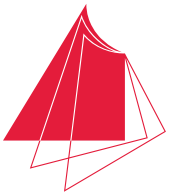


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Metadatenmanagement für Data Science Workflows auf Kubernetes

Bachelor Thesis von

Kevin Exel

30. Januar 2019

an der Fakultät für Informatik und Wirtschaftsinformatik
Fachrichtung Verteilte Systeme (VSYS)

Erstgutachter:	Prof. Dr. rer. nat. Christian Zirpins
Zweitgutachter:	Prof. Dr. rer. nat. Ulrich Bröckl
Betreuer:	Dr. rer. nat. Stefan Igel
Zweiter Betreuer:	Dipl.-Inform. Hans-Peter Zorn

Hochschule Karlsruhe Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik
Moltkestr. 30
76133 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 30.01.2019

. . . . . .

(Kevin Exel)

Zusammenfassung

Technologien wie ModelDB[23] sowie die Arbeiten [31] und [32] zeigen, dass es im Bereich der Metadaten und Verwaltung von Modellen im Bereich der Data Science aktuellen Bedarf gibt. Diese Arbeit entwickelt ein Datenmodell sowie eine Systemarchitektur mit der Metadaten, teils automatisiert, im Einsatzgebiet von Data Science Workflows auf der Kubernetes Plattform geschrieben und gelesen werden können. Hierbei wurde darauf geachtet, dass die Nutzung aus jedem Teilschritt heraus möglich ist und eine potenziell hohe Anzahl Programmiersprachen unterstützt werden können. Weiterhin wurde darauf geachtet das System so zu konzipieren, dass es nicht abhängig von speziellen Technologien, wie zum Beispiel Machine Learning Frameworks ist, sondern lediglich eine Abhängigkeit bezüglich der zur Umsetzung des Systems nötigen Technologien entstand. Das konzipierte System wurde prototypisch umgesetzt und anhand einiger Szenarien auf der Kubernetes Plattform evaluiert. Hierbei konnte gezeigt werden, dass dieses System funktional ist. Aufgrund des angedeuteten Wunsches keiner spezifischen Abhängigkeiten zu speziellen Machine Learning Frameworks wurden in dieser Arbeit keine Extraktoren entwickelt. Hieraus resultiert ein gewisser Mehraufwand bei der Verwendung des Systems, welcher jedoch durch ein potenziell breites Einsatzfeld aufgewogen wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

IaC Infrastructure as Code . . . 1
CRISP-DM Cross-industry standard process for data mining . . . 3
ASUM-DM Analytics Solutions Unified Method . . . 3
EVA Eingabe-Verarbeitung-Ausgabe . . . 5
YAML YAML Ain't Markup Language . . . 5
DAG Directed acyclic graph . . . 5
HA High Availability . . . 8
API Application Programming Interface . . . 8
CRUD Create, Read, Update, Delete . . . 9
EBS (Amazon) Elastic Block Store . . . 9
NFS Network File System . . . 9
IP Internet Protocol . . . 10
CPU Central Processing Unit . . . 16
CLI Command Line Interface . . . 48
I/O input/output . . . 16
REST Representational State Transfer . . . 18
AWS Amazon Web Services . . . 19
UML Unified Modelling Language
ER Entity Relationship . . . 23
NoSQL Not only SQL . . . 18
JSON Javascript object notation . . . 23
ELK Elasticsearch, Logstash, Kibana . . . 29
TCP Transmission Control Protocol . . . 34
IDCP Inovex Data Cloud Plattform . . . 19
HTTP Hypertext Transfer Protocol . . . 34
RPC Remote Procedure Calling . . . 37
RBAC Role Based Access Control . . . 47
EC2 (Amazon) Elastic Compute Cloud . . . 47
S3 (Amazon) Simple Storage Server . . . 31
CNCF Cloud Native Computing Foundation . . . 1
ICS Inovex Cloud Services . . . 55
UUID Universally Unique Identifier . . . 42
DSL Domain Specific Language . . . 43
XOR Exclusive Or

1. Einführung

Diese Arbeit entstand bei inovex GmbH[1] in Karlsruhe. Es wird untersucht, wie Metadaten im Bereich von Data Science[2] Workflows, hier im Sinne einer technischen Umsetzung eines (Teil-) Arbeitsprozesses auf Kubernetes[3] verwaltet werden können. Nachfolgend soll das Thema motiviert, sowie die Ziele dieser Arbeit definiert werden. Dieses Kapitel endet mit einem Überblick über den Aufbau dieser Arbeit.

1.1. Motivation und Ziele der Arbeit

Im Bereich der Data Science ist eine valide Aussage, welche Daten wie häufig für das Training oder die Evaluation eines Modells verwendet wurden, meist nicht trivial durchführbar. Weiterhin verliert man in großen verteilten Systemen den Überblick, wo Ausgangsdaten, Zwischen- oder Endergebnisse zu finden sind. Cloud[4] Infrastrukturen erlauben es Nutzern auf einfache Weise einen Rechnerverbund, auch als Computercluster[36] und im nachfolgend, als Cluster bezeichnet zu betreiben. Infrastructure as Code (IaC) Lösungen wie Terraform[5] erlauben es Entwicklern eigenständig auf der Infrastruktur verschiedenster Cloud Anbieter, virtualisiert oder ohne Virtualisierungsschicht "direkt" auf der Hardware, auch als *Bare Metal* bezeichnet, Cluster zu starten, betreuen und entfernen[41]. Virtualisierungslösungen wie Docker[6] erhöhen die Reproduzierbarkeit im Sinne einer Wiederholbarkeit und Ausführbarkeit von Data Science Arbeitsschritten; denn ein entsprechender Container wird bei gleichbleibenden Umgebungsbedingungen dasselbe Ergebnis liefern und vereinfacht das Deployment der Tools immens [24][9][7]. Kubernetes, eine ursprünglich von Google[7] entwickelte, mittlerweile an die Cloud Native Computing Foundation (CNCF)[8] übergebene Technologie, welche zur Orchestration[9], unter anderem von Docker Containern eingesetzt werden kann. Diese hat sich im Bereich des horizontalen Skalierens von Deep Learning Frameworks als mögliche Wahl gezeigt [16] und erfreut sich im Kontext der Data Science immer größerer Beliebtheit [40]. Technologien wie zum Beispiel Argo[10], eine Kubernetes native Workflow Engine, in welcher jeder Teilschritt einem Container entspricht, ermöglichen das effiziente Ausführen der heterogenen Teilaspekte

[1] https://www.inovex.de/de/
[2] Neue Erkenntnise anhand von Daten zu gewinnen ist Schwerpunkt der Data Science.
[3] https://kubernetes.io/
[4] Über ein Kommunikationsnetz mietbare und ansprechbare Infrastruktur.
[5] https://www.terraform.io
[6] https://www.docker.com/
[7] https://www.google.de/intl/de/about
[8] https://www.cncf.io
[9] Das automatisierte Starten und Verwalten von vielen Containern[39].
[10] https://argoproj.github.io

eines Workflows. Hierbei kommen die verschiedensten Techniken und Technologien zum Einsatz. Die zentralisierte Speicherung der genannten, verteilt entstehenden Metadaten, kann es ermöglichen, sowohl Aussagen über die historische Entwicklung eines Experimentes treffen zu können, als auch Ressourcen effizienter zu nutzen. Es können zum Beispiel Gesamtüberblicke über auf verschiedensten Clustern laufende Experimente ermöglicht werden. Im Hinblick auf Extraktion von Metadaten im Bereich des maschinellen Lernens ermöglicht dies das sogenannte Meta-Learning, welches ein Lernen anhand der Metadaten ermöglicht [31]. Die Infrastruktur Metadaten zu erfassen kann sowohl im Hinblick auf gleichbleibende / ähnliche Konfiguration der Infrastruktur als auch der Vergleichbarkeit von Laufzeiten unterstützen.

Ziel der Arbeit ist es, ein Informationsmodell zu entwickeln, welches idealerweise auch unabhängig von einer expliziten Technologie bzw. Problemdomäne eines Benutzers ist und es erlaubt, Aussagen über die genannten Punkte zu treffen. Weiterhin soll ein prototypisches System entwickelt werden, welches das Konzept implementiert, leicht in einen bestehenden Kubernetes Cluster integrierbar ist und eine Evaluierung anhand eines Workflows umsetzt. Weiterhin soll untersucht werden, wie Metadaten innerhalb der eingesetzten Systeme automatisiert erfasst werden können. Die Kuration bereits geschriebener Metadaten ist ein weiterer wichtiger Aspekt eines solchen Systems und muss theoretisch formalisiert werden. Das zu konzipierende System muss über Schnittstellen verfügen, welche sowohl das Lesen als auch das Schreiben von Metadaten unterstützt.

1.2. Aufbau der Arbeit

Die vorliegende Arbeit umfasst sieben Kapitel. Zu Beginn sollen Grundlagen in den drei Bereichen dieser Arbeit geschaffen werden. Hierbei wird zuerst der Begriff des Data Science Workflows betrachtet. Anschließend soll der Begriff Metadaten vorgestellt und einzelne Kategorien unterteilt werden. Das Grundlagen-Kapitel schließt mit einer Betrachtung der Kubernetes Technologie. Das folgende Kapitel analysiert vorhandene Technologien und Ansätze. Nutzungsszenarien und Anforderungen an das zu konzipierende System sowie eine Analyse möglicher Metadaten in Data Science Workflows sind die folgenden Aspekte dieses Kapitels. Das Konzeptions-Kapitel betrachtet zu Beginn die Datenbankwahl und ihren Einfluss auf die Modellierung des Datenmodells. In der anschließenden Modellkonzeption werden ein Metamodell sowie das objektorientierte Regionenmodell vorgestellt. Zuletzt wird die Architektur des Systems Technologie-agnostisch vorgestellt, um diese Architektur im folgenden Implementierungskapitel dann entsprechend zu füllen. Anschließend wird das Deployment vorgestellt, welches im Kapitel der funktionalen Evaluation verwendet wird. Im Evaluations-Kapitel wird weiterhin das Deployment der restlichen Infrastruktur vorgestellt. Die Verwendung des Systems und nötige Änderungen an den genutzten Workflowbeschreibungen und Machine Learning Skripten wird aufgezeigt und das hierdurch entstandene Ergebnis beschrieben und diskutiert. Diese Arbeit endet mit einem Fazit sowie einem Ausblick.

2. Grundlagen

Dieses Kapitel soll die Grundlagen dieser Arbeit vermitteln und gliedert sich in drei Bereiche. Zu Beginn soll der Begriff des Data Science Workflows erklärt werden. Das anschließende Subkapitel betrachtet den Begriff der Metadaten; hierbei werden auch bereits erste, aus den vorliegenden Problemdomänen kommende Metadaten, anhand eines Beispiels vorgestellt. Dieses Kapitel schließt mit den Grundlagen der Technologie Kubernetes, welche als Plattform, zur effizienten Orchestrierung von Containern, zum Einsatz kommt.

2.1. Data Science Workflows

Der Begriff des Data Science Workflows soll in dieser Arbeit aus zwei Perspektiven betrachtet werden. Auf der einen Seite existieren diese als Methodologie und werden zu Beginn anhand des Cross-industry standard process for data mining (CRISP-DM) veranschaulicht. Auf der anderen Seite existiert dieser Begriff auf technischer Ebene und beschreibt, wie ein Arbeitsablauf auf einem technischen System umgesetzt wird. Hierbei wird eine beispielhafte Beschreibung eines Workflows der später verwendeten Workflow Engine Argo vorgestellt.

2.1.1. Data Science Workflows als Arbeitsprozesse

Data Scientists arbeiten häufig nach ihren eigenen Prozessen[31]. In den letzten Jahren entstanden jedoch einige Standards und Definitionen wie zum Beispiel der nachfolgend betrachtete CRISP-DM[1]. Weitere, wie zum Beispiel Analytics Solutions Unified Method (ASUM-DM)[2] oder der Arbeitsprozess von Uber[3], variieren in Ihrem Abstraktionsgrad, sowie auch im Umfang. Einige betrachten hierbei sowohl die Datenerfassung als auch das Deployment in detaillierter Weise. Für andere wiederum beginnt der Prozess beim Laden der Daten und endet mit dem Modell-Deployment als "Black Box", in welcher das Deployment, Monitoring und weitere Phasen des Deployments in einer Phase zusammengefasst sind. Nachfolgend wird der Ablauf der Methodologie CRISP-DM betrachtet.

2.1.1.1. Cross-industry standard process for data mining

Das CRISP-DM Modell wurde 2000 veröffentlicht und von einem Konsortium aus mehreren Unternehmen entwickelt. Es betrachtet, wie in Abbildung 2.1 auf der nächsten Seite

[1] ftp.software.ibm.com/software/analytics/spss/support/Modeler/Documentation/14/UserManual/CRISP-DM.pdf
[2] ftp.software.ibm.com/software/data/sw-library/services/ASUM.pdf
[3] https://eng.uber.com/scaling-michelangelo/

dargestellt, Data Mining als Zyklus aus sechs Phasen und beschreibt es mit einer Schritt für Schritt Anleitung. Diese werden von einer kontinuierlichen Phase umschlossen, welche die zyklische Charakteristik des Data Minings versinnbildlichen soll. Dabei ist der Ablauf der Phasen nicht streng vorgegeben und ein Wechsel zwischen ihnen vorgesehen. Die Pfeile zwischen den Phasen verdeutlichen Abhängigkeiten. Wie auch Chapman et. al. in ihrem *Step-by-step data mining guide* beschreiben, endet das Data Mining jedoch nicht mit dem Deployment, sondern ermöglicht neue Erkenntnisse, hierdurch ein tieferes Verständnis und somit neue Fragestellungen. [8] Im nachfolgenden sollen die einzelnen, in Abbildung 2.1

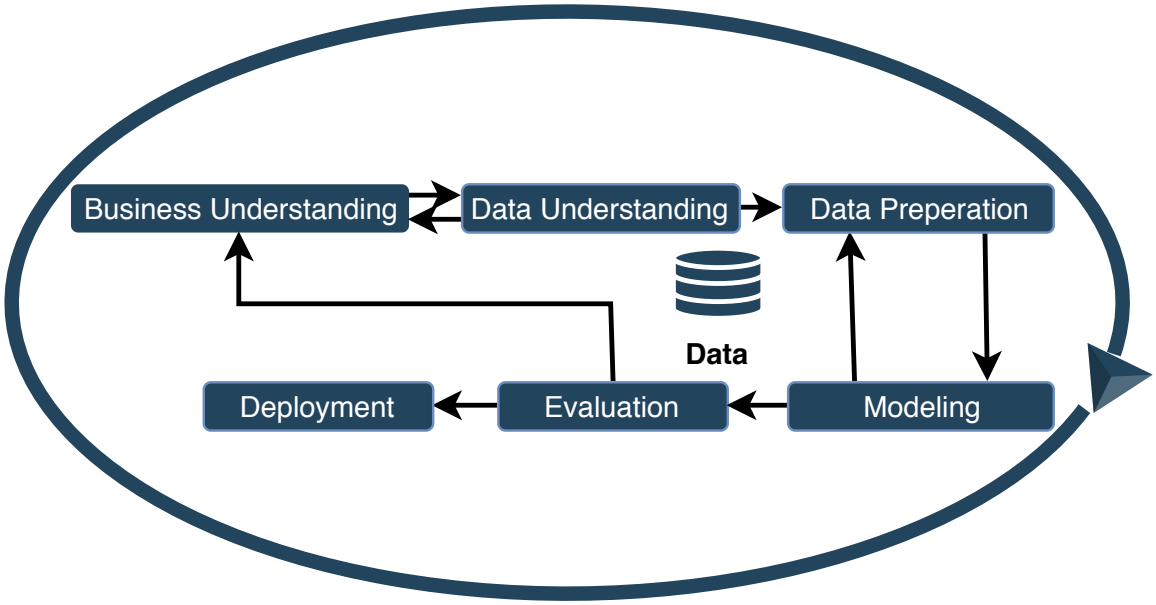


Abbildung 2.1.: Das CRISP-DM Modell angelehnt an Quelle: [8]

zu entnehmenden, Phasen beschrieben werden.

Phase 1 Business Understanding Diese Phase dient dem Verständnis der Projekt-Ziele, sowie der Anforderungen auf betrieblicher Seite. Dies wird anschließend genutzt, um einen vorläufigen Plan, sowie ein Data Mining Problem zu definieren. [8]

Phase 2 Data Understanding Die zweite Phase beginnt mit dem Zusammentragen von Daten, sowie der Gewinnung erster interessanter Kenntnisse über diese. Hierbei entstehen erste Hypothesen über in den Daten befindliche Informationen. [8]

Phase 3 Data Preperation An Phase 2 schließt sich die Data Preperation Phase an, in welcher es um die Konstruktion des für das Modelling Tool finalen Datensatzes geht. Diese vorbereitenden Maßnahmen können mehrere Male wiederholt werden und umfassen neben der Selektion und Transformation auch die Bereinigung der Daten. [8]

Phase 4 Modelling In dieser Phase werden unterschiedlichste Methodiken und Parameter verwendet, mit dem Ziel hierdurch die bestmöglichen Ergebnisse zu erzeugen. Hieraus kann ein häufiger Wechsel zwischen Phase 3 und 4 resultieren. [8]

Phase 5 Evaluation Das fertige Modell wird hinsichtlich der definierten betrieblichen Ziele geprüft und auf etwaige Mängel untersucht. Sind die Ziele erfüllt erfolgt das Deployment. [8]

Phase 6 Deployment Mit dem Deployment endet nicht das Projekt, sondern es werden neue Erkenntnisse gewonnen, welche wiederum zurückfließen können. Diese Phase kann einfach und durch das Schreiben eines Berichts beendet werden oder komplexer sein und einen sich wiederholenden Prozess umfassen. [8]

Das beschriebene Modell wird dieser Arbeit als Methodologie zugrunde gelegt und anhand diesem in Kapitel 3.3.1 auf Seite 19 Nutzungsszenarien veranschaulicht.

2.1.2. Data Science Workflows als Technischer Begriff

Neben der Betrachtung als Methodologie kann unter dem Begriff Workflow auch die technische Beschreibung der einzelnen Arbeitsschritte verstanden werden. Häufig[4][5][6][7][8] wird dies durch die deskriptive Beschreibung eines Zustandsautomaten beziehungsweise Graphen erreicht. Sie folgen dem Eingabe-Verarbeitung-Ausgabe (EVA)-Prinzip und erlauben es einzelne Schritte zu verketten und die Ausgabe des einen, als Eingabe eines anderen Schrittes formal zu beschreiben. Diese deskriptive Beschreibung kann nun von einer Engine eingelesen und entsprechend umgesetzt werden. Im nachfolgenden soll ein einfacher beispielhafter Workflow Technologie Argo gezeigt werden. Das in Abbildung 2.1 auf Seite 11 dargestellte YAML Ain't Markup Language (YAML) – Dokument beschreibt einen einfachen Schritt-basierten Workflow[9]. Ein komplexerer und größerer Workflow, welcher auch in dieser Arbeit Verwendung findet, kann dem Anhang A.11 auf Seite 76 entnommen werden. Die Beschreibung eines Workflows wie in Abbildung 2.1 auf Seite 11 zu sehen besteht zu Beginn aus diversen deskriptiven Metadaten [10]. Das Datum entrypoint zeigt auf das zu verwendende Template. Es folgen eine Reihe technischer Metadaten. Dies beinhaltet zum einen die formale Beschreibung des Zustandsautomaten, mit sequenziellem und parallelem Anteil, sowie der Name des auszuführenden Containers.

Die Technologie Argo wird in dieser Arbeit als Workflow Engine Verwendung finden und in Kapitel 5.1.3 auf Seite 44 detaillierter beschrieben.

2.2. Metadaten

Metadaten sind Daten über Daten. Ihr Zweck dient der Beschreibung von Daten und können beim Verständnis von Daten unterstützen. Richard Gartner beschreibt in seinem

[4] https://azkaban.readthedocs.io/en/latest/createFlows.html#step-1
[5] https://github.com/treasure-data/digdag/blob/master/digdag-docs/src/workflow_definition.rst
[6] https://github.com/creactiviti/piper
[7] https://github.com/fission/fission-workflows/tree/master/examples/
[8] https://github.com/argoproj/argo/tree/master/examples
[9] Directed acyclic graph (DAG) basierte Workflows sind ebenfalls möglich.
[10] Siehe Kapitel 2.2

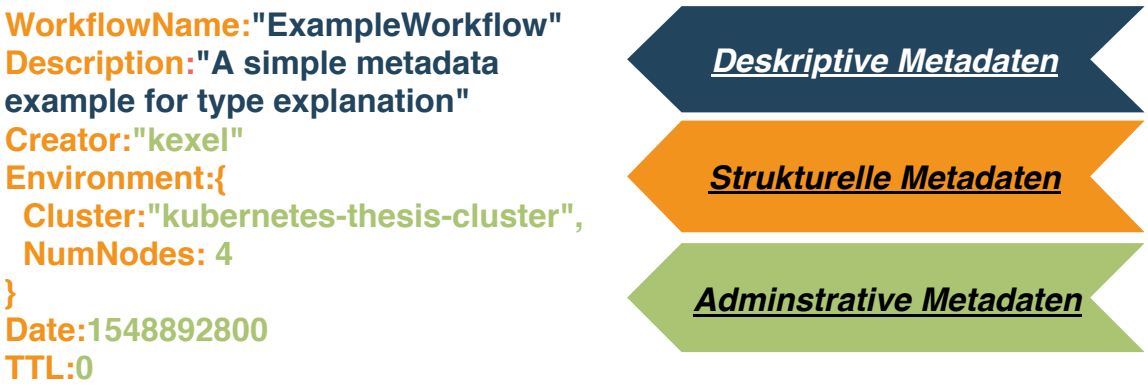


Abbildung 2.2.: Metadaten-Kategorien veranschaulicht anhand einer beispielhaften Beschreibung

Buch *Metadata Shaping Knowledge from Antiquity to the Semantic Web* drei Typen von Metadaten. Diese strukturell, deskriptiv und administrativ genannten Typen werden nachfolgend beschrieben. Abbildung 2.2 veranschaulicht diese Typen an einem fiktiven Beispiel von Workflow-Metadaten. [11]

2.2.1. Deskriptive Metadaten

Deskriptive Metadaten erlauben es Dinge zu finden und werden im Englischen auch als *Finding Metadata*[11] bezeichnet [11]. In der vorliegenden Arbeit könnten das zum Beispiel Informationen darüber sein, wo ein trainiertes Modell zu finden ist oder in welchem Verzeichnis ein bestimmter Datensatz abgelegt ist. Aber auch beschreibende Merkmale wie Namen eines Workflows oder Projektersteller können in dieser Kategorie zu finden sein. [11]

2.2.2. Administrative Metadaten

Der Bereich der administrativen Metadaten beinhaltet die Bereiche der technischen Metadaten, der Rechte-Metadaten sowie der Konservierungs-Metadaten [11].

Technische Metadaten Beschreiben für ein System notwendige Daten um ein technisches Objekt auszuliefern und zu rendern. Hierunter fallen zum Beispiel Informationen wie die Größe, der Datentyp, Kompressionsverfahren und weitere. Diese Metadaten unterscheiden sich je nach Objekttyp und sind für den Nutzer in der Regel transparent.[11]

Rechte-Metadaten Diese Metadaten ermöglichen in einem System den Schutz von geistigen Eigentum. Zugriffsrechte auf Daten, sowie das Urheberrecht werden in diesen festgehalten.[11]

Konservierungs-Metadaten Sie unterstützen und ermöglichen das Speichern von Daten über einen langen Zeitraum. Weiterhin dokumentieren sie, welche Aktionen auf den Daten unternommen wurden, wann dies geschehen ist, sowie von wem. [11]

2.2.3. Strukturelle Metadaten

Diese Metadaten werden genutzt, um aus kleinen Teilen größere Strukturen zu machen [11]. In dieser Arbeit sind dies Schlüsselwerte der Schemata. Vorstellbar wären hier jedoch auch Referenzen auf andere Objekte. Hierdurch könnten zum Beispiel Metadaten einzelner virtueller Maschinen als Knoten eines Kubernetes Clusters beschrieben werden.

2.3. Kubernetes

Kubernetes ist eine Technologie, welche ursprünglich vom Unternehmen Google entwickelt wurde. Diese soll Entwickler und Administratoren bei der Entwicklung und dem Betrieb, verteilter, skalierbarer und hochverfügbarer Systeme unterstützen. Dabei dient Kubernetes als Container Orchestrator. Container sowie die Grundlagen von Kubernetes sollen im nachfolgenden betrachtet werden.

