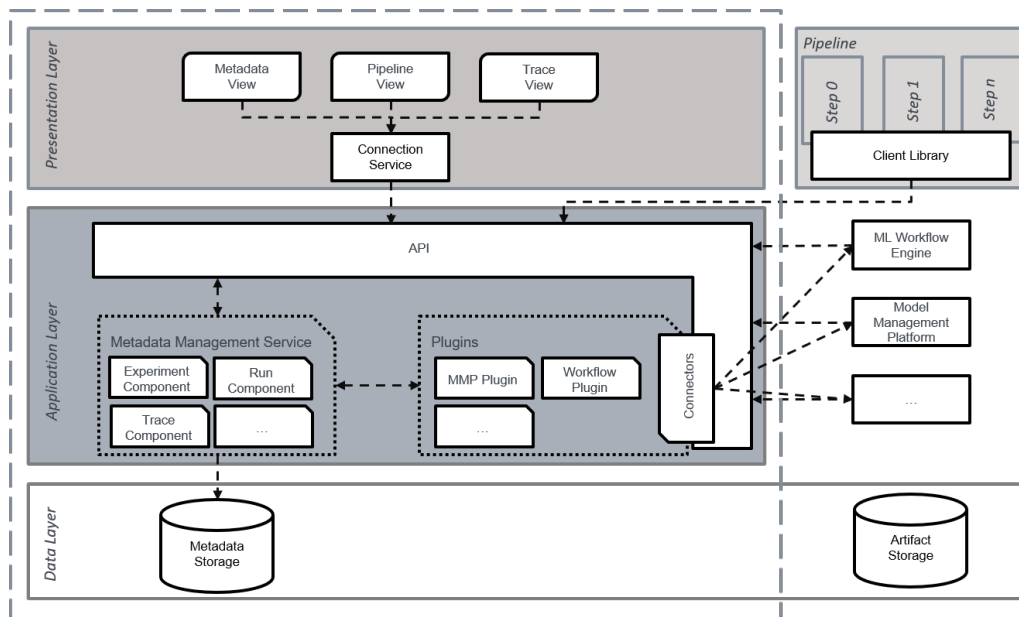
**Table 6.10:** Metadata Metrics

The event entity is a special artifact. The metadata for events contains information about how and why the pipeline was started. There is only a single event entity in each run and it is the input artifact of the first execution.

Name	Type	Required	Comment
All attributes from artifact			
kind	String	Y	event
cause	String	N	causes could be data shift detected or time dependent

**Table 6.11:** Metadata Event





**Figure 6.3:** Architecture Overview

### Customized Artifact Metadata Definitions

To increase the generality of the schema, the concept is designed extensible to create custom artifacts. The annotation attribute can be used, but for validation and analytics, it is not an optimal solution. Users should be able to create new artifacts based on their requirements. Artifacts can then be handled based on the *kind* attribute. Both the implementations and the storage solutions should not make differences between predefined and user-defined artifacts.

## 6.4 Architecture

Figure 6.3 shows the proposed architecture. The architecture is based on the three-tier-architecture pattern [BHS07]. It consists of a presentation, application and data tier. The presentation layer is responsible for the visualization of the metadata and their relations. The visualization is described in more detail in Section 7.2.3. This layer uses a developed client library to communicate with the application layer. The client libraries are described in Section 7.2.2. The application layer contains the logic and is also known as the logic tier. The core functionalities are to provide the API, build the links between artifacts, enrich the metadata, and interact with existing systems (see Section 6.4.2). Each incoming request goes through one metadata management service and stores the metadata in the metadata storage. Moreover, the component is responsible for calling the enabled plugins. The plugin component should be emphasized, it is responsible for the integration with existing model management platforms and workflow management systems. It will be described in more detail in Section 7.3. The data layer includes the data persistence mechanisms and an access layer which allows them to store and retrieve metadata objects (see Section 6.4.3). The artifact

storage is no storage provided by the system but is used to store the artifacts such as serialized models and datasets. Direct connection to the artifact storage is not necessary, the location and connection parameters are stored inside the metadata.

### 6.4.1 Client Libraries

Client libraries for the most common ML language Python should be provided. Pipeline steps or framework components (e.g. hyperparameter tuning framework to store and read results) can use the libraries to send and read metadata from the server. Additionally, an Angular library should be provided to obtain the metadata for visualization purposes.

### 6.4.2 API Server

The communication between client and server should be scalable, flexible, portable, simple, independent, and follow the web standard of the Hypertext Transfer Protocol (HTTP). Moreover, the server API should provide a CRUD interface for creating, reading, updating and deleting metadata. The API interface can be provided by different architectural styles, e.g. REST, SOAP or GraphQL<sup>1</sup>. GraphQL provides a query language to specify which fields should be included in the payload. Unfortunately, the complexity for client and server side is high and therefore GraphQL is not optimal. The disadvantages of SOAP compared to REST are that the components are tightly coupled. Clients need to know the operations and their semantics beforehand, and that the data format is restricted to XML [ZNS05]. Due to the ability to add new custom schemas and the associated consequence of using JSON as the data format, the server should provide a REST API.

### 6.4.3 Storage

The two main requirements for the storage are that the data schemes can be added and changed, and the relation between entities should be considered. For the first requirement, a NoSQL database would be suitable, while for the second a relational SQL database would be suitable. This indicates that a hybrid approach would be a perfect fit. MongoDB is such a hybrid database that supports relational and non-relation concepts [HHL11]. It supports SQL features, like ACID transactions, joins in queries, and reference and embedded entity relations. The powerful query language supports filtering and sorting within a document, as well as aggregations. The most important advantage is the storage of JSON documents, which allows flexible and dynamic schemas.

---

<sup>1</sup><https://graphql.org/>

## 7 Implementation

This chapter introduces a prototypical implementation. It implements all the concepts defined in the previous chapter. The prototypical realization includes the creation of a software prototype in the form of a vertical prototype [BKKZ92]. That means that selected parts of the system are implemented completely across all layers of a three-tier-architecture. The goal is to implement the functional requirements mentioned in Section 6.1 in order to assess the concept in Chapter 8. Furthermore, integrations with existing model management platforms and workflow management systems are prototypically implemented.

The chapter is divided into 4 sections. Section 6.4 deals with the design considerations that led to the selection of specific technologies for the prototype. Section 7.1 introduces the architecture that realizes the system's components. The individual implemented components are discussed in Section 7.2. The last section, Section 7.3, discusses the integration of the backtracking system with existing systems used for machine learning.

