Bachelor

July 20, 2021

Continuous Deep Learning

An in-depth investigation of the deep learning workflow

Janosch Baltensperger

of Brütten, Zürich, Schweiz (15-915-085)

supervised by

Prof. Dr. Harald C. Gall
Dr. Pasquale Salza

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

Continuous Deep Learning

An in-depth investigation of the deep learning workflow

Janosch Baltensperger

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

Author: Janosch Baltensperger, janosch.baltensperger@uzh.ch

URL: https://github.com/janousy/CDL

Project period: 25.01.2021 - 24.07.2021

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First, I would like to thank Dr. Pasquale Salza for supervising my thesis. During regular meetings, he provided valuable inputs and critical feedback to keep me on the right track. Further, I would like to thank Prof. Dr. Gall for allowing me to write my thesis at the department of software evolution and architecture (s.e.a.l).

In addition, I would like to thank my girlfriend Sara Costa Fernandes and my fellow student Jonas Zürcher for proofreading my thesis.

Finally, I would like to thank the MLOps Community for valuable insights into the application of machine learning and interesting discussions on Slack about best practices.

Abstract

Deep learning has gained immense attraction with the emergence of big data and advanced computing power. Through the use of artificial neural networks, various breakthroughs were achieved in fields such as language understanding and image recognition. Nevertheless, it has soon become clear that deep learning and machine learning in general impose various additional challenges besides building an accurate model. Researchers have been highly active to investigate the classical machine learning workflow and integrate best practices from the software engineering lifecycle. However, deep learning exhibits deviations which are not yet covered in this conceptual development process. This includes the requirement of dedicated hardware, dispensable feature engineering, extensive hyperparameter optimization, large-scale data management and model compression to reduce size and inference latency. Individual problems of deep learning are under thorough examination, and numerous concepts and implementations have gained traction. However, the complete end-to-end development process still remains unspecified. In this thesis, we defined a detailed deep learning workflow that incorporates the aforementioned characteristics on the baseline of the classical machine learning workflow. We further transferred the conceptual idea into practice by building a prototypic deep learning system using the latest technologies on the market. To examine the feasibility of the workflow, two use cases were applied to the prototype. The first use case represented a text classification problem, while the second use case focused on image processing. We thereby successfully demonstrated the application of the workflow on distinct examples. In summary, it becomes apparent that the deep learning lifecycle compromises a large set of steps and involves various roles. With our defined workflow, we present a profound guideline for the deep learning development process. Moreover, we conclude that the technologies currently available on the market are not fully mature. Great effort is required to manage all deep learning artifacts and keep versions aligned within continuous iterations over the lifecycle.

Zusammenfassung

Deep Learning hat mit dem Aufkommen von Big Data und fortschrittlicher Rechenleistung immenses Interesse gewonnen. Durch den Einsatz von künstlichen neuronalen Netzwerken wurden verschiedene Durchbrüche in Gebieten wie Sprachverständnis und Bilderkennung erzielt. Dennoch wurde schnell klar, dass Deep Learning und Machine Learning im Allgemeinen neben der Erstellung eines akkuraten Modells verschiedene zusätzliche Herausforderungen mit sich bringen. Forscher waren sehr aktiv, um den klassischen Workflow des maschinellen Lernens zu untersuchen und Best Practices aus dem Lebenszyklus der Softwareentwicklung zu integrieren. Deep Learning weist jedoch Abweichungen auf, die in diesem konzeptionellen Entwicklungsprozess noch nicht berücksichtigt sind. Dazu gehören die Notwendigkeit dedizierter Hardware, entbehrliches Feature-Engineering, aufwendige Hyperparameter-Optimierung, umfangreiches Datenmanagement und Modellkompression zur Reduzierung der Grösse und Latenz. Die einzelnen Probleme von Deep Learning werden bis heute gründlich untersucht, und zahlreiche Konzepte und Implementierungen haben an Zugkraft gewonnen. Der komplette Entwicklungsprozess ist jedoch noch immer nicht spezifiziert. In dieser Arbeit haben wir einen detaillierten Deep-Learning-Workflow definiert, der die oben genannten Eigenschaften auf der Basis des klassischen Machine-Learning-Workflows einbezieht. Des Weiteren haben wir die konzeptionelle Idee in die Praxis übertragen, indem wir ein prototypisches Deep-Learning-System unter Verwendung der neuesten Technologien auf dem Markt entwickelt haben. Um die Anwendbarkeit des Workflows zu untersuchen, wurden zwei Beispiele auf den Prototyp angewendet. Der erste Anwendungsfall stellte ein Textklassifizierungsproblem dar, während sich der zweite Anwendungsfall auf die Bildverarbeitung konzentrierte. Damit konnten wir die Anwendung des Workflows an unterschiedlichen Beispielen erfolgreich demonstrieren. Zusammenfassend wird deutlich, dass der Deep-Learning-Lebenszyklus eine grosse Anzahl von Aktivitäten umfasst und verschiedene Rollen involviert. Mit unserem definierten Workflow stellen wir einen fundierten Leitfaden für den Deep-Learning-Entwicklungsprozess vor. Darüber hinaus kommen wir zu dem Schluss, dass die derzeit auf dem Markt verfügbaren Technologien nicht vollständig ausgereift sind. Es ist ein grosser Aufwand erforderlich, um alle Deep-Learning-Artefakte zu verwalten und die Versionen innerhalb kontinuierlicher Iterationen über den Lebenszyklus hinweg abzustimmen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rise of big data, cloud computing and other topics of the digital age, artificial intelligence (AI) has emerged as a promising field to harness the potential of the tremendous information growth. AI enables humanity to teach machines how to solve problems that may be intuitive, but difficult to describe formally, *i.e.*, feel automatic to human individuals [16]. There are numerous active research topics in AI, and many findings have been successfully adopted in practice. One of these topics is deep learning – a subset of machine learning – which is applied in domains such as image processing, speech recognition, or autonomous driving [21]. The core concept of deep learning is based on biomimicry of interconnected human neurons, so-called neural networks. Neural networks form a computational model composed of multiple processing layers to learn the representation of data and predict the output of raw input [34].

Together with the emergence of various new data-centric fields, the role of a data scientist was introduced in the early 20th century. Their main responsibility is to transform data into insights [31]. By today, as more software applications tend to include AI features, data scientists are often incorporated into software engineering teams to facilitate multidisciplinary development [30]. This incorporation requires software engineers to learn how to work with data science specialists [6]. On one hand, developing AI software requires proficiency in data analysis and statistics, *e.g.*, for neural network design and evaluation. On the other hand, additional engineering aspects are also demanded, such as the configuration of the application environment and deployment solution to facilitate a continuous development lifecycle [21].

The traditional software engineering lifecycle is usually maintained through continuous integration (CI) and continuous delivery (CD) to enable planning, development and deployment of a software artifact. Moreover, DevOps – which stands for development and operations – has manifested as a practice (and even as a culture) to merge continuous lifecycle management into a single set of processes. Although the machine learning lifecycle is different from traditional software development [6], a similar framework named MLOps has been introduced, *i.e.*, machine learning operations. In this process setting, data scientists create the development environment, while software/operation engineers are responsible for the setup of the production environment [28].

Problem Domain. In general, machine learning lifecycles require sophisticated pipelines, that facilitate data management, training, deployment and model integration into the corresponding product. A previous case study at Microsoft [6] was conducted to describe the current concept of machine learning development and proposed an abstract workflow reaching from model requirements up to model monitoring in production. In general, a workflow consists of an ordered

sequence of activities required to achieve one or multiple goals [1]. An activity is defined as a task contributing to the defined objective, which can be performed either automatically or manually by a determined individual. Regarding information technology, a workflow provides systematic organization and reproducibility to the development process, while reducing costs and increasing productivity [11].

However, in the context of deep learning, there is still lack of guidance when it comes to integrating it into the software development process. Such a workflow may deviate from the one proposed by Microsoft. For example, while conventional machine learning is based on manual feature engineering, deep learning is based on an end-to-end approach, *i.e.*, features are learned automatically [38]. Moreover, high performance graphical processing units (GPU) are recommended to be able to train a neural network in reasonable time, since the training process is computationally expensive [3]. Several of these problems specific to deep learning have been addressed in past research. For instance, advanced algorithms have been proposed to accelerate the process of finding the optimal parameters and new platforms have been introduced to make GPU training more accessible to researches and practitioners [26, 44]. However, the concepts and technologies presented are still relatively new and not yet fully mature [22]. Additionally, these individual solutions have not yet been assembled to an end-to-end development process for deep learning. A conceptual representation of the complete workflow and a corresponding implementation still remains undefined.

Contribution. To overcome the above stated knowledge gap, the aim of this thesis is to investigate the current development workflow of deep learning in the context of machine learning lifecycle management. The goal is to specify best-practice guidelines from model development to deployment and execution, *i.e.*, bringing deep learning models into production. Therefore, lifecycle critical differences to conventional machine learning are collected and integrated into an extended workflow for the deep learning lifecycle. Additionally, a prototype will be implemented to demonstrate the practicability of the defined workflow. Overall, this thesis will summarize the current state of research focused on the deep learning workflow and investigate the applicability into practice. We will demonstrate that our abstract definition of the workflow can be successfully utilized in common deep learning applications. Moreover, despite the construction of an effective prototype, we indicate that the technologies currently available on the market are not fully mature.

Outline. This thesis is organized as follows. Chapter 2 summarizes the theoretical baseline for this thesis, which includes the workflow defined for classical machine learning, the roles involved in a machine learning team and the decisive characteristics of deep learning. In Chapter 3 we will describe the correlated research. We will then derive the abstract deep learning workflow in Chapter 4, using the collected particularities of deep learning. This abstract definition is then implemented in a minimum viable prototype in Chapter 5, where we select a set of technologies to facilitate end-to-end deep learning. The usage of our prototype and consequently the applicability of our abstract workflow is then demonstrated on two distinct use cases in Chapter 6. Our findings are discussed in Chapter 7 and ultimately concluded in Chapter 8.

Chapter 2

Background

Within the following sections, we build the theoretical baseline to investigate the deep learning workflow. Therefore, the classical machine learning workflow is outlined together with the roles involved in a machine learning team. Most importantly, we discuss the distinct characteristics of deep learning that have an influence on the workflow.

2.1 Classical Machine Learning Workflow

Although many software engineering principles can be transferred to machine learning development, various new challenges have to be solved [36]. In comparison, machine learning projects exhibit many critical differences. For example, development is data-centric instead of code-centric, individual modules are hard to isolate, and the project team is more divers in terms of the required skill-set [6]. Maintenance is more difficult and costly than development, as a model needs to be continuously improved and adopted to a changing environment [55]. This leads to more frequent iterations over the workflow compared to classical software engineering. Due to the non-deterministic behavior, machine learning becomes a highly experimental process, which brings up the need for reproducibility [62].

Thus, researchers have been actively investigating the machine learning workflow. Despite minor differences, the current literature has agreed on a converging conceptual idea of the workflow [6,7, 33,37]. In the context of this thesis, we rely on the workflow definition of Amershi *et al.* [6], which is composed of the nine steps illustrated in Figure 2.1. As deep learning is a subset of machine learning, all the defined steps can be incorporated into the deep learning workflow. However, the embedded steps will be extended and expressed in more detail to meet the requirements of deep learning. We thus take their workflow definition as a starting point.

The steps defined by Amershi *et al.* can briefly be summarized as follows: the first stage of the workflow, *model requirements*, focuses on the problem analysis from the business perspective and corresponding solutions [6]. The following steps of *data collection*, *cleaning* and *labeling* are targeted towards providing the necessary datasets to train and test a model [33]. The fifth workflow component called *feature engineering* includes the selection and transformation of informative elements of the raw data into an appropriate form for model inputs [66]. During *model training*, a set of algorithms then learns from the selected features and is subsequently tested during *model evaluation*. Such that the produced model can be used by applications or end-users, *model deployment* compromises the tasks to serve a model for inference [6]. As a final stage, the main purpose of

model monitoring is detecting the moment where a model no longer fits its purpose and therefore is required to be revised [33].