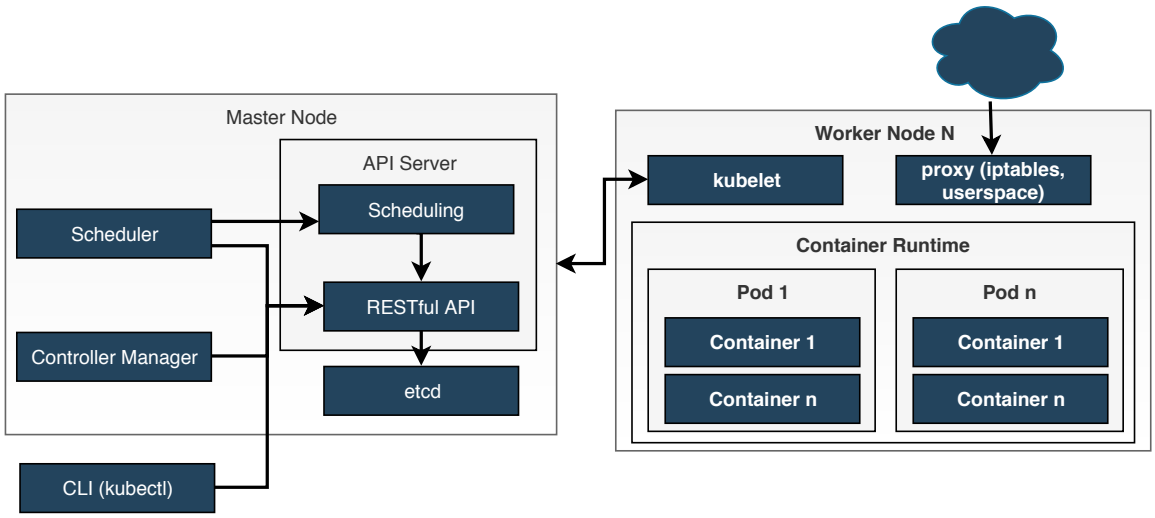


Abbildung 2.3.: Die Kubernetes Architektur angelehnt an Quellen: [21], [30], [29], [6]

2.3.1. Containervirtualisierung

Containervirtualisierung ist eine Technik, mit welcher mehrere Linux-Betriebssysteme auf einem Host schnell und flexibel gestartet werden können. Es werden die Ressourcen des Host-Systems mit den Gast-Systemen geteilt, indem auf Linux Kernel Funktionen wie beispielsweise control groups[11] und Namensräume[12] zugegriffen wird. Diese Funktionen er-

[11] https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process
[12] https://itnext.io/chroot-cgroups-and-namespaces-an-overview-37124d995e3d

lauben die Isolation der einzelnen Gäste. [29] Container sind hierbei eine leichtgewichtigere Alternative zu Virtuellen Maschinen. Ein Image beschreibt hierbei alle benötigten Dinge, welche für den Ablauf des jeweiligen Programms oder auch Services benötigt werden. Dies beinhaltet das Betriebssystem, Tools, Bibliotheken, Konfigurationen, den jeweiligen Quellcode beziehungsweise ausführbare Programme sowie eventuell die benötigten Daten. Letztere können jedoch auch über ein Cloud-Speicher mittels Netzwerkkommunikation oder ein gemeinsam genutztes Dateisystem zur Verfügung gestellt werden. [18] Sculley et. al. beschreiben die Existenz besonderer technischer Schulden in Machine Learning Projekten [34]. Containervirtualisierung ermöglicht es hier, Reproduzierbarkeit zu gewährleisten und erlaubt es Wissenschaftlern, Experimente auf unterschiedlichen Systemen nachzuvollziehen zu können [13] [9] In der Praxis beschreibt häufig nicht ein einzelner Container den Workflow. Viel mehr müssen eine Vielzahl Container, häufig parallel gestartet, untereinander kommunizieren sowie externe Datenquellen zur Verfügung gestellt werden. Kubernetes ist hier eine Möglichkeit dies zu ermöglichen. [18]

Container Runtime Eine Container Runtime ermöglicht das Starten und Verwalten von Containern auf einem Host. Kubernetes erlaubt es neben Docker verschiedenste Container Runtimes zu verwenden. Diese werden auf den jeweiligen Nodes gestartet. [30]

2.3.2. Nodes

Ein Node ist entweder eine physikalische oder virtuelle Maschine. Innerhalb eines Kubernetes Clusters gibt es Master[14] sowie Worker Nodes. Wie in Abbildung 2.3 auf der vorherigen Seite veranschaulicht, enthält ein Worker sowohl die *Kubelet* als auch eine *Proxy* Komponente und wird vom Master verwaltet. Auf den Nodes laufen sogenannte Pods, welche in Kapitel 2.3.3 auf Seite 10 betrachtet werden. [17]

2.3.2.1. Master Node Komponenten

Der Master ist die Zentrale Steuereinheit eines Kubernetes Clusters und besteht aus mehreren Komponenten. In der Regel werden diese zusammen auf einer Maschine gestartet und auf dieser keine User Workloads ausgeführt. [17]

Application Programming Interface (API) Server Diese Komponente ermöglicht es die Kubernetes API zu nutzen. Clients bauen mit diesem eine Verbindung auf und können hierdurch Operationen auf Ressourcen ausüben[17].

etcd Diese Komponente ist ein Key-Value Store, welcher verwendet wird, um den Zustand des Kubernetes Clusters zu halten[17].

[13] Unter der Voraussetzung, dass auf dem System eine entsprechende Runtime zur Verfügung steht.
[14] In High Availability (HA)-Clustern existieren mehrere Master Nodes vgl. https://kubernetes.io/docs/setup/independent/high-availability/

Scheduler Diese Komponente überwacht den Cluster auf neu hinzugekommene Pods, welche noch keinem Node zugewiesen wurden. Sie beachten einige Faktoren wie zum Beispiel Ressourcenanforderungen und weisen den Pod anschließend einem zur Verfügung stehendem Node zu. [17]

(Kube) Controller Manager Ein Controller ist eine Komponente, welche eine steuernde Funktion übernimmt und innerhalb eines Kubernetes Clusters zum Beispiel eine Überwachung eines Deployments übernehmen kann. Der Controller Manager ist ein binary, das aus der logischen Sicht aus mehreren verschiedenen Controllern besteht. Diese können zum Beispiel Nodes überwachen und auf Abstürze reagieren (Node Controller), die korrekte Anzahl Pods im System überwachen (Replication Controller), die Endpunkt Objekte zur Verfügung stellen (Endpoint Controller), sowie sogenannte *default Accounts* und API Zugriffstoken für neue Namensräume erzeugen (Service Account und Token Controllers). [17]

(Cloud) Controller Manager Dieser Controller wurde aus Gründen der Einfachheit in Abbildung 2.3 auf Seite 7 nicht aufgenommen. Er dient der Kommunikation mit dem zugrundeliegenden Cloud Provider und umfasst folgende Funktionalitäten: Kommunikation um die Löschung eines Nodes in der Cloud zu ermitteln (Node Controller), Routen in der Cloud Infrastruktur anzulegen (Route Controller), Create, Read, Update, Delete (CRUD) Operationen auf Cloud-Provider Loadbalancer (Service Controller) sowie Erzeugen, Zuweisen, Mounten und Interagieren mit Volumes. Volumes sind eine Kuberneters Abstraktion welche verschiedene Speichertypen, wie Cloud-Speicher zum Beispiel *Amazon (Amazon) Elastic Block Store (EBS)*[15], *cephfs* [16] oder auch *Network File System (NFS)*[17] umfasst und mittels sogenanntem *Volume Mount* einem Container zur Verfügung gestellt werden können. Sie lösen das Problem, dass Daten innerhalb eines Containers flüchtig sind und bei einem Neustart der Container mit einem frischen Zustand startet. [17]

2.3.2.2. Worker Node Komponenten

Worker Nodes werden vom Master verwaltet und sind für die Umsetzung der User Workloads verantwortlich. Die einzelnen Komponenten haben folgende Funktion. [17]

Kubelet Die Kubelet Komponente läuft auf jedem Node des Clusters und stellt sicher, dass die Container in einem Pod laufen. Hierfür existieren sogenannte PodSpecs, welche eine formale Beschreibung der Container darstellen. [17]

Proxy Proxy erlaubt durch Netzwerkregeln und Weiterleitung Forwarding auf dem Host, die später verwendete und in Kapitel 5.2 auf Seite 51 erklärte Kubernetes Service Abstraktion. [17]

[15] https://aws.amazon.com/de/ebs/
[16] https://kubernetes.io/docs/concepts/storage/volumes/#cephfs
[17] http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-de-4/ch-nfs.html

2.3.3. Pods

Pods dienen, als kleinste Einheit dazu mehrere Container zu verwalten. Dies ermöglicht es eng zusammengehörende Komponenten gemeinsam zu halten. Innerhalb eines Pods haben Container dieselbe Internet Protocol (IP)-Adresse, können mittels Interprozesskommunikation, sowie via localhost kommunizieren. Weiterhin haben sie Zugriff auf den geteilten Local Storage des Nodes, auf welchem der Pod läuft. Hierbei wird der Storage in jeden einzelnen Container entsprechend gemountet. Container in Pods sowie ihre Local Storages sind flüchtig; der Storage wird bei Löschen des Pods entsprechend auch gelöscht. [30] [17]