### 7.1 Infrastructure

Due to the non functional requirements *flexibility*, *scalability*, and *hybrid environments* (see Section 6.1.2) every service must be run stateless and containerized. The state of the art technology for containerization is Docker [TRA15]. Docker uses containers to isolate code and its dependencies from compute environments. It solves the problems of portability and consistency between environments. By using Docker, we can scale all our layers (presentation, application, and data) and run their components in different environments. The scaling of the data layer is slightly more complicated than the other two because a sharded database must be used [DG92]. Because we use Argo as a workflow framework, a container native workflow engine for orchestrating jobs on Kubernetes, and because features like automating deployment, scaling, and management of containers are required, we chose Kubernetes as our container orchestration. Kubernetes is available from all big cloud providers and can also be installed on bare metal servers. Connections between the different layers and later the integration, Kubernetes provides a DNS service. For the definition of the Kubernetes resources, we use Helm<sup>1</sup>. With Helm was it also easy to use Istio<sup>2</sup>. Istio is a service mesh and we use it for traffic management (e.g. request routing, ingress, ...) and metrics (collecting and visualizing).

---

<sup>1</sup><https://helm.sh/>

<sup>2</sup><https://istio.io/>

## 7.2 Components

In this section we will describe the components in more detail. Starting with the backend (metadata management 7.2.1), followed by the client libraries 7.2.2 and the graphics user interface 7.2.3.

### 7.2.1 Metadata Management

The metadata management component is located on the application layer. It provides the API and is connected to the storage. Moreover, it also manages the metadata schema. Its main task is to process the incoming metadata and establish their relationships. Furthermore, missing metadata and attributes are added using the plugin component, such as executables and creation date. The plugin component is located within the metadata management component and is used for integration with other frameworks. (see Section 7.3). The API follows the REST specification as defined by the concept. For the design of the API, we followed the design rulebook from Mark Masse [Mas11]. The example POST interface 7.1 is used to add a new execution to an existing run. The payload is an execution as JSON format and if the execution is successfully added, the server returns the response status code 201 (created). The second example 7.2 can be called to get all executions from the specified run. The response body is a list of executions in JSON format with status code 200 (ok). The last example 7.3 is an update interface. It updates the specified execution, declared in the path (executionId), and the updated execution is in the payload.

*POST/experiments/runs/{runId}/executions* (7.1)

*GET/experiments/runs/{runId}/executions* (7.2)

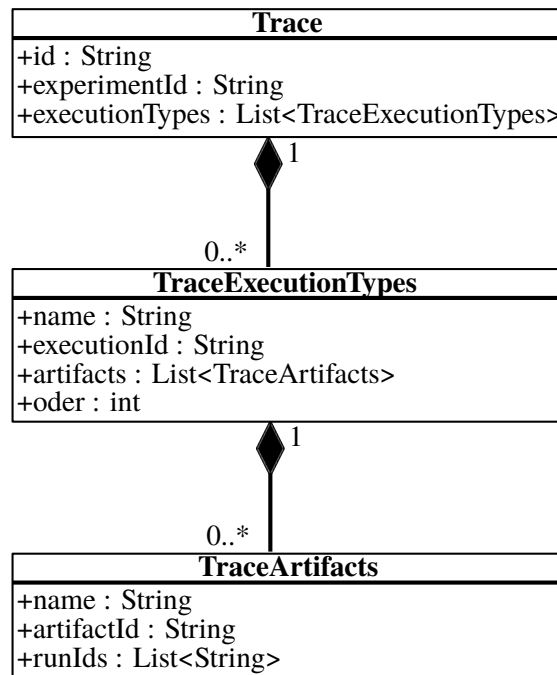
*PUT/experiments/runs/executions/{executionId}* (7.3)

To define the complete API and the metadata schema we used the OpenAPI specification (see Appendix A) [Ini+18]. The OpenAPI Initiative is a vendor neutral, portable and open specification for providing metadata via REST Web APIs [EIC17]. Two additional functions, besides the basic CRUD functions, are also integrated. The first function is a trace function (requirement FR10). The metadata component can be queried to trace all runs of an experiment by specifying the corresponding experiment id. As shown in Figure 7.1 the trace entity contains a list of execution types (e.g. transform, train, ...). Each execution type contains a list of artifacts (e.g. a dataset, a model, ...). Equal artifacts are merged into a trace artifact. Currently, artifacts are equal, if all attributes except the creation date are the same. Therefore, a trace artifact can have multiple run ids. The run ids within the artifacts can be used to track artifacts throughout their entire lifecycle.

The second function is the workflow function (requirement FR3). For each run, the metadata management component can return a workflow as a DAG. The node ids represent the execution ids and can then be used to load the corresponding artifacts. A visualization approach is shown in Section 7.2.3.

### 7.2.2 Client Libraries

Client libraries in Python and TypeScript are provided. These libraries serve several purposes. They can be used from external services (e.g. Argo or Kubeflow) to send metadata to the API. The libraries can also be used in the pipeline steps to receive and send data to the server. An example of



**Figure 7.1:** Trace Structure

how to create a model artifact and send it to the metadata management service via Python is shown in Listing 7.1. Furthermore, metadata can be queried for visualizations, such as our implemented user interface in Section 7.2.3. The TypeScript code 7.2 shows an example service to query executions filtered by an explicit run. As mentioned before that we used OpenAPI as standard to describe our REST API, therefore the libraries can be generated with the OpenAPI Generator<sup>3</sup> automatically. This also allows us to generate additional client libraries with little effort (e.g. java, go, c, ...). In addition, if the metadata schema changes, the libraries are also updated and support the new schemas with the newer version. Each entity is validated against the schema to reduce unnecessary traffic and failure risks. We are currently using the libraries in our user interface implementation and in every step of our created example pipeline.

### 7.2.3 Graphical User Interface

For the visualization of the metadata the pipeline trace, we created a graphical user interface. To access the data, we used the created TypeScript library (see Section 7.2.2) to query the metadata component. As shown in Figure 7.2, we used basic tables to display the metadata of experiments, runs, executions and artifacts. Further details are shown after selecting a row. Furthermore, the selected row serves as a filter for the next table as well. As described in Section 7.2.1, a structured pipeline can be queried via the metadata management component. This data is used to visualize each step and its connections to other steps. An example pipeline is shown in Figure 8.1. Each node represents one execution with its output artifacts. Details about the artifacts and the executions are

<sup>3</sup><https://openapi-generator.tech/>

## 7 Implementation

---

---

### Listing 7.1 Example usage of the Python client library.

---

```
from mmp_metadata_client import *

api = ApiClient(configuration=Configuration(host="http://192.168.221.29/v1/metadata"))
artifactApi = ArtifactApi(api)

model = Model(name="Model",
              version="1.0",
              kind="model",
              framework=Pair("Tensorflow", "v1"),
              hyper_parameters={'learning-rate': str(0.001),
                                'training-iters': str(20000),
                                'batch-size': str(128)},
              uri='http://192.168.221.29/minio/model-output/' + 'modelExample.pmml')

artifactApi.add_artifact(execution=execution.id, artifact=model)
```