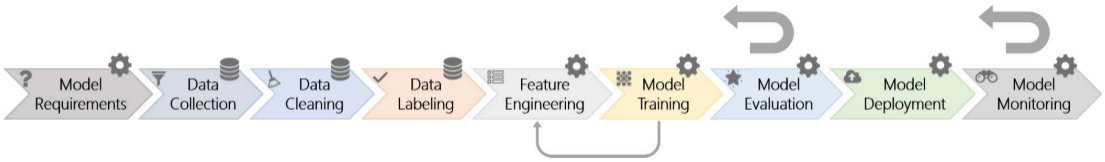

Figure 2.1: The classical machine learning workflow by Amershi *et al.* [6]

Although these steps are theoretically executed in sequence, the machine learning workflow is not necessarily tied to a linear timeline. The workflow involves several feedback loops, illustrated by backward arrows in Figure 2.1. For example, *model monitoring* may uncover a distribution shift of the data, which indicates that the training and/or testing data needs to be updated. On the other hand, the stage of *model evaluation* may reveal insufficient performance and compel a return to previous steps of the workflow. Because of these feedback loops, the machine learning lifecycle exhibits higher complexity than traditional software engineering [55].

2.2 Roles in Machine Learning

There are many actors involved in the machine learning lifecycle, each having various and possibly overlapping responsibilities. Machine learning is a fast changing field, thus the engaged roles and their corresponding tasks are not absolutely well-defined [54]. In the context of this thesis, we define six important roles based on the current literature [7, 33, 37]. These roles are later matched to the deep learning workflow based on their responsibilities.

Data Engineer. The main essential responsibility of a data engineer – in the context of machine learning – is to provide the required data to build a model. To do so, they construct scalable pipelines that acquire, prepare and store data securely [33]. Data engineers should also be able to help continuously train and deploy existing models [5]. Thus, they work in close collaboration with data scientists.

Data Labeler. Semi-supervised and supervised machine learning approaches require labeled data to train and evaluate a model. Most often, human effort is necessary to provide labels for the datasets prepared by the data engineer. Thereby, it is essential to have a highly accurate ground-truth for a model to become performant [5]. Labeling data is thus a crucial step within the machine learning workflow. The role of a data labeler can be filled internally, *e.g.*, by data engineers or domain experts, or externally through crowd-sourcing [5].

DevOps Engineer. In general, a DevOps Engineer supports machine learning by introducing and maintaining processes, tools, and methodologies [52]. They set up the initial infrastructure that will be used by other roles during the machine learning workflow. Moreover, they support model deployment and monitoring. Depending on the project environment, a DevOps Engineer is not necessarily part of the machine learning team but still involved in the workflow [52].

Data Scientist. When it comes to data analysis, exploration, feature engineering and model building, the data scientist serves as the central role [33]. They are responsible for prototyping a machine learning model, commonly in an experimental environment such as a notebook. Besides statistical and mathematical knowledge, data scientists are further required to possess domain knowledge for their model to create business value [7].

Software Engineer. Within the machine learning team, software engineers (or developers) with the appropriate machine learning knowledge help productionizing a model. As data scientists often only build prototypes within an experimental environment, their work needs to be put into a deployable form [33]. This includes for example tasks such as API design for effective prediction requests.

Model Validator. Also referred to as the data science manager, a model validator takes responsibility of the project. They possess knowledge from both field – machine learning and software engineering – and additionally bring domain expertise to ensure the business requirements are accomplished [12].

2.3 Workflow Critical Aspects of Deep Learning

In this thesis, we focus on supervised deep learning. Nevertheless, unsupervised approaches have been proposed in research, such as clustering analysis, sample specificity analysis, self-supervised learning and generative models [25]. The following sections introduce the characteristics of deep learning that have an impact on the development process. These will later be used to derive the deep learning workflow.

Data Management. A decisive property of deep learning is the ability to automatically learn multiple levels of data representations and abstract information, which diminishes the need for manual feature engineering [16]. This advantage in turn increases the necessity of having high-quality data to train a model on. Moreover, deep learning algorithms often rely on large amounts of data to make accurate predictions [42]. Therefore, data management at scale is crucial within the deep learning workflow. This, however, imposes the need for data pipelines to reduce human effort and automate the process of handling batch or streaming data [41]. Such pipelines should ingest the collected data into an appropriate storage solution, which can handle structure or unstructured objects together with their metadata [7]. In case of regulated industries or sensitive data in general, a secure storage location is required [42]. Furthermore, data needs to be continuously updated as the model environment changes. One common problem is concept drift, where the relationship between model input and output changes. This occurs for example due to a cultural transformation, which results in altered user behavior [29].

Computing Resources and Scalability. Machine learning algorithms in general are very resource demanding, which is especially true for deep learning [64]. When it comes to massive datasets and complex models, appropriate hardware is indispensable to stay within reasonable training periods. On that account, researchers and practitioners often resort to GPUs for parallelization and distributed training [3]. Several frameworks and platforms have evolved to simplify the use of GPUs [43]. However, specialized hardware may not only be required when training or optimizing a model. Other steps within the deep learning workflow can become computationally intensive as well, such as large-scale data preparation or model serving when response

time performance is critical [17]. Thus, various roles involved in the workflow should be able to access compute resources on demand and at scale.

Hyperparameter Optimization. Deep learning algorithms require the configuration of numerous hyperparameters, for example the number of hidden layers and nodes, a learning rate and an activation function [26]. These variables are set in advance of training and can have great influence on model performance [51]. Thus, there is a need for optimization. However, as the model performance can only be validated during training, hyperparameter optimization can become costly in terms of time and compute resources. Over the years, various frameworks have emerged in the context of *AutoML* to reduce costs and accelerate this process, *e.g.*, through early stopping of unpromising configurations or automated neural architecture search. While building a model, a data scientist should therefore have access to such a platform or framework to automate and simplify the process of hyperparameter tuning. Moreover, it should be comprehensible how the optimal configuration was deduced.

Model Compression. There are various ways to deploy and serve machine learning models. Depending on the use case, a model can be restricted by its environment in production. For example, when deploying to edge devices, memory and processing power is limited [48]. On the other hand, latency is crucial when a model serves in real-time [23]. As existing deep learning models tend to be large and resource intensive, various approaches have evolved to overcome these restrictions. In general, the techniques applied to compress a model and reduce resource requirements include: pruning and quantization, low-rank factorization, convolutional filters and knowledge distillation [9]. The methods of choice usually depend on the application domain, accuracy requirements, type of model and dataset size. While some methods can be applied to pretrained models, others do require a training from scratch.

Chapter 3

Related Work

While algorithms and frameworks to build machine learning models evolved quickly, other stages of the workflow have been neglected for a long time. However, to integrate machine learning into the current software applications, a need for a conceptual development process emerged.

Therefore, Amershi and his colleagues [6] investigated the development of machine learning applications at Microsoft. Through a case study, a high-level concept of the machine learning workflow composed of nine steps was deduced. Furthermore, they highlighted the current challenges imposed during the machine learning development and introduced a model to measure the machine learning process maturity.

Salama and his collaborators [53] took the machine learning workflow a step further. From a more practical perspective, they presented a conceptual representation of a fully integrated machine learning system targeted towards continuous adaption to the business environment. Within their work, it is illustrated what artifacts are produced during the workflow and how data is moved and transformed between stages.

Garcia *et al.* [14] argue that a crucial piece currently missing in the machine learning workflow was context. Unstructured and offhand transitions between stages – and consequently between roles – could impair productivity and reproducibility. Therefore, artifacts produced by an individual role should not appear as a black-box to other team members.

Furthermore, within the work of Haakman and his colleagues [22], it is argued that several steps within the machine learning lifecycle had been neglected up to now. The authors interviewed 17 machine learning practitioners at ING, a company which operates in the fintech industry. These interviews revealed that many existing workflow models do not compromise crucial steps such as data collection, feasibility study, documentation, risk assessment, model evaluation and monitoring. They stress that the machine learning development process should not only focus on algorithms, but the complete lifecycle. Additionally, it is stated that the existing tools for machine learning were not mature enough. Many practitioners would still rely on manual solutions, despite the existence of automating technologies. Indeed, there is a broad set of tools available on the market, with many still being in their early development phases [40, 59]. Moreover, very few are specifically targeted towards deep learning.

Regarding the deep learning lifecycle, Miau *et al.* [38] addressed the issue of managing models and their corresponding artifacts. They built a lifecycle management system for versioning models and a domain-specific language to query created deep learning models. Thereby, users can explore and compare hyperparameter tuning experiments using external frameworks and publish models.

Zhang *et al.* [64] conducted an empirical study concerning common challenges within deep learning development. By collecting questions and answers on *Stack Overflow* and building a classification model, they concluded five categories of common issues: API misuse, incorrect hyperparameter selection, GPU computation, static graph computation and limited debugging and profiling support. It is further stated that the current tool chain was not fully mature.

Moreover, Guo *et al.* [21] examined the influence of platforms and frameworks on deep learning development. Findings showed that a model suffers from diminishing accuracy when converting to another deep learning framework. They resulted to the conclusion that there would be a demand for a universal deep learning platform. Additionally, various practical guidelines are presented regarding stability and robustness of existing frameworks.

In summary, we argue that no investigation has yet been conducted on the complete workflow specifically for deep learning. Most research focuses on either a high-level abstraction of classical machine learning or the implementation of specific stages of the deep learning workflow.

Chapter 4

Definition of the Deep Learning Workflow

In this thesis, we want to deduce the workflow required to perform end-to-end deep learning and demonstrate the usage of the workflow through a minimal viable prototype. Therefore, this chapter outlines the abstract deep learning workflow which complies with the requirements listed in Chapter 2. First, we give a high-level overview of the components and roles required. Within further sections, we provide a more detailed illustrations of all activities and interactions.

4.1 A High Level Overview

We describe the deep learning workflow as a flow chart which defines the order of all activities and the corresponding roles to take responsibility. Furthermore, the flow chart demonstrates what data and operations may be involved at each step. The complete overview flow chart is visualized in Figure 4.1. It is important to mention that some steps described in this abstract workflow can be optional. We define the responsibility based on the general definition of a role in a data science team, as listed in Section 2.2. However, the allocation vastly depends on the project setting, the roles responsible for a specific activity are therefore not fixed.

Within the workflow, we distinguish between workflow steps and persistence entities, which store data produced by a workflow step. We define the following types of persistence entities:

- **Code Repository** stores source code within version control and allows sharing.
- **ML Data** holds versioned testing and training data prepared by a data engineer, including the corresponding metadata and labels.
- **Transformation Registry** stores preprocessed data specific to a model, produced by a data scientist for faster and more convenient access.
- **Experiment Registry** tracks configuration, metrics and results from an experiment conducted by a data scientist.
- **Model Registry** holds model artifacts of all models registered. This includes model definition (source code), configuration (hyperparameters, environment, etc.), metadata (version, creator, time, etc.), dependencies (*e.g.*, software packages, files), and most importantly the serialized model, *i.e.*, the trained weights in binary format.