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: steps-
spec:
  entrypoint: hello-hello-hello
  templates:
  - name: hello-hello-hello
    steps:
    - - name: hello1
        template: whalesay
        arguments:
          parameters: [{name: message, value: "hello1"}]
    - - name: hello2a
        template: whalesay
        arguments:
          parameters: [{name: message, value: "hello2a"}]
      - name: hello2b
        template: whalesay
        arguments:
          parameters: [{name: message, value: "hello2b"}]
  - name: whalesay
    inputs:
      parameters:
      - name: message
    container:
      image: docker/whalesay
      command: [cowsay]
      args: ["{{inputs.parameters.message}}"]
```

Listing 2.1: Einfache Beschreibung eines Workflows für die Workflow Engine Argo Quelle: https://github.com/argoproj/argo/tree/master/examples

3. Analyse

Diese Sektion betrachtet zu Beginn verschiedene Metadaten aus dem Bereich Data Science. Bestehende Ansätze werden diskutiert sowie mögliche Nutzungsszenarien des zu konzipierenden Systems anhand des in Kapitel 2.1.1.1 auf Seite 3 gezeigten CRISP-DM Modells veranschaulicht. Eine Analyse der Anforderungen an das System sind Kern der anschließenden Betrachtung.

3.1. Analyse verschiedener Metadaten im Bereich Data Science Workflows

Nachfolgend soll nun betrachtet werden, welche möglichen Metadaten hierbei im Kontext der Arbeit vorkommen können. Es sollen an diesem Punkt auch nötige Termini beschrieben und Grundwissen vermittelt werden. Dabei werden nachfolgend vier Kategorien erarbeitet:

1. Datensatz
2. Machine Learning
3. Laufzeitumgebung
4. Experiment und Workflow

3.1.1. Datensatz Metadaten

In Tabelle 3.1 auf der nächsten Seite ist ein Ausschnitt des IRIS Datensatzes zu sehen, welcher stellvertretend für zweidimensionale Datensätze betrachtet wird. Der komplette Datensatz umfasst 150 Instanzen [14]. Dieser Datensatz ist ein gelabelter Datensatz.Solche Datensätze kommen beim Verfahren des überwachten Lernens zum Einsatz [12]. Für das vorliegende Beispiel wurden für jede im Datensatz vorkommende Klasse eine Instanz extrahiert.

Instanz und Beispiel Eine Instanz bzw. Beispiel enthält alle zu einer Klasse gehörenden Features. Hierbei kann bei Datensätzen für überwachtes Lernen auch ein sogenanntes class label enthalten sein. Aus diesem Grund unterscheidet man diese auch in gelabelte und ungelabelte Beispiele. [20] Im in Tabelle 3.1 auf der nächsten Seite zu sehenden Ausschnitt des Iris Datensatzes sind 3 Instanzen zu sehen.

[1] angelehnt an Quelle: [15]

Tabelle 3.1.: Ausschnitt des IRIS Datensatzes[1]

sepallength	sepalwidth	petallength	petalwidth	class label
5.1	3.5	1.4	0.2	Iris-setosa
7.0	3.2	4.7	1.4	Iris-versicolor
6.3	3.3	6.0	2.5	Iris-virginica

Feature und Attribut Jede Instanz verfügt über ein oder mehrere Features [12]. Ein Attribut ist die Bezeichnung des Datums, welches den Inhalt eines Features spezifiziert [12]. Für den in Tabelle 3.1 zu sehenden Datensatz sieht die Menge der Attribute wie folgt aus:

$$M_{attributes} = \{sepallength, sepalwidth, petallength, petalwidth\}$$

Ein (diskretes) Feature kann, abhängig des Kontextes, jedoch als Tupel aus dem Attribut und dem entsprechenden Wert betrachtet werden[12].

class labels Die Klasse, in Tabelle 3.1 mit class zu sehen, entspricht dem Ausprägungsmerkmal einer Instanz und ist bei gelabelten Daten vorhanden. Jede Instanz gehört folglich zu einer Klasse. Dabei kann für einen endlichen Datensatz eine endliche Menge an Klassen definiert werden. Betrachtet man den vorliegenden Datensatz, so stellt man fest, dass die Menge der möglichen Klassen wie folgt aussehen:

$$M_{classes} = \{Iris\text{-}setosa, Iris\text{-}versicolour, Iris\text{-}virginica\}$$

Das Ziel eines Algorithmus ist es in der Regel anhand der Features eine Prognose (Prediction) über die Zugehörigkeit einer Instanz zu einer bestimmten Klasse zu fällen [12]. Es gibt jedoch auch Bereiche, in den ein Feature selbst das Ziel des Trainings ist; dies wird als Feature Learning bezeichnet [25].

Missing Values Ein nicht vorhandenes Feature innerhalb einer Instanz wird als *Missing Value* bezeichnet. Die Anzahl der *Missing Values* innerhalb eines Datensatzes ist eine wichtige Information, denn der Umgang mit *Missing Values* muss vor dem Training durchdacht werden [12].

3.1.2. Machine Learning Metadaten

Metadaten aus dem Bereich des maschinellen Lernens als Teil eines Data Science Workflows sollen im nachfolgenden näher betrachtet werden. Es existieren verschiedene Domänenspezifische Termini, welche teilweise unterschiedlich besetzt sind und im nachfolgenden kurz erläutert werden sollen. Dabei soll nicht detailliert auf die einzelnen Techniken eingegangen werden; hierfür sei auf genannte Fachliteratur verwiesen. Viel mehr soll das

Tabelle 3.2.: Einige Machine Learning Methoden mit Beispiel Algorithmen [2]

Methode	Algorithmik
Supervised	Lineare Regression, SVM, einige Deep Learning Verfahren (Diese können jedoch auch Semisupervised oder Unsupervised sein)...
Unsupervised	Clustering (k-means, HCA, ...), Visualisierung und Dimensionsreduzierung, ...
Reinforcement Learning	Strategie des Agenten

Spektrum maschinellen Lernens aufgezeigt und veranschaulicht werden, welche interessante Informationen ein Metadaten-Schema für maschinelles Lernen beinhalten könnte.

Lernverfahren Aurélien Géron beschreibt in seinem Buch *Hands-On Machine Learning with Scikit-Learn and TensorFlow Concepts, Tools, and Techniques to Build Intelligent Systems* verschiedene Klassifizierungsmerkmale von maschinellen Lernverfahren. Er unterscheidet hierbei hinlänglich der Art, wie Daten für ein Lernverfahren organisiert sein müssen. Tabelle 3.1 auf der vorherigen Seite veranschaulicht einen Ausschnitt des Iris Datensatzes. Dieser ist ein sogenannter gelabelter Datensatz, wie er in Supervised Lernverfahren verwendet wird. In Supervised Lernverfahren ist für jede Instanz bekannt, welcher Klasse diese entspricht. In der Regel teilt man einen solchen Datensatz in zwei Hälften. 80 Prozent werden für das Training verwendet und als Trainingsdatensatz bezeichnet.[12] 20 Prozent bilden den Testdatensatz[12]. Ziel des Verfahrens ist es nun durch Betrachtung der Features des Trainingsdatensatzes eine Klassifizierung zu ermöglichen und Instanzen des Testdatensatzes korrekt einer Klasse zuordnen zu können [12]. Unsupervised Lernverfahren hingegen ermöglichen eine Prognoseerstellung ohne den Bedarf gelabelter Daten. Der Algorithmus lernt somit selbst Datenpunkte einer bestimmten Klasse zuzuordnen. Reinforcement Learning ist ein Verfahren in welchem ein sogenannter Agent (das Lernsystem) innerhalb seiner Umgebung Aktionen durchführen kann. Über einen Mechanismus wird der Agent hierbei für seine Aktionen belohnt oder bestraft. Das Ziel des Agenten ist es eine Strategie zu entwickeln mit welcher er innerhalb einer bestimmten Zeit die größte Belohnung erhält. [12] Tabelle 3.2 zeigt hier einen kleinen Ausschnitt an Methoden und Algorithmen.

Reaktivität auf neue Daten Die Art der Reaktivität auf neue Daten wird in online learning und batch learning unterteilt. Hierbei ist das Klassifizierungsmerkmal die Tatsache, ob das System inkrementell durch einen Datenstrom lernen kann. Online learning erlaubt das inkrementelle Lernen mittels sequentiell hinzugefügter Dateninstanzen. Werden diese

[2] vgl. Quelle: [12]

nicht einzeln, sondern in Gruppen hinzugefügt, nennt man diese *mini batch*. Das sogenannte *batch learning* hingegen erlaubt kein inkrementelles Lernen. Hierbei muss für jedes adaptieren auf neue Daten mit dem gesamten Dataset gelernt werden. Dies erfordert in der Regel viel Ressourcen (Central Processing Unit (CPU), Memory, Storage, input/output (I/O), Time). Ein auf diese Weise trainiertes Modell wird in der Regel ausgeliefert, ohne weiterzulernen und lediglich das gelernte zu Nutzen. Hierbei wird auch von *offline learning* gesprochen. [12]

Training Der Prozess, in welchem ein bestimmter Algorithmus Muster in den Daten lernt, wird Training genannt. Das Ergebnis des gesamten Trainings wird als das Modell bezeichnet. [12]

Model Das Modell ist die Repräsentation des mithilfe der Trainingsdaten Gelerntem [20]. Eine andere Beschreibung ist, dass das Model eine vereinfachte Repräsentation des beobachteten ist [12].

Prognose Wie bereits in Kapitel beschrieben wird die Antwort des Modells auf eine Fragestellung als Prognose [3] bezeichnet [12].

Evaluation und Validation Das Ziel maschinellen Lernens ist es ein Modell zu entwickeln, welches anhand der vorhandenen Daten eine Generalisierung erlaubt und eine möglichst fehlerfreie Prognose erstellen kann. Um dies zu überprüfen wird mit dem in Tabelle 3.1 auf Seite 14 gezeigtem Testdatensatz die Qualität des Modells überprüft. [12] Für eine Erklärung der verschiedensten Techniken sei auf genannte Literatur verwiesen.

Model Parameter und Hyperparameter Zu unterscheiden sind zwei Arten von Parametern. Sogenannte Model Parameter sind interne Parameter eines entsprechenden Modells und beschreibt den Freiheitsgrad. Hyperparameter hingegen konfigurieren die Regularisierung und sind Parameter des Lernalgorithmus, wird vom Nutzer im Voraus festgelegt und bleibt für die Dauer eines Trainings konstant. [12]

Feature Tranformation, Reduktion, Extraktion Feature Transformation ist eine Tätigkeit, in welcher versucht wird, die Daten zu vereinfachen ohne zu viele Informationen zu verlieren. Bei der *dimensionality reduction* werden dabei korrelierte Features zu einem gemeinsamen Feature zusammengefasst. Die wird auch *feature extraction* genannt. Vorteile hiervon ist ein schneller Trainingslauf, geringerer Festplatten- sowie Arbeitsspeicher. [12]

Technologiewahl Ein weiterer Aspekt, welcher im Hinblick auf historische Entwicklungen eines Experimentes von Interesse sein kann ist die Wahl der Technologien. Hierbei können zum Beispiel die Wahl der Programmiersprache und ihre Version sowie das verwendete Framework und seine Version bedeutsame Metadaten sein. Diskutabel ist hierbei,

[3] Im Englischen als Prediction bezeichnet.

Tabelle 3.3.: Verschiedene Komponenten und ihre, jeweilige Laufzeitumgebung

Komponente	Laufzeitumgebung
Quellcode	Übersetzer
Maschinencode	Betriebssystem
Bytecode	Interpreter
Interpreter	Betriebssystem
Container Runtime	Betriebssystem, Orchestrator
Orchestrator	Cloud / Virtuelle Maschinen / Hardware
Cloud / Virtuelle Maschinen	potenzielle weitere virtuelle Layer => Hardware

ob solche Daten eher der Laufzeitumgebung und somit der nachfolgenden Environment Region zuzuordnen sind. [4]

3.1.3. Laufzeitumgebung Metadaten

Dieser Bereich soll alle notwendigen Informationen über die Umgebung, in der Workflows ausgeführt werden, enthalten. Dies kann mehrere Vorteile bringen. Eine bessere Nutzung der Ressourcen könnte entstehen, indem sich Workflows hinsichtlich ihrer Parallelität zum Beispiel adaptiv an die entsprechende Clustergröße anpassen. Ein weiterer Vorteil, welcher hierdurch ermöglicht wird ist, dass durch diese Daten Experimente vergleichbarer gemacht werden können. Diese Arbeit betrachtet den Begriff Laufzeitumgebung wie in Tabelle 3.3 auf verschiedenen Ebenen und konzentriert sich auf die Extraktion der Metadaten von Container Engine bis Cloud. Weiter können in der Laufzeitumgebung installierte Komponenten in diesen Bereich fallen. Hier kann zum Beispiel festgehalten werden, unter welcher URL man ein Monitoring Dashboard erreichen kann. Ein interessanter Aspekt kann hierbei sein mittels Metadaten (base URL, workflow timestamp, dashboard name, ...) URLs zu erzeugen mit denen ein Nutzer Dashboards aufrufen kann, um die Auslastung des Clusters zu bestimmtem Zeiten seines Workflows zu zeigen. Eine weitere mögliche Betrachtung ist es die Zeitreihen Daten zu archivieren und in der Metadaten-Datenbank den Ort des Archivs abzulegen.

[4] Die Entscheidung für diese Arbeit befindet sich in Kapitel 4.2.2 auf Seite 26.

3.1.4. Experiment und Workflow Metadaten

Diese Metadaten dienen der Anreicherung von Informationen über die genutzten Technologien wie zum Beispiel, welche Workflow Engine zum Einsatz gekommen ist. Art und Aufbau der jeweiligen Workflow sowie ihr Ziel bzw. Nutzen für das Experiment sowie die Dauer der einzelnen Teilschritte wird hierbei genauso festgehalten. Informationen darüber wo Beschreibungen eines Workflows oder Artefakte der Experimente zu finden sind, kann ebenfalls in den Aufgabenbereich dieser Region fallen. Diese Region kann die historische Entwicklung von beispielsweise einzelnen Fragestellungen im Bezug auf die Daten oder aber auch die Entwicklung der Laufzeiten betrachtet werden.

3.2. Diskussion bestehender Ansätze

Es existieren verschiedene Lösungsansätze für das vorliegende Problem. Nachfolgend sollen diese in Kürze vorgestellt werden und untersucht werden, welche Technologien und Techniken zum Einsatz kommen.

Metacat Metacat [5] ist eine Technologie, welche vom Unternehmen Netflix [6] entwickelt wurde und es unter anderem erlaubt Metadaten aus und über verschiedene Systeme mittels sogenannter Konnektoren nutzbar zu machen. Dabei ist der Quell Metdatenstore die sogenannte "Source of Truth". Es kommt ein Event-basiertes System zum Einsatz, um über Änderungen der Datensätze informiert zu werden. Schema Metadaten werden in diesem System mittels Elasticsearch durchsuchbar gemacht. [2]

Metastore Prabhune beschreibt in seiner Dissertation ein Generisches und Adaptives Metadaten Repository für wissenschaftliche Experimente. Dabei wird, neben vielen weiteren Aspekten, in dieser Arbeit gezeigt, wie mittels Representational State Transfer (REST)-API Schemata an eine Registry geschickt und in einer ArangoDB[7], sogenannte *Collections* angelegt werden. Es verfügt darüber hinaus über einen Elasticsearch[8] Konnektor wodurch es die Metadaten in ein Elasticsearch speisen kann und hiermit eine Volltext Suche auf den Metadaten erlaubt. In seiner Arbeit beschreibt er weiterhin, wie Endpunkte für akzeptierte Schemata, für das Schreiben von Metadaten automatisiert zur Verfügung gestellt werden können. [26]

ModelDB ModelDB[9] ist ein Projekt am *MIT Computer Science and Artificial Intelligence Laboratory* und beschreibt sich selbst als System für das Management von Machine Learning Modellen. Dabei unterstützen sie SparkML[10] sowie Scikit-learn[11].[23] Ein Problem hierbei ist jedoch, dass das Projekt kaum noch Aktivitäten aufweist und entsprechende

[5] https://github.com/Netflix/metacat
[6] https://www.netflix.com/de/
[7] Not only SQL (NoSQL)-Technologie siehe https://www.arangodb.com/
[8] https://www.elastic.co/de/products/elasticsearch
[9] https://senselab.med.yale.edu/modeldb/
[10] https://spark.apache.org/docs/latest/ml-guide.html
[11] https://scikit-learn.org/stable/

Versions-Abhängigkeiten der Machine Learning Frameworks verlangt werden. Für die Speicherung der Metadaten wird hierbei eine SQLite[12] [13] Datenbank verwendet, welches eine Skalierung und Entwicklung eines Mehrbenutzersystems erschwert, da sie die Daten in einer Datei hält.

Schelter et. al. Machine Learning Tracking Schelter et. al. beschreiben in ihren beiden Arbeiten [31], sowie [32] Möglichkeiten die das Metadaten Tracking ermöglichen. Sie beschreiben ein Datenmodell und eine Architektur Metadaten aus Machine Learning Experimenten zu extrahieren. Dabei beschreiben sie in ihrem Schema fünf Bereiche namens TrainingRun, Model-Metadata, Evaluation, Prediction-Metadata sowie Dataset-Metadata. Sie beschreiben wie sie mittels sogenannter High-Level Clients automatisiert Metadaten aus Frameworks wie SparkML, scikit-learn, MXNet[14] und UIMA [15] extrahieren und in einem zentralisierten Dokumenten-basierten Metadaten-Store mit REST-API persistieren. Sogenannte Low-Level Client erlauben die Integration in Java und Python und ermöglichen es dem Benutzer explizit Metadaten zu schreiben und zu suchen. Diese sind auto generierte REST-Clients. Weiterhin erlaubt das System die Integration von Daten aus OpenML[16]. Das Backend des Systems wird in einer als serverless genannten Art in der Amazon Web Services (AWS) Cloud[17] ausgeführt. [32] [31]

3.3. Nutzungsszenarien und Anforderungen an das zu entwickelte System

Die zu konzipierende Software soll auf der beschriebenen Inovex Data Cloud Plattform (IDCP) zum Einsatz kommen. Aus diesem Grund sollen nachfolgend die Nutzungsszenarien als auch Anforderungen an das System analysiert werden.

3.3.1. Nutzungsszenarien

Diese Arbeit betrachtet die Speicherung und Nutzung von Metadaten als unterstützende Maßnahme bei der Umsetzung von Data Mining Prozessen.Aufgrund dieser Tatsache sollen im nachfolgenden Nutzungsszenarien analysiert und in Kapitel 2.1.1.1 auf Seite 3 vorgestellte CRISP-DM Modell eingeordnet werden. Hierbei wurde das Modell wie in Abbildung 3.1 auf der nächsten Seite um eine Metadaten Datenbank erweitert. Die schwarzen Pfeile symbolisieren wie auch im CRISP-DM Modell Phasenübergänge [8]. Die hinzugekommenen blauen Pfeile sollen Datenflüsse und Informationsflüsse in das und aus der Metadaten-Datenbank symbolisieren. In jeder Phase können Informationen in die Metadaten-Datenbank gespeichert werden sowie aus der Metadaten-Datenbank abgefragt werden. Diese sollen sowohl in derselben Phase, in der nächsten Phase als auch bei

[12] https://www.sqlite.org/index.html
[13] vgl. https://github.com/mitdbg/modeldb/issues/303
[14] https://mxnet.apache.org/
[15] https://uima.apache.org/
[16] https://www.openml.org/
[17] https://aws.amazon.com/de/

Rücktransitionen in eine ältere Phase zur Beantwortung verschiedenster Fragestellungen unterstützen.

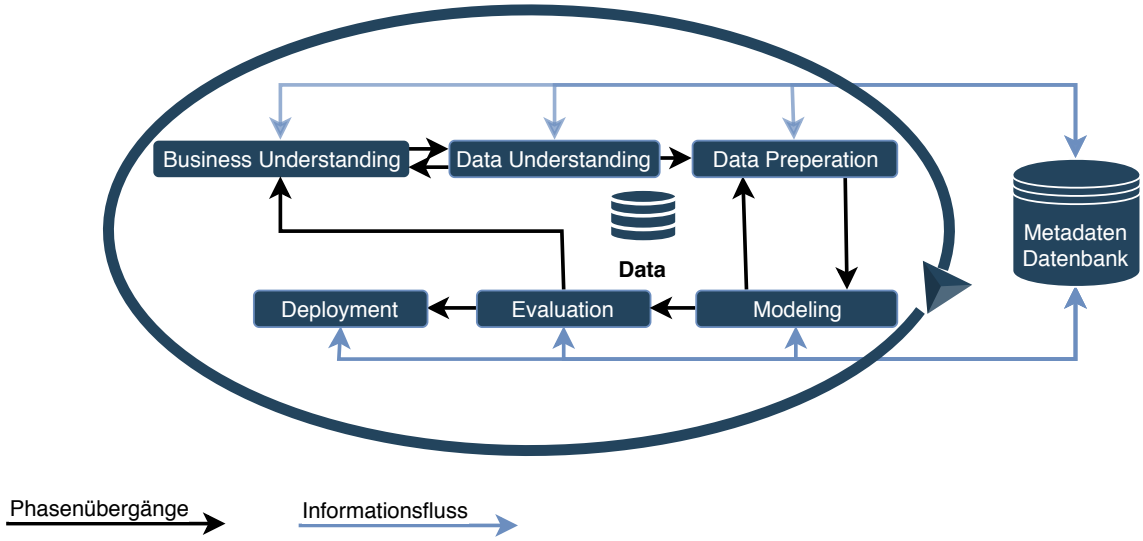


Abbildung 3.1.: Nutzungsszenarien des Systems anhand des CRISP-DM Modells.

3.3.2. Anforderungsanalyse

Nachfolgend sollen die wichtigsten Anforderungen an das System definiert werden:

1. Die Komponenten des Systems müssen auf beliebigen System lauffähig, verteilbar, skalierbar und einfach zu installieren sein
2. Es muss dem Nutzer ermöglicht werden, Datensätze und Resultate zu speichern und einfach zu beschreiben. Änderungen an den Datensätzen und Resultaten müssen automatisiert Aktualisierungen der entsprechenden Metadaten zur Folge haben können.
3. Das System erlaubt CR(UD) Operationen für Metadaten. Es müssen zu jeder Zeit beliebige Teilinformationen geschrieben werden können und alle Metadaten müssen wiedergefunden und zugeordnet werden können
4. Das System muss die Ausführung von Workflows ermöglichen und bei Aufkommen neuer Daten Event-basiert reaktiv Workflows starten können und hierbei Metadaten schreiben
5. Data Science Experimente werden in verschiedensten Sprachen entwickelt. Das System sollte hinlänglich unterstützter Sprachen und Schemata erweiterbar sein
6. Die Datenbanklogik soll von allen Client-Komponenten verwendet werden können
7. Die Persistenz Technologie muss austauschbar sein und darf keine Änderung am Applikationscode zur Folge haben

8. Eine automatisierte Metadaten Extraktion wird zugunsten nicht entstehender Abhängigkeiten von Frameworks und Bibliotheken aus den Domänen der Benutzer, wie es in Eingangs erwähnten Technologien teilweise vorkommt, nicht gefordert. Es dürfen nur Bibliotheken zum Einsatz kommen, welche für die Umsetzung des Systems vonnöten sind.

9. Das System muss so konzipiert werden, dass Benutzer auf einfache Weise neue Schemata hinzufügen können und das System erweitern kann.

10. Die Systemarchitektur soll für einen Benutzer transparent sein. Die Benutzung so einfach wie möglich gehalten werden.

11. Das System muss in der Lage sein, bestimmte Metadaten selbständig zu erfassen und diese zu persistieren

12. Es soll auf leichte Weise ermöglicht werden verschiedenste Metadaten Komponenten (Logging, Monitoring, Infrastruktur, Applikationen, Schnittstellen, ...) zu beschreiben.

4. Konzeption des Informationssystems

Das Konzept der Systemarchitektur, der Komponenten sowie des Datenmodells soll Inhalt dieses Kapitels sein. Zu Beginn soll eine Datenbanktechnologie ausgewählt werden und hierbei untersucht werden, welche Folgen dies für die Konzeption des Schemas bedeutet.

4.1. Wahl der Datenbanktechnologie und Einfluss auf den Modellierungsansatz des Datenmodells

Zu Beginn wurde in einem ersten Versuch mittels Entity Relationship (ER)-Modellen in relationaler Weise modelliert. Hierbei hat sich gezeigt, dass sich für das vorliegende Problem der Modellierung aufgrund der dynamischen Ausprägungen der Datenelemente des Modells eine flexiblere Technologie gebraucht wird. Ein Aspekt des langfristigen Fokus dieses Metadaten-System besteht daraus, die Entwicklung von Experimenten über einen größeren Zeitraum hinweg zu betrachten, sowie viele Experimente zu vergleichen. Eine weitere Sichtweise war es, dass eine Metadaten-Beschreibung eines Workflows als Dokument aufgefasst werden kann und zu einem späteren Zeitpunkt von einem anderen System ausgelesen bzw. als Konfiguration verwendet werden könnte. Aus diesem Grund wurden Datenbanken aus dem Bereich der NoSQL Technologien betrachtet, welche ein solches flexibleres Datenmodell sowie die Ablage als Dokument erlauben. Aus den vielen am Markt erhältlichen Technologien wurden die beiden Technologien MongoDB und Elasticsearch in die engere Auswahl genommen. Diese beiden Technologien, die Entscheidung sowie der Einfluss werden im nachfolgenden beschrieben.

4.1.1. MongoDB

MongoDB[1] ist eine freie, Dokumenten-basierte, verteile Datenbank. Durch die Ablage als Javascript object notation (JSON) Dokumente wird es ermöglicht flexibel Daten abzulegen und das Datenschema zu ändern. [39]

4.1.2. Elasticsearch

Eine weitere Überlegung bestand darüber hinaus Elasticsearch zu verwenden. Elasticsearch ist eine verteilte Dokumentendatenbank und eine Volltext Suchmaschine. Es bietet die Möglichkeit Schemata vorzugeben. Weiterhin bietet es die Möglichkeit dynamisch Daten einzupflegen und die Schemata für diese automatisch zu generieren. Es bietet eine mächtige

[1] https://www.mongodb.com/de

Suche, insbesondere Volltextsuche, welches einem Benutzer ermöglicht nach beliebigen Inhalten zu suchen. Es bietet einfache Skalierbarkeit und Verteilbarkeit. [10]

4.1.3. Entscheidung

Es wurde für die Entscheidungsfindung der Datenbanktechnologie für diese Arbeit, sowohl ein Testclient für MongoDB, als auch für Elasticsearch geschrieben. Beide Technologien arbeiten Dokumenten-basiert und erlauben es CRUD Operationen auf JSON Dokumenten durchzuführen. [39] [10] Es hat sich herausgestellt, dass Elasticsearch hinsichtlich der Suche, im Vergleich zu MongoDB, für diese Arbeit schneller zu verwenden ist. Das automatische Mapping von Dynamisch eingetragenen Daten wurde als weiterer Vorteil von Elasticsearch empfunden. [10] Elasticsearch erlaubt es die Dokumente zu versionieren wodurch es auf einfache Weise ermöglicht wird einen historischen Durchblick zu einem Experiment zu gewinnen. Elasticsearch ist auf Verteilbarkeit und Skalierbarkeit ausgelegt, welches für die vorliegende Arbeit ein wichtiges Kriterium ist. [10] Aus den genannten Gründen wurde entschieden, das Proof of Concept mit Elasticsearch umzusetzen.

4.1.4. Einfluss

Die Wahl einer bestimmten Datenbanktechnologie hat einen Einfluss auf die Art und Weise der Gestaltung des Schemas. Nachfolgend sollen auf einige Aspekte eingegangen werden.

Clientseitiges Schema Beide NoSQL Technologien verwenden JSON als Datenformat. Für viele Programmiersprachen existiert, wie in Tabelle 4.1 abgebildet, eine leichte Möglichkeit der Serialisierung und Deserialisierung von Objekten der jeweiligen Sprache zu und aus JSON. Die ubiquitäre Verwendung dieses Schemas, liegt nahe das Datenmodell für dieses System so nah wie Möglich an JSON zu modellieren. Aus diesem Grund wurde entschieden die Modellierung Objektorientiert umzusetzen und so die Clientseitige Verwendung für den Nutzer einfach zu gestalten.

Datenbankseitiges Schema Betrachtet man nun die Möglichkeit des Automapping von Elasticsearch so führt dies dazu, dass für das Proof of Concept keine Datenbank Schemata/Mappings konzipiert werden müssen.

Referentielle Integrität Die Wahrung referentieller Integrität ist bei der Nutzung des Systems Aufgabe des Datenbank Clients. Wie in Kapitel 3.3.2 gezeigt wird auf eine Umsetzung der Update- und Delete-Funktionalität verzichtet. Dies führt dazu, dass bei der Umsetzung des Clients auf eine Integritätswahrung vorerst nicht geachtet werden muss. Der Verzicht auf Löschung zugunsten der Integritätswahrung beschreiben auch Schelter et. al. in [31].

Identifier Metadaten Diese Metadaten sollen es ermöglichen, zu einem Experiment gehörende Metadaten zu identifizieren. Dies kann wie in Abbildung 4.1 auf der nächsten Seite

Tabelle 4.1.: Technologien zur Verwendung von JSON in verschiedenen Programmiersprachen

Sprache	Technologie	Referenz
Python 2	Nativ	[2]
Python 3	Nativ	[3]
Python 3	jsonpickle	[4]
Java	javax/json	[5]
Go	Nativ	[6]
Rust	Serde	[7]

zu sehen auf verschiedenen Arten erreicht werden. Tabelle 4.2 auf der nächsten Seite stellt hierbei die Vorteile und Nachteile der einzelnen Arten gegenüber.

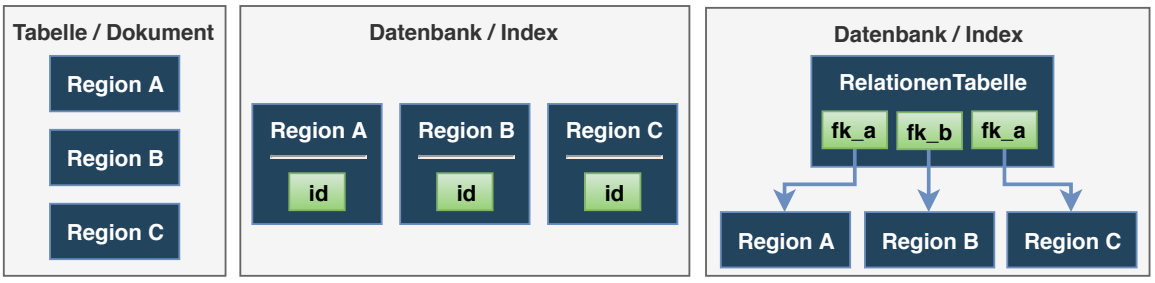


Abbildung 4.1.: Möglichkeiten der Schlüsselkonzeptionen

4.2. Konzeption eines Datenmodells

Das zu konzipierende Informationssystem soll es ermöglichen beliebige Informationen zu speichern. In Kapitel 3.1 auf Seite 13 wurden vier Metadaten Bereiche erarbeitet. Kapitel 4.2.1 auf der nächsten Seite erarbeitet ein für alle gültiges Metamodell und Kapitel 4.2.2 auf der nächsten Seite ordnet diese einzelnen Kategorien in ein Regionenmodell ein.

Tabelle 4.2.: Gegenüberstellung verschiedener Referenzierungen

Typ	Beschreibung	Vorteile	Nachteile
Denormalisiert	Speicherung aller Partitionen in einem Dokument/Tabelle.	Zugriff und Aggregation einfach	häufige Updates desselben Dokuments / Tabelle
Globaler Schlüsselraum	Identifier bilden einen globalen Schlüsselraum. Jedes dem System hinzugefügte Datum enthält diese Identifier und kann so eindeutig zugeordnet werden.	Teildaten können einfach hinzugefügt werden, ohne eine Relationentabelle zu pflegen	Suche unter Umständen komplexer, Indizierung wichtig
Normalisiert	In einem Dokument bzw. einer Tabelle werden Referenzen auf die Partitionen gespeichert.	Teilbeschreibungen leichter auffindbar / geringerer Overhead da Lookup mit jeweiligem Key	Aggregation aller Teile komplexer / zeitaufwendiger, häufige Updates der Linktabelle/Realtionentabelle

4.2.1. Metamodell

Für die Entwicklung eines Datenmodells des Systems soll zunächst ein Metamodell konzipiert werden. Dieses wurde als Kompositum[8] umgesetzt. Das Modell besteht hierbei aus zwei hintereinander geschalteten Komposita Ebenen. Die erste, das sogenannte Metadata-Objekt besteht aus einem Core-Objekt und einem Region-Objekt. Wie Abbildung 4.2 auf der nächsten Seite zu entnehmen sind beides wiederum Komposita. Das Core-Objekt enthält hierbei Identifier-Felder sowie Core-Felder. Das Regionen-Objekt enthält einen Regionen-Core. Diese drei sind vom System vorgegeben. Der veränderliche Teil des Schemas, welcher vom Nutzer vorgegeben werden können soll, soll mittels der RegionMetadata definiert sein.

4.2.2. Regionenmodell

Die in Kapitel 3.1 auf Seite 13 analysierten Kategorien werden nachfolgend in ein Regionenmodell überführt. Dabei wurde versucht für die einzelnen Kategorien generischere Begriffe zu finden. Tabelle 4.3 auf Seite 28 veranschaulicht hierbei die Einordnung der Kategorien in das Modell.

[8] Ein Kompositum ist ein Entwurfsmuster für Baumartige Strukturen, mit welcher sich Teil-Ganze Beziehungen modellieren lassen. [13]

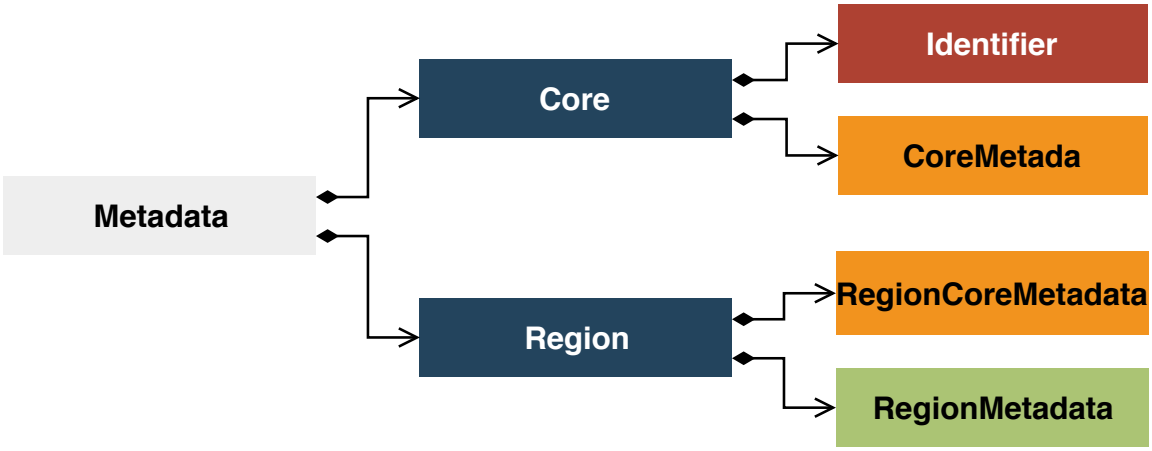


Abbildung 4.2.: Metamodell des Regionenmodells. Zu sehen ist die Zweistufige Komposition, sowie der Aufbau des Core und der Region.

Bei der Konzeption des Modells wurde der Fokus auf zwei Bereiche gelegt. Der erste Fokus ist eine gewisse Mindestvorgabe des Schemas vorzugeben, ohne den Nutzer in der Gestaltung seines Schemas einzuschränken. Aus diesem Grund sieht das Modell verschiedene Schichten vor. Prabhune beschreibt in [26] bei der Entwicklung seines Systems, einen Core und einen Expansion Layer. Die Idee der Unterteilung in verschiedene Layer wurde auch beim vorliegenden System übernommen und nennen sich wie in der Beschreibung des Metamodells in Kapitel 4.2.1 auf der vorherigen Seite gezeigt Core sowie Region. Der zweite Fokus wurde darauf gelegt, dass jeder am Bereich des Workflows im Idealfall nur eine Region des Schemas laden muss, um die Benutzung aus Nutzersicht so einfach wie möglich zu halten. So kann ein Crawler die EnvironmentRegion laden, im Training die RunResult Region oder in einer Dataset Beschreibung die DatasetRegion verwendet werden. Sinnvoll kann diese Unterteilung auch im Hinblick darauf erscheinen, dass die unterschiedlichen Bereiche von unterschiedlichen Entwicklern genutzt werden. Zum Beispiel das Deployment des Systems durch einen Adminstrator, die Entwicklung eines Crawlers für eine andere Infrastruktur-Technologie durch einen DevOps-Entwickler oder die Beschreibung der Trainings Metadaten durch einen Data Scientist.

Pflichtfelder des Modells Das Metamodell in Abbildung 4.2 zeigte, das jedes Metadaten-Objekt, welches dem System hinzugefügt werden soll, einer Komposition aus einem Core-Objekt sowie einem Regionen-Objekt entspricht. Die jeweils darauffolgenden Objekte ebenfalls Komposita sind. Hierraus folgt, dass eine Umsetzung des Schemas auf technischer Ebene dies berücksichtigen und Benutzer diese Felder setzen müssen.

Aufbau des Regionenmodells Das Modell enthält wie in Abbildung 4.3 auf Seite 35 gezeigt zwei Schichten mit jeweils zwei Bereichen und vier unterschiedlichen Regionen. Die innere Schicht besteht aus zwei Bereichen und dient der Identifizierung hinzugefügter Metadaten. Hiermit soll es ermöglicht werden gewisse Vorgaben zu erzwingen. Diese können zum Beispiel aus identifizierenden Schlüsseln (Experiment ID, Workflow ID, ...), Informationen über Zugriffsrechte oder auch Informationen über Herkunft der Daten bestehen. Diese

Tabelle 4.3.: Überführung der Kategorien aus Kapitel 3.1 auf Seite 13 in das Regionenmodell

Kategorie	Region	Zweck
Dataset	Dataset	Speicherung von Datensatz Metadaten. Mögliche Entwicklung eines Datensatzrepositorys
Machine Learning	RunResult	Speicherung von Metadaten aus Läufen wie zum Beispiel eine Datenanalyse oder ein Training sowie Resultaten.
Laufzeitumgebung	Environment	Beschreibung verschiedenster Laufzeitumgebungen und Infrastruktur-orientierter Metadaten
Experiment und Workflow	ExperimentWorkflow	Metadaten ein Experiment und den Workflow betreffend.

beiden werden *Identifier Metadata* und *Core Metadata genannt*. Alle dem System hinzuzufügenden Informationen müssen über diese Elemente verfügen. Weiterhin verfügt der Core über das Feld *user_defined_data*. Dieses Feld, auf welches noch genauer eingegangen wird, ermöglicht die Hinzunahme beliebiger Nutzerdaten. Schelter et. al. beschreiben in ihrer Arbeit die von 'open' Data Modellen inspirierte Möglichkeit Nutzern beliebige Applikations Annotationen zu erlauben[31]. Diese Idee wurde in der vorliegenden Arbeit übernommen und in jedem Core, sowohl im Globalen als auch in den Regionen, eingebaut. Darüber hinaus kann ein Nutzer dies in seinem Regionenschema ebenfalls einarbeiten. Die äußeren beiden Bereiche bilden die Regionen Schicht. Dabei existiert für jede Region ein für alle Domänen gemeinsam genutztes Core Schema. Dieses ermöglicht es gemeinsame Metadaten abzubilden. Im äußersten Bereich können vom Nutzer verschiedene Schemata für die verschiedensten Problemdomänen definiert und somit das System erweitert werden [9]. Eine Region entspricht in Abbildung 4.3 auf Seite 35 einem Orangenen und Grünem Viertel und entspricht der in Abbildung 4.2 auf der vorherigen Seite dargestelltem Regionen Klasse. Jede Region kann hierbei als eigenständiges Schema verstanden werden. Zum Beispiel kommt für die RunResult Region in dieser Arbeit das von Schelter in [31] beschriebene Schema, in einer leicht angepassten Version, zum Einsatz. So ist es weiterhin möglich, dass zu einer bestimmten Region mehrere Schemata existieren, welches in Abbildung 4.2 auf der vorherigen Seite an der gestuften Darstellung des Domänen Bereichs veranschaulicht wurde. Dies findet sich in dieser Arbeit mit den Schemata für Kubernetes und dem Openstack Schema wieder.

[9] Umsetzung der Erweiterung siehe Kapitel 5.1.1 auf Seite 37

Globaler Schlüsselraum mittels Experiment ID In Kapitel 4.1.4 auf Seite 24 wurde beschrieben wie Experiment und Workflow ID Metadaten genutzt werden können, um alle Metadaten zu einem Experiment/Workflow wiederzufinden. In diesem System soll die Variante des Globalen Schlüsselraum umgesetzt werden. Hierzu enthält jedes Metadata-Objekt entsprechende Felder, um Schlüsselwerte abzulegen.

Generik vs. Spezifizierung Das Schema erlaubt es im Core Layer beliebige Daten abzulegen. Dies wird mittels eines String Feldes ermöglicht. Hierbei ist jedoch der Informationsgehalt der abgelegten Daten nicht anhand des Schemas ersichtlich, sondern es müssen die Daten analysiert werden. Das Modell bietet auf der anderen Seite, wie auch schon in [31] beschrieben, aus Gründen des Informationsgehalts und eventuellen maschinellen Weiterverarbeitung der Metadaten, mit dem Domain-Layer die Möglichkeit, spezifisches und explizites Domänenwissen in die Metadaten-Datenbank zu speisen. Dies erfordert jedoch einen Mechanismus, mit welchem die entsprechenden Schemata dem System bekannt gegeben werden können und dieses dadurch erweitern [10].

4.2.3. Schemaumsetzungen

In Kapitel 3.1 auf Seite 13 wurden vier für diese Arbeit relevante Regionen ausgearbeitet, deren Aufbau nachfolgend erläutert wird. Abbildung 4.5 auf Seite 31 sowie 4.4 auf der nächsten Seite zeigen wie sich die konzipierten Schemata an dem Metamodell orientieren und dieses in der jeweiligen Domäne umsetzen. Es werden zu jeder Region ein paar kleine Ausschnitte in Form von Tabellen gezeigt, die vollständigen Schemata finden sich ab Anhang A.17 auf Seite 82

4.2.3.1. Dataset Region

Diese Region wurde angelehnt an die Klasse Dataset-Metadata des von Schelter et. al. beschriebenen Schemas[11]. Dabei bleibt die Klasse Dataset-Metadata Teil der RunResult Region und existiert darüberhinaus als eigenständige Region. Erscheint diese Extraktion als eigenständige Region auf den ersten Blick redundant, ermöglicht sie jedoch die Entwicklung eines Dataset-Repository.

4.2.3.2. Environment Region

Diese Region besteht aus mehreren Schemata. Es existieren Schemata zur Beschreibung der Infrastruktur, insbesondere des Cluster Aufbaus sowie des Monitorings. Im vorliegenden Falle ist dies Openstack, Kubernetes, Docker, der Elasticsearch, Logstash, Kibana (ELK)-

[10] Kapitel 7 auf Seite 63 beschreibt in Kürze eine mögliche Erweiterung des Systems.
[11] vgl. Abbildung A.1 auf Seite 69

Stack [12] sowie Grafana [13] und Prometheus [14]. Es wurde dabei versucht die Schemata nahe an den jeweiligen original Schemata zu belassen. Weiterhin ermöglicht es die Beschreibung der Applikationsinfrastruktur sowie der Beschreibung vorhandener Systemschnittstellen wie zum Beispiel eines User Interfaces in Form eines Angular[15] Frontends oder Jupyter[16] Notebooks.

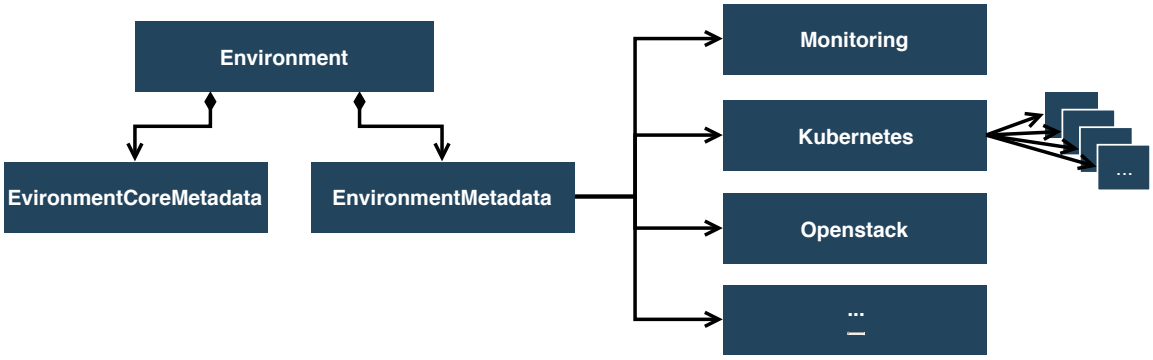


Abbildung 4.4.: Aufbau der Region Environment. Zu sehen ist wie einzelne Objekte der Region in sich wiederum Objekte beinhalten und so eine teils komplexe Struktur umsetzen.

Beispielhafte Felder der Region Nachfolgend sollen ein paar Felder des Environment Objekts veranschaulicht werden. Diese können Tabelle 4.4 auf der nächsten Seite entnommen werden.

4.2.3.3. Experiment und Workflow Region

Kapitel 3.1.4 auf Seite 18 zeigte, dass Metadaten auf Ebene des gesamten Experiments und auf Ebene des/der technischen Workflow(s) existieren. Ein Experiment besteht, in Anlehnung an das Metamodell wie in Abbildung 4.2 auf Seite 27 in Kapitel 4.2.1 auf Seite 26 gezeigt, aus einen ExperimentCore sowie einem ExperimentMetadata. Workflow Metadaten können, wie in Abbildung 4.5 auf der nächsten Seite zu entnehmen, innerhalb eines ExperimentMetadata beschrieben werden. Sollen mehrere verschiedene Workflows innerhalb eines Experimentes beschrieben werden, so kann ein Nutzer ein weiteres MetadataObject erzeugen und dem System hinzufügen.

[12] Logstash dient zur Zusammenführung von verschiedensten Datenströmen, wie dies bei Logging Informationen häufig der Fall ist und Kibana ermöglicht das Anzeigen von in Elasticsearch gespeicherten Daten. - https://www.elastic.co/de/products

[13] Technologie zu Darstellung und Analyse von Time Series Daten, welche von verschiedensten Monitoring Komponenten extrahiert werden können. - https://grafana.com/

[14] Technologie zur Extraktion (und weiteren Möglichkeiten) von Metriken der Infrastruktur. - https://prometheus.io/

[15] https://angular.io/

[16] https://jupyter.org/

[17] Monitor ist vom Typ Union.

Tabelle 4.4.: Beispielhafte Felder des Environment Objekt. Kubernetes-Felder orientiert an Quelle: [17]

Klasse	Feldname	Feldtyp	Pflichtfeld
Openstack	openstack_uuid	string	☑
Monitor	grafana	Grafana	⊖ OR ☑[17]
Grafana	grafana_service_url	string	☑
Node	noide_name	string	☑
Pod	pod_scheduled_node	string	⊖

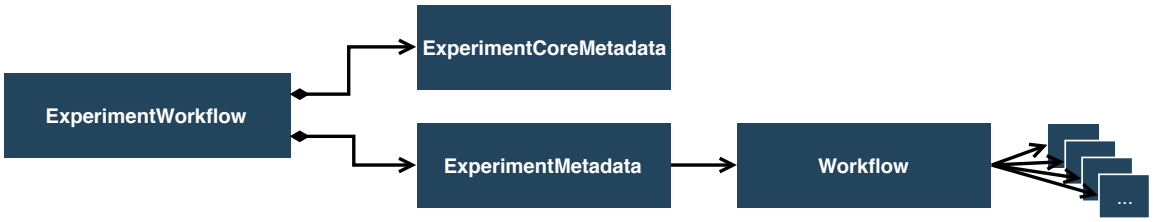


Abbildung 4.5.: Aufbau der Region ExperimentWorkflow. Zu sehen ist, wie sich die Region am Metamodell einer Region orientiert und dieses Umsetzt.

Beispielhafte Felder der Region Innerhalb der genannten Objekte sollen verschiedenste Metadatan gespeichert werden. So kann zum Beispiel das Feld *Schritt* des Objekts *Workflow* Metadaten zu den Schritten enthalten wie zum Beispiel Name, Input-Argumente oder auch welche Container ausgeführt wurden. Bei der Konzeption dieses Schemas wurde an der Beschreibung des Datenmodells der Engine Argo [3] orientiert und dieses in ein weniger komplexes Format überführt, indem die Referenz auf ein Template Objekt des Argo Datenmodells explizit als Felder der Klasse Schritt modelliert wurde. Weiterhin wurden nur benötigte Felder übernommen. Zum Beispiel wird in dieser Arbeit mit Artefakten gearbeitet die innerhalb eines (Amazon) Simple Storage Server (S3) kompatiblen ObjectStorgage liegen. Aus diesem Grund wurde die Klasse S3Artifact, des Argo Datenmodells übernommen. Da jedoch zum Beispiel kein HDFS[18] Artefakt zum Einsatz kommt, wurde auf die Übernahme solcher Felder vorerst verzichtet. Es wird an einigen Stellen ersichtlich wie Teile der Environment Region, wie zum Beispiel die Klasse Container in diesem Schema wiederverwendet werden[19].Tabelle 4.5 auf der nächsten Seite zeigt einen Ausschnitt des ExperimentWorkflow Objekts.

[18]https://hadoop.apache.org/docs/current1/hdfs_design.html
[19]Auch im originalen Schema werden hier Verweise auf die Kubernetes Schemadefinition beschrieben.

Tabelle 4.5.: Beispielhafte Felder des ExperimentWorkflow Objekts. Workflow-Felder orientiert an Quelle: [3]

Klasse	Feldname	Feldtyp	Pflichtfeld
WorkflowMetadata	workflow_engine	string	⊖
WorkflowMetadata	workflow_type	string	⊖
Step	name	string	☑
Step	Container	Environment.Contair	☑

4.2.3.4. RunResult Region

Diese Region soll dazu dienen spezifische Informationen eines Programmes (zum Beispiel einem Training) bzw. eines Resultats (zum Beispiel einem trainierten Modell) zu speichern. In Kapitel 3.1.2 auf Seite 14 wurden nötige Grundlagen und Termini für maschinelles Lernen vorgestellt. Kapitel 3.2 auf Seite 18 zeigte auf, dass Schelter et. al. in Ihrer Arbeit *Automatically Tracking Metadata and Provenance of Machine Learning Experiments*, ein passendes und in Abbildung A.1 auf Seite 69 zu sehendes Schema beschreiben. Dieses Schema ist von einigen wenigen Änderungen abgesehen ausreichend für die Anforderungen dieser Arbeit und soll für diese Region nahezu unverändert zum Einsatz kommen. Kapitel 3.1.2 auf Seite 14 beschrieb die Problematik der Zuordnung von Sprachlaufzeitumgebung sowie Framework/Libraries. Es wurde die Frage aufgestellt, ob diese nicht eher der Environment Region zuzuordnen sind. Hierbei kann gegenargumentiert werden, dass eine Einordnung dieser Daten in die Machine Learning Metadaten die Komplexität des Schemas vereinfacht. Ein Benutzer muss innerhalb eines Trainings nach wie vor nur den RunResult Teil des Schemas laden. Würde man nun die genannten Metadaten als Teil der Environment Region beschreiben, so müsste er auch dieses laden und in der RunResult Region entsprechende Referenzen ablegen. Betrachtet man nun das Schema von Schelter et. al. so wird ersichtlich, dass diese die Beschreibung verwendeter Framework-Informationen als Teil der Model-Metadata betrachten. Eine Speicherung der zum Einsatz kommenden Sprache/Laufzeitumgebung sieht das Schema lediglich als einfache string Map mittels des Feldes environment in der TrainingRun Klasse. Es wurde somit das Schema von Schelter et. al., mit den in Tabelle 4.6 auf der nächsten Seite gezeigten Änderungen, in das zu konzipierende Gesamtschema aufgenommen.

4.3. Konzeption einer Architektur

Nachfolgend sollen die einzelnen Komponenten des Systems vorgestellt werden. Hierbei wird die Architektur zuerst ohne Technologiewahl beschrieben. Diese werden in Kapitel

Tabelle 4.6.: Übersicht über die Änderungen am Schema von Schelter et. al.

Klasse	Feldname	Änderung
TrainingRun	environment	entfällt => modelliert durch EnvironmentRegion
ModelMetadata	language	Für die Erstellung dieses Modells zum Einsatz kommende Sprache
ModelMetadata	languageVersion	Version der zum Einsatz kommenden Sprache

5.1 jeweils anhand der spezifischen Komponente vorgestellt. Die jeweiligen Zahlen zu Beginn der Paragraphen entsprechen denen in der Abbildung 4.6.

2.2 und 4.2 Workflow Engine und Event Handling In Kapitel 3.3.2 auf Seite 20 wurde die Ausführung von Workflows sowie das reaktive Ausführen von Workflows bei Eintreffen neuer Daten als Anforderung definiert. Hieraus folgen für die Architektur zwei Komponenten. Eine Event Handling Komponente wartet auf eingehende Events einer Storage Komponente. Hieraus resultiert, dass die Storage Komponente in der Lage sein muss Komponenten über Änderungen zu informieren. Die beiden Komponenten können in Spalte zwei und vier[20] jeweils in der zweiten Reihe der Abbildung 4.6 auf Seite 36 entnommen werden. Für diese Komponenten können die Regionen *Experiment und Workflow* sowie *RunResult* des in Kapitel 4.2.2 auf Seite 26 vorgestellten Regionenmodells zum Einsatz kommen.

2.3 und 2.4 Datensatz-, Resultatsspeicherung und Uploadservice Sowohl Rohdaten als auch Resultate sollen in einem eventfähigen Storage, zu sehen in Spalte zwei Zeile zwei, gespeichert werden. Hierbei ermöglicht die Komponente Event Handling die reaktive Ausführung von Workflows. Ein UploadService dient als Schnittstelle zwischen Metadaten Client und Storage und ermöglicht das komfortable Hochladen von Daten und zeitgleiche Setzen von Metadaten. Für diese Komponente können die Regionen *Environment* zur Beschreibung der Storage Infrastruktur, *Dataset* zur Beschreibung von hochgeladenen Daten und *RunResult* zur Beschreibung eines Restultat, des in Kapitel 4.2.2 auf Seite 26 vorgestellten Regionenmodells zum Einsatz kommen.

5.4 Crawler Die Aufgabe des Crawlers, welcher sich in Abbildung 4.6 auf Seite 36 auf der rechten Seite befindet, kann sowohl den statischen Zustand als auch dynamische Änderungen von Cluster Zuständen ermöglichen. Hiermit kommt diese Komponente der

[20] Zählweise der Spalten und Zeilen beginnend oben links mit eins.

Anforderung nach, dass das System eigenständig Metadaten erfassen und speichern kann. Der Crawler kann zum Beispiel die Verknüpfung der Virtuellen Maschinen aus Sicht von Openstack mit den Nodes aus Sicht von Kubernetes in der Metadaten-Datenbank herstellen und somit die Experiment-Metadaten mit Informationen über den Clusteraufbau anreichern. Auch wenn es mit dem Crawler prinzipiell möglich wäre auf dynamische Änderungen zu reagieren und diese in die Metadaten-Datenbank zu schreiben wurde entschieden, dass dies in den Bereich des Monitorings fällt und das System hierfür auf ubiquitäre Lösungen zurückgreift. Hierbei können jedoch Referenzen auf entsprechende Nutzerschnittstellen [21] gespeichert werden.

3.1 bis 3.3 Monitoring und Logging Wie beschrieben soll es nicht die Aufgabe des Crawlers sein den Cluster Zustand zu überwachen. Aus diesem Grund wird eine Monitor Komponente für die Auslastung der Infrastruktur benötigt. Eine Logging Komponente ermöglicht das Zusammentragen der Applikationslogs aus den Containern heraus. Metadaten eines Experimentes können hier zu einem bestimmten Workflows speichern wo Logging und Monitoring Informationen zu einem Zeitpunkt t zu finden sind und wie entsprechende Schnittstellen anzusprechen sind. Dies kann es zum Beispiel ermöglichen aktuelle Laufzeiten und Auslastungen mit historischen Daten zu vergleichen. Für diese Komponenten kann die Region *Environment* des in Kapitel 4.2.2 auf Seite 26 vorgestellten Regionenmodells zum Einsatz kommen.

1.4 Metadaten Client Der Metadaten Client soll als Schnittstelle zwischen System und Benutzer dienen. Er soll eine einfache Benutzung des Systems erlauben und es Nutzern ohne Detailwissen über die Systemarchitektur auf transparente Weise ermöglichen Daten hinzuzufügen und zu suchen. Diese Komponente soll für verschiedene Sprachen angeboten werden können.

1.3 Metadaten Service Diese Komponente dient als Schnittstelle zwischen Client und Datenbank und erlaubt es die Datenbanklogik für verschiedene Client-Implementierungen zu verwenden. Hierbei soll die Kommunikation zwischen Client und Metadaten Service mittels gängiger Protokolle, zum Beispiel Transmission Control Protocol (TCP) oder Hypertext Transfer Protocol (HTTP) ausgeführt werden und somit auch der Service nur einmal implementiert werden.

1.2 Datenbankclient Der Datenbankclient dient der Ausführung der jeweiligen im Client angestoßenen und dem Service übergebenen Operationen.

1.5 User Schnittstellen Mittels geeigneter Schnittstellen kann ein User den Metadaten Client instanziieren und hierdurch mit dem System kommunizieren.

[21] Dies kann zum Beispiel ein konfiguriertes Grafana Dashboard sein, welches auf Daten zugreift, welche von Prometheus gesammelt wurden.

Abbildung 4.3.: Aufbau des Regionenmodells und Darstellung möglicher Regionen

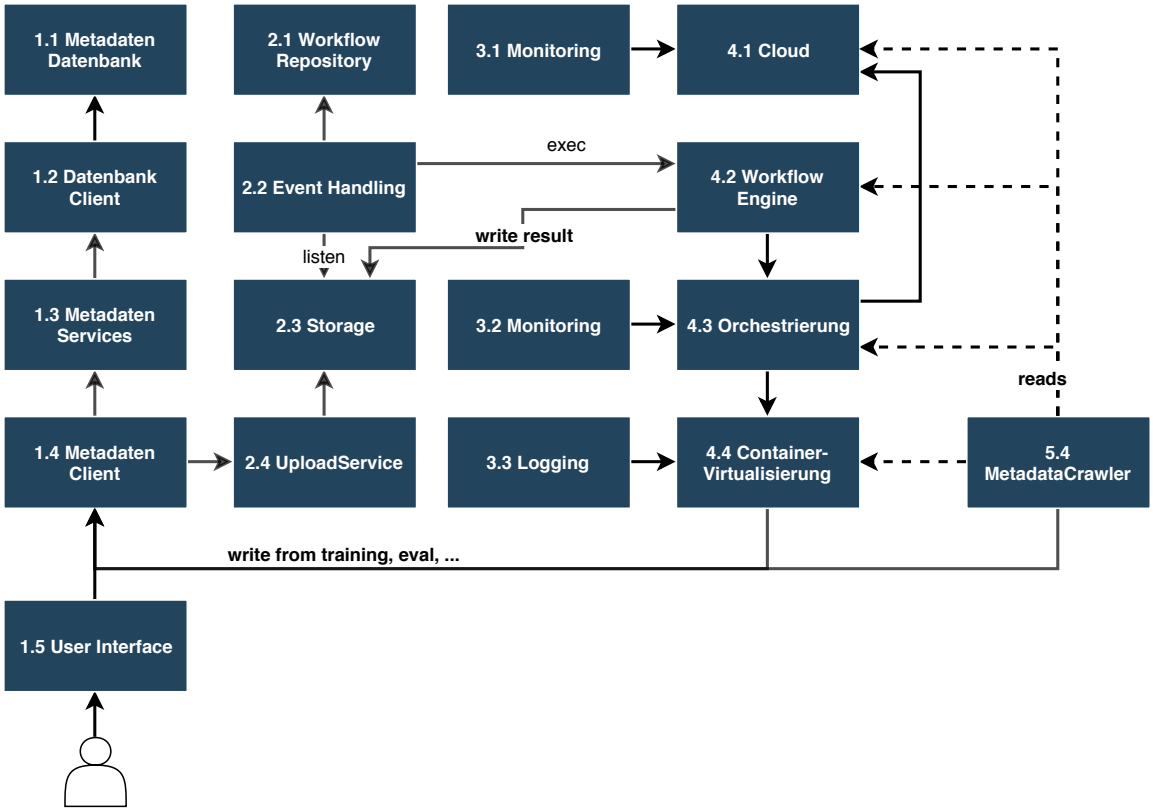


Abbildung 4.6.: Gesamtarchitektur des Systems ohne Technologien.

5. Implementierung

Nachdem in Kapitel 4 auf Seite 23 das System und das Informationsmodell konzipiert wurden, soll im nachfolgenden gezeigt werden mit welchen Technologien diese umgesetzt wurden. Abbildung 5.1 auf der nächsten Seite veranschaulicht hierbei, für welche Komponente, welche Technologie zum Einsatz kam und Kapitel 5.1 wie diese eingesetzt wurden.

5.1. Umsetzung der Komponenten

Es werden im nachfolgenden Komponentenweise die Implementierung erklärt. Hierbei wird, wann immer nötig auf Grundlagen eingegangen.

5.1.1. Umsetzung der Schema- und Schnittstellenbeschreibung sowie der Clients und Server

Die Schemabeschreibung sowie die Schnittstellenbeschreibung soll mittels der Technologie Thrift umgesetzt werden. Diese ermöglicht es anhand der Beschreibung auch entsprechende Clients sowie die Server Stubs umzusetzen und ermöglicht hierdurch ein Remote Procedure Calling (RPC) System.

Thrift Thrift[1] ist eine Bibliothek und ein Code Generator, welcher 2007 beim Unternehmen Facebook [2] entwickelt wurde und mittlerweile ein Apache [3] Projekt ist. Das Ziel von Thrift ist es sowohl die Kommunikation zwischen verschiedenen Sprachen zu ermöglichen als auch die Entwicklung skalierbarer Backenlösungen zu unterstützen. Hierbei werden die Services und ihre zugehörigen Typen in einer Datei oder mehreren Dateien beschrieben. Auf der Grundlage dieser formalen Spezifikation werden RPC Clients und Server erstellt. Die Aufgabe des Entwicklers besteht nun noch aus der Entwicklung des spezifischen Aktionscodes, welcher in der vorliegenden Arbeit aus der Implementierung sogenannter Handler besteht. [35] Thrift verfügt auf der einen Seite über viele Sprachmittel, konzentriert sich auf der anderen Seite jedoch auf Klarheit und einfache Struktur. Es existiert zum Beispiel keine Vererbung auf struct Ebene, sondern es kommt eine Komposition zum Einsatz. [35] Listing 5.2 auf Seite 39 veranschaulicht wie eine Datenklasse und ein Service mittels Thrift umgesetzt werden kann.

[1] https://thrift.apache.org/docs/
[2] https://de-de.facebook.com/
[3] https://www.apache.org/

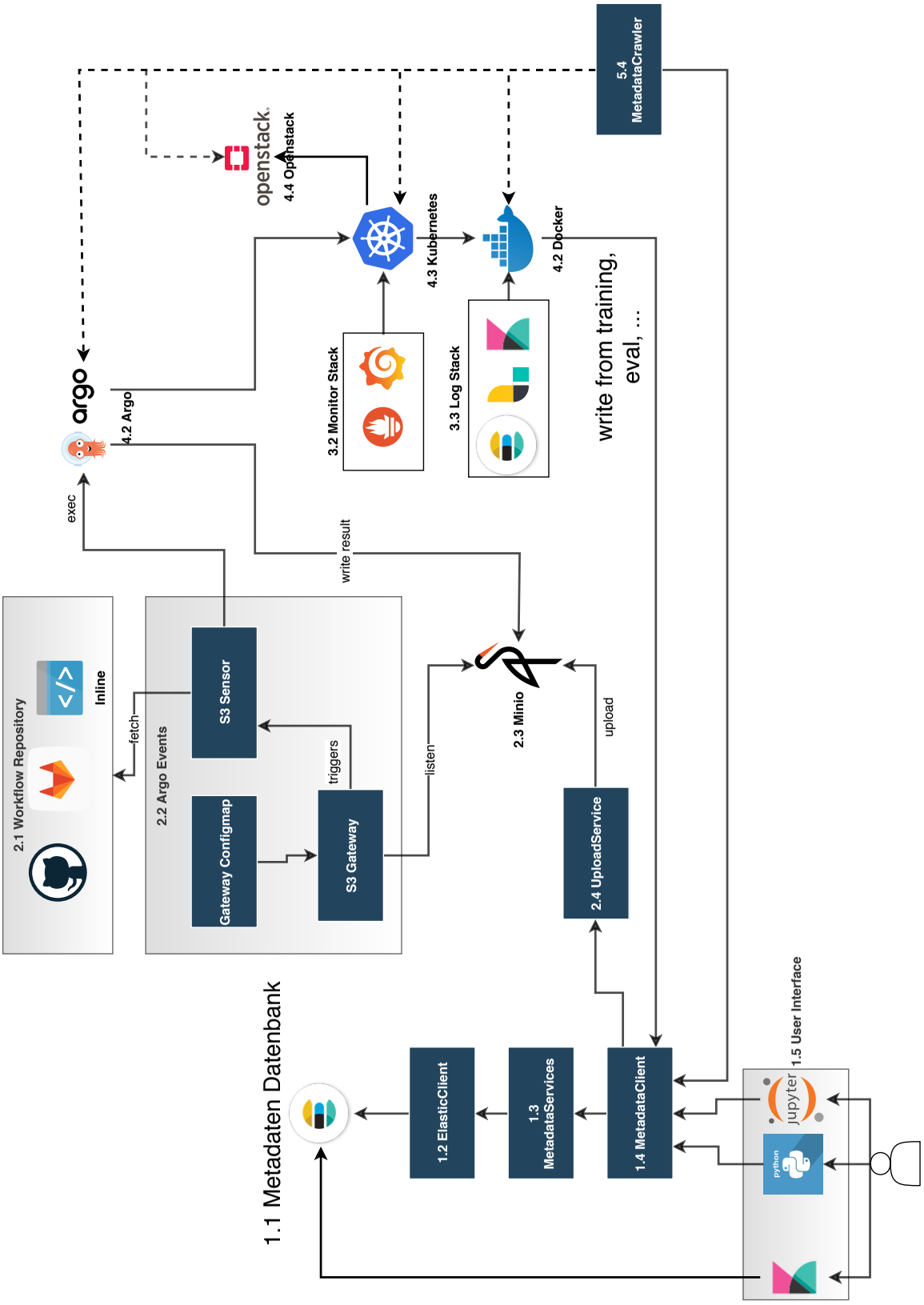


Abbildung 5.1.: Gesamtarchitektur des Systems mit Technologien

Pflichtfelder Pflichtfelder des konzipierten Schemas können in einem Thrift Schema, wie in Abbildung 5.2 anhand der Zeilen 5 – 7 zu sehen, mittels der Schlüsselworte *optional* und *required* definiert werden. Hierbei ist sichergestellt, dass ein *required* Feld immer geschrieben wird.