---

---

### Listing 7.2 Example usage of the TypeScript client library.

---

```
import { Execution, ExecutionService } from 'mmp-metadata';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

export class MmpMetadataService {

  private executionService: ExecutionService;

  constructor(private http: HttpClient) {
    this.executionService =
      new ExecutionService(http, 'http://192.168.221.29/v1/metadata', null);
  }

  getExecutionsByRunId(runId: string): Observable<Array<Execution>> {
    return this.executionService.getExecutionsByRun(runId);
  }
}
```






---

displayed on click. A more complex trace visualization is shown in Figure 8.2. As described in Section 7.2.1 it is possible to query the trace over all artifacts in different runs of one experiment. In contrast to the pipeline visualization, the time progression is displayed horizontally. The reason for this is that the focus is more on the connection between artifacts than on the executions. Therefore, the lines represent dependencies between artifacts. Lines connect different artifacts that are produced in the same run. Detailed views of the artifact and the executions can be displayed as in the pipeline visualization. Another feature is the highlighting of artifacts that depend on the selected artifact.

Metadata

Experiments Runs Executions Artifacts

Runs

#id	Name	Version	Amount of Executions	Create Time ↕	Status	Pipeline
35cf8171-568a-47b3-8858-6478e691424a	First run in experiment	1.0	3	2019-09-25T15:39:01.999Z	SUCCESS	
1149d454-9e56-448c-9681-fad8fed0c140	Second run in experiment	1.0	3	2019-09-25T19:04:01.199Z	SUCCESS	
7ea04990-7a37-41c0-9870-2fba28855aca	Run with Validation Check	1.0	5	2019-09-25T20:56:30.456Z	SUCCESS	
d4abef5b-39fb-4868-9213-41cc3a8bf101	Run with new Dataset	1.0	4	2019-09-26T15:24:58.752Z	SUCCESS	
dde0468b-0de0-4118-a88b-0fbb84361d58	Run with optimal Parameters	1.0	5	2019-09-26T16:45:03.243Z	SUCCESS	

**Figure 7.2:** Metadata are listed in a table for an overview.

## 7.3 Integration

This section describes the exemplary integration of our created prototype with the workflow management system Argo and the MMP. For both integrations, we use the plugin component as mentioned in Chapter 6. For each created or updated metadata, the suitable function was called. For example, the implemented Argo plugin adds to each run automatically the corresponding pipeline. In addition, executables are also automatically created or linked to each execution. A powerful feature of Argo is the event-based dependency manager. It helps to define multiple triggers that start a specified pipeline, e.g. webhooks, data changes, schedules, and others. Our integration is able to extract the exact event and save it along with other metadata of the run. With these features, users do not have to manually create all metadata and linkages, instead, most metadata can be automatically extracted and stored. Another example implementation is the integration to the existing MMP (see Section 2.2). Once a new model artifact is created and the pipeline is successful, the plugin creates a new model in the MMP.

Further integrations are easy to implement and make sense if no adjustments can be made to existing systems. This means that if the system is modifiable (e.g. open source tool or in-house system), it is often easier to transfer the data to the metadata management system than to create a plugin that fetches the required data from the system.





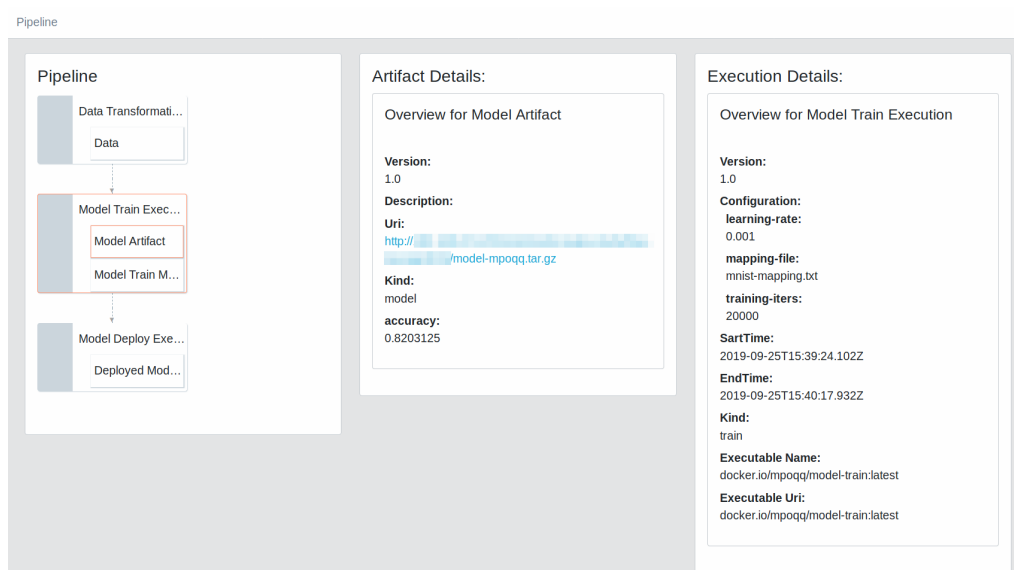
## 8 Case Study

In this chapter we evaluated our prototype and concept based on a case study, the non functional requirements 6.1.2 and functional requirements 6.1.1. The case study is based on scenario 1 (see Section 4). All other scenarios will also be considered in the case study and will also be briefly discussed afterward.

### 8.1 Root Cause Analysis

As in Scenario 1, we created a handwriting recognition model in the case study. We first use the MNIST dataset and build a pipeline with three steps. The first step would be the data preparation step, which is used to clean and transform the MNIST dataset in its raw format. Since the data set is already cleaned and transformed, this step is discarded. Instead, we check whether the data set is accessible. In the second step we, use Tensorflow to train a CNN on a sample of the MNIST dataset. We serialize the model and store it in an object storage. In the last step, we deploy the model to its target environment. For the target environment, we use Seldon as a data scoring engine which instantiates the model as a web service for online prediction (see Section 5.3). The web service is then ready to consume handwritten digits and predict their digital value. In order to execute the steps sequentially, we use Argo Workflows as a container-native workflow engine (see Section 3.2). Input parameters are used to specify the learning rate, training iterations, and model storage location. The pipeline can be viewed via the graphical user interface (see Section 7.2.3). Figure 8.1 shows the created pipeline. Each step can be visualized, as well as their sequential execution and dependencies. In addition, each step shows the artifacts that result after executing that step. For a more detailed view, users can select the model artifact and the corresponding training step. Important information, such as the storage location of the model and the configuration of the execution can be displayed. This means that it is now easy to trace with which configuration the model stored in the specific location and id was trained.