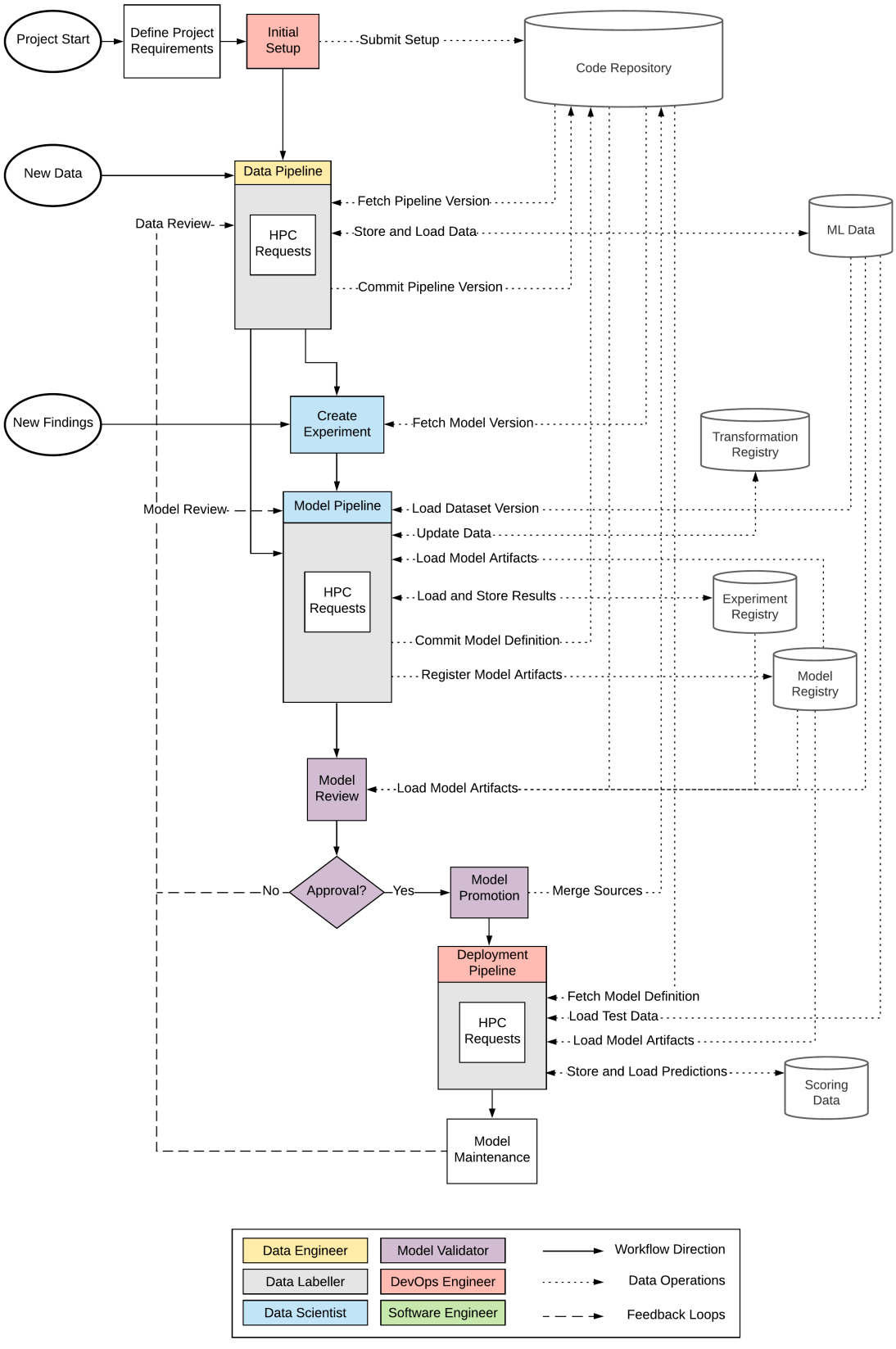

Figure 4.1: High-level overview of the deep learning workflow

- **Scoring Data** stores prediction requests and results for analysis and monitoring.

Our workflow initially only allows one starting point, namely the "Project Start". Moreover, there is no point of termination as deep learning – as well as classical machine learning – is a cyclic process due to continuous model improvements and adaptions to a possibly changing environment. Upon further iterations of the lifecycle, various roles have the option to step in at almost any stage of the workflow, either due to feedback loops or individual initiative.

At the beginning of every deep learning project, the requirements need to be defined. This is usually an interdisciplinary activity, involving business, research and engineering [37]. After coming to an agreement, a DevOps engineer is responsible for the initial setup. This includes the version-controlled code repository and other technical infrastructure. The code repository represents a central location to store, version and share source code, such that the complete workflow remains reproducible. Once the setup is complete, other team members can start with their implementations.

Theoretically, the deep learning workflow compromises three fundamental pipelines, namely the data, model and deployment pipeline, which are explicitly discussed in their respective section. All pipelines share the requirement of high-performance computing (HPC) requests, involve a subset of roles and interact with the defined persistence entities. These pipelines are not necessarily tied to a specific order on a linear timeline and can be executed independently. Nevertheless, the objective is to derive to a model in production. Once a model is deployed for inference, the final step of a workflow iteration, the "Model Maintenance", is determining when and where to re-enter the workflow. This can occur at various stages, illustrated by feedback loops on the flow chart.

4.2 Data Pipeline

The data pipeline, displayed as a flow chart in Figure 4.2, is the fundamental element of the deep learning workflow, as a deep learning model directly depends on the supplied data [42]. There are essentially two roles present within the data pipeline: the data engineer and the data labeler, whereas the latter is solely responsible for labeling data, as the name suggests.

The first step in the data pipeline – "Data Collection" – is defining the external sources for the data. If the test and training set do not have different origins, the data engineer divides the collected data in a reproducible manner at the subsequent step of "Data Splitting". This step may not be required on further iterations if the two datasets are updated independently. Once the sources for the training and test data are defined, the "Data Ingestion" step is targeted towards loading the external data into a suitable storage option, illustrated as "Machine Learning (ML) Data" in Figure 4.2. "Data Versioning" is indispensable and critical for data lineage [33] and thus performed directly after data ingestion, presumably in an automated fashion. Afterwards, the data engineer defines how the ingested data is to be cleaned and validated, before deciding whether the amount and quality of the data is sufficient for training a deep learning model. If this is not the case, the data engineer returns to the step of "Data Collection". Otherwise, the prepared datasets are approved and released for "Data labeling" by the internal or external data labeler.

The datasets generated by the data pipeline are required to be reproducible. Thus, not only the datasets themselves require versioning, but also the process that produced these datasets. The data engineer thus commits the definitions of all steps performed within the data pipeline to the version controlled code repository, initially set up by the DevOps engineer. On further iterations of the deep learning lifecycle, when there is already a model available, one can optionally trigger

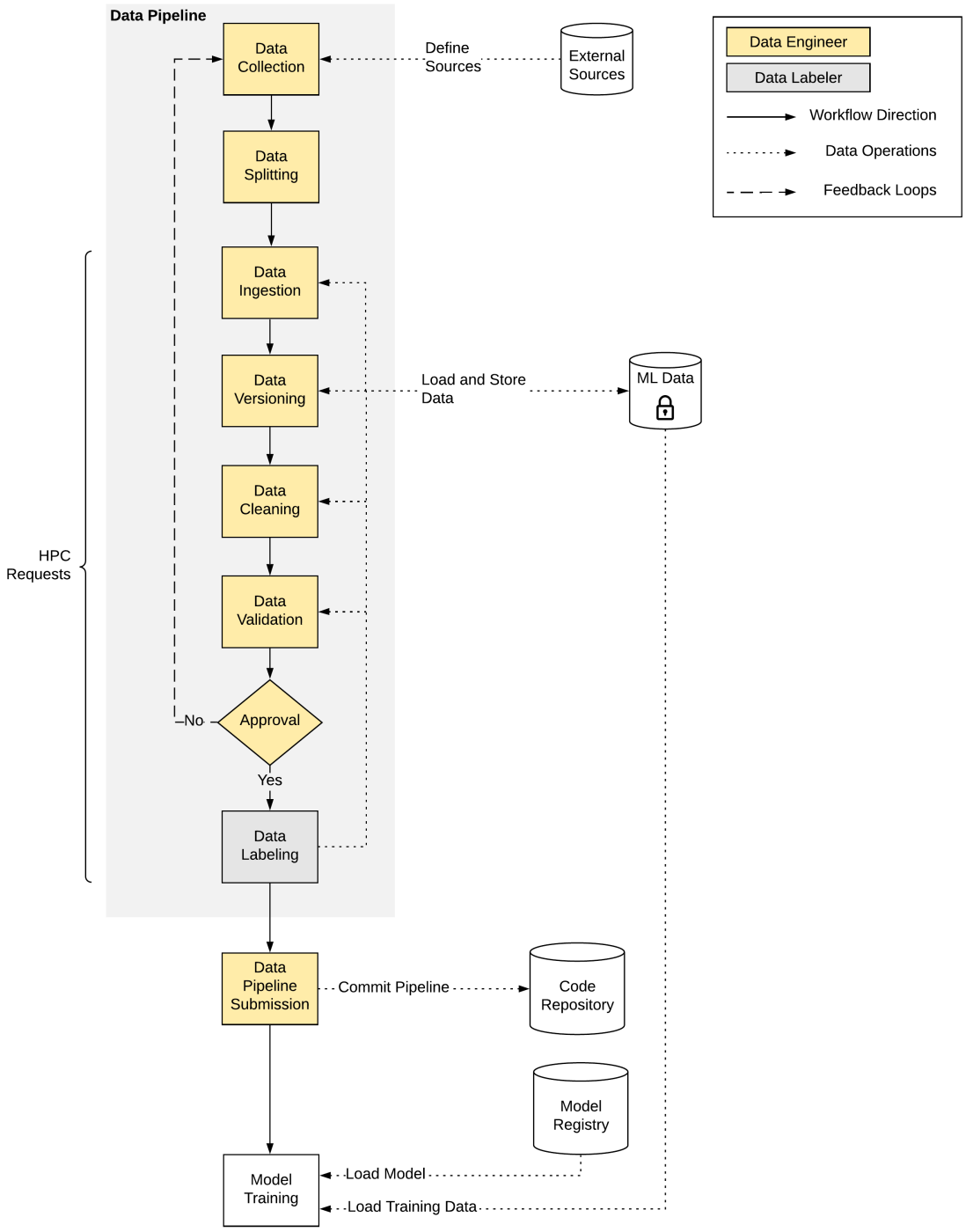

Figure 4.2: Definition of the abstract data pipeline

the execution of "Model Training" to analyze the performance with the new or updated dataset as a form of *Continuous Integration*.

When working with large and/or complex data, various steps within the data pipeline such as "Data Ingestion", "Data Cleaning" and "Data Validation" become resource demanding. Therefore, a data engineer should be able to request computing resources on demand within the steps of the pipeline.

4.3 Model Pipeline

Similar to the data pipeline, the model pipeline abstracted in the overview flow chart is defined in detail in Figure 4.3. There are two main actors within the model pipeline: the data scientist, with the aim to build a performant model, and the model validator, with the role of reviewing the model created by the data scientist. Similar to the data pipeline, a data scientist can request high-performance computing resources on demand throughout the course of all steps of the model pipeline.

Initially, an experiment is created by either modifying an existing version of the "Code Repository" or creating an experiment from scratch. As a first step of the pipeline, the data scientist analyzes the provided training data, whereby sensitive information may be hidden or masked. Direct access to the testing data may as well not be granted, *e.g.*, due to privacy reasons. After the "Data Analysis", the data is preprocessed to be compatible as model input if necessary. At this step, the data scientist may choose to store a version of the transformed data within the "Transformation Registry" for faster and more convenient access on further iterations. If the data transformation changes on subsequent iterations, the data needs to be stored again, otherwise it can simply be loaded into the experimentation environment. As a next step, the data is split into a training and validation set, before the data scientist starts building a model. In contrast to the "Data Splitting" step of the data pipeline, this step further divides the training data and is not relevant to other pipelines. As opposed to the test dataset, the validation set is not further used.

Upon "Model Building", one has the option to load pretrained models from the model registry. Within the flow chart in Figure 4.3, the model registry is illustrated as a single persistence entity. However, pretrained models can be loaded from any private or public registry. Thus, there may be multiple registries available to the data scientist for loading pretrained models. As an optional step before training a model, "Hyperparameter (HP) Optimization" helps a data scientist finding the optimal configuration of a defined model. Subsequently, a training job is launched, which can optionally be distributed over computing instances, given the training is resource-intensive.

Whenever a training or hyperparameter tuning job is executed, the corresponding metadata, metrics and results are stored within the "Experiment Registry". This acts as a central location for experimentation history, not only available to the data scientist building the current model, but also to other data scientists and model validators for review. Thereby, the evolution of a model remains comprehensible.