```
1  include "Data.thrift"
2  include "Environment.thrift"
3  include "Run.thrift"
4  include "Core.thrift"
5
6  struct MetadataObject {
7      1: required string experiment_id
8      2: required Core.CoreBaseValues core
9      3: required MetadataPartition metadata
10 }
11
12 union MetadataPartition {
13     1: Data.Data data_metadata
14     2: Environment.Environment environment_metadata
15     3: Run.TrainingRun run_metadata
16     4: Experiment.ExperimentWorkflow experiment_workflow_metadata
17 }
18
19 service IngestService {
20     IngestResult add(1: MetadataObject mdObj)
21 }
```

Listing 5.2: Ausschnitt der Schemabeschreibung sowie einer Schnittstellenbeschreibung anhand des Ingestservice mittels Thrift

Server Der Metadaten Service ist eine logische Komponente, welche aus verschiedenen voneinander unabhängigen Services besteht. Diese Arbeit sieht, wie der Abbildung 5.2 auf der nächsten Seite zu entnehmen, die prototypische Umsetzung eines SearchService und eines IngestService vor. Wie in Abbildung 5.2 können diese in einem Thrift Schema beschrieben werden. Die beiden genannten Services ermöglichen das Hinzufügen eines Metadaten-Objektes sowie das Suchen von Metadaten zu einer bestimmten Experiment-ID. Bei der Umsetzung der Server kam der TSimpleServer zum Einsatz. Abbildung 5.2 auf der nächsten Seite zeigt eine Klasse *ServerBase*, welche für die einzelnen Server der Services als Basisklasse dienen soll. Jeder Server eines Services erbt von diesem und kann in der jeweiligen run Funktion die Funktion serve der Basisklasse aufrufen. Weiterhin importieren sie den von Thrift generierten Code sowie den jeweiligen programmierten

handler. Diese werden in den Servern instanziiert und führen die Logik des jeweiligen Services wie zum Beispiel einen Such-Aufruf des Datenbankclients aus.

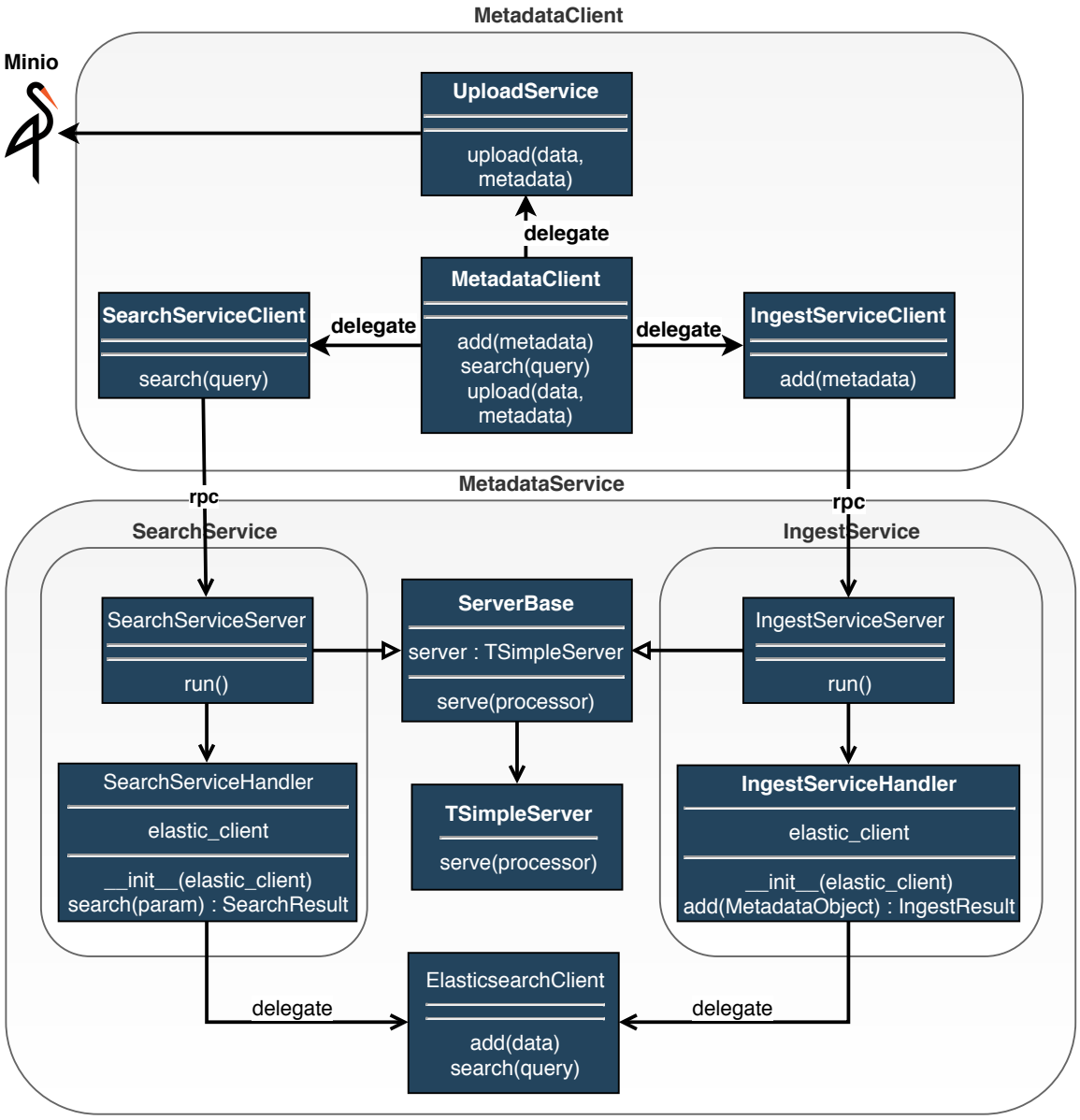


Abbildung 5.2.: Konzeption des Client und Server Aufbaus. Zu sehen ist, wie der Metadaten Client die einzelnen Service-Clients kapselt. Unter Zuhilfenahme des Delegate Patterns, welches hier als Kompositum implementiert wurde, werden die Aufrufe an die jeweiligen Zielklassen weitergeleitet und mittels RPC an die Server gesandt.

IngestServiceHandler Dieser Handler implementiert die Delegation des add Funktionsaufruf eines Clients an des Elasticsearch Client und erzeugt zusammen mit dem Resultat der Delegation ein Returnwert vom Typ IngestResult. Der *IngestServiceHandler* wird im

IngestServiceServer instanziiert und zusammen mit einer Instanz des Elasticsearch Clients dem Processor des IngestService, welcher ebenfalls hier instanziiert wird, übergeben.

SearchServiceHandler Die Funktionalität des SearchServiceHandlers implementiert die lesende Funktion des Metadata Service. Diese Komponente wurde ähnlich aufgebaut wie der IngestServiceHandler und übergibt die vom Nutzer übergebenen Suchparameter an den Datenbank Client.

Kapselung der Thrift-Clients mittels MetadatenClient Der Metadata Client hat die Aufgabe die Systemnutzung transparent zu gestalten. Abbildung 5.2 stellt den Aufbau des Metadaten-Clients dar. Hierbei ist erkenntlich, dass dieser sowohl die einzelnen durch Thrift generierten Thrift-Clients, als auch den UploadService kapselt. Dabei importiert er alle notwendigen Ressourcen, instanziiert diese und delegiert die jeweiligen Aufrufe des Users an die spezifischen Thrift-Clients.

Kompilierung des Services und des Datenmodell sowie Erweiterung des Systems Docker Hub[4] stellt einen Docker Container für die Thrift Library zur Verfügung[5]. Listing 5.3 zeigt, wie dieser Container genutzt werden kann, um lokale Thrift Dateien in die jeweilige Zielsprache zu kompilieren. Zeile 2 zeigt hierbei, wie der Container von Docker Hub heruntergeladen werden kann. Zeile 3 veranschaulicht, wie der Ordner *schema* des lokalen Arbeitsverzeichnisses in den Conainer unter dem Verzeichnis */data* gemounted wird. Der Parameter $-r$ sorgt hierbei für eine Auflösung aller include Befehle. Mittels des Parameters $--gen\ \ py\ \ /data/IngestServiceDef.thrift$ wird nun die Service Beschreibung wie bereits beschrieben in Code für die jeweilige Zielspache[6], übersetzt. Dabei werden auch die durch den Service genutzten Datenklassen erzeugt. Das jeweilige Kompilat liegt nun im Verzeichnis *gen-py* des *schema* Verzeichnisses.