Next, we show how our system supports root cause analysis when a pipeline generates an undesired model output (see Section 4.1). Therefore, we change the pipeline parameter to use the EMNIST instead of the MNIST dataset. Especially when the execution of pipelines is automatically triggered through events such as daily cron-jobs or change detectors for data, a pipeline trace can be very useful for root cause analysis. However, before that, we also created an Argo Event which triggers automatically the pipeline if we change something in the storage where the datasets are stored. After that, we store the EMNIST dataset in the storage and the pipeline starts automatically. As shown in the pipeline trace visualization in Figure 8.2, two models were created in this experiment. Both are based on different data sources, one on the MNIST dataset and the currently deployed one on the EMNIST dataset. To further support root cause analysis, the question of how and why the pipeline was started can be answered by the shown events, which are also tracked by the system. The first



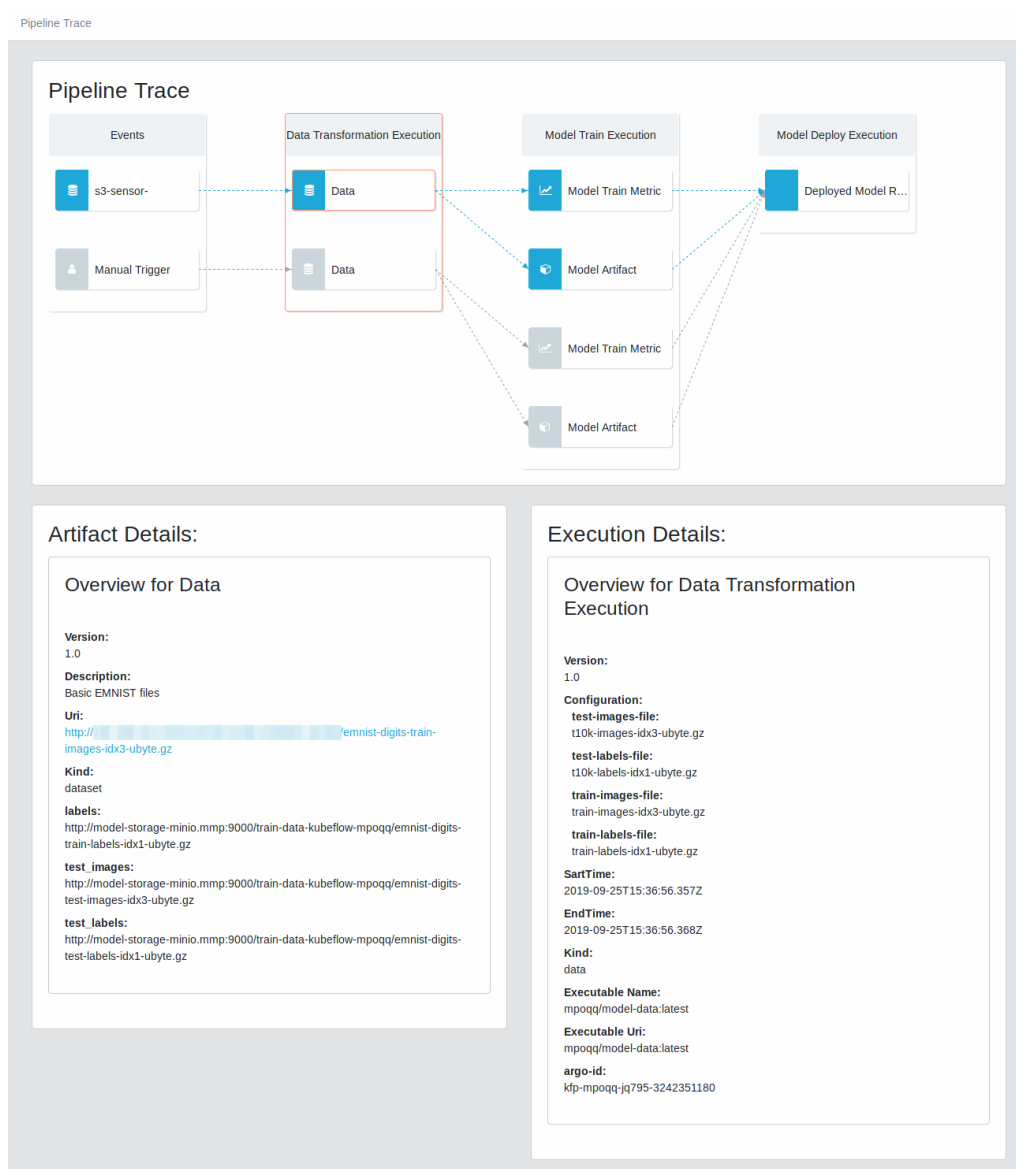
**Figure 8.1:** Sample pipeline comprising three steps: 1) preprocessing the MNIST dataset [LBBH98], 2) training the model, and 3) deploying the model. Details of the generated steps and artifacts are displayed on the right.

run was triggered manually and the second was triggered by a data change detector attached to the object storage.

By comparing the two metric artifacts, we found out that the old model had better accuracy. To prevent that newly created models with a lower accuracy are deployed, we created a new step that validates the model and it is only deployed if the model has better accuracy than the one currently use. With similar intent as Scenario 4.5, we create a new artifact called validation, extend the metadata schema, and register it with the metadata management component. On the one hand, it contains the result of the validation, valid or failed, and on the other hand a history of all previous accuracies. Because training times of CNNs are often high as shown in the Stanford DAWN project [CNK+17], we create a step that notifies users when the execution of the pipeline finishes and the result of the validation. Because we now have an extended schema, we can analyze the artifact more easily. Therefore our prototype contains a view that provides details about the kind of artifact (see Figure 8.3). Additional to the artifact information it shows the result and the history. All previous model accuracies are represented in a line diagram so that increases and decreases are better recognizable over time. For example, Figure 8.3 shows that our most recently trained model has an accuracy of about 0.914, resulting in an increase of about 0.152 over the previous model. Next, we check if our model has been deployed correctly. Figure 8.4 shows a basic request to our model that is instantiated as a web service on Seldon. As an example of data to be predicted, we send the digit 5 to the web service.