At the step of "Experiment Evaluation", the data scientist reviews his training experiment based on the validation results and other metrics [65]. If not satisfied, they return to previous stages of the model pipeline. Otherwise, the model is registered on the "Model Registry" to make it available for "Model Evaluation". At this point, the testing data is loaded together with the model artifacts to test the produced model. In case the testing data contains sensitive information, the "Model Evaluation" can be conducted in a secure environment. Consequently, the test results are

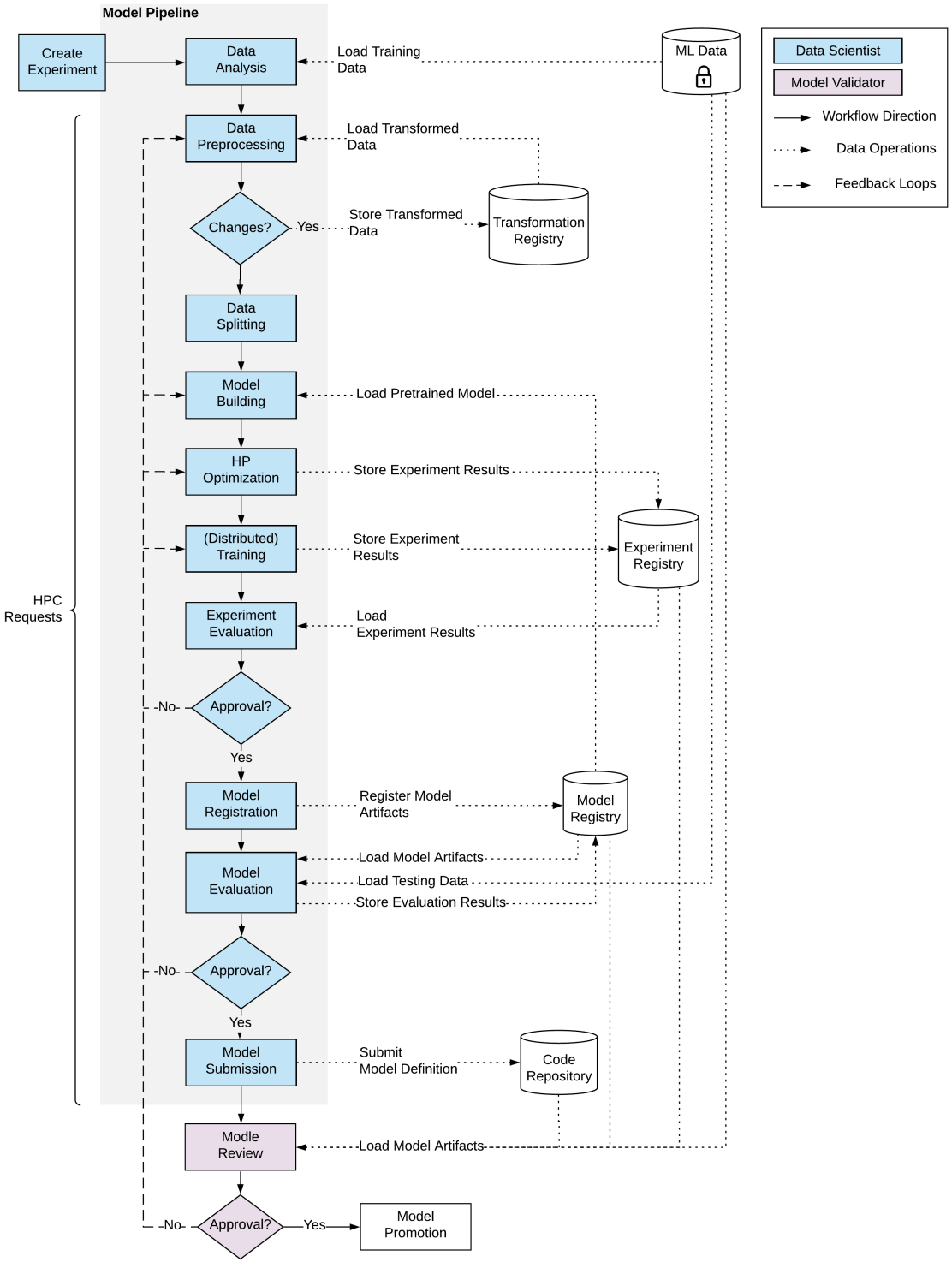

Figure 4.3: Definition of the abstract model pipeline

stored to the metrics of the model within the registry. Based on the evaluation results, the data scientist then decides to either submit the model for review or return to previous steps of the model pipeline.

During the "Model Submission" step, the corresponding source code of the registered model is submitted to the code repository, although the source code is included in the model artifacts. However, the "Code Repository" should hold all source code required to reproduce a workflow iteration.

After the data scientist submitted his model, the model validator reviews the registered and evaluated model by loading corresponding artifacts. Such a review focuses on model quality and can include various metrics such as accuracy, sensitivity, precision, different error measures or ranking methods [57]. These metrics can be compared to other produced models, possibly already deployed to production. If the results are not satisfactory, data scientists can return to specific steps within the model pipeline and improve the model version at the model validator's request. Furthermore, the model validator can instruct the data engineer to collect new or more qualitative data, as illustrated in the overview flow chart in Figure 4.1. In case of approval, the created model is promoted to production, which triggers the deployment pipeline.

4.4 Deployment Pipeline

Once a model has been approved and promoted to production, the goal is to deploy the model. Besides other roles involved in the deployment pipeline, the DevOps engineer is primarily responsible for bringing a model into the deployment environment. As in the data and model pipeline, there are certain steps within the pipeline that require computing resources on demand, such as "Model Deployment". The deployment pipeline is illustrated in Figure 4.4.

In case the model is not yet suitable for future retraining, the source code needs to be refactored into a performant, automation and testing friendly form. This task called "Model Implementation" is performed by a software engineer as a first step of the deployment pipeline. A software engineer has advanced knowledge on runtime performance and memory usage and can therefor make the source code more efficient for retraining. The implementation is further reviewed by the model validator and stored to the model registry with the corresponding model.

As an optional step within the deployment pipeline, the model can be compressed to reduce size and latency, *e.g.*, by using the methods listed in Section 2.3: pruning and quantization, low-rank factorization, convolutional filters and knowledge distillation. In this case, the model needs to be tested and compared to the initial model to prevent a significant decrease in accuracy. For certain model compression techniques, retraining the model is additionally required before testing [9]. This is conducted during the stage of "Model Revision". The model artifacts in the "Model Registry" have to be updated, if the model definition has been refactored.

After the preceding preparation steps, the DevOps engineer packages the model into an appropriate form for inference, which wraps the loaded model with an additional layer to serve prediction requests. Subsequently, they write a manifest which defines the deployment configuration and finally deploy the model to production.

Once the model is deployed, a data scientist is required to monitor the model for environment changes, *e.g.*, for concept drifts as described in Section 2.3. Therefore, input data from prediction requests together with their metadata and results are stored within a database for analysis. In case the model performance declines or other issues occur, the final step of "Model Maintenance" initializes the next lifecycle iteration based on the interpretation of the "scoring data".

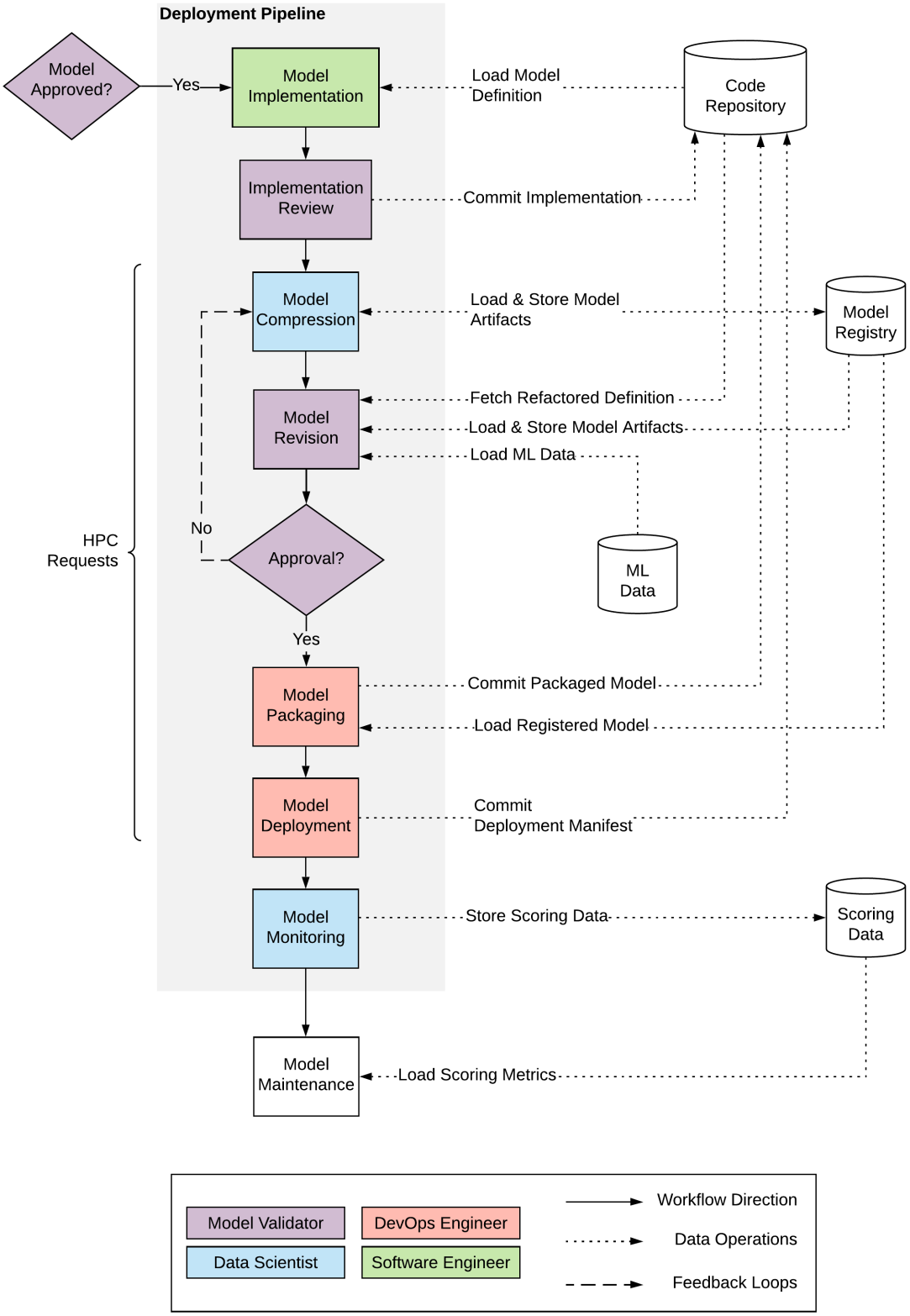

Figure 4.4: Definition of the abstract deployment pipeline

Chapter 5

Prototype Implementation

Throughout the following sections, we implement a minimum viable prototype that enables the deep learning workflow defined in Chapter 4. Thereby, we demonstrate the transfer of the conceptual workflow into a practical implementation. To do so, we select available technologies that fulfill the requirements given through the characteristics of deep learning listed in Section 2.3 and our defined workflow. Technical instructions on the prototype can be found in the associated *GitHub* repository at `https://github.com/janousy/CDL`.

5.1 Technologies of Choice

There is a relatively new, but fast-growing market for technologies that partially or completely facilitate the machine learning workflow. At the time of this thesis, the landscape of tools available is broad and not fully mature [59]. Few are targeted towards deep learning only, as their vendors provide generic solutions to problems of the machine learning workflow. Nevertheless, these solutions can often be transferred to various subtypes of machine learning. There are some emerging platforms that try to cover the complete machine learning workflow, such as *Vertex AI* or *Sagemaker* maintained by Google and Amazon respectively. However, these solutions can introduce immense costs, especially with frequent usage of GPUs [4, 18]. In behalf of reproducibility, we restrict our selection of tools to being deployable on-premise, available for free and preferably open-source.

5.1.1 Hardware and Infrastructure

We use the computing and storage resources provided by the ScienceCloud [58] of the University of Zurich (UZH), which lets us provision virtual machines (VM). In fact, two VMs are used:

- a large VM with 32 vCPUs, a single Nvidia Tesla T4 GPU and 128 GB of RAM;
- a small VM with 2 vCPUs and 8 GB of RAM.

The specifications of the large VM are chosen regarding the minimal resources to host all technologies required to run the deep learning workflow and meet the demands described in section Section 2.3. A single GPU is the maximum amount available per VM on the ScienceCloud. However, to demonstrate distributed training, at least two GPUs would be required. To independently and securely host a code repository, a second VM is introduced with near minimal specification to spare resources. Both VMs run *Ubuntu* 18.04 as their operating system.

To meet the requirement of scalability within the deep learning workflow, we select *Kubernetes* as our infrastructure of choice. *Kubernetes* serves an open-source system for scalable container orchestration [32]. In the context of this thesis, we use *Microk8s*, a lightweight upstream *Kubernetes* with low operation costs and GPU support [8]. However, the prototype is portable to any *Kubernetes* version. For our use case, a *Microk8s* single-node cluster is hosted on the large VM. As a container technology of choice, we use *Docker* due to being widely used across the developer community. Furthermore, *DockerHub* serves as the container registry to push and pull images.

5.1.2 Workflow Tools

Hereinafter, the technologies that directly support the implementation of the workflow tasks and persistence entities are described. Our selection is based on the recommendations of the MLOps Community [40] and Visengeriyeva *et al.* [59]. A summary of the technologies used, their purpose and what persistence entity they represent can be found in Table 5.1. It is important to note that this selection is not fixed and could be swapped with any tools with similar features.

Table 5.1: Technologies used to implement the deep learning workflow, not including underlying infrastructure.

Technology	Purpose	Persistence Entity
GitLab	version control, CI/CD	Code Repository
Minio	object storage	Model Registry
Postgres	store labels and experiment data	ML Data
LabelStudio	data labelling	-
Pachyderm	data lineage, object storage	ML Data, Transformation Registry
Determined AI	model pipeline	Model Registry
Seldon	model deployment	-

We choose *GitLab* [15] as our version control system (VCS), thereby representing the "Code Repository". *GitLab* can be hosted on premise, provides mature features for CI/CD and integrates well with *Kubernetes*.