```
1 #!/bin/bash
2 docker pull thrift
3 docker run -v $(pwd)/schema:/data thrift thrift -r -o /data --gen py
  ↪ /data/IngestServiceDef.thrift
```

Listing 5.3: Darstellung des Kompiliervorganges der Thrift Beschreibungen unter Einsatz eines Docker Containers

Metadata Klasse Thrift erlaubt keine Überladung von Funktionen. Um nun nicht für jedes Schema eine extra add Funktion zu definieren wurde die Klasse Metadata, wie in Abbildung 5.3 sowie Listing 5.2 auf Seite 39 zu sehen, definiert. Diese kapselt alle Schemata und ermöglicht hierdurch eine generischen add Funktion. Die Klasse Metadata

[4] https://hub.docker.com/
[5] https://hub.docker.com/_/thrift/
[6] py bedeutet hierbei python

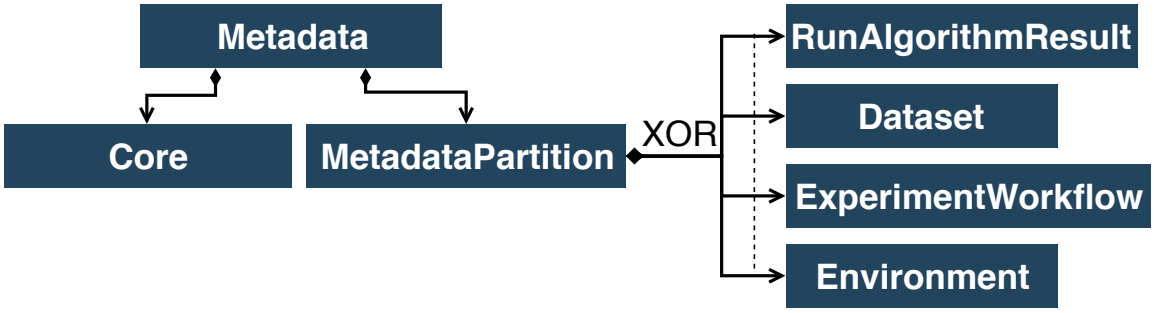


Abbildung 5.3.: UML Diagramm der Metadaten Klasse. Zu sehen ist, wie das Feld MetadataPartition als Stellvertreter einer Region fungiert. Die hier gezeigte XOR Verknüpfung der einzelnen Regionen enstspricht semantisch dem Thrift Union.

als Parameter der Funktion in der Service Beschreibung genannt. Eine weitere Idee bestand daraus in der Metadata Klasse ein Feld vom Thrift Typ Union zu definieren in welcher die einzelnen Schemata als Felder genannt werden. Der Vorteil hier ergibt sich, dass definitiv sichergestellt ist, dass ein Datenobjekt übergeben wird, denn der Typ Union sorgt, dass exakt ein Feld gesetzt ist. Bei der Konzeption des Schemas im Generellen und insbesondere bei der Verwendung von Unions muss jedoch darauf geachtet werden die einzelnen Feldnamen mit Prä oder Suffixen zu versehen und so eine Feldkollision zu verhindern. Es wurde entschieden Feldnamen den jeweiligen Klassennamen als Suffix enthalten.

5.1.2. Elasticsearch

Die nachfolgende Beschreibung von Elasticsearch beschreibt nur das für die Implementierung und Deployment nötige Grundwissen über Elasticsearch. Für ein tieferes Verständnis, vor allem im Hinblick auf Shards und Replikas sei auf die Dokumentation verwiesen.[7] Wie bereits in 4.1.2 beschrieben ist Elasticsearch eine verteilte Dokumenten-basierte Volltext Suchmaschine. Um nicht direkt mit der REST-API des Elasticsearch Clusters kommunizieren zu müssen kommt eine Python Low-Level Client Implementierung[8] zum Einsatz. Diese wird als schlanke Kapselung der Elasticsearch REST-API beworben. [28]

Grundlagen Ein Elasticsearch **Cluster** besteht aus einem bis beliebig vielen sogenannten **Nodes**. Beide haben jeweils einen Namen und Nodes können einem Cluster anhand seines Namens beitreten. Der Name eines Nodes muss eine Universally Unique Identifier (UUID) sein. Nodes sind Server welche die Indizierung- und Suchfähigkeiten von Elasticsearch umsetzen. Ein **Index** ist eine Sammlung von Dokumenten und ein Cluster kann theoretisch beliebig viele Indizes enthalten. Ein **Dokument** entspricht einem indiziertem JSON-Dokument. Ein **Mapping** ist die Beschreibung der möglichen Felder eines Dokumentes. Mit diesem wird ein Index konfiguriert. Elasticsearch ist hierbei in der Lage die Felder

[7] https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html
[8] https://elasticsearch-py.readthedocs.io/en/master/

für einen Index automatisch zu mappen. Es ist hierbei möglich Teile eines Dokumentes streng vorzugeben und andere Felder automatisch mappen zu lassen. Als **Query** werden Anfragen an Elasticsearch genannt. Es gibt viele verschiedene Arten Queries, die mittels einer Domain Specific Language (DSL) beschrieben werden können. [10] Diese Arbeit verwendet eine bool Query mit einem must Term wie sie in Listing 5.4 veranschaulicht. Diese ermöglicht es ein Dokument zu finden, welches über einen Schlüssel *key* mit einem Wert *value* verfügt.

```
{
    "query": {
        "bool": {
            "must": {
                "term": {
                    "key": "value"
                }
            }
        }
    }
}
```

Listing 5.4: Verwendete bool Query der Suchfunktion

Datenbank Client Der Elasticsearch Client soll die *add* und *search* Funktionalität umsetzen. Zum Einsatz kommen soll eine Python Low-Level Client Implementierung [9]. Da das Thrift Schema darauf achtet, dass Feldnamen einzigartig sind, können Dokumente hierbei in einen gemeinsamen Index geschrieben werden. Dabei soll auf die auto mapping Funktion von Elasticsearch zurückgegriffen werden, sodass hierbei keine weitere Arbeit nötig ist. Zur Entwicklung des Datenbank Client konnte sich an der Dokumentation orientiert werden. Es wurde eine Klasse *ElasticClient* definiert und in dessen Konstruktor[10] die Klasse Elasticsearch instanziiert. Dies gibt der ElasticClient Funktion Zugriff auf die add und search Funktion der Elasticsearch Klasse. Nachfolgenden sind die beiden Signaturen dieser Funktionen zu sehen.

$$index(**kwargs)$$

$$search(**kwargs)$$

An den Signaturen wird ersichtlich, dass der Nutzer einen gewissen Freiheitsgrad bei den übergebenen Parametern besitzt. Die Dokumentation listet für die Funktion index, welche es erlaubt ein Dokument einem Index hinzuzufügen, 15 Parameter. Für die Funktion search, welche es ermöglicht nach Dokumenten auf einem oder mehreren Indizes zu

[9] https://elasticsearch-py.readthedocs.io/en/master/
[10] In Python ist dies die __init__ Funktion

suchen, werden 41 Parameter gelistet. Die Klasse ElasticClient bietet den Handlern daher die beiden folgenden Funktionen an:

$$add(self,\ data,\ index = "thesis")$$

Die Funktion *add* ruft dabei die oben gezeigte Funktion *index* wie folgt auf[11]:

$$index(index = index,\ doc_type = "\_doc",\ body = data)$$

Die Funktion *search_with_param* ermöglicht dem Nutzer eine Suche mittels eines Suchparameters und hat folgende Signatur:

$$search\_with\_param(self,\ search\_param,\ index = "thesis")$$

Hierbei wird der Parameter *search_param* als Kolon-getrennter String erwartet und kann wie folgt aussehen:

$$"experiment_id : TestExperiment31012018"$$

Es wird hierbei der übergebene Suchparameter mit der in Python vorhandenen String Funktion split[12] und dem Trennzeichen ":" in die lokalen Variablen namens *key* und *value* gespeichert. Diese beiden werden nun in die in Listing 5.4 gezeigte Query eingesetzt und als lokale Variable *query* gespeichert. Ein Aufruf der obigen Funktion search kann dann wie folgt die Suche umsetzen:

$$search(body = query,\ index = "thesis",\ doc\_type = "\_doc\_type")$$

5.1.3. Workflow Engine, Event Handling und Storage

Als Workflow Engine kommt Argo zum Einsatz und ermöglicht hierdurch das Anstoßen der Containerausführung. Argo Events ist eine Erweiterung Argos und ermöglicht das System reaktiv zu gestalten. Minio ein S3 kompatibler Storage kommt als Storage Lösung zum Einsatz und kann von Argo direkt verwendet werden. Das Zusammenspiel der einzelnen Komponenten wird im folgenden erklärt.

Minio Im vorherigen Abschnitt wurde gezeigt, dass Argo eine S3 kompatible Schnittstelle benötigt. Aus diesem Grund kommt, wie auch in den Beispielen der Argo Dokumentation zu sehen, die Technologie Minio[13] zum Einsatz. Minio ist eine Objectstore Cloud Storage Lösung, welche über eine S3 kompatible Schnittstelle verfügt. Hierbei werden (unstrukturierte) Daten in sogenannten Buckets verwaltet und per Namen auf diese zugegriffen. Diese Daten können dabei im ganzen aber auch Byteweise gelesen werden. [27]

[11] Der Parameter doc_type bezieht sich hierbei auf den Typen für das Mapping. Ein Mapping konnte, bis zur aktuellen Elasticsearch Version noch über unterschiedliche Typen verfügen was nun nicht mehr möglich ist [10]

[12] https://python-reference.readthedocs.io/en/latest/docs/str/split.html

[13] https://www.minio.io/

Uploadservice Minio erlaubt es zwar über das Minio Frontend Daten hochzuladen. Ein Uploadservice kann den User jedoch darin unterstützen zusätzliche Metadaten in das System zu speisen. Hierfür kann die Python Client Implementierung [14] von Minio genutzt werden. Eine weitere Aufgabe des Uploadservice kann es sein, entsprechende Bucket Notifications[15] zu konfigurieren. Diese können die Kuration der gespeicherten Metadaten ermöglichen. In Abbildung 4.6 kann dieser Service dem unteren linken Bereich entnommen werden. Für diese Komponente kann die Region *Dataset* des in Kapitel 4.2.2 vorgestellten Regionenmodells zum Einsatz kommen.

Argo und Argo Events Für die Umsetzung der in Kapitel 2.1.2 vorgestellten technischen Workflows soll die Workflow Engine Argo[16] zum Einsatz kommen. Für die automatische Ausführung von definierten Workflows bei Eintreten von bestimmten Events, wie zum Beispiel dem Upload neuer Daten kommt die Erweiterung von Argo, Argo Events[17] zum Einsatz. Argo ist eine Kubernetes native Workflow Engine, in welcher jeder Schritt durch einen Container repräsentiert wird. Es ermöglicht wie bereits in Kapitel 2.1.2 gezeigt die Definition von Schritt-basierten Workflows als auch die Definition von DAGs. Es erlaubt das Schreiben von Resultaten in ein S3 oder S3 kompatibles Speichermedium. [4]

Gateway Gateways sind überwachende Komponenten. Sie können hierbei auf unterschiedlichste Quellen wie das vorgestellte Minio, ein Stream aber auch ein Kalender Event achten. [5]

Sensor Sensoren beschreiben, welche Events auftreten müssen um einen bestimmten Workflow zu starten. [5]

5.1.4. Collector / Main Crawler

Der Collector ist eine logische Komponente, die hierfür zum Einsatz kommenden Technologien werden im nachfolgenden Paragraphen beschrieben. Zur statischen Erfassung des Cluster-Aufbaus aus Sicht von Kubernetes sollen ein Kubernetes Python Client [18] zum Einsatz kommen und mit diesem entsprechende Cluster Metadaten vom in Kapitel 2.3 vorgestellten API Servers abgefragt werden. Die Openstack Metadaten können mittels Openstack Metadaten Service abgefragt werden. Dieser Service läuft, wie bereits in Kapitel 5.1.4.2 gezeigt, auf jedem Knoten und ist mittels HTTP GET unter einer entsprechenden lokalen IPv4 Aderesse [19] abrufbar. Hieraus resultiert, dass der Crawler somit auf jedem Knoten einen HTTP Aufruf durchführen muss. Hierfür soll die Python Technologie Requests [20], eine leicht verwendbare HTTP Bibliothek, zum Einsatz kommen. Für diese Komponente kann die Region *Environment* des in Kapitel 4.2.2 vorgestellten Regionenmodells zum

[14] https://docs.minio.io/docs/python-client-api-reference.html
[15] https://docs.minio.io/docs/minio-bucket-notification-guide.html
[16] https://github.com/argoproj/argo
[17] https://github.com/argoproj/argo-events
[18] https://github.com/kubernetes-client/python
[19] Diese wird im vorliegenden System mittels Terraform Skript konfiguriert siehe Kapitel 6.
[20] http://docs.python-requests.org/en/master/

Einsatz kommen. Ein Crawler für Openstack braucht nicht notwendigerweise die Kubernetes Bibliothek. Aus diesem Grund wurde für jede Technologie ein spezifischer Container geschrieben und somit die logische Collector Komponente in Technologie spezifische Komponenten unterteilt wird. Jeder Crawler nutzt den Metadaten Client und ist somit in der Lage Metadaten abzufragen und zu schreiben. Durch diesen Aufbau werden die spezifischen Crawler so minimal wie möglich gehalten und es existiert nur eine schreibende Komponente. Hierbei wäre auch vorstellbar, dass dieser sich zu Beginn bestimmte Konfigurationen, wie zum Beispiel die Adresse des Openstack Metadaten Servers unter Einsatz des MetadatenClients liest.

5.1.4.1. Kubernetes Crawler

Der Crawler für Kubernetes nutzt den in Tabelle 5.1 gezeigten Python Client in der Version 7. Dabei wird, wie der in der Dokumentation der Bibliothek beschrieben, eine In-Cluster Konfiguration verwendet. Unter Einsatz der CoreV1API werden entsprechende List-Funktionen ausgeführt und mit diesen Daten die Datenklassen des Kubernetes Schemas gefüllt und an den MetadatenService geschickt. Listing 5.5 veranschaulicht hierbei eine stark verkürzte Version des Crawlers.