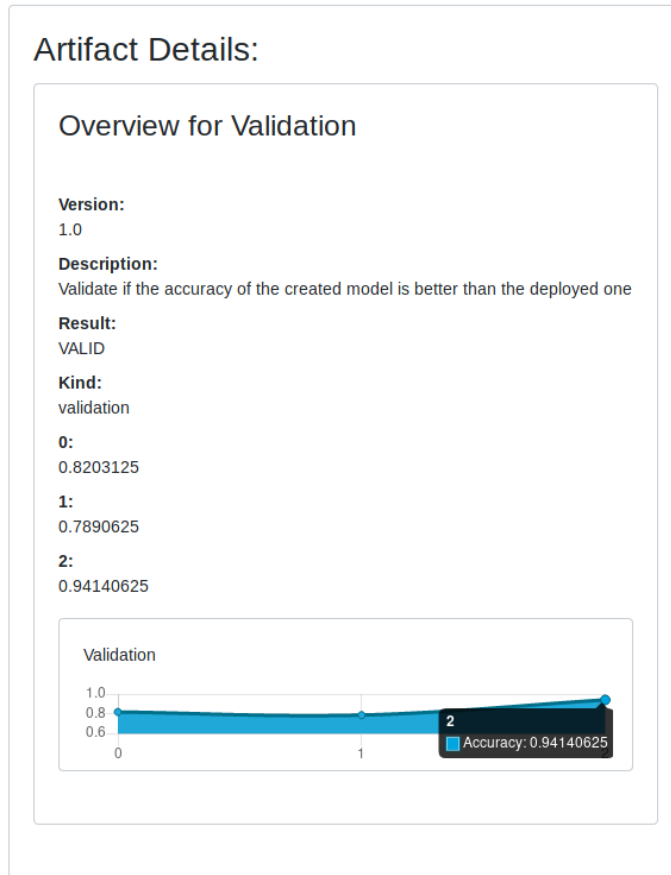
To achieve a better model performance, the training script was constantly improved. For the pipelines to use the improved script, we have created an event that triggers the pipeline when a script is committed to our version control system. This is shown in Figure 8.5 on the left side below the events.

After making changes to the ML model training script, unusual predictions have been detected. We observe that when an image of a digit was sent for prediction, the prediction is mostly higher than nine. To detect the root cause, we open the pipeline trace view and select the model currently in use.

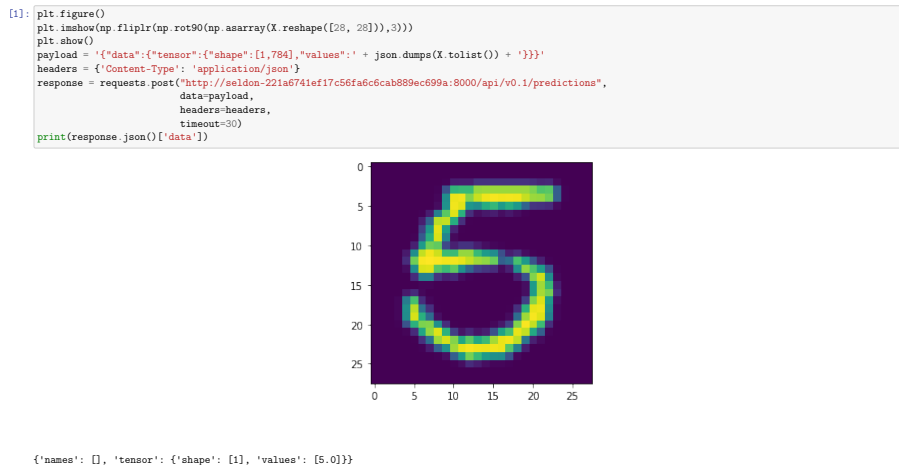


**Figure 8.2:** Pipeline trace of an CNN model

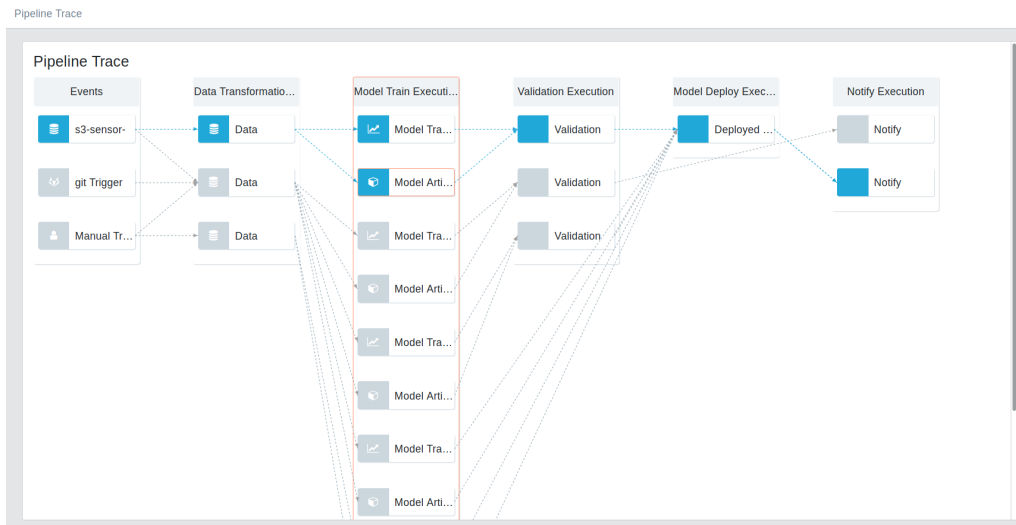
As shown in Figure 8.5, we can detect which artifacts were created in the same run. That makes it very simple to notice that a different data source was used than in the previous runs. With the help of the detail view of the dataset, the difference between the used dataset could be determined fast. We can now identify that the EMNIST letters dataset was used to train the CNN model. This allows us to explain the false prediction, which in this case represents the index of the letter in the alphabet instead of the real digit.



**Figure 8.3:** A detail view of the validation artifact



**Figure 8.4:** An example predict call



**Figure 8.5:** Pipeline trace visualization to detect the root cause of an incorrect model

## 8.2 Discussion of further Cases and Requirements

Section 4.2 describes the scenarios of collaboration between data scientists and the problem of being up to date. Collaboration can be enhanced by our concept because every step and every artifact is stored and retrievable. The metadata schema is conceptualized in such a way that metadata for an experiment is easily accessible. For example, the API has an endpoint to query all artifacts in an experiment. Due to the artifact attributes version and creation time, artifacts can be filtered and sorted to always work with the latest resources. The GUI currently implements sorting of the create time attribute, but no filtering. For some use cases, it would also be good to have the ability to sort and filter via the API. Another problem mentioned in the scenario is the lack of documentation about which dataset was used and with which hyperparameter the model was trained. If our metadata management system would have been used, the hyperparameters and the employed dataset could be traced and queried.

Section 4.3 describes the use of an automated workflow and the problem of traceability and documentation. If the company would have used our prototype, they could have been able to trace the event that led to the last deployed model and how often this pipeline has been started since then. By visualizing the execution in the trace visualization (see Figure 8.5) without differentiating between the executables, it is difficult to determine whether a change in the scripts is the cause. An improvement could be to add a visualization of the executables within the executions that shows which executable was used to create the artifact. To detect such behavior at an early stage, one can use the plugin component can be used to implement an alert system.