To achieve data lineage, *i.e.*, data pipelines and version-controlled datasets, *Pachyderm* [45] is selected. *Pachyderm* allows us to execute containerized tasks on a *Kubernetes* cluster in a scalable, parallel and distributed manner. *Pachyderm* can serve as a general object storage technology, thus can be used to store unstructured datasets and as the "Transformation Registry" by the data scientist to cache transformed data.

Furthermore, we use *LabelStudio* [24] to integrate data labelling into our workflow implementation. *LabelStudio* is compatible with various types of data, especially unstructured data commonly used in deep learning applications such as computer vision, natural language processing and audio processing [5]. The labels – together with their metadata – are stored to a *PostgreSQL* [50] database for fast and convenient queries. We further refer to *PostgreSQL* as *Postgres*.

For tasks related to the model pipeline, we use a cloud-native platform specifically targeted towards deep learning called *Determined AI* [10], further referred to as *Determined*. This platform addresses the need for distributed training, hyperparameter tuning and compute resource management. *Determined* automatically tracks experiments for analysis and additionally provides a

model registry to store model artifacts. Thereby, a data scientist can focus on building and optimizing a model. Under the hood, a *Postgres* instance represents the "Experiment Registry", whereas a *Minio* [39] bucket is configured to store artifacts of the "Model Registry". *Minio* is a widely used, cloud-native object store.

To deploy models at scale to *Kubernetes*, *Seldon* [56] is used. *Seldon* supports a large spectrum of machine learning libraries and deployment configurations. A model can be brought to production by simply building a language wrapper around the model and specifying the container environment. Although there are alternative technologies to deploy machine learning models on *Kubernetes*, *Seldon* currently appears to be the most mature solution.

We do not implement the "Scoring Data" persistence entity, as "Model Monitoring" would exceed the scope of this thesis. Nevertheless, a *Postgres* database holding results of requests and possibly references to provided files stored to *Pachyderm* would be applicable.

5.2 Mapping Abstraction and Implementation

With the technologies selected in Section 5.1.2, we can build a deep learning system that implements our abstract workflow of Chapter 4. By mapping the technologies to tasks and persistence entities, we demonstrate how these integrate into the deep learning workflow. For each pipeline, we will walk through the practical utilization of the technologies.

5.2.1 Data Pipeline Implementation

Besides other tools for CI/CD, the main technologies within the data pipeline are *Pachyderm* and *LabelStudio*. Figure 5.1 illustrates how these technologies are integrated into the data pipeline.

The data engineer defines all steps of the pipeline with a programming language of choice, from "Data Collection" to "Data Validation". As mentioned in Section 4.2, the steps "Data Collection" and "Data Splitting" are not necessarily part of the automated pipeline. In our case, the "Data Labelling" also remains a manual step. The data engineer can define different sources for training and testing data – which implicitly splits the data – and then build separate pipelines. However, it is important that both data sets are processed the same way, *i.e.*, the same scripts for each step are used. Otherwise, the datasets could exhibit different characteristics, for example when the training and testing data are validated differently.

Once all steps are defined, the data engineer packages the scripts into a pipeline by writing a manifest complying to the *Pachyderm* format, which has either JSON or YAML format. Within this manifest, they optionally specify the resources to be used at each step, such as GPU, CPU and memory. Additionally, one can define how ingested data is processed, *e.g.*, as streams or in batches. The data engineer further defines the *Docker* container, wherein the pipeline is executed. They then commit their work to the *GitLab* code repository. This triggers the build of the *Docker* image and subsequently a push to *DockerHub*. Moreover, the pipeline is indirectly deployed to *Kubernetes* via *Pachyderm*. The execution of the pipeline is initiated each time data is ingested.

Pachyderm presents input and output repositories for each pipeline. Thus, the output of the "Data Validation" can automatically be ingested into *LabelStudio*. Once the data has been annotated by the data labeler, the labels are exported in a format of choice into another *Pachyderm* repository. From there, the labels are stored to a *Postgres* database, where the testing and training data reside in separate tables. Within a table, a row keeps information about the label, the corresponding

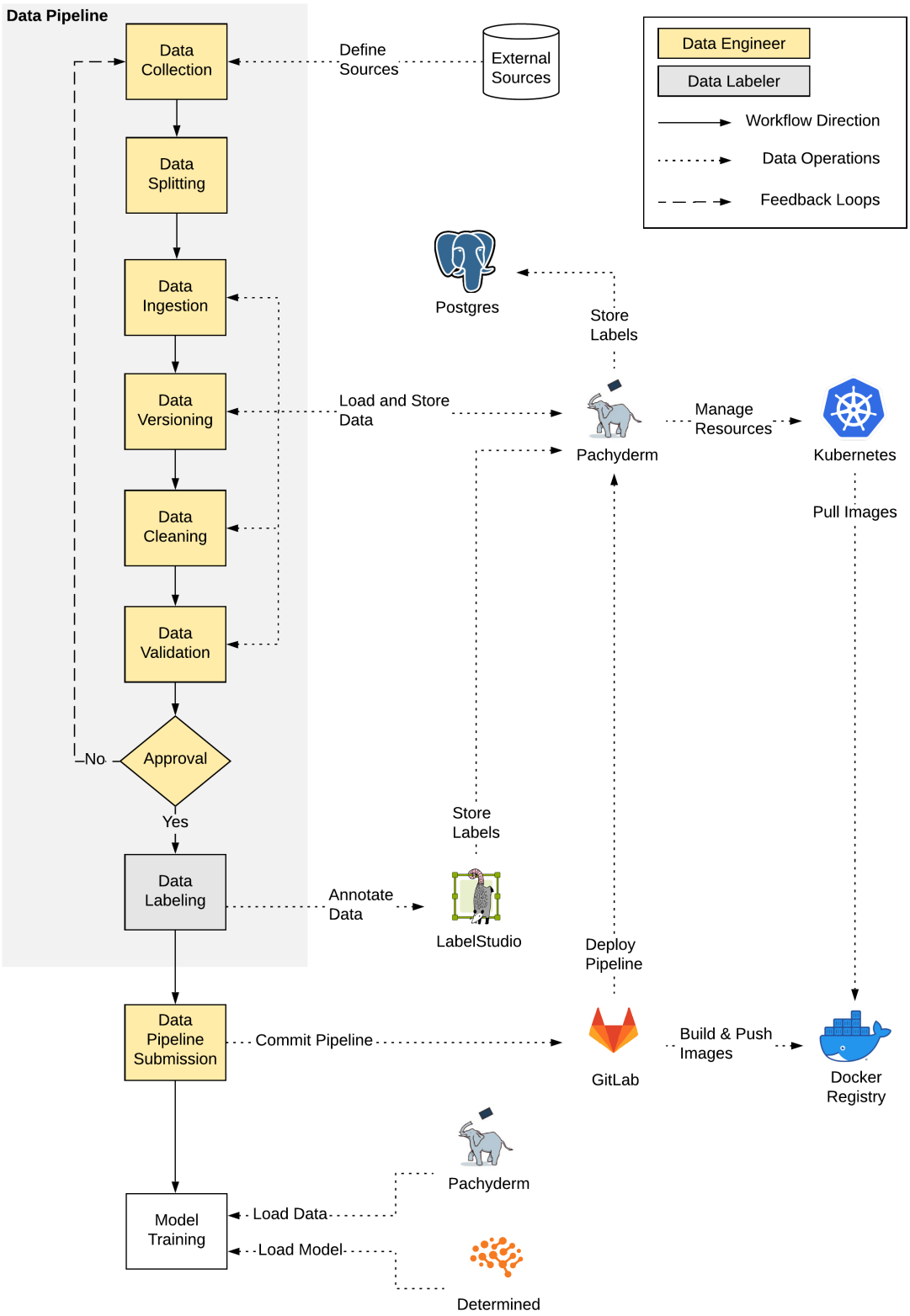

Figure 5.1: Implementation of the data pipeline, with the selected technologies mapped to the flow chart.

file path to the validated output repository and metadata such as labeler, date and dataset version.

"Data Versioning" is automatically performed by *Pachyderm* in a git-like manner. Each repository, *i.e.*, bucket, allows branching and annotates ingested data with commit IDs. Upon further iterations of the data pipeline, a data engineer can ingest on a new branch or distinguish between dataset versions using the commit ID within the same branch. Similarly, the pipeline manifest is versioned as well, thereby it remains fully comprehensible how a specific dataset version was produced. Data lineage is therefore guaranteed.

5.2.2 Model Pipeline Implementation

Within the implementation of the model pipeline, a data scientist mainly interacts with *Determined* and *Pachyderm*, as illustrated in Figure 5.2. To initiate an experiment, a new branch within *GitLab* is created, either from an existing branch or from scratch. While operating locally, they can choose to work in a notebook environment offered by *Determined* or directly write scripts as their source code. However, a notebook will have to be downloaded manually and checked into version control.

At the first step of the model pipeline, the training data is loaded from *Pachyderm*, analyzed and preprocessed. The transformed data can be stored to a *Pachyderm* repository and accessed on further iterations for faster development. After splitting the data into a train and validation set, they start building the model using a *Determined* compatible definition. Therefor, a trial class must be built that implements a predefined set of member functions for initialization, training, evaluation and data loading. Through these restrictions, a data scientist does not need to take care of logging and visualizing metrics or saving model checkpoints. Within the process of *Model Building*, pretrained models can be loaded from the internal *Determined* model registry or any external model registry, such as the Hugging Face transformer library [60].

Besides a model definition, *Determined* additionally requires a configuration file in YAML format, which specifies hyperparameters, resource requests, data source and version, etc. The data scientist can then use the same model definition with different configuration files for hyperparameter tuning and (distributed) training. Moreover, *Determined* executes jobs on agents within containers scheduled by a master. A data scientist can either specify software dependencies within a startup script or build a container image for the agent to run on. If a *Docker* image is used, it is built and pushed upon a commit to *GitLab* and subsequently pulled by *Determined* on a run execution. Within the process of "HP Optimization" and "(Distributed) Training", data scientists can review and compare their executed jobs on the *Determined* UI until they arrive at a satisfying model.

Once a data scientist approves of the validated model, they can register a selected model checkpoint through the *Determined* CLI. To synchronize the code repository with the latest model version, the corresponding model definition has to be downloaded manually from the model registry before committing. At the step of "Model Evaluation", a commit to the experiment branch on *GitHub* triggers the model testing, which loads the testing data from *Pachyderm* and the latest model version from the registry for evaluation. This commit is required as a model should be evaluated remotely and *GitLab* cannot listen for changes in the *Determined* registry due to a lack of change events. The results are then written to the model metrics and additionally presented as a *GitLab* pipeline artifact to the whole team.

If a data scientist agrees with the test results, they perform a *Model Submission* by creating a merge request on *GitLab*. The model validator then reviews the experiment. In case of approval, the deployment pipeline is triggered.