```
1  from kubernetes import client, config
2  from metadata import MetadataClient
3
4  config.load_incluster_config()
5  node = client.CoreV1Api().list_node(watch=False)[0]
6
7  node = MetadataClient.Node(provider_id=node.providerID, ...)
8  # Generation of Metadata Object pruned
9  metadata_object.metadata.environment_metadata.node[0] = node
10 MetadataClient().add(metadata_object)
```

Listing 5.5: Stark verkürzte Version des Kubernetes Crawlers.

Metadaten Abfrage innerhalb des Kubernetes Clusters Wie in [17] beschrieben, existieren sowohl offizielle als auch inoffizielle Bibliotheken[17]. Es wurden die offiziellen Client-Implementierung untersucht, welche in Tabelle 5.1 veranschaulicht werden. Hierbei ist zu sehen, welche offizielle Client Version, mit der Kubernetes Version 1.9 sowie 1.10 voll kompatibel bedeutet hierbei, dass alle Features sowohl in der Client Version als auch in der Kubernetes Version des Clusters vorhanden ist. Durch die Nutzung eines Client kann komfortabel aus einem Programm heraus mit dem Kubernetes API-Server kommuniziert werden. Dabei stehen einem Nutzer verschiedenste Möglichkeiten der

Tabelle 5.1.: Gegenüberstellung verschiedener Client Implementierungen

Details	client-go	client-python	client-java
Client-Version für Kubernetes 1.9	6	5	⊖
Client-Version für Kubernetes 1.10	7	6	2.0.0
Quelle	[22]	[23]	[24]

Abfrage zur Verfügung. Dabei ist für einen mittels Role Based Access Control (RBAC)[21] konfigurierten Cluster, wie er in der vorliegenden Arbeit zum Einsatz kommt, zu beachten, dass die verschiedenen Anfrage-Arten unterschiedliche Berechtigungen brauchen, welche vor der Nutzung entsprechend dem ausführenden Cluster-User gegeben werden müssen.

5.1.4.2. Openstack Crawler

Metadaten Abfrage der Cloud Umgebung anhand des Openstack Metadata Server Openstack [25] ermöglicht es Nutzern mit dem Metadaten Service die Abfrage verschiedenster Metadaten. Hierbei ist der Metadaten Service ein auf einem Knoten laufender lokaler Service und kann mittels HTTP GET Befehl angefragt werden. Es existieren hierbei in der Regel (Amazon) Elastic Compute Cloud (EC2) kompatible Schemata als auch Openstack Schemata im JSON Format. Es wird empfohlen für API-Consumer die aktuellste Version anzufragen und im Falle, dass diese nicht vorhanden ist, eine ältere Version zu nutzen. [22]

Ablauf Der Openstack Crawler liest zu Beginn die Umgebungsvariable

OPENSTACK_METADATA_IP

des Betriebssystems aus. Falls diese nicht gesetzt ist, nutzt er als Default die 169.254.169.254 und führt einen GET Request auf die in Tabelle 5.2 auf der nächsten Seite gezeigten Routen und gibt die entsprechenden Daten zurück. Da der Metadaten Server als lokaler Service auf jedem Knoten läuft, wird dieser Crawler als Daemonset[28] deployed. Dies wurde in dieser Arbeit innerhalb einer Argo Workflow Beschreibung als Step umgesetzt. Hierbei wurde eine Kubernetes Daemonset Ressource geschrieben, welche in Anhang A.6 auf Seite 74 zu sehen ist und innerhalb eines Workflows eingefügt. Diese Ressouce könnte zum Beispiel auch in einem Helm Chart als Template dienen. Da die RestartPolicy von

[21] RBAC ist eine Art der Zugriffssteuerung siehe https://kubernetes.io/docs/reference/access-authn-authz/rbac/
[25] Eine freie Architektur zur Umsetzung von Cloud Computing siehe https://www.openstack.org/
[26] https://docs.openstack.org/nova/latest/user/metadata-service.html
[28] Dies bedeutet, dieser Crawler wird auf jedem Knoten ausgeführt.

Tabelle 5.2.: Routen des Openstack Metadata Service unter der Anname, dass dieser unter http://169.254.169.254 erreichbar ist. Entnommen von [26]

Route	Erklärung
/	Auflistung aller verfügbarer EC2 kompatibler Schemata.
/openstack	Auflistung aller verfügbarer Openstack kompatibler Schemata.
/openstack/<version>	Auflistung aller verfügbarer Openstack kompatibler Schemata.
/<version>	Auflistung aller verfügbarer EC2 Metadatenkategorien[27]
/openstack/<version>/meta_data.json	Abruf der Metadaten im Openstack Format

DaemonSets keinen anderen Wert als Always annehmen kann, wurde ein Lösung gesucht, zu verhindern, dass der Container nach erfolgreichem Crawling neu gestartet wird. Dies wurde mittels Kombination von zwei Techniken erreicht:

1. Nutzung einer Schleife in welcher der entsprechende Prozess nach dem Crawling dauerhaft schläft[29]

2. Nutzung des daemon Felds eines Argo Workflow, welches erlaubt dass der Schritt im Hintergrund ausgeführt wird und den Workflow nicht blockiert [30].

Ein weiterer Aspekt war es, dieses DaemonSet als Schritt innerhalb eines Workflows zu starten. Argo erlaubt es in einer Workflow Beschreibung beliebe Ressourcen zu deklarieren. Dies wurde sich zunutze gemacht und eine entsprechende in Anhang A.6 auf Seite 74 zu sehende Konfiguration geschrieben. Ein entsprechender Schritt am Ende des Workflow löscht das DaemonSet aus dem Cluster. Hier kann eine weitere Arbeit den Collector so umschreiben, dass dieser wiederholbar genutzt werden kann.

5.1.4.3. Argo Crawler

Metadaten und Workflow Informationen in Argo können kurzlebig sein. Ein User kann mittels Command Line Interface (CLI) Aufruf sämtliche Informationen auf einmal löschen.

[29] https://github.com/kubernetes/kubernetes/issues/50689
[30] https://github.com/argoproj/argo/tree/master/examples#daemon-containers

Aus diesem Grund soll ein Crawler geschrieben werden, welcher diese Informationen am End eines Workflowas zusammenträgt und in die Metadatenbank schreibt. Workflow Metadaten können Informationen wie Gesamtlaufzeit, Laufzeiten der einzelnen Schritte, Artefakt-Informationen, Pod Metadaten der am Workflow beteiligten Schritte sowie Informationen über Workflow, Schritte und weitere Informationen enthalten. [3]

Metadaten Abfrage von Argo Workflows Nachfolgend sollen zwei Möglichkeiten betrachtet werden Workflow Metadaten zu speichern. Abbildung 2.1 auf Seite 11 veranschaulichte einen Ausschnitt einer Workflow-Beschreibung und Anhang A.11 auf Seite 76 den in dieser Arbeit eingesetzten. Bei diesem werden Metadaten in den Container als Umgebungsvariablen injiziert. Es wird ersichtlich das Argo es erlaubt Kubernetes Ressourcen abzufragen und in den Container zu injizieren. Ein API-Server existiert für Argo nicht. Vielmehr wird Argo als controller und Custom Ressource implementiert. Das bedeutet das Kubernetes durch eine neue Gruppe/Version *argorproj.io/v1alpha1* und einen neuen Typ *Workflow* erweitert wird und Zugriff auf diese über den Kubernetes API Server möglich ist. [5] Ein Zugriff auf die Liste aller Workflows eines namespaces ist zum Beispiel mittels GET Befehl auf folgende Route möglich [5]:

$$< k8s{-}api{-}server > /apis/argoproj.io/v1alpha1/namespaces/< namespace > /workflows$$

Die vorgestellte Kubernetes Python Bibliothek erlaubt den Umgang mit sogenannten *Custom Objects*[31] wie es die Ressource Workflow in diesem Falle ist. Aus diesem Grund wurde für die Abfrage der Workflow Metadaten ebenfalls diese Komponente verwendet. Die Funktion `get_namespaced_custom_object(group, version, namespace, plural, name)` ermöglicht hierbei die Abfrage des jeweiligen Workflows anhand des Namens und weiteren Informationen, welche wie in Listing A.11 auf Seite 76 in Zeile 77 – 90 sowie 215 – 222 und 226 – 238 veranschaulicht, in den Container injiziert wurden.

5.1.5. Nutzerschnittstellen

Die Kommunikation mit dem System kann auf drei Arten erfolgen. Wie in Abbildung 4.6 auf Seite 36 zu sehen kann ein Nutzer mittels Client sowohl aus einem Python Programm heraus mit dem System interagieren als auch mittels eines Jupyter Notebooks. Thrift bietet die Möglichkeit einen Typescript[32] Client zu erzeugen; eine spätere Arbeit kann hier aufsetzen und mittels zum Beispiel Angular ein für dieses System optimiertes Frontend entwickeln.

5.1.5.1. Jupyter Notebook

Jupyter Notebook ist eine im Data Science Bereich weit genutzte Technologie, welche verschiedenste Tätigkeiten des Data Science Workflows unterstützt. Ein Notebook ist ein

[31] https://github.com/kubernetes-client/python/blob/master/kubernetes/docs/CustomObjectsApi.md
[32] https://www.typescriptlang.org/

Dokument und kann live ausführbaren Code, Gleichungen, Grafiken sowie Text beinhalten. Es werden verschiedenste Sprachen unterstützt. [37] In dieser Arbeit dient diese Technologie der Interaktion mit den Metadatenservices.

Beschreibung des Notebooks Ein Notebook kann Python Code enthalten und ausführen. Es kann also der Client instanziiert und Funktionen aufgerufen werden, als würde man diesen in einem Python Script aufrufen wollen. Hierbei wird angenommen, dass das Netzwerk in welchem der MetadatenService läuft vom Pod, welcher den Jupyter Server ausführt erreichbar ist. Dies ist im vorliegenden Fall gegeben, denn beide Pods laufen im selben Cluster. Kapitel 6.2.2.6 auf Seite 61 zeigt hierbei, wie ein solches Notebook aussehen kann.

Erweiterung des Jupyter Kernels Um innerhalb des Notebooks den Metadatenclient verwenden zu können, muss auf dem Betriebssystem des ausführenden Containers die Thrift Library vorhanden sein. Hierfür gibt es zwei Möglichkeiten.

1. Ein Terminal auf dem Container mittels `kubectl exec <jupyter-pod> -it /bin/bash` starten und mittels Paketverwaltung installieren.

2. System-Befehle in einem Jupyter Notebook mittels vorangestelltem Ausrufezeichen ausführen.

3. Das Dockerfile des Jupyter Containers erweitern

Alle drei Möglichkeiten wurden getestet. Hierbei ist zu beachten, dass der ausführende User des Containers keine Administrator-Rechte besitzt. Diese hätten beim Starten zwar ermöglicht werden können, jedoch hätte man diese dann auch wieder entziehen müssen, um keine unnötigen Risiken einzugehen. Es wurde versucht die nötigen Ressourcen zu installieren, ohne Administrator Rechte zu benötigen. Das Ziel war es mittels der Paketverwaltung pip[33] die Bibliothek thrift zu installieren sowie das argo CLI innerhalb des Jupyter Notebooks zu ermöglichen. Dabei konnten alle mittels pip installierbaren Pakete durch voranstellen eines Ausrufezeichens ausgeführt werden. Die Installation der Argo CLI wird in der Dokumentation[34] mittels curl[35] beworben. curl war nicht im Jupyter Kernel vorhanden und eine Ausführung des Linux Paketmanagers apt[36] aufgrund einer Sperre der Datei */var/lib/dpkg/lock* nicht möglich. Durch einen Aufruf des Programmes *which*[37] wurde jedoch festgestellt, dass auf dem System wget[38] zur Verfügung steht. Unter Zuhilfenahme von wget konnte die Argo CLI installiert werden. Zu beachten ist weiterhin, dass eine entsprechende Kubernetes CLI Configfile[39] auf dem System zur Verfügung steht.

[33] https://pypi.org/project/pip/
[34] https://github.com/argoproj/argo/blob/master/demo.md
[35] https://curl.haxx.se/
[36] https://wiki.ubuntuusers.de/APT/
[37] https://linux.die.net/man/1/which
[38] https://www.gnu.org/software/wget/
[39] https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

5.2. Bereitstellung(Deployment) der Systemkomponenten

Die implementierten Ressourcen müssen zum Teil in einen Kubernetes Cluster installiert werden. Hierfür kommen die Technologien Docker sowie Helm zum Einsatz. Nachfolgend wird gezeigt, wie unter Einsatz dieser Technologien implementierten Komponenten bereitstellbar gemacht werden können.

5.2.1. Dockercontainer

Es wurde gezeigt wie unter Einsatz eines Docker Containers, der jeweilige Zielcode erzeugt werden kann. Nachfolgend wird gezeigt wie diese nun in einem Dockerfile beschrieben, das Image gebaut und auf eine Registry geschickt werden kann. Dabei beschreibt Listing 5.6 ein Dockerfile, welches mittels der Befehle `docker build -t tag` und `docker push registry` gebaut, getagt und gepusht werden kann.

```
1 FROM python:3.5-jessie
2 WORKDIR /app
3 ENV PATH="/root/.local/bin/:${PATH}" PYTHONPATH="/app:${PYTHONPATH}"
  ↪ PYTHONUNBUFFERED=0
4 RUN pip3.5 install pipenv
5 RUN pipenv install thrift jsonpickle elasticsearch
6 COPY ["build/gen-py/", "IngestServiceHandler.py", "IngestServiceServer.py",
  ↪ "ServerBase.py", "ElasticClient.py", "./"]
7 CMD pipenv run python3.5 IngestServiceServer.py
```

Listing 5.6: Dockerfile eines Servers anhand des IngestService veranschaulicht.

5.2.2. Kubernetes als Laufzeitumgebung

Kapitel 2.3 auf Seite 7 zeigte Architekturelle Grundlagen der Technologie Kubernetes. Nun werden auf genutzte Implementierungsdetails bestimmter Ressourcen eingegangen. Ressource im Kubernetes Kontext sind Objekte, welche mittels HTTP POST an den API-Server geschickt werden und anschließend als REST-Ressource im Cluster zur Verfügung stehen. [17] Nachfolgend sollen die *Deployment* und *Service* Abstraktionen sowie *Labes* und *Selektoren* untersucht werden.

Labels und Selektoren Labels sind Key/Value Paare und können einem Kubernetes Objekt zu Erstellungszeit und Laufzeit hinzugefügt werden. Während der Laufzeit können sie jederzeit geändert werden und sind im *metadata* Feld des Objektes zu entnehmen. Queries und Watches[40] können mittels Labels effizient umgesetzt werden. Mittels Label Selektoren

[40] Die Beobachtung eines Kubernetes Objektes über einen längeren Zeitraum

können Sets, wie zum Beispiel im Paragraph Service zu sehen, von Objekten identifiziert werden. [17]

Deployment Ein *Replication Controller* bzw. ein *ReplicaSet*[41] sorgen dafür, dass eine Mindestanzahl an Pods zur Verfügung stehen. Dies bedeutet, sollte ein Pod gelöscht oder aus irgendeinem Grund beendet werden, so sorgen diese beiden Controller dafür, dass neue Pods gestartet und so diese Mindestanzahl, auch als *Gewünschter Zustand*[42] bezeichnet, erfüllt ist. Der Unterschied dieser beiden Controller ist in der Art des Umgangs mit Selektoren. Das *Deployment* ist nun eine Abstraktionsstufe darüber und verwendet ein ReplicaSet um den gewünschten Zustand zu erhalten. Darüber hinaus ermöglicht es aber komfortabel verschiedenste Arten von Updates und Upgrades durchzuführen. [17]

Service Jeder Pod erhält eine IP-Adresse, welche jedoch nicht langfristig gewährleistet wird. Im Falle eines Deployments, werden abstürzende Pods durch neue ersetzt. Die Service Abstraktion erlaubt das dynamische Verwalten der Pod Adressen. In der Regel und auch in dieser Arbeit werden die durch das Service Objekt anzusprechenden Pods mittels eines Label Selektors definiert. [17]

Helm Charts Helm[43] ist ein Paketmanager für die Kubernetes Plattform. Die Weiterentwicklung wird von der CNCF in Zusammenarbeit mit Microsoft[44], Google, Bitnami [45] sowie der Helm Community Group umgesetzt. Es werden drei Helm Konzepte beschrieben bestehend aus *Chart* und *Release*. Das dritte Konzept wird entweder als *Repository* oder als *Config* beschrieben. Die komplette Beschreibung einer Applikation wird als Helm Chart bezeichnet. Diese erlauben Versionierung, ein einfaches Deployment sowie Upgrades und Updates von laufenden Applikationen. Eine in einen Kubernetes Cluster installierte Instanz eines Helm Charts wird als Release bezeichnet. Ein *Repository* dient der Speicherung, Verteilung sowie dem Finden von Helm Charts. Eine *Config* enthält Konfigurationen, welche zusammen zu einem installierbaren Objekt gebündelt werden können. Helm als Tool besteht aus verschiedenen Komponenten. Die Client Komponente wird *helm* genannt und ermöglicht lokale Entwicklung von Charts, das Verwalten von Repositories sowie die Kommunikation mit der Server Komponente, welche als *tiller* bezeichnet wird. *Tiller* wartet auf ankommende Requests des Clients und ermöglicht das installieren, deinstallieren sowie upgraden von Releases. Die dritte Komponente ist helms *Template Sprache*. Die erwähnten Charts werden in YAML und einer leicht erweiterten Version der Go Template Language[46] beschrieben. [38] Das Deployment der Metadaten-Server wird unter Verwendung der gezeigten Deployment und Service Abstraktion umgesetzt. Dabei wurde wie in 5.7 auf der nächsten Seite gezeigt das Deployment in einer Template Sprache verfasst. Es ist zu sehen wie der Namespace und die Anzahl der Replicas[47] aus einer Konfigurationsdatei namens

[41] Nachfolger des Replication Controllers.
[42] Im englischen als Desired State bezeichnet.
[43] https://helm.sh
[44] https://www.microsoft.com/de-de
[45] https://bitnami.com/
[46] https://golang.org/pkg/text/template/
[47] Anzahl zu startender Pods.

Listing 5.7: Ausschnitt eines deployment YAML-Dokuments. Das Templating ist anhand der Felder namespace und replicas zu sehen.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: ingest
  namespace: {{.Values.namespace}}
spec:
  replicas: {{.Values.replicas}}
  ...
```

Listing 5.8: Ausschnitt eines values YAML-Dokuments, welche die Konfigurationen der Templates enthalten.

```
# values.yaml
namespace: "kubeyard"
replicas: 3

# service.yaml
apiVersion: v1
kind: Service
metadata:
  # pruned
spec:
  # pruned
  selector:
    app: ingest
```

values.yaml gelesen werden, welche in Abbildung 5.8 im oberen Bereich ausschnitthaft zu sehen ist. Hier ist weiterhin ein Ausschnitt des beschriebenen Services im unteren Bereich zu sehen. Man kann erkennen, wie dieser Service Pods mit dem Schlüssel/Werte Paar *app:ingest* anspricht.

6. Evaluation

6.1. Untersuchung der Anforderungserfüllung

Kapitel 3.3.2 auf Seite 20 beschrieb einige Anforderungen an das zu entwickelnde System. Tabelle 6.1 auf der nächsten Seite zeigt nun, auf welche Art die einzelnen Anforderung umgesetzt wurden.

6.2. Funktionale Evaluation der Systemkomponenten anhand eines Data Science Workflows

Es wurde gezeigt, wie die Komponenten konzipiert, implementiert und installiert werden können. Diese Umsetzung soll nun evaluiert werden. Vom Aufsetzten der Cloud Infrastruktur, der Installation des Systems in diese Infrastruktur, über die Integration des Systems in einen bestehenden Workflow, dem Ausführen eines Workflows bis hin zur Abfrage eines Benutzers über eine geeignete Schnittstelle werden die jeweiligen Schritte beschrieben, das Ergebnis aufgezeigt sowie diskutiert.

6.2.1. Infrastruktur und Deployment verwendeter Komponenten

Bevor die Komponenten evaluiert und genutzt werden können, müssen diese zuerst noch in den Cluster installiert und bereitgestellt werden. Nachfolgend soll gezeigt werden, welche Vorarbeiten hierfür nötig waren.

6.2.1.1. Ressource Planning

Als Cloud Infrastruktur kommt die Inovex Cloud Services (ICS)[1] zum Einsatz. Mithilfe eines Accounts der inovex Cloud Services kann in dieser ein Kubernetes Cluster provisioniert werden. Für diesen, für die Evaluation angelegten, Account musste ein geeignetes *Ressource Planning* durchgeführt werden. Dies bedeutet es musste definiert werden, über welche Flavors[2] eine zu startende virtuelle Maschine verfügen muss. Es musste beachtet werden, dass sowohl für die eigentlichen Komponenten des Metadaten Services als auch für den ausführenden Workflow genug Ressourcen zur Verfügung stehen. Bei der Auswahl der Instanzen in der inovex Cloud gibt es verschiedene Kategorien von Flavors zur Auswahl. Hierbei kann, wie in Tabelle 6.2 auf Seite 57 zwischen Instanzen mit viel vCPUs und

[1] https://www.inovex.de/de/leistungen/inovex-cloud-services/

[2] Flavors sind eine Beschreibung einer Virtuellen Maschine hinlänglich der Anzahl vCPUs, des Arbeitsspeichers sowie der Festplattenkapazität. Siehe https://docs.openstack.org/horizon/latest/admin/manage-flavors.html

Tabelle 6.1.: Übersicht über Umsetzung der Anforderungen.

Anforderung	Umgesetzt durch
1	Docker, Kubernetes, Helm
2	MetadatenClient. Fähigkeit von Minio BucketNotifications zu senden.
3	Generische add Funktion. Globaler Schlüsselraum.
4	Argo und Argo Events
5	Thrift
6	Client - Server Architektur
7	Client - Server Architektur
8	implizit
9	Regionenmodell
10	Kapselung des Metadatenclients
11	Crawler
12	Regionenmodell

Tabelle 6.2.: Auswahl einiger Flavors in der inovex Cloud

Name	vCPUs	Memory	Storage
m4.large	2	8GB	10GB
c4.large	2	4GB	10GB
r4.medium	1	8GB	10GB
r4.large	2	16GB	10GB
ix1.xlarge	4	12GB	10GB

wenig RAM auf der einen Seite, sowie wenig vCPUs und viel RAM auf der anderen Seite entschieden werden. Weiterhin gibt es jene, welche einen Kompromiss darstellen. Der Argo Workflow kann als ein Beispiel für einen Rechenintensiven Job betrachtet werden, welcher von ersterem Typen profitieren würde. Die Elasticsearch Datenbank hingegen profitiert eher von einer Instanz mit viel RAM. Da in den Helm Charts die zur Verfügung stehenden Ressourcen ohnehin limitiert werden muss und unter der Betrachtung des Aufbaus des zur Provisionierung des Kubernetes Clusters zum Einsatz kommenden Terraform Skriptes[3] wurde entschieden, als Kompromiss Instanzen vom Typ mit mittlerer RAM und mittlerer vCPU Anzahl zu verwenden. Es kamen 5 Instanzen vom Typ *m4.large* zum Einsatz. Dies resultiert in 10 zur Verfügung stehenden vCPUs sowie 40GB RAM.

6.2.1.2. Anpassungen des Terraform Skriptes

Um das Terraform Skript in der Cloud Umgebung nutzen zu können mussten noch in einer Konfigurationsdatei entsprechende Metadaten gesetzt werden. Diese bestehen aus dem zu verwendende Cloud Projekt ID, Cluster Namen, User Namen, Authrisationsinformationen, Instanztyp, Anzahl, Image sowie die zu verwendende Kubernetes Version. Mit einem anschließenden ausführen der Befehle `terraform init`, `terraform plan` und `terraform apply` kann nun die Infrastruktur hochgefahren und der Kubernetes Cluster in diese installiert und gestartet werden.