Section 4.4 describes the problem of reusing pipeline steps and their outcomes. The trace visualization (see Figure 8.5) is a good starting point since identical artifacts are visualized as a single artifact. That is, if a step always creates the same artifact, it might be useful to delete the step in the script file and use the created artifact instead to reduce computational costs. To compare artifacts, executions or executables across experiments, it is not yet implemented, but the concept and the metadata schema offer the possibilities.

In Section 4.5 data scientists want to create a validation step similar to the one we created in our

case study. The created metadata schema can be used to automatically analyze and determine if the model is valid. An example of this is provided by the visualization shown in Figure 8.3. Further user-defined artifacts such as a deployment artifact or an extended model artifact are conceivable. Within the scope of the case study described, we covered the most functions of our current prototype of the metadata management system and showed how users can benefit from it. Those not considered are briefly evaluated in the following.

To address FR8, one must be able to reproduce every artifact based on the stored metadata. The metadata contains information about the order of the executions, the configuration for each step and the executable. This information is enough to reproduce every execution and therefore also every artifact.

NFR3 requests reusability and our case study shows only one scenario with selected tools, but due to the extensible schema and the plugin component, other tools can also be used.

NFR4 requests flexibility and the aforementioned scenarios are a good indication of this. We mostly consider the use case scenarios from the perspective of a data scientist, but other roles like software developers or quality assurance can use the metadata as well. For example, a software developer can query the URL of a deployed ML model. They can also implement new plugins that communicate with their firmware as we did with the MMP.

NFR5 requests scalability. This is ensured because the metadata management service is stateless. Therefore, the service is not only vertically scalable, but also horizontally scalable. As already mentioned in the concept (see Chapter 6), distributed databases can be used to scale the storage.

NFR6 requests hybrid environments. Our case study was conducted in a private cloud environment. The implementation section, Section 7.1, describes the technologies used to make it possible to achieve independence from different environments. We also deployed and ran a pipeline within a managed Kubernetes cluster, to test our system in fulfilling the requirement. Advantages such as auto-scaling can be more easily used in the managed environment. If more pipelines are running in parallel, more nodes can be added to the cluster. Besides, various pipeline steps can be executed on predefined nodes, e.g. some ML algorithms require more GPU power and thus the training step can be executed explicitly on a node with more GPU power.

## 9 Conclusion and Outlook

We propose a concept and a prototype of a ML metadata schema that support the comprehensive ML model lifecycle. The central goal of the thesis was the development of the concept, which enables functions such as backtracking from the deployed model to the used dataset. It distinguishes itself from existing research and solutions by providing a metadata schema for the most common ML pipeline steps and the possibilities to extend the schema for additional use cases. The goal was achieved by firstly researching and creating a knowledge base. Secondly, setting up exemplary ML model building use cases. Then the concept was developed based on the extracted requirements from research and existing solutions. Lastly, the concept was implemented and afterward evaluated based on a case study.

The established set of scenarios are root cause analysis, data scientist collaboration, automated workflows, metadata collection, and custom artifact schemas use cases. Each of them shows the necessity of our concept and our solution. They always form a part of the ML model lifecycle, *plan model*, *build & test model*, *deploy model*, *use & monitor model*, and *retire model*. These steps were manually or automated iterative executed and had the same objective to create a ML model. The steps are often described in pipelines, which can then be interpreted and automatically executed. In the process, they all run into problems, which could have been avoided with our solution. Existing technologies trying to solve the problems as well and there are approaches currently under development which aggregate more metadata out of the ML model lifecycle. These approaches are mostly focused on specific technologies and are strongly coupled. This makes it difficult to expand on the one hand and inflexible on the other.

The developed concept includes a metadata schema, the required components and the architecture of the components. The predefined functional and no-functional requirements are the base of the concept. They describe all the functions that can be used to solve the problems that have occurred in the scenarios. The non-functional ones describe the expandability to other solutions as well as the scalability, pluggability, flexibility, reusability, and hybrid environments. The main part of the concept is the metadata schema. The different object descriptions cover meta information and the relations between them. On the one hand, this information can be used to store and query valuable information and on the other hand, relations can be automatically analyzed. This allows the prototype to implemented backtracking. The concept includes also an approach on how to interact with existing tools and frameworks. It supports the pull and the push principal to collect data. Single steps in a pipeline can send data directly to the system or the system can query data from the used technology.

The implemented prototype and the concept is evaluated with a case study. For this, Scenario 1 was used. We created an example model development pipeline and send the metadata, e.g. database and model, to our prototype. The prototype enriched the data with an implemented plugin, which interacts with the used pipeline framework. With the created GUI the metadata could be displayed

accordingly. With the help of metadata and their representation, it was possible to find out why the model created did not work as desired. The other presented scenarios were also shortly discussed, as well as the requirements which were not presented directly.

**Limitations.** A limitation of our approach is that we are dependent about the data we get and the data we can extract from provided APIs. If the pipeline is running in a closed system our approach is unable to receive any data. Another limitation is that our prototype is currently not able to validate newly created artifacts without downtime. The prototype must be recompiled to be able to validate new artifacts with all the new attributes. Additionally, we are also not yet able to handle different versions of metadata schemas. The storage is able to handle different versions, but the logic component and the plugin component cannot handle different versions.

The GUI also has its limitations. The representation of the metadata in the table should be improved. As soon as the amount of data increases, it quickly becomes unmanageable. A sorting and filtering would be desirable. For some use cases, it would also be good to have the ability to sort and filter via the API.

**Future work.** During integration of our scenarios, we have come across a set of challenges, which we will address in the near future. Examples include the better and easier creation of metadata in the steps. Integration to common ML libraries to extract metadata and send it to our system automatically would improve the usability. The optimal solution would be, that the user only provides the location of the pipeline framework and all the data can be extracted automatically or the framework communicated with our solution. The metadata will also be extended with additional artifact types, e.g. validation and ML model variants. Additional views such as a list of all artifacts and more specific detail views of the artifact types are planned for visualization.