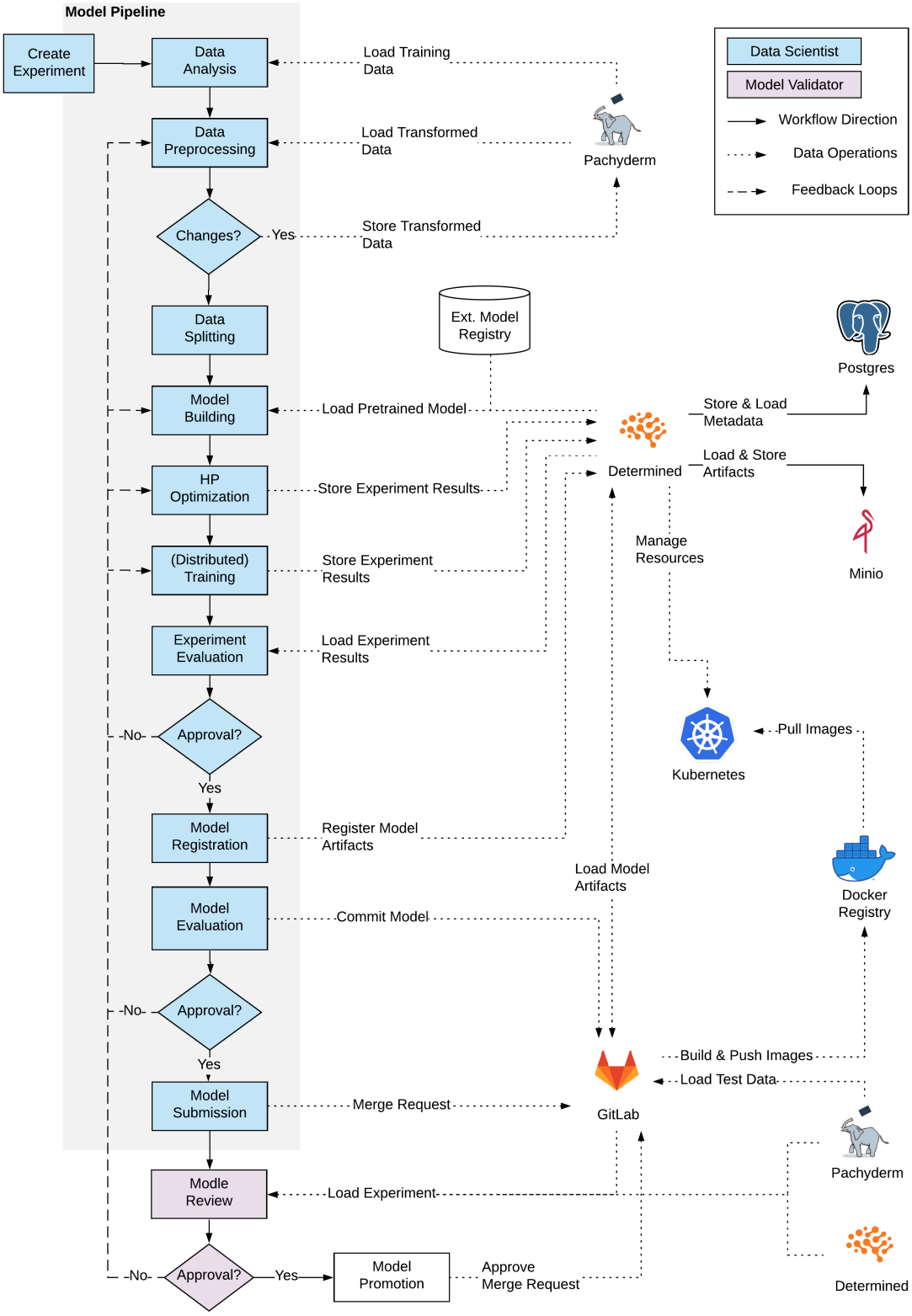

Figure 5.2: Implementation of the model pipeline, with the selected technologies mapped to the flow chart.

5.2.3 Deployment Pipeline Implementation

After the model validator has approved a model, it is prepared for production. An illustration of the implemented deployment pipeline is available in Figure 5.3. We point out that in the context of our minimum viable prototype, we skip two steps of the deployment pipeline. First, we do not perform "Model Compression", as our model is deployed to *Kubernetes* and inference latency is neglected. Secondly, "Model Monitoring" is omitted as this would exceed the scope of this thesis.

First, a software engineer fetches the model definition from *GitLab* to refactor the source code, which also includes the sources for data preprocessing. After the "Implementation Review" and "Model Compression", we retrain, evaluate and test the model for performance, *e.g.*, model size and inference latency, at the step of "Model Revision". The model validator therefor loads model artifacts from *Determined*, the refactored model definition from the code repository and the datasets from *Pachyderm*.

Once the revision is approved, the DevOps engineer builds the model wrapper for *Seldon*. The wrapper is essentially a Python class that at minimum defines how the model is loaded and how inputs are preprocessed for prediction. Additionally, a *Docker* image defines the container for the model environment at run-time. At the step of "Model Deployment", the DevOps engineer commits the deployment manifest to the main branch of the code repository to trigger the deployment. The deployment manifest specifies what resources are available to the model. A *GitLab* pipeline then builds and pushes the *Docker* image and deploys the packaged model to *Kubernetes* using *Seldon*.

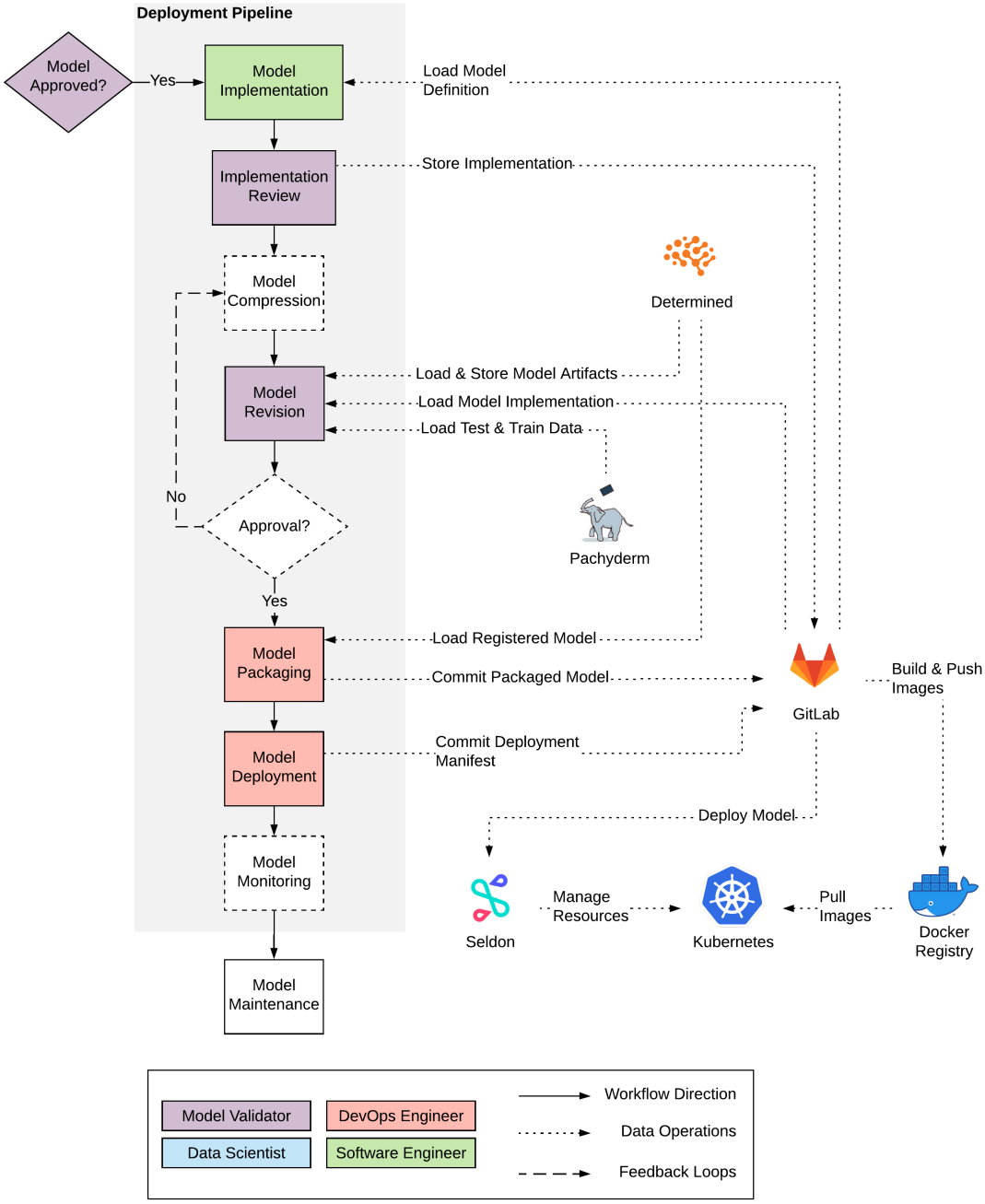

Figure 5.3: Implementation of the deployment pipeline, with the selected technologies mapped to the flow chart.

Chapter 6

Use Cases

To investigate the practicability of our implementation and the embedded pipelines established in Chapter 5, we apply two use cases to the prototype. These use cases address common but distinguished problems of deep learning, using different frameworks to provide variety. The source code for each use case including the setup is available on *GitHub* at `https://github.com/janousy/CDL`.

6.1 News Classification

In our first use case, the goal is to address a natural language processing (NLP) problem. Therefore, a multinomial news classification model trained and evaluated on the BBC datasets [20] is constructed using *PyTorch* [46].

The BBC datasets consists of 2225 text documents that represent news articles from the years 2004 and 2005, collected from the official BBC news website [20]. The documents in English have various length and are divided into the following categories: business, entertainment, politics, sport and tech. The datasets are available for download in two versions [19]: a pre-processed version, which includes stemming, stop-word removal and low term frequency count, and a raw version which provides the unprocessed articles. To be able to demonstrate the complete data pipeline, we use the raw version.

As a DevOps engineer, we initially set up a *GitLab* repository with a main folder and a *GitLab* pipeline for CI/CD. The main folder includes subdirectories for data, model, deployment and test code. The code repository has two branches, one for development (dev) and one for production (master).

6.1.1 Data Pipeline

A visualization of the data pipeline corresponding to this use case can be found in Figure 6.1. We assume the role of a data engineer and perform the first steps of the data pipeline manually using Python script. By downloading the dataset locally, giving each file a unique ID, merge each category and randomize the order, we prepare the data set for ingestion. Then, the dataset is split into a train and test set, *i.e.*, hold out set, using an 80-20 ratio [7]. Undoubtedly, there would be more sophisticated approaches [33]. Simultaneously to generating the train and test set, we automatically create labels in *LabelStudio* JSON format for each article, as we do not possess the resources for manual labeling.

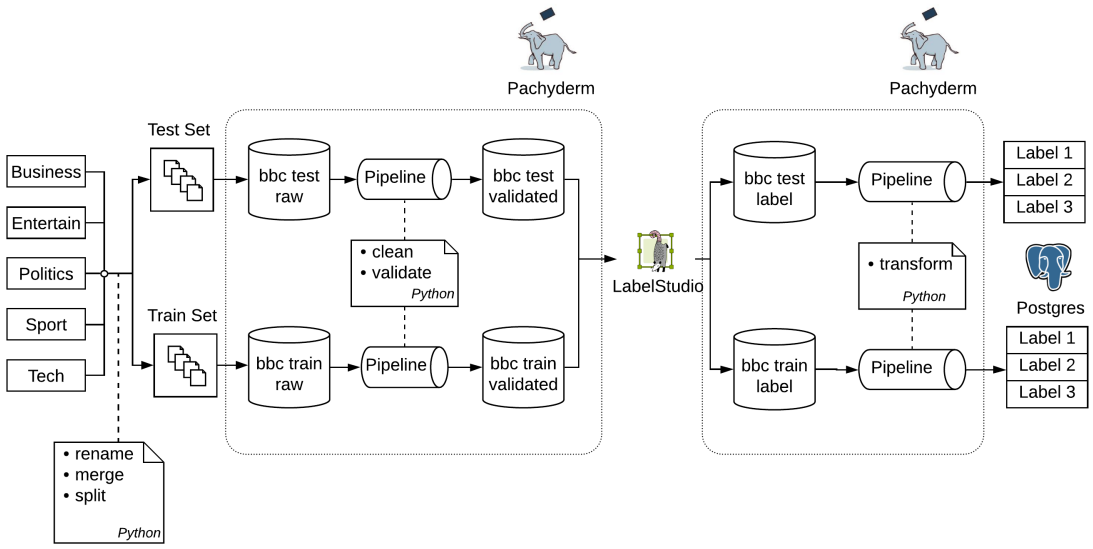

Figure 6.1: The data pipeline of news classification use case

A simple Python script then defines how data is ingested, cleaned and validated. We replace characters not being UTF-8 conform, ensure that the articles have a minimum character length and TXT file format. Articles not corresponding to the minimum character length or file format are invalid and therefore discarded. Valid articles are copied to the output repository defined by *Pachyderm*. After the steps of "Data Ingestion", "Data Cleaning" and "Data Validation" are defined, we initialize two separate *Pachyderm* repositories for the train and test datasets. Two pipelines for each dataset are then constructed that essentially execute the same aforementioned script with different input paths, as illustrated in Figure 6.1. For demonstration purposes, the validation pipeline requests one CPU and processes ingested data in batches.

We then prepare the *Docker* container for both pipelines to run on. The image installs the required packages and pulls the source code defining the pipeline. In this use case, the pipeline is directly committed to the production branch in the *GitLab* repository. Certainly, a more realistic application would require review and testing of the pipeline. *GitLab* then automatically builds and pushes the *Docker* image and deploys the pipelines to *Pachyderm* on *Kubernetes*.

Subsequently, both training and testing data can be ingested into the corresponding pipeline with a single *Pachyderm* CLI command. The validated data is synchronized into *LabelStudio* and available for manual labeling. An additional *Pachyderm* pipeline facilitates the export of the labels from *LabelStudio* into the respective table within the *Postgres* database. A single row holds the label itself, the file path to the article in the *Pachyderm* repository, additional metadata about the label and the matching *Pachyderm* branch for data lineage. Note that in this use case, a model is not automatically retrained upon ingestion of new data or changes to the pipeline, although this could be enabled using an additional *GitLab* pipeline stage.