6.2.1.3. Installation der benötigten Komponenten

In diesen frisch provisionierten Kubernetes Cluster müssen nun noch nötige Tools und Konfigurationen vorgenommen werden. Für die initiale Installation und Konfiguration von Docker, Helm, Namespaces, Serviceaccounts, Rolebinding und des Kubernetes Dashboard konnte auf das kube-setup.sh[4] Skript zurückgegriffen werden. Kapitel 5.1.1 auf Seite 37

[3] https://github.com/johscheuer/kubernetes-on-openstack
[4] Das Init Skript des IDCP Projekts

Tabelle 6.3.: Übersicht über das Deployment verschiedene Komponenten

Komponente	Fremd	Quelle	Helm
Elasticsearch	☑	[5]	⊖
Jupyter	☑	[6]	⊖
Services	⊖	[7]	☑
Minio	☑	[8]	☑
Argo	☑	[9]	☑
Argo Events	☑	[10]	☑
Prometheus	☑	[11]	☑
Grafana	☑	[12]	☑

zeigte, wie ein Service kompiliert werden kann. Kapitel 5.2 auf Seite 51 beschreibt wie mittels eines Dockerfile ein Image erzeugt und auf eine Registry gepusht werden kann sowie ein passendes Helm Chart hierfür. Wurde ein Service auf eine Registry gepusht, so kann mittels des Befehls `helm install <chartname> --namespace <namespace>` das Helm Paket in den entsprechenden namespace installiert werden. Für die Installation der Komponenten dieser Arbeit wurde ein Skript geschrieben, welches zu einem späteren Zeitpunkt in das kube-setup Skript integriert werden kann. Es besteht aus den entsprechenden Aufrufen von *helm install* falls ein Chart zur Verfügung steht oder der entsprechenden *kubectl* Befehle. Die Tabelle in 6.3 zeigt hierbei die Quelle der Komponente, ob ein Helm Chart zur Verfügung steht, sowie ob eine Fremd oder Eigenentwickelte Komponente ist.

6.2.2. Nutzung Systems, Ergebnisse und Diskussion

Es wurde gezeigt wie die jeweiligen Komponenten erzeugt und installiert werden. Nachfolgend soll beschrieben werden, wie diese einzelnen Komponenten verwendet werden können. Kapitel 2.1.1.1 auf Seite 3 stellte die Methodologie CRISP-DM vor. Im nachfolgenden soll ausschnitthaft ein Teilbereich des CRISP Prozesses betrachtet werden und hierbei untersucht werden, inwiefern das konzipierte System einen Benutzer hierbei unterstützt beziehungsweise eingesetzt werden. Abbildung 6.1 auf der nächsten Seite veranschaulicht hierbei die einzelnen vorgestellten Schritte.

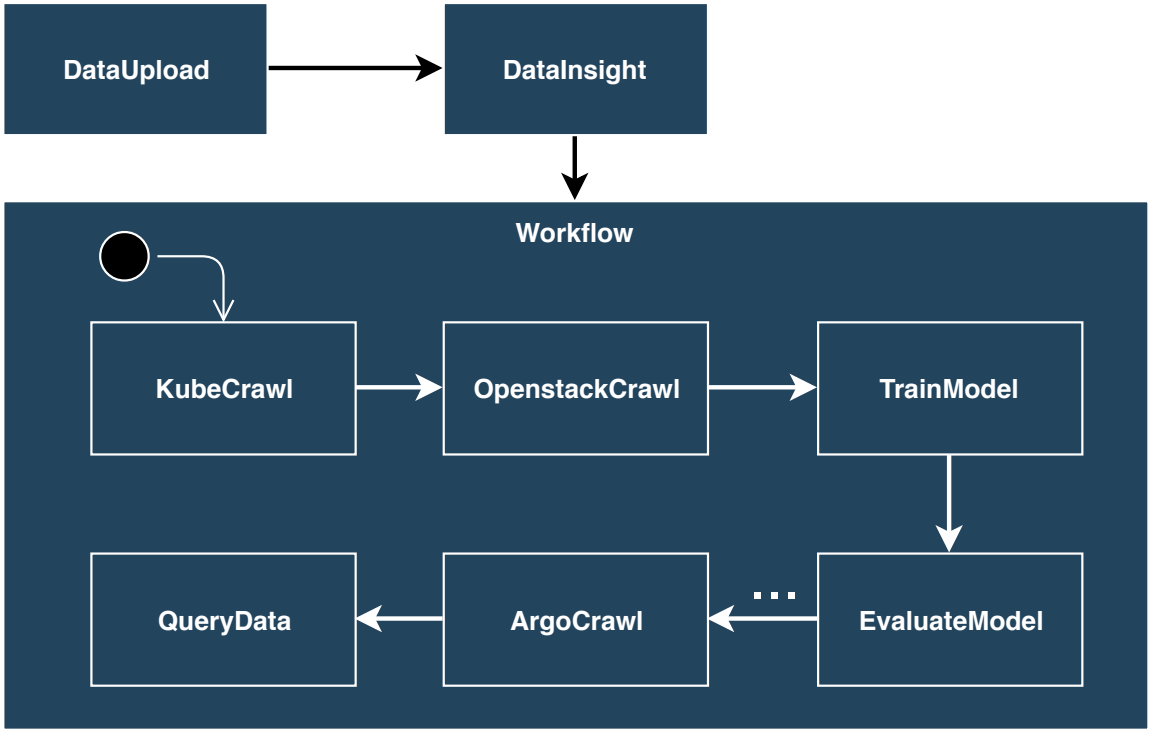


Abbildung 6.1.: Übersicht über die einzelnen Teilaspekte der Evaluation

6.2.2.1. Datensatzbeschreibung und Upload mittels Client am Beispiel des Iris Datensatzes

Beschreibung der Verwendung In Kapitel 3.1 auf Seite 13 wurde der Datensatz Iris vorgestellt und hinsichtlich der Metadaten untersucht. Betrachtet man Anhang A.2 auf Seite 70, welches die Beschreibung eines Datensatzes mittels eines Jupyter Notebook veranschaulicht kann man erkennen, wie dies mit dem konzipierten System umgesetzt werden kann. Zu Beginn werden die Datenklassen instanziiert und entsprechend gefüllt. Anschließend wird die Funktion *upload* des Clients mit dem Pfad des Datensatzes sowie des Metadaten Objektes aufgerufen. Im Hintergrund wird nun ein Aufruf auf den entsprechenden Minio Client durchgeführt, sowie die Metadaten mittels *add* Funktion in die Datenbank geschrieben.

Ergebnis der Verwendung Im Minio Frontend kann anschließend im jeweiligen Bucket untersucht werden, ob der Datensatz hochgeladen wurde. Die geschriebenen Metadaten können durch Nutzung des Metadaten-Clients wiedergefunden werden. Dies wird in Kapitel 6.2.2.6 auf Seite 61 gezeigt.

Diskussion der Verwendung Es konnte gezeigt werden, dass eine explizite Beschreibung der Datensatz Metadaten sowie ein Upload mit dem System ermöglicht werden kann. Hierbei müssen jedoch Metadaten noch händisch gesetzt werden. Eine spätere Arbeit könnte hierbei, wie es auch bei OpenML möglich ist, hochzuladende Daten automatisiert untersuchen und dem Nutzer interaktiv einige Metadaten vorschlagen. Weiterhin können anhand von Bucket Notifications auch eine Kuration der Metadaten durchgeführt werden.

6.2.2.2. Verständnisgewinnung und Speicherung eines Plots aus einem Notebook am Beispiel des Iris Datensatzes

Python besitzt einige Bibliotheken mit denen Datensätze untersucht werden können. In Abbildung A.5 auf Seite 73 ist dies veranschaulicht. Hierbei kann die Funktionalität des MetadatenClients genutzt werden und eine gespeicherte Grafik an einem zentralen Ort gespeichert werden.

6.2.2.3. Anpassungen des Training-Skript und Extraktion von Machine Learning Metadaten

Das Trainings-Skript wurde um verschiedene Aspekte erweitert. Es wurden sowohl die Hyperparameter, die Evaluation als auch das verwendete Dataset beschrieben und mittels der injizierten Identifier miteinander verknüpft und an den Service geschickt.

Ausschnitt des Trainingsskriptes
Listing A.10 auf Seite 75 veranschaulicht die Verwendung des Clients und Datenmodells in einem Trainings-Skript. Zeile 5 – 10 zeigt wie Hyperparameter in einem Python Dictionary gespeichert werden. Zeile 13 instanziiert den Client. Hierbei liest dieser für den User transparent die injizierten Metadaten aus den Umgebungsvariablen weshalb der Nutzer diese in Zeile 16 – 17 verwenden kann. Zeile 16 – 31 zeigt wie die einzelnen Datenklassen instanziiert und gefüllt werden können und in Zeile 32 an den Server geschickt werden.

Anpassungen des Dockerfiles Wie gezeigt, verwendet das Trainingsskript den Metadatenclient. Es musste somit sowohl der Client und das Datenmodell als auch die nötigen Abhängigkeiten in den Container installiert werden. Hierbei wurde das Dockerfile um die in Listing 6.9 gezeigten Zeilen erweitert und somit die thrift Bibliothek installiert sowie die nötigen Abhängigkeiten in den Container kopiert.

```
1 RUN pip install thrift
2 COPY services/ ./services/
3 COPY MetadataClient.py meta-train.py ./
```

Listing 6.9: Auschnitt der hinzukommenden Zeilen im Dockerfile für das Trainingsskript

Diskussion Es zeigt sich, dass bei der Verwendung des Clients, die genutzte Technologie nahezu transparent ist und innerhalb eines Trainings-Skriptes keine Konfiguration des Clients seitens des Users nötig ist. Weiterhin wird die Mehrarbeit der händischen Metadaten Erzeugung ersichtlich.

6.2.2.4. Anpassungen der Workflow Beschreibung um Metadaten Injizierung und Verwendung der Crawler

Anpassungen des Workflows Die bestehende Workflow-Beschreibung wurde dahingehend geändert, dass für das Training der neu erstellte Docker Container verwendet werden muss. Weiterhin wurde der Workflow um einen Initialen Schritt sowie einen End Schritt erweitert. Ersterer Crawled hierbei die statischen Cluster-Informationen und letzterer die Informationen zum ausgeführten Argo Workflow. Notwendige Metadaten wie zum Beispiel die Identifier wurden hierbei ebenfalls gesetzt und in die Container injiziert.

Ergebnis der Verwendung Es konnte gezeigt werden, dass die Komponenten einfach in einer Workflow Engine einsetzbar sind. Weiterhin konnte gezeigt werden, dass durch Argo Events das System auf Events reagieren kann. Dies konnte durch das Hinzufügen eines Datensatzes und dem Auslesen der Systemlogs sowie des gestarteten Workflows gezeigt werden.

6.2.2.5. Einsatz von Argo Events und Nutzung eines Workflows im Kontext der eventgesteuerten Ausführung

Für den Einsatz der Event-gesteuerten Ausführung von Workflows konnten die Beispiele von Argo verwendet werden. Hierbei musste lediglich der Inline Workflow um den auszuführenden Workflow ersetzt werden.

6.2.2.6. Abfrage und Setzen von Metadaten anhand einer Nutzerschnittstelle mittels Jupyter Notebook

Anhand der Ausgabe des Jupyter Notebooks wurde erkenntlich, dass die Environment Metadaten sowie die Workflow Metadaten durch den Crawler zusammengetragen wurden.

Schreiben des Notebooks Nachdem der Kernel des Notebook Servers erweitert wurde, konnte der Metadaten Client innerhalb eines Jupyter Notebooks instanziert und verwendet werden. Das Schreiben des Notebooks entspricht weitestgehend dem Schreiben eines Python Skripts.

Ergebnis der Verwendung Es konnten Metadaten aus einem Jupyter Notebook heraus gelesen und geschrieben werden. Hierbei sei auf Anhang A.4 auf Seite 72 verwiesen, in welchem das Notebook inklusive der Ausgabe angehängt wurde.

Diskussion der Verwendung Jupyter Notebook hat sich als leichtes, intuitives und schnell verwendbares Frontend herausgestellt. Zu beachten ist, dass unter Umständen eine Anpassung des Pfades in welchem Python nach Bibliotheken sucht, stattfinden muss, wenn das Einbinden der Skripte fehlschlägt.

7. Fazit und Ausblick

Diese Arbeit entwickelte einen Ansatz ein Metadatenmanagement System umzusetzen, welches einfach in einen bestehenden, für den Data Science Kontext genutzten, Kubernetes Cluster installiert werden kann. Dieses wurde umgesetzt und konnte seine Funktionalität in einer Evaluation unter Beweis stellen.

Hierbei wurden zuerst Grundlagen in den drei Bereichen *Data Sciene Workflow*s, Metadaten und Kubernetes geschaffen. Eine folgende Analyse verschiedener Metadaten ermöglichte es, das im anschließenden analysierte von Schelter et. al. in [31] vorgestellte Schema zu nutzen. Eine Diskussion bestehender Ansätze erlaubte die Analyse von Nutzungsszenarien und Anforderungen an das zu konzipierende System.

Die in der Konzeption gewählte Datenbank Elasticsearch hat sich hinlänglich der Verwendung als gute Wahl erwiesen und soll es in der Zukunft ermöglichen verschiedenste Metadaten zu suchen und zu finden. Das konzipierte Datenmodell ist in der Lage leicht erweitert zu werden. Es wurden Problematiken wie Kollisionen der Schlüssel des aktuellen Schemas beachtet. Aufgrund der verwendeten auto mapping Funktionalität von Elasticsearch muss ein zu entwickelndes Mapping jedoch noch zeigen, inwiefern das konzipierte Modell skaliert und ob hierbei noch Änderungen nötig sind.

Eine Technologie-agnostische Konzeption veranschaulichte, welche Komponenten für die Entwicklung eines entsprechenden Systems nötig sind. Eine weitere Arbeit kann untersuchen, ob und welche Komponenten bei einem bestehenden System nicht nötig sind. Zum Beispiel seien hier oftmals bereits vorhandene Monitoring und Logging Lösungen genannt.

Bei der Umsetzung und Verwendung des Systems hat sich die Technologie Thrift als geeignete Wahl herausgestellt um das Datenmodell und die Schnittstelle deklarativ zu beschreiben, eine automatisiert Client Server (Teil-)Generierung sowie Kommunikation anhand von RPC zu ermöglichen. Der so entstehende Client entspricht dem in [31] beschriebenen Low-Level Client. Der Unterschied zwischen beiden ist, dass Schelter et. al. einen HTTP/REST orientierten Ansatz, diese Arbeit jedoch einen TCP/RPC basierten Ansatz umsetzte. Auf die Umsetzung eines Domänenspezifischen High Level Clients wurde aufgrund der Diskussion in Kapitel 3.2 auf Seite 18 sowie der beschriebenen Anforderungen in Kapitel 3.3.2 auf Seite 20 verzichtet und können in einer späteren Arbeit entwickelt werden. Dies erlaubt eine Technologie-agnostische [1] Verwendung des Systems erfordert beim Erzeugen der Metadaten jedoch Mehrarbeit.

[1] Im Sinne spezifischer Domänen Technologien.

Das Schema ist für verschiedenste Metadaten erweiterbar, hierbei müssen jedoch Regeln die in der Thrift Dokumentation beschrieben sind eingehalten werden. Eine weiterführende Arbeit kann hier ansetzen und eine DSL zur einfacheren Erweiterung des Schemas konzipieren. Hierbei ist auch ein Übersetzer denkbar, welcher anhand eines Thrift Schemata ein Elasticsearch Mapping konzipiert. Weiterhin kann hier ein Automatismus konzipiert werden, welcher bei einem Update des Thrift Schemas die Services des Clusters aktualisiert oder startet[2] sowie Clients in einem Repository zur Verfügung stellt.

Argo stellte sich als für dieses System gute Wahl heraus, denn auf der einen Seite sind wichtige Informationen wie bestimmte Identifier auf bequeme und einfache Weise in Container injizierbar, auf der anderen Seite ermöglicht es durch die abrufbaren Ressourcen des Workflows eine Möglichkeit der durch den Crawler automatisierten Metadatenerfassung. Dieser ermöglichte auch die Erfassung bestimmter Metadaten aus dem Bereich Kubernetes/Openstack. Eine weitere Arbeit kann hier untersuchen, wie anhand der Metadaten Infrastrukturen und Cluster automatisiert gestartet werden können. Dies könnte es ermöglichen bestimmte Experimente Umgebungen jederzeit zu wiederholen und diese durch Nutzer einfach auszuführen. Argo Events zeigte die Leichtigkeit der Eventbasierten Ausführung bereits bestehender Workflows. Eine weitere Arbeit kann es ermöglichen dieses Systems in Experimenten zu verwenden in denen ein solcher Bedarf existiert.

[2]Dies wäre nahe am von Prabhune beschrieben automatisierten Starten von Endpunkten

Literatur

[1] *A JSON-based schema for storing declarative descriptions of machine learning experiments*. 2018. URL: https://github.com/awslabs/ml-experiments-schema (besucht am 02. 01. 2019).

[2] Zhen Li Ajoy Majumdar. *Metacat: Making Big Data Discoverable and Meaningful at Netflix*. 2018. URL: https://medium.com/netflix-techblog/metacat-making-big-data-discoverable-and-meaningful-at-netflix-56fb36a53520 (besucht am 18. 01. 2019).

[3] *Argo OpenAPI Swagger Spezifikation der Workflow Ressource*. 2018. URL: https://github.com/argoproj/argo/blob/master/api/openapi-spec/swagger.json (besucht am 20. 01. 2019).

[4] *Argo - The Workflow Engine for Kubernetes*. 2018. URL: https://argoproj.github.io/argo/ (besucht am 02. 01. 2019).

[5] *Argo - The Workflow Engine for Kubernetes*. 2018. URL: https://argoproj.github.io/docs/argo/docs/rest-api.html (besucht am 02. 01. 2019).

[6] Jonathan Baier. *Getting Started with Kubernetes Second Edition*. 2017. ISBN: 978-1-78728-336-7.

[7] Ciara Byrne. *Development Workflows for Data Scientists*. 2017. URL: https://resources.github.com/downloads/development-workflows-data-scientists.pdf (besucht am 16. 01. 2019).

[8] Pete Chapman u. a. *CRISP-DM 1.0 Step-by-step data mining guide*. 2000. URL: ftp://ftp.software.ibm.com/software/analytics/spss/support/Modeler/Documentation/14/UserManual/CRISP-DM.pdf (besucht am 10. 12. 2018).

[9] *Data Science - Accelerate data analysis into actionable insights and outcomes*. 2018. URL: https://www.docker.com/solutions/data-analytics (besucht am 18. 12. 2018).

[10] *Elasticsearch Reference*. 2018. URL: https://www.elastic.co (besucht am 12. 01. 2019).

[11] Richard Gartner. *Metadata*. 2016. ISBN: 978-3-319-40891-0.

[12] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. 2017. ISBN: 978-1-491-96229-9.

[13] Joachim Goll und Manfred Dausmann. *Architektur- und Entwurfsmuster der Softwaretechnik: mit lauffähigen Beispielen in Java*. Springer Vieweg, 2014.

[14] *iris*. 2017. URL: https://www.openml.org/ (besucht am 10. 12. 2018).

[15] *iris*. 2017. URL: https://www.openml.org//d/61 (besucht am 10. 12. 2018).

[16] Sebastian Jäger. *Horizontales Skalieren von Deep Learning Frameworks*. 2018.

[17] *Kubernetes Documentation*. 2018. URL: https://kubernetes.io/docs/home (besucht am 16. 01. 2019).

[18] *Kubernetes for Machine Learning, Deep Learning and AI*. 2018. URL: https://twimlai.com/kubernetes/ (besucht am 14. 01. 2019).

[19] *LSTM Tutorial*. 2018. URL: https://neon.nervanasys.com/docs/latest/lstm.html (besucht am 20. 01. 2019).

[20] *Machine Learning Glossary*. 2018. URL: https://developers.google.com/machine-learning/glossary/ (besucht am 18. 12. 2018).

[21] Carol McDonald. *Kubernetes, Kafka Event Sourcing Architecture Patterns and Use Case Examples*. 2018. URL: https://mapr.com/blog/kubernetes-kafka-event-sourcing-architecture-patterns-and-use-case-examples/ (besucht am 10. 12. 2018).

[22] *Metadata service*. 2018. URL: https://docs.openstack.org/nova/latest/user/metadata-service.html (besucht am 02. 01. 2019).

[23] *ModelDB: A system to manage ML models*. 2018. URL: https://github.com/mitdbg/modeldb (besucht am 30. 12. 2018).

[24] Jon Ander Novella u. a. "Container-based bioinformatics with Pachyderm". In: *Bioinformatics* (2018), bty699. DOI: 10.1093/bioinformatics/bty699. eprint: /oup/backfile/content_public/journal/bioinformatics/pap/10.1093_bioinformatics_bty699/3/bty699.pdf. URL: http://dx.doi.org/10.1093/bioinformatics/bty699.

[25] Josh Patterson und Adam Gibson. *Deep Learning: A Practitioner's Approach*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491914254, 9781491914250.

[26] Ajinkya Prabhune. "GENERIC AND ADAPTIVE METADATA MANAGEMENT FRAMEWORK FOR SCIENTIFIC DATA REPOSITORIES". Diss. 2018.

[27] *PRIVATE CLOUD STORAGE*. 2018. URL: https://www.minio.io/ (besucht am 02. 01. 2019).

[28] *Python Elasticsearch Client*. 2018. URL: https://elasticsearch-py.readthedocs.io/en/master/ (besucht am 02. 01. 2019).

[29] Hideto Saito, Hui-Chuan Chloe Lee und Cheng-Yang Wu. *DevOps with Kubernetes*. 2017. ISBN: 978-1-78839-664-6.

[30] Gigi Sayfan. *Mastering Kubernetes*. 2017. ISBN: 978-1-78646-100-1.

[31] Sebastian Schelter u. a. "Automatically Tracking Metadata and Provenance of Machine Learning Experiments". In: 2017.

[32] Sebastian Schelter u. a. "Declarative Metadata Management: A Missing Piece in End-To-End Machine Learning". In: 2018.

[33] David Schmidt. *Evaluation von Data Science Workflow Engines für Kubernetes*. 2018.

[34] D. Sculley u. a. "Hidden Technical Debt in Machine Learning Systems". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, S. 2503–2511. URL: http://dl.acm.org/citation.cfm?id=2969442.2969519.

[35] Mark Slee, Aditya Agarwal und Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation*. 2007. URL: https://thrift.apache.org/static/files/thrift-20070401.pdf (besucht am 02. 01. 2019).

[36] Andrew S. Tanenbaum und Maarten van Steen. *Verteilte Systeme : Prinzipien und Paradigmen*. 2., aktualisierte Aufl. IT Informatik. München [u.a.]: Pearson Studium, 2008. ISBN: 978-3-8273-7293-2.

[37] *The Jupyter Notebook*. 2018. URL: https://jupyter.org/ (besucht am 02. 01. 2019).

[38] *The package manager for Kubernetes*. 2018. URL: https://helm.sh/ (besucht am 14. 01. 2019).

[39] *What is MongoDB?* 2017. URL: https://www.mongodb.com/ (besucht am 18. 12. 2018).

[40] *Why data scientists love Kubernetes*. 2019. URL: https://opensource.com/article/19/1/why-data-scientists-love-kubernetes (besucht am 18. 01. 2019).

[41] *Write, Plan, and Create Infrastructure as Code*. 2019. URL: https://www.terraform.io/ (besucht am 16. 01. 2019).

A. Anhang

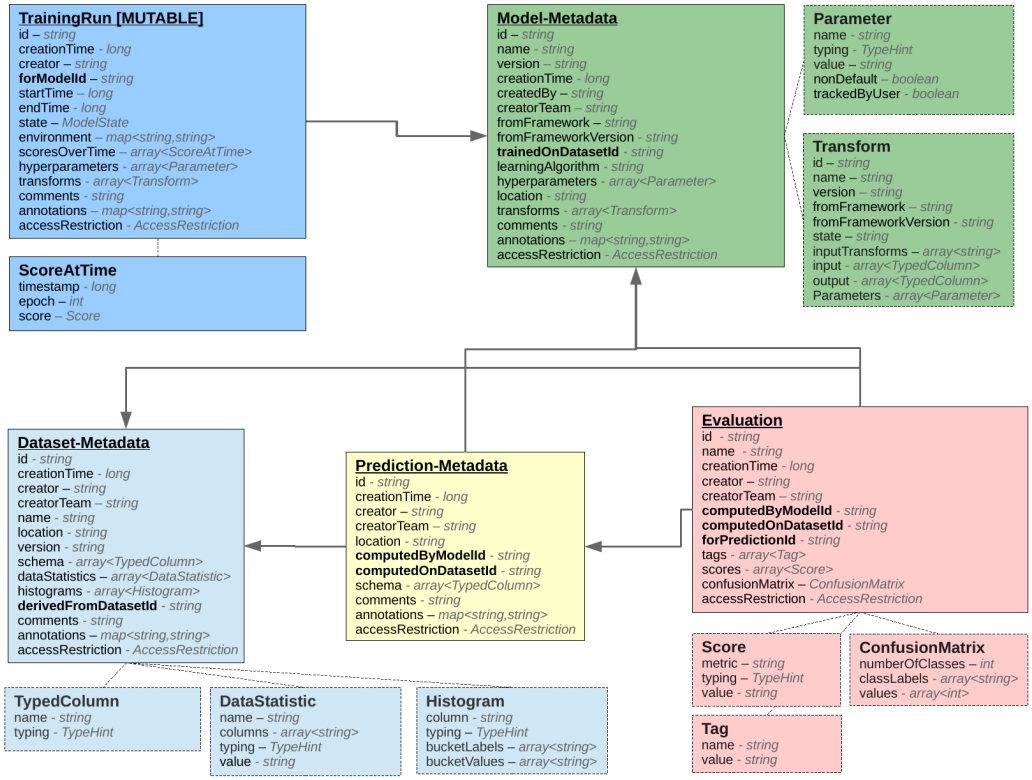


Abbildung A.1.: Metadaten Schema von Schelter et. al. entnommen aus Quelle: [1]

IrisUpload

January 30, 2019

```
In [46]: import sys
         sys.path.append( '/home/jovyan/src/' )
         sys.path.append( '/home/jovyan/src/model' )
         from MetadataClient import MetadataClient
         from src.model.Data.ttypes import Data, CoreData, FileMetadata
         from src.model.MetadataObject.ttypes import MetadataObject, MetadataPartition
         from src.model.Core.ttypes import CoreBaseValues
         import hashlib
         import pprint
         import json

In [58]: data = Data(coreData= CoreData(description="The IRIS Dataset used for classification"
                              contentType="text",
                              fileMetadata=FileMetadata(location="open-quokka:9000", fileformat='

         mc=MetadataClient(upload_host="open-quokka:9000")
         core = CoreBaseValues(metadata_id="demo_test",
                               timestamp=mc.time(), creator="kexel",
                               creator_team="idcp", origin="from_script")
         md_obj = MetadataObject(experiment_id="iris",
                                 core=core,
                                 metadata=MetadataPartition(data_metadata=data))
         mc.upload(metadata=md_obj,
                   bucket="data",
                   name="iris",
                   local_path="dataset_61_iris.csv")
```

1

Abbildung A.2.: Nutzung des MetadatenClients innerhalb eines Jupyter Notebooks für den Upload eines Datensatzes und dem Hinzufügen von Metadaten.

SearchHyperparameter

January 30, 2019

```
In [14]: import sys
         sys.path.append( '/home/jovyan/src/model' )
         from MetadataClient import MetadataClient
         import pprint
         import json
         metadata = MetadataClient().search("experiment_id:demofinal")
         pprint.PrettyPrinter(indent=1).pprint(json.loads(metadata.result)
                                               .get('hits')
                                               .get('hits')[3]
                                               .get('_source')
                                               .get('metadata')
                                               .get('run_metadata')
                                               .get('hyperparameters'))
```

```
{'batch_size': {'annotations': None,
                'comments': None,
                'name': 'batch_size',
                'nonDefault': False,
                'trackedByUser': False,
                'type': 'int',
                'value': '32'},
 'clip_gradients': {'annotations': None,
                    'comments': None,
                    'name': 'clip_gradients',
                    'nonDefault': False,
                    'trackedByUser': False,
                    'type': 'bool',
                    'value': 'True'},
 'embedding_dim': {'annotations': None,
                   'comments': None,
                   'name': 'embedding_dim',
                   'nonDefault': False,
                   'trackedByUser': False,
                   'type': 'int',
                   'value': '128'},
 'embedding_update': {'annotations': None,
                      'comments': None,
```

1

Abbildung A.3.: Nutzung des MetadatenClients innerhalb eines Jupyter Notebooks zur Suche von Hyperparametern.

SearchMetadata

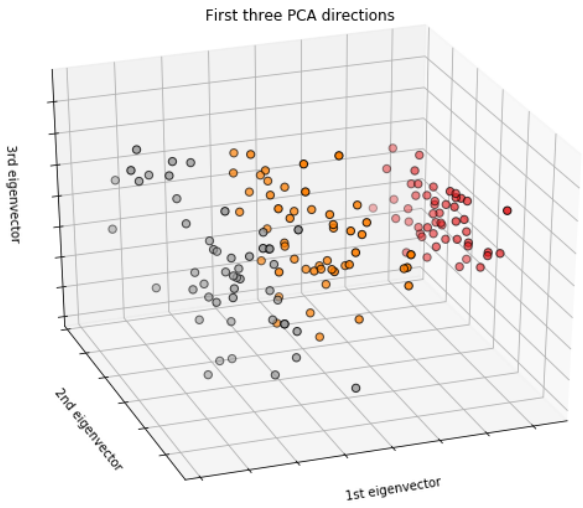
January 30, 2019

```
In [51]: import sys
         sys.path.append( '/home/jovyan/src/model' )
         from MetadataClient import MetadataClient
         import pprint
         import json
         metadata = MetadataClient().search("experiment_id:demofinal")
         pprint.PrettyPrinter(indent=1).pprint(json.loads(metadata.result)
                                               .get('hits')
                                               .get('hits')[8]
                                               .get('_source')
                                               .get('metadata')
                                               .get('environment_metadata')
                                               .get('openstack_metadata'))
```

```
{'openstack_availability_zone': 'az1',
 'openstack_devices': [],
 'openstack_hostname': 'thesis-kube-node-3.fra.private.vpc.host',
 'openstack_instance_type': 'm4.large',
 'openstack_launch_index': None,
 'openstack_meta': {'cluster': 'thesis', 'kubernetes': 'node'},
 'openstack_name': 'thesis-kube-node-3',
 'openstack_project_id': '0b8c76aa525547a68925f1f6bb3e80b6',
 'openstack_uuid': '7ab0debf-8036-47e2-a5db-ef09e48dc4cc'}
```

1

Abbildung A.4.: Nutzung des MetadatenClients innerhalb eines Jupyter Notebooks zur Suche von Nodes.




```
In [35]: import sys
         sys.path.append( '/home/jovyan/src/model' )
         from MetadataClient import MetadataClient

         mc = MetadataClient()
         from src.model.Data.ttypes import Data, CoreData
         from src.model.MetadataObject.ttypes import MetadataObject, MetadataPartition
         from src.model.Core.ttypes import CoreBaseValues
         import hashlib
         hashlib.sha256()

         plt.savefig('training_points_and_pca.png')

         coreData = CoreData(description="First Plots of Iris, Containing Training Points and I
                             contentType="image")
         data = Data(coreData=coreData)
         core = CoreBaseValues(metadata_id="demo_test", timestamp=mc.time(), creator="kexel",
                               creator_team="idcp", origin="from_jupyter")
         mc=MetadataClient(upload_host="open-quokka:9000")
         md_obj = MetadataObject(experiment_id="demofinal", core=core, metadata=MetadataPartit:
         mc.upload(metadata=md_obj, bucket="data", name='training_points_and_pca', local_path=
```

3

Abbildung A.5.: Nutzung des MetadatenClients innerhalb eines Jupyter Notebooks für den Upload eines Plots des IRIS Datensatzes.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: openstack-collector
  namespace: kubeyard
spec:
  selector:
    matchLabels:
      name: openstack-collector
  template:
    metadata:
      labels:
        name: openstack-collector
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: openstack-collector
        nodeSelector:
        image: registry.inovex.de:4567/kexel/bachelor_thesis_code/
        ↪ openstack_crawler:demo
        imagePullPolicy: Always
        env:
          - name: EXPERIMENT_ID
            value: demofinal
        command: [ "sh", "-c" ]
        args: ["pipenv run python3.5 /app/crawler.py; while true; do sleep
        ↪ 3600; done"] # dead loop needed because restartpolicy in
        ↪ daemonsets is always Always...
        ↪ https://github.com/kubernetes/kubernetes/issues/50689
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      terminationGracePeriodSeconds: 30
```

Abbildung A.6.: Openstack Collector als Kubernetes Daemonset Ressource angelehnt an: https://github.com/argoproj/argo/blob/master/examples/k8s-orchestration.yaml

```
from Run.ttypes import RunCore, TrainingRun, ModelMetadata, Transform,
↪  Parameter, PredictionMetadata, Evaluation, Score, ScoreAtTime,
↪  ConfusionMatrix
from MetadataObject.ttypes import MetadataObject, CoreBaseValues
from MetadataClient import MetadataClient

hyperparameters = {
    "hidden_size": Parameter(name="hidden_size", type="int", value=str(128),
                             nonDefault=0, trackedByUser=1),
    # ... PRUNED
    "num_epochs": Parameter(name="num_epochs", type="int", value=str(1),
                            nonDefault=0, trackedByUser=1)}

# ... Training pruned
mc = MetadataClient()

# Generate and send metadata to the server
rc = RunCore(run_id="thesis", container_id=mc.container_id,
             pod_id=mc.pod_id, node_id=mc.node_id,  # ... pruned
             )

md = ModelMetadata(id="thesis-model",
                   name="IMDB-LSTM-SentinelAnalysis-Model",  # ... pruned
                   hyperparameters=hyperparameters)

run = TrainingRun(run_core=rc,  # ... PRUNED
                  model_metadata=md)

c = CoreBaseValues(metadata_id="TestId", timestamp=mc.time(),
                   creator="kexel", creator_team="idcp", origin="script")

mo = MetadataObject(experiment_id=mc.experiment_id, core=c,
                    run_metadata=run)
mc.add(mo)
```

Listing A.10: Auschnitt der hinzukommenden Zeilen im Trainingsskript

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: argo-parametrized-wf-
spec:
  arguments:
    parameters:
    - name: experimentId
      value: demofinal
    - name: parallelism
      value: 4
    - name: minio-endpoint
      value: open-quokka:9000
    - name: input-bucket
      value: dataset
    - name: output-bucket
      value: dataset
    - name: train-dataset
      value: labeledTrainData.tsv
    - name: test-dataset
      value: testData-small-10.tsv
    - name: num-epoch
      value: 1
  entrypoint: neon-sentiment-analysis
  templates:
  - name: neon-sentiment-analysis
    steps:
    # execute static crawling in parallel
    - - name: kube-crawling
        template: kube-crawl
        arguments:
          parameters:
          - name: experimentId
            value: "{{workflow.parameters.experimentId}}"
      - name: openstack-crawling
        template: daemon-crawl
        arguments:
          parameters:
          - name: experimentId
            value: "{{workflow.parameters.experimentId}}"
```

Listing A.11: Workflow Beschreibung. Training entnommen [19], Workflow Beschreibungen und Go Code von [33], manifest angelehnt an https://github.com/argoproj/argo/blob/master/examples/k8s-orchestration.yam

```
- - name: train-model
        template: train
        arguments:
          parameters:
          - name: experimentId
            value: "{{workflow.parameters.experimentId}}"
      - name: generate-test-data-files
        template: test-data-generator
    - - name: create-data-distribution-files
        template: distribute-data-files
        arguments:
          artifacts:
          - name: test-data-files
            from:
            ↪  "{{steps.generate-test-data-files.outputs.artifacts.test-data-files}}"
    - - name: generate-list-of-distribution-files
        template: generate-list-of-distribution-files
        arguments:
          artifacts:
          - name: distribution-files
            from:
            ↪  "{{steps.create-data-distribution-files.outputs.artifacts.distribution-fil
    - - name: infer
        template: infer
        withParam:
        ↪  "{{steps.generate-list-of-distribution-files.outputs.result}}"
        arguments:
          parameters:
          - name: dist-file
            value: "{{item}}"
          artifacts:
          - name: imdb-model-in
            from: "{{steps.train-model.outputs.artifacts.imdb-model-out}}"
          - name: test-data-files
            from:
            ↪  "{{steps.generate-test-data-files.outputs.artifacts.test-data-files}}"
          - name: distribution-files
            from:
            ↪  "{{steps.create-data-distribution-files.outputs.artifacts.distribution-fil
```

Listing A.12: Fortsetzung Workflow Beschreibung

```
- - name: argo-crawling
        template: argo-crawl
        arguments:
          parameters:
            - name: workflow-namespace
              value: "{{workflow.namespace}}"
            - name: workflow-name
              value: "{{workflow.name}}"
            - name: group
              value: argoproj.io
            - name: version
              value: v1alpha1
            - name: plural
              value: workflows
            - name: experimentId
              value: "{{workflow.parameters.experimentId}}"
    - - name: clean
        template: delete-daemon
  - name: train
    inputs:
      parameters:
      - name: experimentId
      artifacts:
      - name: labeledTrainData
        path: /tmp/train/labeledTrainData.tsv
        s3:
          endpoint: "{{workflow.parameters.minio-endpoint}}"
          insecure: true
          bucket: "{{workflow.parameters.input-bucket}}"
          key: "{{workflow.parameters.train-dataset}}"
          accessKeySecret:
            name: open-quokka
            key: accesskey
          secretKeySecret:
            name: open-quokka
            key: secretkey

```

Listing A.13: Fortsetzung Workflow Beschreibung

```
  container:
      image: registry.inovex.de:4567/kexel/bachelor_thesis_code/train:demo
      imagePullPolicy: Always
      command: [ "sh", "-c" ]
      args: [ "python /root/neon/meta-train.py -f
      ↪  /tmp/train/labeledTrainData.tsv -e
      ↪  {{workflow.parameters.num-epoch}} -eval 1 -s /tmp/out/imdb.p
      ↪  --vocab_file /tmp/out/imdb.vocab" ]
      env:
      - name: EXPERIMENT_ID
        value: "{{inputs.parameters.experimentId}}"
    outputs:
      artifacts:
      - name: imdb-model-out
        path: "/tmp/out/"
  - name: argo-crawl
    inputs:
      parameters:
      - name: group
      - name: version
      - name: plural
      - name: workflow-name
      - name: workflow-namespace
      - name: experimentId

```

Listing A.14: Fortsetzung Workflow Beschreibung

```
container:
     image: registry.inovex.de:4567/kexel/bachelor_thesis_code/argo_crawler:demo
     imagePullPolicy: Always
     env:
     - name: GROUP
       value: "{{inputs.parameters.group}}"
     - name: VERSION
       value: "{{inputs.parameters.version}}"
     - name: PLURAL
       value: "{{inputs.parameters.plural}}"
     - name: WORKFLOW_NAME
       value: "{{inputs.parameters.workflow-name}}"
     - name: WORKFLOW_NAMESPACE
       value: "{{inputs.parameters.workflow-namespace}}"
     - name: EXPERIMENT_ID
       value: "{{inputs.parameters.experimentId}}"
     command: [ "sh", "-c" ]
     args: ["pipenv run python3.5 /app/crawler.py"]
 - name: daemon-crawl
   daemon: true
   inputs:
     parameters:
     - name: experimentId
   resource:
     action: create
     manifest: |
       # siehe DaemonSet.yaml
 - name: delete-daemon
   resource:
     action: delete
     manifest: |
       apiVersion: apps/v1
       kind: DaemonSet
       metadata:
         name: openstack-collector

```

Listing A.15: Fortsetzung Workflow Beschreibung

```
- name: kube-crawl
    inputs:
      parameters:
      - name: experimentId
    container:
      image: registry.inovex.de:4567/kexel/bachelor_thesis_code/
      ↪ kube_crawler:demo
      imagePullPolicy: Always
    env:
    - name: EXPERIMENT_ID
      value: "{{inputs.parameters.experimentId}}"
    command: [ "sh", "-c" ]
    args: ["pipenv run python3.5 /app/crawler.py"]
```

Listing A.16: Fortsetzung Workflow Beschreibung

```
include "Data.thrift"

struct RunResult {
    1: required RunCore run_core
    2: required TrainingRun training_run
}

struct TrainingRun {
    1: required string training_id
    2: required string creationTimestamp
    3: required string creator
    4: required string forModelId
    5: required string startTime
    6: required string endTime
    7: required map<string, Parameter> hyperparameters
    8: optional string comments
    9: optional map<string, string> annotations
    10: optional map<string,ScoreAtTime> scoresOverTime
}

struct RunCore {
    1: optional string run_id
    2: optional string container_id
    3: optional string pod_id
    4: optional string node_id
    5: optional string step_id
    6: optional string timestamp
    7: optional string comments
    8: optional map<string, string> annotations
}

struct ModelMetadata {
    1: required string model_id
    2: required string name
    3: required string version
    4: required string creationTimestamp
    5: required string creator
    6: required string creationTeam
    7: required string framework
    8: required string frameworkVersion
    9: required string trainedOnDatasetId
    10: required string learningAlgorithm
    11: required map<string,Parameter> hyperparameters
    12: required string location
    13: optional string comments
    14: optional map<string, string> annotations
}
```


Listing A.17: Thrift Umsetzung des Schemas von Schelter et. al..

```
struct Transform {
    1: required string transform_id
    2: required string name
    3: required string version
    4: required string framework
    5: required string frameworkVersion
    6: required string state
    7: required set<string> inputTransforms
    8: required set<string> outputTransforms
    9: required map<string,Parameter> parameters
}

struct Parameter {
    1: required string name
    2: required string type
    3: required string value
    4: required bool nonDefault
    5: required bool trackedByUser
    6: optional string comments
    7: optional map<string, string> annotations
}

struct PredictionMetadata {
    1: required string prediction_id
    2: required string creationTimestamp
    3: required string creator
    4: required string creatorTeam
    5: required string location
    6: required string computedByModelId
    7: required string computetOnDatasetId
    8: required map<string,Data.TypedColumn> schema
    9: required string comments
    10: required map<string, string> annotations
}
```

Listing A.18: Fortsetzung Thrift Umsetzung des Schemas von Schelter et. al..

```
struct Score {
    1: required string metric
    2: required string type
    3: required string value
}

struct ScoreAtTime {
    1: required string timestamp
    2: required i32 epoch
    3: required Score score
}

struct ConfusionMatrix {
    1: required i32 numberOfClasses
    2: required set<string> classLabels
    3: required set<i32> values
}


struct Evaluation {
    1: required string evaluation_id
    2: required string name
    3: required string creationTimestamp
    4: required string creator
    5: required string creatorTeam
    6: required string computedByModelId
    7: required string computetOnDatasetId
    8: required string forPredictionId
    9: required map<string, string> tags
    10: required map<string,Score> scores
    11: optional ConfusionMatrix confusionMatrix
}
```

Listing A.19: Fortsetzung Thrift Umsetzung des Schemas von Schelter et. al..

```
struct Data {
    1: required CoreData coreData
    2: optional set<TypedColumn> schema
    3: optional string derivedFromDatasetId
    4: optional set<DataStatistic> dataStatistics
    5: optional set<Histogram> histograms
    6: optional string comments
    7: optional  map<string, string> annotations
}

struct CoreData {
    1: optional FileMetadata fileMetadata
    2: optional string description
    3: optional string contentType # text, image, video, ... maybe enum
↪  later?
}

struct FileMetadata {
    1: optional string location # maybe storage later
    2: optional string fileformat
    3: optional string filename
    4: optional string hash
    5: optional string version
    6: optional string spaceInMegabyte
    7: optional string licence
    8: optional Restriction restriction
}

struct Restriction { # maybe more later
    1: optional string owner
    2: optional string group
}

struct TypedColumn {
    1: optional string name
    2: optional string type
}
```

Listing A.20: Thrift Umsetzung des Data Schemas.

```
struct DataStatistic {
    1: optional string name
    2: optional set<string> columns
    3: optional string type
    4: optional string value
}

struct Histogram {
    1: optional string column
    2: optional string type
    3: optional set<string> bucketLabels
    4: optional set<string> bucketValues
}
```

Listing A.21: Fortsetzung Thrift Umsetzung des Data Schemas.

```
include "Data.thrift"
include "Environment.thrift"
include "Run.thrift"
include "Experiment.thrift"
include "Core.thrift"

# TODO ANNOTATIONS
struct MetadataObject {
    1: required string experiment_id
    2: required Core.CoreBaseValues core
    3: required MetadataPartition metadata
}

union MetadataPartition {
    1: Data.Data data_metadata
    2: Environment.Environment environment_metadata
    3: Run.RunResult run_metadata
    4: Experiment.ExperimentWorkflow experiment_workflow_metadata
}
```

Listing A.22: Thrift Umsetzung des MetadataObjekts.

```
union EnvironmentMetadata {
    1: Openstack openstack_metadata
    2: Kubernetes kubernetes_metadata
    3: Monitor monitor
    4: UserInterface ui
}

struct EnvironmentCore {}

struct Environment {
    1: required EnvironmentCore environment_core
    2: required EnvironmentMetadata environment_metadata
}

struct Openstack {
    1: required string openstack_uuid
    2: optional string openstack_availability_zone
    3: optional string openstack_hostname
    4: optional i32 openstack_launch_index
    5: optional set<string> openstack_devices
    6: optional map<string, string> openstack_meta
    7: required string openstack_project_id
    8: optional string openstack_name
    9: optional string openstack_instance_type
}

struct Kubernetes {
    1: required string kubernetes_cluster_name
    2: optional set<Node> kubernetes_nodes
    3: optional set<Pod> kubernetes_pods
    4: optional Monitor kubernetes_monitor
}

union Monitor {
    1: Grafana grafana
}

struct Grafana {
    1: required string grafana_service_url
    2: required string grafana_dashboard
    3: required string grafana_params
}
```

Listing A.23: Thrift Umsetzung des Environment Schemas.

```
union EnvironmentMetadata {
    1: Openstack openstack_metadata
    2: Kubernetes kubernetes_metadata
    3: Monitor monitor
    4: UserInterface ui
}

struct EnvironmentCore {}

struct Environment {
    1: required EnvironmentCore environment_core
    2: required EnvironmentMetadata environment_metadata
}

struct Openstack {
    1: required string openstack_uuid
    2: optional string openstack_availability_zone
    3: optional string openstack_hostname
    4: optional i32 openstack_launch_index
    5: optional set<string> openstack_devices
    6: optional map<string, string> openstack_meta
    7: required string openstack_project_id
    8: optional string openstack_name
    9: optional string openstack_instance_type
}

struct Kubernetes {
    1: required string kubernetes_cluster_name
    2: optional set<Node> kubernetes_nodes
    3: optional set<Pod> kubernetes_pods
    4: optional Monitor kubernetes_monitor
}

union Monitor {
    1: Grafana grafana
}

struct Grafana {
    1: required string grafana_service_url
    2: required string grafana_dashboard
    3: required string grafana_params
}
```

Listing A.24: Fortsetzung Thrift Umsetzung des Environment Schemas.

```
include "Environment.thrift"

struct ExperimentWorkflow {
    1: optional ExperimentCore experiment_workflow_core
    2: optional ExperimentMetadata experiment_workflow_metadata
}

struct ExperimentCore{
    1: required string experiment_id
    2: required string experiment_name
    3: optional string experiment_started
    4: optional string experiment_ended
}

struct ExperimentMetadata {
    1: optional WorkflowMetadata workflow_metadata
}

struct WorkflowMetadata {
    1: optional string workflow_engine
    2: optional string workflow_type # dag, step, ...
    3: optional string workflow_url
    4: optional string workflow_created
    5: optional string workflow_started
    6: optional string workflow_ended
    7: optional string workflow_name
    8: optional string workflow_namespace
    9: optional WorkflowTemplate workflow_template
    12: optional string workflow_arguments
    13: optional set<string> workflow_parameter
    14: optional string workflow_results_as_string # todo write the classe
    15: optional string workflow_ressource_version
    16: optional string workflow_result
}

struct StepTemplate {
    1: required string step_inputs
    2: required string step_outputs
    3: required string step_metadata
    4: required string step_name
    5: optional string step_script
    6: optional Environment.Container step_container
}
```

Listing A.25: Thrift Umsetzung des Experiment Schemas.

```
include "Environment.thrift"

struct ExperimentWorkflow {
    1: optional ExperimentCore experiment_workflow_core
    2: optional ExperimentMetadata experiment_workflow_metadata
}

struct ExperimentCore{
    1: required string experiment_id
    2: required string experiment_name
    3: optional string experiment_started
    4: optional string experiment_ended
}

struct ExperimentMetadata {
    1: optional WorkflowMetadata workflow_metadata
}

struct WorkflowMetadata {
    1: optional string workflow_engine
    2: optional string workflow_type # dag, step, ...
    3: optional string workflow_url
    4: optional string workflow_created
    5: optional string workflow_started
    6: optional string workflow_ended
    7: optional string workflow_name
    8: optional string workflow_namespace
    9: optional WorkflowTemplate workflow_template
    12: optional string workflow_arguments
    13: optional set<string> workflow_parameter
    14: optional string workflow_results_as_string # todo write the classe
    15: optional string workflow_ressource_version
    16: optional string workflow_result
}

struct StepTemplate {
    1: required string step_inputs
    2: required string step_outputs
    3: required string step_metadata
    4: required string step_name
    5: optional string step_script
    6: optional Environment.Container step_container
}
```

Listing A.26: Fortsetzung Thrift Umsetzung des Experiment Schemas.

```
include "MetadataObject.thrift"

struct IngestResult {
    1: required i16 code
    2: required string message
}

struct ValidationResult {
    1: required i16 code
    2: required string message
}

service IngestService {
    IngestResult add(1: MetadataObject.MetadataObject mdObj)
    IngestResult addToDatabase(1: MetadataObject.MetadataObject mdObj)
    ValidationResult validate(1: MetadataObject.MetadataObject mdObj)
}
```

Listing A.27: Thrift Umsetzung des IngestService.

```
struct SearchResult {
    1: required i16 code
    2: required string result
}

service SearchService {
    SearchResult search(1: string parameter)
}
```

Listing A.28: Thrift Umsetzung des SearchService.