## Bibliography

- [ACP19] R. Ashmore, R. Calinescu, C. Paterson. “Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges”. In: (2019). URL: <http://arxiv.org/abs/1905.04223> (cit. on pp. 22, 24).
- [Air] Airbnb. *Bighead*. URL: <https://databricks.com/session/bighead-airbnbs-end-to-end-machine-learning-platform/> (cit. on p. 34).
- [Ama] Amazon. *Amazon SageMaker*. URL: <https://aws.amazon.com/sagemaker/> (cit. on p. 34).
- [BCN+17] E. Breck, S. Cai, E. Nielsen, M. Salib, D. Sculley. “The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction”. In: *2017 IEEE International Conference on Big Data (Big Data)* (2017), pp. 1123–1132 (cit. on p. 17).
- [Ber14] D. Bernstein. “Containers and cloud: From LXC to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. ISSN: 23256095. DOI: [10.1109/MCC.2014.51](https://doi.org/10.1109/MCC.2014.51) (cit. on p. 37).
- [BHS07] F. Buschmann, K. Henney, D. Schimdt. *Pattern-oriented Software Architecture: on patterns and pattern language*. Vol. 5. John Wiley & sons, 2007 (cit. on p. 49).
- [BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven. “Prototyping”. In: *Prototyping*. Springer, 1992, pp. 33–46 (cit. on p. 51).
- [CATV17] G. Cohen, S. Afshar, J. Tapson, A. Van Schaik. “EMNIST: Extending MNIST to handwritten letters”. In: *Proceedings of the International Joint Conference on Neural Networks 2017-May* (2017), pp. 2921–2926. DOI: [10.1109/IJCNN.2017.7966217](https://doi.org/10.1109/IJCNN.2017.7966217) (cit. on p. 31).
- [CGM13] CGMA. “CGMA Briefing: Big Data”. In: *Accounting Horizons* 29.2 (2013), pp. 97–107. ISSN: 00178012. DOI: [10.1109/TKDE.2013.109](https://doi.org/10.1109/TKDE.2013.109) (cit. on p. 36).
- [CNK+17] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, M. Zaharia. “DAWNBench: An End-to-End Deep Learning Benchmark and Competition”. In: *Thirty-first Annual Conference on Neural Information Processing Systems (NIPS) Nips* (2017) (cit. on p. 58).
- [DG92] D. J. DeWitt, J. Gray. *Parallel database systems: The future of high performance database processing*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1992 (cit. on p. 51).
- [DRAV19] B. Derakhshan, T. Rabl, M. Alireza Rezaei, M. Volker. “Continuous Deployment of Machine Learning Pipelines”. In: (2019). DOI: [10.5441/002/edbt.2019.35](https://doi.org/10.5441/002/edbt.2019.35) (cit. on p. 37).

- [EIC17] H. Ed-douibi, J. L. C. Izquierdo, J. Cabot. “Example-driven web API specification discovery”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10376 LNCS (2017), pp. 267–284. ISSN: 16113349. DOI: [10.1007/978-3-319-61482-3](https://doi.org/10.1007/978-3-319-61482-3) (cit. on p. 52).
- [Fac] Facebook. *FBLearner Flow*. URL: <https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/> (cit. on p. 34).
- [Gooa] Google. *Google AI Platform*. URL: <https://cloud.google.com/ai-platform/> (cit. on p. 34).
- [Goob] Google. *Kubeflow*. URL: <https://www.kubeflow.org/> (cit. on pp. 34, 38).
- [GZLW09] A. Guazzelli, M. Zeller, W. C. Lin, G. Williams. “PMML: An open standard for sharing models”. In: *R Journal* 1.1 (2009), pp. 60–65. ISSN: 20734859. DOI: [10.32614/rj-2009-010](https://doi.org/10.32614/rj-2009-010) (cit. on p. 36).
- [HHL11] J. Han, E. Haihong, G. Le, J. Du. “Survey on NoSQL database”. In: *Proceedings - 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011* (2011), pp. 363–366. DOI: [10.1109/ICPCA.2011.6106531](https://doi.org/10.1109/ICPCA.2011.6106531) (cit. on p. 50).
- [HMR+] W. Hummer, V. Muthusamy, T. Rausch, P. Dube, K. E. Maghraoui. “ModelOps : Cloud-based Lifecycle Management for Reliable and Trusted AI”. In: () (cit. on pp. 28, 34).
- [HWS+06] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, T. Oinn. “Taverna: A tool for building and running workflows of services”. In: *Nucleic Acids Research* 34.WEB. SERV. ISS. (2006), pp. 729–732. ISSN: 03051048. DOI: [10.1093/nar/gk1320](https://doi.org/10.1093/nar/gk1320) (cit. on p. 28).
- [HZG+18] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, M. Fritz. “MLCapsule: Guarded Offline Deployment of Machine Learning as a Service”. In: *July 2017* (2018) (cit. on p. 37).
- [IBM] IBM. *IBM Watson*. URL: <https://www.ibm.com/watson> (cit. on p. 34).
- [Ini+18] O. Initiative et al. “OpenAPI Specification”. In: 1 (2018) (cit. on p. 52).
- [KK06] S. B. Kotsiantis, D. Kanellopoulos. “Data preprocessing for supervised learning”. In: *International Journal of ...* 1.2 (2006), pp. 1–7. ISSN: 1306-4428. DOI: [10.1080/02331931003692557](https://doi.org/10.1080/02331931003692557) (cit. on p. 36).
- [LAB+06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao. “Scientific workflow management and the Kepler system”. In: *Concurrency Computation Practice and Experience* 18.10 (2006), pp. 1039–1065. ISSN: 15320626. DOI: [10.1002/cpe.994](https://doi.org/10.1002/cpe.994) (cit. on p. 28).
- [LBBH98] Y. Lecun, L. Bottou, Y. Bengio, P. Ha. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* November (1998), pp. 1–46. ISSN: 00189219. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791) (cit. on pp. 31, 58).
- [LV04] K. R. Lakhani, E. Von Hippel. “How open source software works: “free” user-to-user assistance”. In: *Produktentwicklung mit virtuellen Communities*. Springer, 2004, pp. 303–339 (cit. on p. 33).
- [Mas11] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O’Reilly Media, Inc., 2011 (cit. on p. 52).