6.1.2 Model Pipeline

We take over the role of a data scientist and start the first step of "Create Experiment". We create a new branch within the *GitLab* repository, in this case from the main branch. As we are already provided with sufficient knowledge about the BBC dataset, the step of "Data Analysis" is omitted. Nevertheless, the labels that reside in the *Postgres* training table are loaded together with the corresponding news articles from *Pachyderm*.

To transform the text data into a model conform input, a simple NLP approach is applied. This includes Porter stemming [49] and token count vectorization using the feature extraction capabilities of *scikit-learn* [47]. As a vocabulary, the one provided by Greene [19] is used. The target classes are similarly encoded into numeric values. The preprocessed data is then again split into a training and validation set at an 80-20 ratio [7]. In this use case, the preprocessed data is not stored to a "Transformation Registry" for reasons of simplicity. However, the data frame could certainly be stored and versioned in a *Pachyderm* repository.

During "Model Building", we locally create the required trial class for *Determined* using *PyTorch*. Our basic model is defined as a neural network of five linear layers with layer normalization and dropout applied. Additionally, a separate configuration file defines the *Pachyderm* repository and branch, hyperparameters such as dropout rate, hidden layer size and learning rate, and metadata about the experiment.

To start a hyperparameter tuning job, we specify a reasonable search space for each hyperparameter and then submit our configuration to the cluster using the *Determined* CLI. Vocabulary, classes and a list of required software packages are automatically uploaded to the "Model Registry". In the context of this use case, we use the ASHA algorithm, which supports early stopping of low performing configurations [35]. *Determined* then presents various visualizations and metrics to find the optimal hyperparameter configuration. Because parameters in the configuration are loaded during run-time, we can consequently use multiple configuration files for "HP Optimization" and "(Distributed) Training" with the same code for preprocessing and model definition. We thus write an additional configuration file for training using the results of the hyperparameter tuning job, and again submit everything to the cluster. As our infrastructure only compromises one GPU, we do not demonstrate distributed training. However, *Determined* enables networking, data loading and fault tolerance with the specification of a single configuration argument.

When finally arriving at a satisfying performance, we select the checkpoint UUID of the preferred experiment, register the model through the *Determined* CLI and push our local changes to the remote repository to trigger the *GitLab* pipeline and evaluate our model. *Model Evaluation* is facilitated through a Python unit test, which loads the model and the test data and only passes if a minimum accuracy threshold of 0.7 is reached. This value is set arbitrarily and usually dependent on the project requirements. Within an initial iteration of the deep learning workflow, either the data scientist himself or a DevOps engineer writes test cases. Who is responsible for testing a model highly depends on whether the test data is confidential.

Subsequently, we can request a merge onto the development branch. The *GitLab* pipeline produces a text document with the test results and a list of all model versions with their corresponding checkpoint UUID. Thereby, as the model validator, we can conclude the experiment within the *Determined* UI. If the model is adequate for production, the merge request is accepted.

6.1.3 Deployment Pipeline

The experiment conducted in Section 6.1.2 is now approved and merged into the development branch. As this is only a demonstration use case, we do not consider model performance. Thus,

the steps of *Model Implementation*, *Implementation Review* and *Model Revision* are omitted.

We further assume the role of a DevOps engineer. To package the model for deployment, we build a Python class that has two functions: one function initializes the model, therefor loads the model including its artifacts by downloading from the model registry within *Determined*. This includes the vocabulary and classes used during training and the registered model checkpoint. The other function defines how a prediction is performed. In this context, the input is transformed the same way as during model training, *i.e.*, stemmed and tokenized using the vocabulary. The numerical output returned from the model is resolved to the respective category using the label encoding.

To deploy a model using *Seldon*, we define a *Docker* image that copies the model wrapper and installs the required software dependencies. The deployment manifest is specified using a YAML file, whereby we serve the packaged model as a REST API and specify the Docker image as the surrounding container. At this stage, we could additionally specify resource requests such as the number of GPUs to be used for model serving. However, as only one GPU is available, this is omitted.

For continuous delivery of future model improvements, the steps of building and pushing the *Docker* images as well as deploying to *Kubernetes* are automated through *GitLab* pipeline jobs. These jobs are specified to be executed only on pushes to the production branch. In the context of testing, the model could additionally be deployed to a development environment as an additional staging process.

6.2 Fashion Classification

The second use case derives a multinomial image classification model using *TensorFlow* [2] and the Fashion-MNIST dataset provided by Zalando Research [61]. Thereby, a classical computer vision example of deep learning is tackled.

The Fashion-MNIST dataset consist of 60000 training and 10000 testing images of clothing articles [61]. The grayscale 28x28 images are divided into ten categories: T-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The dataset is available for download on *GitHub* [63] as separate binary files representing labels and images.

As in the first use case, we set up a *GitLab* repository which includes a *GitLab* pipeline and a main folder containing the sources in subdirectories and start with two branches for development and production.

6.2.1 Data Pipeline

As we already demonstrate a fully implemented data pipeline with unstructured data in our first use case, the approach in this second use case is simplified. In fact, the pipeline resembles in many aspects, but instead of storing text documents, we would store image files in the *Pachyderm* repositories. To simplify this use case, we directly store the compressed binary files containing the preprocessed fashion images and labels in a *Pachyderm* repository. We assume that the data has already been cleaned and validated, thus no data pipeline is constructed.

6.2.2 Model Pipeline

The model pipeline of this use case is very similar to the first use case in Section 6.1.2. Initially, an experiment branch is created from the production branch. Assuming the role of a data scientist, we load the labels and source images from the respective *Pachyderm* repository and decompress them. Again, the step of "Data Analysis" is omitted as sufficient knowledge about the dataset is already provided. To preprocess the dataset, the pixel values within a scale of 0 to 255 are normalized to a scale of 0 to 1. Similar to the first use case, we do not store the preprocessed data frame into a "Transformation Registry", although *Pachyderm* could very well be used therefor. A validation set is then split off at an 80-20 ratio.

To demonstrate the application of different machine learning frameworks, *TensorFlow Keras* is used to build our image classification model, based on an example provided in the official documentation [13]. For "Model Building", a sequential model is constructed composed of a flat input layer, a dense hidden layer of variable size and a dense layer fixed at the number of output categories. The remaining steps follow the process of the first use case: we optimize the hyperparameters, train the model, review experiments, register a model, evaluate it and finally request a merge to the development branch in the *GitLab* code repository.

6.2.3 Deployment Pipeline

As in the first use case in Section 6.1.3, we omit the first four steps of the deployment pipeline. Again, we - as DevOps engineers - build a Python class that packages, *i.e.*, wraps the model. At this step, only the classes defined in the model pipeline are loaded as model dependencies. We then construct a *Docker* image that starts the *Seldon* microservice and serves the model as a REST API. Merging the source code from the development branch onto the main production branch ultimately triggers the build of the *Docker* image and finally the deployment to *Kubernetes*.

Chapter 7

Discussion

In this thesis, we demonstrate a detailed step-by-step deep learning workflow derived from the high-level concepts of classical machine learning. Through a prototype, we show that it is possible to translate the conceptual idea to practice using the latest technologies available on the market.

We split the overall workflow into three pipelines, such that each component is independently reproducible and thereby enable fast iterations over the deep learning workflow. However, these pipelines in turn can further include sequences of automated steps. For example, the data management pipelines constructed in Section 5.2.1 include separated downstream pipelines for preparing data and storing labels, respectively. Together with additional pipelines for CI/CD, a project team can quickly face the challenge of "pipeline jungles" introduced by Sculley and his colleagues [55]. Managing these pipelines thus requires frequent knowledge exchange, versioning and well-connected interfaces.

Considering our abstraction of the deep learning workflow derived in Chapter 4, it becomes apparent that there is a dependency on various persistence entities with different functions. This increases data management costs and complicates collaboration, as knowledge about storing and retrieving data is required. On the contrary, these persistence entities are required to keep the workflow reproducible. With continuous iterations of the deep learning lifecycle, it becomes important to align the related data between persistence entities. As an example, additional effort is required to keep the registered model and the version in the code repository aligned. From a technical perspective, both persistence entities – "Model Registry" and "Code Repository" are crucial: On one hand, most version control system do not allow storing large artifacts such as a deep learning model. On the other hand, a "Model Registry" cannot provide a history of code changes and efficiently hold all source code of the deep learning workflow. An adopted version control concept that manages machine learning data, model artifacts, workflow source code, etc. as a single set of dependencies in a central location would facilitate the deep learning and the machine learning workflow in general. There are certain tools such as *DVC* and *CML* [27] that focus on this concept. However, these are incompatible with the requirements of the deep learning workflow since they do not provide scalability and compute resource orchestration.

In the context of the abstract model pipeline in Section 4.3, we decided to limit interactions with "Code Repository" to the last step of "Model Submission". Still, triggering hyperparameter optimization or training jobs could require a commit to the "Code Repository" in case a data scientist does not have direct access to the training platform. We argue that an interaction with the "Code Repository" is not necessarily required in case of direct access, as the "Model Registry" already keeps track of the model artifacts and thereby allows rollbacks to previous model versions. Most

often, the model definition does not change drastically. Committing to the VCS with every parameter change would be time-consuming. Furthermore, through direct submission of optimization or training jobs, debugging becomes less complex. As an example, we could trigger model training with *Determined* through a *GitLab* pipeline job, but then a data scientist would not have direct access to the training logs of *Determined*.

Regarding our prototype, it becomes clear that a large amount of different technologies is necessary to execute the complete deep learning lifecycle. Although there are all-in-one solutions available, these technologies often suffer from vendor lock-in and are associated with high costs. Moreover, these solutions are not directly target towards deep learning. The large technology stack imposes the need for interfaces between tools, activities and roles. This in turn introduces additional hazards to the implemented workflow. Taking "Data Labeling" as an example: we use *Pachyderm* since it allows building scalable data pipelines and managing compute resources. As *Pachyderm* does not provide any labeling features, *LabelStudio* is introduced. To store our labels to a *Postgres* table, *Pachyderm* again facilitates export and import. We therefore use a complex constellation of technologies within the data pipeline, which can become confusing and error-prone.

A minor shortage of using *Determined* is that – with the current version – we do not have control over experiment termination. It is thus not possible to notify a data scientist when training ends or trigger subsequent downstream tasks, such as "Model Evaluation". However, we argue that this is crucial as HP optimization or model training can be protracted.

Within our research, we presented an in-depth transfer of the machine learning workflow to deep learning. Through the implementation of a prototype and the application of two distinguished use cases, we demonstrated the practicability of our workflow. However, the use cases do not represent the complexity of real-world challenges. For example, we worked with exemplary datasets that do not require sophisticated data pipelines. Additionally, because our models in production are not exposed to actual requests, we did not implement "Model Monitoring" in our prototype. Thus, we cannot tackle problems such as concept drift. Therefore, further evaluation is required to investigate the validity of the presented deep learning workflow and extend the prototype with the not yet implemented steps. Furthermore, we restricted our investigation to supervised deep learning. However, unsupervised approaches may exhibit a deviating development workflow and reveal limitations.

Chapter 8

Conclusion

In general, the deep learning workflow follows the same high-level conceptual idea as the classical machine learning workflow. However, we summarized that deep learning exhibits many decisive deviations that have influence on the development process, *i.e.*, compute resource demands, sophisticated data management, dispensable feature engineering, extensive hyperparameter tuning and a potential need for model compression. Using these distinct characteristics and the current research on the classical machine learning development process, we derived a detailed deep learning workflow. Our workflow is represented as an abstract flow chart, which includes three pipelines for data, model and deployment. Responsibilities are assigned to each task based on the common roles in the data science team. Our flow chart additionally illustrates what data storage and operations are performed.

To investigate the practicability of the derived workflow, we proposed a *Kubernetes*-native prototype using suitable technologies available on the market to meet the requirements of scalability and compute resources. Through discussions with other members of the MLOps community on Slack, we were able to consider different perspectives and evaluate advantages and disadvantages of the considered tools. We demonstrated the utilization of the prototype by applying two use cases: first, we built a text classification model using *PyTorch* based on the BBC dataset [20]. Second, an image classification model based on the Fashion-MNIST dataset [61] was constructed with the use of *TensorFlow*.

Through the investigations of this thesis, we provide a general guideline on the deep learning development process that helps to build deep learning models and continuously improve them through efficient and reproducible iterations. Additionally, our recommended set of technologies can be used as a reference for future implementations.

As our two use cases do not represent real-world problems, further evaluation is required. Future work should therefor focus on evaluating the proposed deep learning workflow in various industries to find possible alterations, missing steps or inconsistencies. For instance, a field study could compare the proposed workflow to the processes within companies that have already brought deep learning into use. To do so, one could select a set of different companies – possibly with varying cultures – and analyze their workflow with the associated data science team. By accompanying multiple projects and workflow iterations, all performed activities are collected and mapped to our components to see whether each step is actually present and performed by the defined role. Interviews with the involved developers could protocol the reasons for their actions. This procedure would ultimately highlight abundant or missing steps within the proposed workflow of this thesis. Moreover, this field study would reveal inconsistencies in our suggested roles and liabilities. Such an evaluation could additionally lead to the findings of more applicable

technologies for an improved prototype. On the other hand, further research could address the development of new tools to overcome the limitations of the current tool landscape. For example, managing model-related data, *i.e.*, training and testing data, large model artifacts, related source code, etc. as a coherent set is still a major challenge. From a broader perspective, an all-in-one solution for the deep learning workflow could resolve the limitations of loosing context due to the high number of interfaces.

Bibliography

[1] Wil van der Aalst and Kees Max van Hee. *Workflow management: models, methods, and systems*. Cooperative information systems. MIT Press, Cambridge, Mass, 2002.

[2] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, September 2015.

[3] Kamil Aida-Zade, Elshan Mustafayev, and Samir Rustamov. Comparison of deep learning in neural networks on CPU and GPU-based frameworks. *11th IEEE International Conference on Application of Information and Communication Technologies, AICT 2017 - Proceedings*, pages 27–30, 2019. ISBN: 9781538605011 Publisher: IEEE. doi:10.1109/ICAICT.2017.8687085.

[4] Amazon Web Services Inc. Amazon SageMaker Pricing, 2021. URL: https://aws.amazon.com/sagemaker/pricing/?nc1=h{_}ls.

[5] Amazon Web Services Inc. Data Labeling, 2021. URL: https://aws.amazon.com/de/sagemaker/groundtruth/what-is-data-labeling/.

[6] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, pages 291–300, 2019. ISBN: 9781728117607 Publisher: IEEE. doi:10.1109/ICSE-SEIP.2019.00042.

[7] Andriy Burkov. *Machine learning engineering*. True Positive Inc., Québec, Canada, 2020.

[8] Canonical Ltd. MicroK8s - Zero-ops Kubernetes for developers, edge and IoT, 2021. URL: https://microk8s.io/.

[9] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv:1710.09282 [cs]*, June 2020. arXiv: 1710.09282. URL: http://arxiv.org/abs/1710.09282.

[10] Determined AI. High-Performance Distributed Deep Learning, 2021. URL: https://determined.ai/product.

[11] Asuman Doğaç, Leonid Kalinichenko, M. Tamer Özsu, and Amit Sheth, editors. *Workflow Management Systems and Interoperability*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. URL: http://link.springer.com/10.1007/978-3-642-58908-9, doi:10.1007/978-3-642-58908-9.

[12] Kirill Dubovikov. *Managing data science: effective strategies to manage data science projects and build a sustainable team*. Packt, November 2019. OCLC: 1126341974.

[13] François Chollet. Basic classification: Classify images of clothing | TensorFlow Core, 2017. URL: https://www.tensorflow.org/tutorials/keras/classification.

[14] R Garcia, V Sreekanti, N Yadwadkar, Daniel Crankshaw, Joseph E. Gonzalez, and Joseph M. Hellerstein. Context: The missing piece in the machine learning lifecycle. *Kdd Cmi*, 2018. URL: https://rlnsanz.github.io/dat/Flor{_}CMI{_}18{_}CameraReady.pdf.

[15] GitLab Inc. Open DevOps Platform, 2021. URL: https://about.gitlab.com/.

[16] Ian. Goodfellow. *Deep learning*. Adaptive computation and machine learning. MIT P., Cambridge, Mass, 2016.

[17] Google Cloud. Minimizing real-time prediction serving latency in machine learning, 2020. URL: https://cloud.google.com/solutions/machine-learning/minimizing-predictive-serving-latency-in-machine-learning.

[18] Google Cloud. Compute engine GPU pricing, 2021. URL: https://cloud.google.com/compute/gpus-pricing.

[19] Derek Greene. Insight - BBC datasets, 2020. URL: http://mlg.ucd.ie/datasets/bbc.html.

[20] Derek Greene and Pádraig Cunningham. Practical solutions to the problem of diagonal dominance in kernel document clustering. In *Proc. 23rd international conference on machine learning (ICML'06)*, pages 377–384. ACM Press, 2006.

[21] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *Proceedings - 2019 34th IEEE/ACM international conference on automated software engineering, ASE 2019*, pages 810–822. IEEE, 2019. arXiv: 1909.06727 tex.arxivid: 1909.06727. doi:10.1109/ASE.2019.00080.

[22] Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. AI lifecycle models need to be revised: An exploratory study in Fintech. *Empirical Software Engineering*, 26(5):95, July 2021. URL: https://link.springer.com/10.1007/s10664-021-09993-1, doi:10.1007/s10664-021-09993-1.

[23] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. *Proceedings - International Symposium on High-Performance Computer Architecture*, 2018-Febru:620–629, 2018. ISBN: 9781538636596. doi:10.1109/HPCA.2018.00059.

[24] Heartex Inc. Open source data labeling tool, 2021. URL: https://labelstud.io/.

[25] Jiabo Huang, Qi Dong, Shaogang Gong, and Xiatian Zhu. Unsupervised Deep Learning by Neighbourhood Discovery. *arXiv:1904.11567 [cs]*, May 2019. arXiv: 1904.11567. URL: http://arxiv.org/abs/1904.11567.

[26] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning*. The springer series on challenges in machine learning. Springer, Cham, 2019.

[27] Iterative. Developer tools for Machine Learning, 2021. URL: https://iterative.ai/.

[28] Ioannis Karamitsos, Saeed Albarhami, and Charalampos Apostolopoulos. Applying DevOps practices of continuous automation for machine learning. *Information-an International Interdisciplinary Journal*, 11(7):363, 2020. doi:10.3390/info11070363.

[29] Anton Khritankov. Analysis of hidden feedback loops in continuous machine learning systems. *ArXiv*, pages 1–7, 2021. arXiv: 2101.05673 ISBN: 9783030658540 tex.arxivid: 2101.05673. URL: http://arxiv.org/abs/2101.05673{%}0Ahttp://dx.doi.org/10.1007/978-3-030-65854-0{_}5, doi:10.1007/978-3-030-65854-0_5.

[30] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. *Proceedings - International Conference on Software Engineering*, 14-22-May-:96–107, 2016. ISBN: 9781450339001 Publisher: ACM. doi:10.1145/2884781.2884783.

[31] Miryung Kim, Thomas Zimmermann, Robert Deline, and Andrew Begel. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, 44(11):1024–1038, 2018. Publisher: IEEE. doi:10.1109/TSE.2017.2754374.

[32] Kubernetes. Production-grade container orchestration, 2021. URL: https://kubernetes.io/.

[33] Valliappa Lakshmanan. *Machine learning design patterns: Solutions to common challenges in data preparation, model building, and MLOps.* O'Reilly Media, Inc., Sebastopol, CA, October 2020.

[34] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi:10.1038/nature14539.

[35] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, B. Recht, and Ameet S. Talwalkar. Massively parallel hyperparameter tuning. *ArXiv*, abs/1810.05934, 2018.

[36] Giuliano Lorenzoni, P. Alencar, N. Nascimento, and D. Cowan. Machine learning model development from a software engineering perspective: A systematic literature review. *ArXiv*, abs/2102.07574, 2021.

[37] Treveil Mark, Nicolas Omont, Clément Stenac, Kenji Lefevre, Du Phan, Joachim Zentici, Adrien Lavoillotte, Makoto Miyazaki, and Lynn Heidmann. *Introducing MLOps*. O'Reilly Media, Inc., 2020.

[38] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. Towards unified data and lifecycle management for deep learning. In *Proceedings - international conference on data engineering*, pages 571–582. IEEE, 2017. arXiv: arXiv:1611.06224v1 ISSN: 10844627 tex.arxivid: arXiv:1611.06224v1. doi:10.1109/ICDE.2017.112.

[39] MinIO Inc. High performance, kubernetes native object storage, 2021. URL: https://min.io.

[40] MLOps Community. Best practice real-world machine learning operations., 2021. URL: https://mlops.community/.

[41] Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka, editors. *Product-Focused Software Process Improvement*, volume 12562 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2020. URL: https://link.springer.com/10.1007/978-3-030-64148-1, doi:10.1007/978-3-030-64148-1.

[42] Aiswarya Munappy, Jan Bosch, Helena Holmstrom Olsson, Anders Arpteg, and Bjorn Brinne. Data management challenges for deep learning. *Proceedings - 45th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2019*, pages 140–147, 2019. ISBN: 9781728132853. doi:10.1109/SEAA.2019.00030.

[43] Nvidia Corporation. Deep Learning, February 2015. URL: https://developer.nvidia.com/deep-learning.

[44] Nvidia Corporation. Inference: The next step in GPU-Accelerated deep learning | NVIDIA developer blog, 2015. URL: https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/.

[45] Pachyderm Inc. The data foundation for machine learning, 2021. URL: https://www.pachyderm.com/.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H Wallach, H Larochelle, A Beygelzimer, F d\textquotesingle Alché-Buc, E Fox, and R Garnett, editors, *Advances in neural information processing systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[48] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, abs/1802.05668(2015):1–21, 2018. arXiv: 1802.05668 tex.arxivid: 1802.05668.

[49] M.F. Porter. An algorithm for suffix stripping. *Program*, 40(3):211–218, January 2006. Publisher: Emerald Group Publishing Limited. doi:10.1108/00330330610681286.

[50] PostgreSQL Global Development Group. Advanced open source relational database, June 2021. URL: https://www.postgresql.org/.

[51] Philipp Probst, Anne Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20:1–32, 2019. arXiv: 1802.09596 tex.arxivid: 1802.09596.

[52] Red Hat Inc. Who is a DevOps engineer?, 2021. URL: https://www.redhat.com/en/topics/devops/devops-engineer.

[53] Khalid Salama, Jarek Kazmierczak, and Donna Schut. Practitioners guide to MLOps: A framework for continuous delivery and automation of machine learning., May 2021. URL: https://cloud.google.com/resources/mlops-whitepaper.

[54] Jeffrey S. Saltz and Nancy W. Grady. The ambiguity of data science team roles and the need for a data science workforce framework. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2355–2361, Boston, MA, December 2017. IEEE. URL: http://ieeexplore.ieee.org/document/8258190/, doi:10.1109/BigData.2017.8258190.

[55] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 2015-Janua:2503–2511, 2015.

[56] Seldon Technologies Ltd. Open source machine learning deployment for kubernetes, 2021. URL: https://www.seldon.io/tech/products/core/.

[57] Saleh Shahinfar, Paul Meek, and Greg Falzon. "How many images do I need?" Understanding how sample size per class affects deep learning model performance metrics for balanced designs in autonomous wildlife monitoring. *Ecological Informatics*, 57:101085, May 2020. URL: https://linkinghub.elsevier.com/retrieve/pii/S1574954120300352, doi:10.1016/j.ecoinf.2020.101085.

[58] University of Zurich. UZH ScienceCloud, 2021. URL: https://www.zi.uzh.ch/en/teaching-and-research/science-it/infrastructure/sciencecloud/.

[59] Laryza Visengeriyeva, Anja Kammer, Isabel Bär, Alexander Kniesz, and Michael Plöd. MLOps: Machine learning operations, 2021. URL: https://ml-ops.org/.

[60] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: System demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6, doi:10.18653/v1/2020.emnlp-demos.6.

[61] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017. arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747.

[62] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with MLflow. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 39–45, 2018.

[63] Zalando Research. Fashion-MNIST GitHub Page, June 2021. original-date: 2017-08-25T12:05:15Z. URL: https://github.com/zalandoresearch/fashion-mnist.

[64] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–115, October 2019. ISSN: 2332-6549. doi:10.1109/ISSRE.2019.00020.

[65] Alice Zheng. *Evaluating machine learning models*. O'Reilly Media, Inc., Sebastopol, CA, 2015.

[66] Alice Zheng. *Feature engineering for machine learning*. O'Reilly Media, Inc., Sebastopol, CA, 2018.