- [MBY+15] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, A. Talwalkar. “MLlib: Machine Learning in Apache Spark”. In: 17 (2015), pp. 1–7 (cit. on p. 29).
- [Mic] Microsoft. *Azure Machine Learning*. URL: <https://azure.microsoft.com/> (cit. on p. 34).
- [MLDD17] H. Miao, A. Li, L. S. Davis, A. Deshpande. “Towards Unified Data and Lifecycle Management for Deep Learning”. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2017, pp. 571–582. ISBN: 978-1-5090-6543-1. DOI: [10.1109/ICDE.2017.112](https://doi.org/10.1109/ICDE.2017.112) (cit. on pp. 22, 23).
- [MSK+15] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, R. K. Bocinsky, Y. Cao, J. Cheney, F. Chirigati, S. Dey, J. Freire, C. Jones, J. Hanken, K. W. Kintigh, T. A. Kohler, D. Koop, J. A. Macklin, P. Missier, M. Schildhauer, C. Schwalm, Y. Wei, M. Bieda, B. Ludäscher. “YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts”. In: *International Journal of Digital Curation* 10.1 (2015), pp. 298–313. DOI: [10.2218/ijdc.v10i1.370](https://doi.org/10.2218/ijdc.v10i1.370) (cit. on p. 28).
- [NDB+19] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. López García, I. Heredia, P. Malík, L. Hluchý. “Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey”. In: *Artificial Intelligence Review* 52.1 (2019), pp. 77–124. ISSN: 15737462. DOI: [10.1007/s10462-018-09679-z](https://doi.org/10.1007/s10462-018-09679-z) (cit. on p. 35).
- [PEŁ+18] G. C. Publio, D. Esteves, A. Ławrynowicz, P. Panov, L. Soldatova, T. Soru, J. Vanschoren, H. Zafar. “ML-Schema: Exposing the Semantics of Machine Learning with Schemas and Ontologies”. In: 1 (2018), pp. 1–5 (cit. on pp. 27, 28, 42).
- [Pol] Polyaxon. *Polyaxon*. URL: <https://polyaxon.com/> (cit. on pp. 34, 38).
- [RGC15] M. Ribeiro, K. Grolinger, M. A. Capretz. “MLaaS: Machine Learning as a Service”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Dec. 2015, pp. 896–902. ISBN: 978-1-5090-0287-0. DOI: [10.1109/ICMLA.2015.152](https://doi.org/10.1109/ICMLA.2015.152) (cit. on p. 24).
- [SBK+17] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, S. Seufert. “Automatically Tracking Metadata and Provenance of Machine Learning Experiments”. In: *Machine Learning Systems Workshop at NIPS (2017)* (cit. on pp. 27, 29, 42).
- [SPVS19] S. Schelter, N. Polyzotis, M. Vartak, S. Seufert. “DEEM 2019”. In: *Proceedings of the 2019 International Conference on Management of Data - SIGMOD '19*. Ed. by P. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, T. Kraska. New York, New York, USA: ACM Press, 2019, pp. 2066–2067. ISBN: 9781450356435. DOI: [10.1145/3299869.3323598](https://doi.org/10.1145/3299869.3323598) (cit. on p. 18).
- [TRA15] A. Tosatto, P. Ruiu, A. Attanasio. “Container-Based Orchestration in Cloud: State of the Art and Challenges”. In: *Proceedings - 2015 9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015* (2015), pp. 70–75. DOI: [10.1109/CISIS.2015.35](https://doi.org/10.1109/CISIS.2015.35) (cit. on p. 51).
- [Ube] Uber. *Michelangelo*. URL: <https://eng.uber.com/michelangelo/> (cit. on p. 34).
- [Var] M. Vartak. *Verta.ai*. URL: <https://www.verta.ai/> (cit. on p. 34).

- [VM] M. Vartak, S. Madden. “MODEL DB : Opportunities and Challenges in Managing Machine Learning Models”. In: (), pp. 16–25 (cit. on pp. 17, 34, 38).
- [VRBT14] J. Vanschoren, J. N. van Rijn, B. Bischl, L. Torgo. “OpenML: networked science in machine learning”. In: *dl.acm.org* (2014). doi: [10.1145/2641190.2641198](https://doi.org/10.1145/2641190.2641198) (cit. on p. 27).
- [VSL+16] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, M. Zaharia. “Model DB”. In: (2016), pp. 1–3. doi: [10.1145/2939502.2939516](https://doi.org/10.1145/2939502.2939516) (cit. on p. 24).
- [WGLM18] H. Wang, J. Gonzalez, G. Li, A. Meliou. “On Challenges in Machine Learning Model Management”. In: (2018), pp. 5–13 (cit. on pp. 24, 27).
- [WHRS19] C. Weber, P. Hirmer, P. Reimann, H. Schwarz. “A New Process Model for the Comprehensive Management of Machine Learning Models”. In: *Proceedings of the 21st International Conference on Enterprise Information Systems Iceis* (2019), pp. 415–422. doi: [10.5220/0007725304150422](https://doi.org/10.5220/0007725304150422) (cit. on pp. 17, 22, 23).
- [ZCD+18] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, C. Zumar. “Accelerating the Machine Learning Lifecycle with MLflow”. In: (2018), pp. 39–45 (cit. on pp. 17, 34, 38).
- [ZCF+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. “Spark Cluster Computing with Working Sets hotcloud\_spark.pdf”. In: *HotCloud* 10 (2010), p. 95 (cit. on pp. 18, 22, 36).
- [Zen04] M. Zeng. “Metadata types and functions”. In: *NISO*. Retrieved from: [https://en.wikipedia.org/wiki/MetadataRetrieved\\_on:\\_February\\_1](https://en.wikipedia.org/wiki/MetadataRetrieved_on:_February_1) (2004), p. 2017 (cit. on p. 42).
- [ZNS05] M. Zur Muehlen, J. V. Nickerson, K. D. Swenson. “Developing web services choreography standards - The case of REST vs. SOAP”. In: *Decision Support Systems* 40.1 SPEC. ISS. (2005), pp. 9–29. ISSN: 01679236. doi: [10.1016/j.dss.2004.04.008](https://doi.org/10.1016/j.dss.2004.04.008) (cit. on p. 50).

All links were last followed on October 27, 2019.

# A Appendix

---

## Listing A.1 Metadata Management Component API and Schema defined with the OpenAPI specification

---

```
1 openapi: 3.0.1
2 info:
3   description: This is a API for MMP Metadata
4   version: "1.0.0-oas3"
5   title: MMP Metadata
6 servers:
7   - url: 'https://localhost:8080/v1/metadata'
8   - url: 'http://localhost:8080/v1/metadata'
9 paths:
10  ...
11  /experiments/runs/{run}/executions:
12    parameters:
13      - name: run
14        in: path
15        required: true
16        description: The run's id
17        schema:
18          type: string
19    get:
20      tags:
21        - Execution
22      operationId: getExecutionsByRun
23      summary: list of executions
24      description: Return list of executions
25
26      responses:
27        '200':
28          description: OK
29          content:
30            application/json:
31              schema:
32                type: array
33                items:
34                  $ref: '#/components/schemas/Execution'
35  ...
36 components:
37   schemas:
38     Artifact:
39       discriminator:
40         propertyName: kind
41       mapping:
```

```
42     artifact: '#/components/schemas/Artifact'
43     model: '#/components/schemas/Model'
44     dataset: '#/components/schemas/Dataset'
45     metrics: '#/components/schemas/Metrics'
46     event: '#/components/schemas/Event'
47   required:
48     - name
49     - version
50     - kind
51   properties:
52     id:
53       type: string
54       format: uuid
55       example: d290f1ee-6c54-4b01-90e6-d701748f0851
56     name:
57       type: string
58       example: K-Nearst Model
59     version:
60       type: string
61       example: v1
62     kind:
63       type: string
64       default: artifact
65     description:
66       type: string
67       example: This is a Model.
68     createTime:
69       type: string
70       format: date-time
71       example: '2016-08-29T09:12:33.001Z'
72     uri:
73       type: string
74       example: s3:/bucket/model
75     annotations:
76       type: object
77     additionalProperties:
78       type: string
79   ...
```

---

### **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature