Continuous Deep Learning:
A Workflow to Bring Models into Production

Janosch Baltensperger
University of Zurich
Switzerland
janosch.baltensperger@uzh.ch

Pasquale Salza
University of Zurich
Switzerland
salza@ifi.uzh.ch

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

ABSTRACT

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 that 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. Unfortunately, the complete end-to-end development process still remains unspecified. In this paper, we define 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 some of the latest technologies on the market. To examine the feasibility of the workflow, two use cases are applied to the prototype.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**.

KEYWORDS

Deep learning, machine learning, continuous integration, software maintenance and evolution

1 INTRODUCTION

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 many software engineering principles can be transferred to machine learning (ML) development, various new challenges have to be solved [24]. In comparison, ML 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 diverse in terms of the required skill-set [5]. Maintenance is more difficult and costly than development, as a model needs to be continuously improved and adapted to a changing environment [38]. This leads to more frequent iterations over the workflow compared to classical software engineering. Due to the non-deterministic behavior, ML becomes a highly experimental process, which brings up the need for reproducibility [46].

In general, ML lifecycles require sophisticated pipelines, which facilitate data management, training, deployment, and model integration into the corresponding product. A previous case study at Microsoft [5] was conducted to describe the current concept of ML development and proposed an abstract workflow reaching from model requirements up to model monitoring in production. Principally, a workflow consists of an ordered sequence of activities required to achieve one or multiple goals [42]. 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 [6].

In the context of deep learning (DL), there is still a 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 [5] for multiple reasons: (1) while conventional ML is based on manual feature engineering, DL is based on an end-to-end approach, i.e., features are learned automatically [26]; (2) a large amount of high-quality data is required to accurately learn data representations, which introduces the need for scalable data management pipelines; (3) as neural networks (NNs) are computationally intensive, specialized hardware is required. This includes the use of graphics processing units (GPUs) for parallelization and distributed training [2]; (4) DL requires the configuration of numerous hyperparameters (HPs), which need to be optimized; (5) trained NNs tend to be large and resource-intensive, introducing the need for model compression, e.g., pruning and quantization, low-rank factorization, convolutional filters, or knowledge distillation [9].

Several of these critical aspects of DL have been addressed independently 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 researchers and practitioners [20, 30]. However, the concepts and technologies presented are still relatively new and not yet fully mature [18]. Additionally, these individual solutions have not yet been assembled to an end-to-end development process for DL. A conceptual representation of the complete workflow and a corresponding implementation remains undefined.

To overcome the above-stated knowledge gap, this paper aims at investigating the current development workflow of DL in the context of ML lifecycle management. The goal is to specify best-practice guidelines from model development to deployment and execution, i.e., bringing DL models into production. Therefore, lifecycle critical differences to conventional ML are collected and integrated into an extended workflow for the DL lifecycle. Additionally, a prototype is

implemented to demonstrate the practicability of the defined workflow. Overall, this paper summarizes the current state of research focused on the DL workflow and investigates its applicability in practice. Then, we demonstrate that our abstract definition of the workflow can be utilized in common DL applications. The technical instructions on the prototype and source code for each of the use cases are published at the address https://github.com/janousy/CDL.

This paper is organized as follows. Section 2 describes the correlated research. We derive the abstract DL workflow in Section 3, using the collected particularities of DL. This abstract definition is then implemented in a minimum viable prototype in Section 4, where we select a set of open source technologies to facilitate end-to-end DL. The usage of our prototype and consequently the applicability of our abstract workflow is then shown through two distinct use cases in Section 5, based on a text classification and image processing problem. Our findings are ultimately concluded in Section 6.

2 RELATED WORK

While algorithms and frameworks to build ML models evolved quickly, other stages of the workflow have been neglected for a long time. However, to integrate ML into the current software applications, a need for a conceptual development process emerged. In this section, we describe the work related to our research.

Machine learning workflow. Amershi et al. [5] investigated the development of ML applications at Microsoft. Through a case study, a high-level concept of the ML workflow composed of nine steps was deduced. Furthermore, they highlighted the current challenges imposed during the ML development and introduced a model to measure the ML process maturity.

Later, Salama et al. [37] took the ML workflow a step further. From a more practical perspective, they presented a conceptual representation of a fully integrated ML 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.

Compatibly, Garcia et al. [13] argue that a crucial piece currently missing in the ML workflow was the context. Unstructured and off-hand 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 et al. [18], it is argued that several steps within the ML lifecycle had been neglected up to now. The authors interviewed 17 ML 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 ML development process should not only focus on algorithms, but the complete lifecycle. Additionally, it is stated that the existing tools for ML 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 [28, 43]. Moreover, very few specifically target DL.

Deep learning lifecycle. Miao et al. [26] 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 DL models. Thereby, users can explore and compare hyperparameter tuning experiments using external frameworks and publish models.

Instead, Zhang et al. [48] conducted an empirical study concerning common challenges within DL development. By collecting questions and answers on *Stack Overflow* and building a classification model, they concluded five categories of common issues: application programming interface (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.

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

3 A DEEP LEARNING WORKFLOW

In this paper, we want to deduce the workflow required to perform end-to-end DL and demonstrate the usage of the workflow through a minimum viable prototype. Therefore, this section outlines the abstract DL workflow that complies with the requirements listed in Section 1. First, we give a high-level overview of the components and roles required. Then, we provide a more detailed description of all activities and interactions.

3.1 High Level Overview

We describe the DL 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 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.

Within the workflow, we distinguish between `workflow steps` and `persistence entities`, which store data produced by a workflow step, once an `actor` performed an action. 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 (HPs, 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.
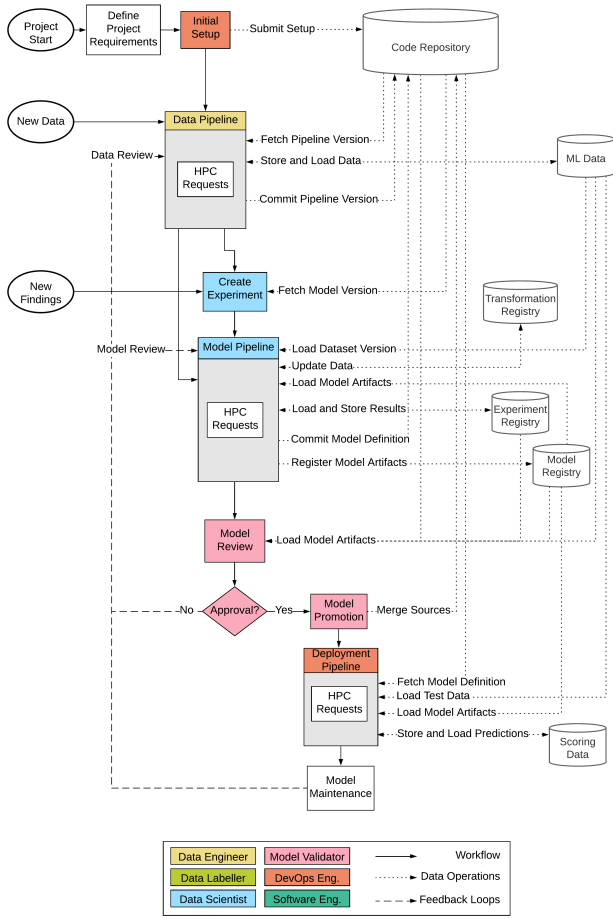
Figure 1: High-level overview of the DL workflow.

Figure 2: Definition of the abstract data pipeline.

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 DL, as well as classical ML, 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 initiatives.

At the beginning of every DL project, the requirements need to be defined. This is usually an interdisciplinary activity, involving business, research, and engineering [25]. After coming to an agreement, i.e., Define Project Requirements, a DevOps Engineer is responsible for the Initial Setup. They introduce and maintain tools, methodologies and processes for the team to support development, deployment, and monitoring of ML models [36]. 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, so that the complete workflow remains reproducible. Once the setup is complete, other team members can start with their implementations.
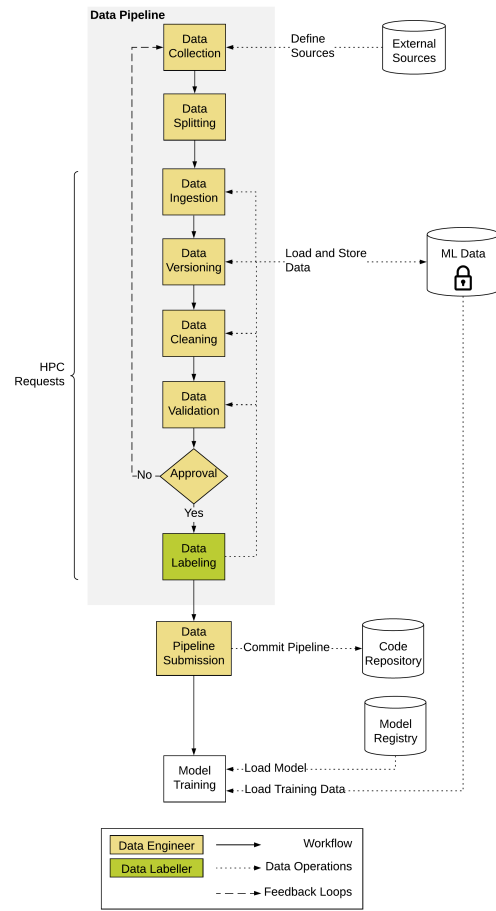
Theoretically, the DL workflow comprehends three fundamental pipelines, namely the Data Pipeline, Model Pipeline, and Deployment Pipeline, which are explicitly discussed in their respective sections. All pipelines share the requirement of high performance computing (HPC) Requests, involve a subset of 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.

3.2 Data Pipeline

The data pipeline, displayed as a flow chart in Figure 2, is the fundamental element of the DL workflow, as a DL model directly depends on the supplied data [29]. There are essentially two roles present within the data pipeline: the Data Engineer and the Data Labeler. A Data Engineer's responsibility is to construct, acquire, prepare, and store data securely [22]. A Data Labeler, on the other hand. provides an accurate ground-truth for supervised or semi-supervised learning.

The first step in the data pipeline, i.e., Data Collection, is defining the external sources for the data. If the test and training sets 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 for 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 ML Data in Figure 2. Data Versioning is indispensable and critical for data lineage [22], 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 DL 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 DL lifecycle, when there is already a model available, one can optionally trigger 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 complex data, various steps within the data pipeline such as Data Ingestion, Data Cleaning, and Data Validation are resource demanding. Thus, a Data Engineer should request computing resources on-demand within the steps of the pipeline.

3.3 Model Pipeline

Similar to the Data Pipeline, the Model Pipeline abstracted in the overview flow chart is defined in detail in Figure 3. There are two main actors within the model pipeline: the Data Scientist and the Model Validator. The Data Scientist's aim is to analyze and explore data, extract features, and prototype models in an experimental manner [7]. The Model Validator takes responsibility for the project and ensures that the business requirements are fulfilled [12]. Analogously to the Data Pipeline, a Data Scientist can request HPC Resources on-demand throughout 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. Next, the data is split into a training and validation sets, before the Data Scientist starts building a model. In contrast to the Data Splitting step of the Data Pipeline, this Data Splitting step further
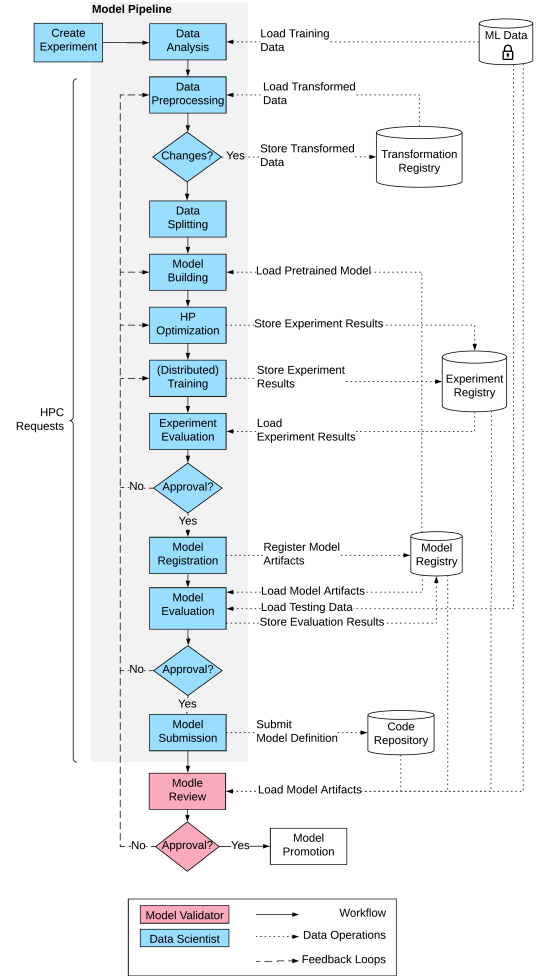

Figure 3: Definition of the abstract model pipeline.

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 pre-trained models from the Model Registry. Within the flow chart in Figure 3, the Model Registry is illustrated as a single persistence entity. However, pre-trained models can be loaded from any private or public registry. Thus, there may be multiple registries available to the Data Scientist for loading pre-trained models. As an optional step before training a model, 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 that the training is a resource-intensive activity.

Whenever a Training or HP Optimization 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 others for review. Thereby, the evolution of a model remains comprehensible.

At the step of Experiment Evaluation, the Data Scientist review their training experiment based on the validation results and other metrics [49]. If not satisfied, they return to the 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 stored to the metrics of the model within the Model 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 source code of the registered model is submitted to the Code Repository, although the source code is within the model artifacts. However, the Code Repository should hold all the code needed to reproduce a workflow iteration.

After the Data Scientist submitted their model, the Model Validator reviews the registered and evaluated model by loading corresponding artifacts. A Review focuses on model quality and can include various metrics such as accuracy, sensitivity, precision, different error measures, or ranking methods [40]. These metrics can be compared to other produced models, possibly already deployed to production. If the results are not satisfactory, the Data Scientist 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 1. In case of approval, the created model is promoted to production, which triggers the Deployment Pipeline.

3.4 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 Pipeline 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.

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 therefore make the source code more efficient for retraining [22]. They further provide expertise in API design for effective prediction requests. The implementation is further Reviewed by the Model Validator and stored into 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., 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
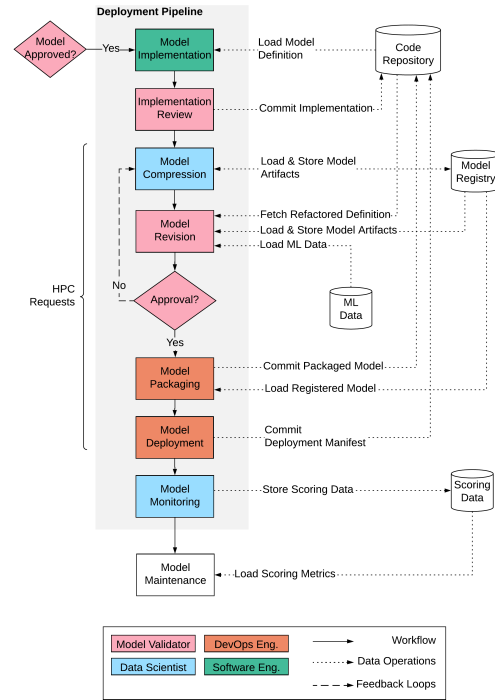

Figure 4: Definition of the abstract deployment pipeline.

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 that 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. Thus, input data from prediction requests, together with their metadata and results, are stored in a database for analysis. In case the model performance declines or other issues occur, the final step of Model Maintenance initializes the next iteration based on the interpretation of the Scoring Data.

4 PROTOTYPE IMPLEMENTATION

Throughout the following sections, we describe the implementation of a minimum viable prototype that enables the DL workflow defined in Section 3. Thereby, we demonstrate the transfer of the conceptual workflow into practical implementation. To do so, we select available technologies that fulfill the requirements given through the characteristics of DL listed in Section 1, and our defined workflow. Technical instructions on the prototype can be found in the associated *GitHub* repository at https://github.com/janousy/CDL.

4.1 Technologies of Choice

There is a relatively new, but fast-growing market for technologies that partially or completely facilitate the ML workflow. At the time of this paper, the landscape of tools available is broad and not fully mature [43]. Few are targeted towards DL only, as their vendors provide generic solutions to problems of the ML workflow.

Nevertheless, these solutions can often be transferred to various subtypes of ML. There are some emerging platforms that try to cover the complete ML 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, 15]. On behalf of reproducibility, we restrict our selection of tools to being deployable on-premise, available for free, and preferably open-source.

4.1.1 Hardware and Infrastructure. We used the computing and storage resources provided by the ScienceCloud [41] of the University of Zurich (UZH), which lets us provision virtual machines (VMs): (1) a large VM with 32 virtual central processing units (vCPUs), a single Nvidia Tesla T4 GPU and 128 GB of random-access memory (RAM); (2) 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 DL workflow and meet the demands described in Section 1. A single GPU is the maximum amount available per VM on the ScienceCloud, but to demonstrate distributed training, at least two GPUs would be required. To independently and securely host a Code Repository, a second VM was introduced with a near minimal specification to spare resources. Both VMs run *Ubuntu* 18.04 as their operating system.

To meet the requirement of scalability within the DL workflow, we selected *Kubernetes* as our infrastructure of choice. It serves as an open-source system for scalable container orchestration [21]. In the context of this paper, 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.

4.1.2 Workflow Tools. Hereinafter, the technologies that directly support the implementation of the workflow steps and persistence entities are described. Our selection is based on the recommendations of the MLOps Community [28] and Visengeriyeva et al. [43]. It is important to note that this selection is not fixed and could be swapped with any tools with similar features.

We choose *GitLab* [14] as our version control system (VCS), representing the Code Repository. *GitLab* can be hosted on premise, provides mature features for CI/CD and integrates with *Kubernetes*.

To achieve data lineage, i.e., Data Pipeline and version-controlled datasets, *Pachyderm* [31] is selected. It 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 be used to store unstructured datasets, and as the Transformation Registry by the Data Scientist to cache transformed data.

We used *Label Studio* [19] to integrate Data Labeling into our workflow implementation. It is compatible with various types of data, especially unstructured data commonly used in DL applications such as computer vision, natural language processing, and audio processing [3]. The labels, together with their metadata, are stored to a *PostgreSQL* [35] database for fast and convenient queries.

For tasks related to the Model Pipeline, we used a cloud-native platform specifically targeted towards DL called *Determined* [11].

This platform addresses the need for distributed Training, HP Optimization, 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 *PostgreSQL* instance represents the Experiment Registry, whereas a *MinIO* [27] 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* [39] was used. It supports a large spectrum of ML 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 alternative technologies to deploy ML models on *Kubernetes*, *Seldon* appeared to be the most mature solution.

We did not implement the Scoring Data persistence entity, as Model Monitoring would exceed the scope of this paper. However, a *PostgreSQL* database holding results of requests, and possibly references to provided files stored to *Pachyderm*, would be applicable.

4.2 Mapping Abstraction and Implementation

With the technologies selected in Section 4.1.2, we can build a DL system that implements our abstract workflow of Section 3. By mapping the technologies to tasks and persistence entities, we demonstrate how these integrate into the DL workflow. For each pipeline, we will walk through the practical utilization of the technologies.

4.2.1 Data Pipeline Implementation. Besides other tools for CI/accd, the main technologies within the Data Pipeline are *Pachyderm* and *Label Studio*. 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 3.2, 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 JavaScript object notation (JSON) or yet another markup language (YAML) format. Within this manifest, they optionally specify the resources to be used at each step, such as GPU, central processing unit (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 *Label Studio*. 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 *PostgreSQL* database, where the testing and training data reside in separate tables. Within a table, a row keeps information about the label, the corresponding file path to the validated output repository, and metadata, e.g., 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.

4.2.2 Model Pipeline Implementation. Within the implementation of the Model Pipeline, a Data Scientist mainly interacts with *Determined* and *Pachyderm*. 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. Therefore, 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, pre-trained models can be loaded from the internal *Determined* Model Registry or any external model registry, such as the Hugging Face transformer library [44].

Besides a model definition, *Determined* additionally requires a configuration file in YAML format, which specifies HPs, resource requests, data source, and version, etc. The Data Scientist can then use the same model definition with different configuration files for HP Optimization 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, a Data Scientist can review and compare their executed jobs on the *Determined* user interface (UI) until a satisfying model.

Once a Data Scientist approves of the validated model, they can register a selected model checkpoint through the *Determined* command-line interface (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 Model 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, they perform a Model Submission with a merge request on *GitLab*. The Model Validator then reviews the experiment and, if approved, the Deployment Pipeline is triggered.

4.2.3 Deployment Pipeline Implementation. After the Model Validator has approved a model, it is prepared for production. In the context of our minimum viable prototype, we skip two steps of the Deployment Pipeline: (1) we do not perform Model Compression, as we deploy to *Kubernetes* and inference latency is neglected; (2) Model Monitoring is omitted as this would exceed the scope of this paper.

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. Therefore, the Model Validator 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 defines at minimum 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 to trigger the deployment. The deployment manifest specifies what resources are available to the model. Then, a *GitLab* pipeline builds and pushes the *Docker* image, and deploys the packaged model to *Kubernetes* using *Seldon*.

5 USE CASES

To investigate the practicability of our implementation and the embedded pipelines established in Section 4, we apply two use cases to the prototype. These use cases address common but distinguished problems of DL, 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.

5.1 News Classification

In our first use case, the goal is to address an natural language processing (NLP) problem. Therefore, a multinomial news classification model trained and evaluated on the BBC datasets [17] is constructed using *PyTorch* [32].

The BBC datasets consists of 2,225 text documents that represent news articles from the years 2004 and 2005, collected from the official BBC news website [17]. The documents in English have various lengths and are divided into the following categories: business, entertainment, politics, sport, and tech. 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/accd. 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 (main).

Data Pipeline. We assume the role of a Data Engineer and perform the first steps of the Data Pipeline manually using Python scripts. 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] and we automatically create labels in *LabelStudio* JSON format for each article, as we do not possess the resources for manual labeling.

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, which essentially execute the same afore-mentioned script with different input paths

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. *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 *Label Studio* and available for manual labeling. An additional *Pachyderm* pipeline facilitates the export of the labels from *Label Studio* into the respective table within the *PostgreSQL* 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, but this could be enabled using an additional *GitLab* pipeline stage.

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 *PostgreSQL* training table are loaded together with the corresponding news articles from *Pachyderm*.

To transform the text data into input conform to the model, a simple NLP approach is applied. This includes Porter stemming [34] and token count vectorization using the feature extraction capabilities of *scikit-learn* [33]. As a vocabulary, we use the one provided by Greene [16]. The target classes are similarly encoded into numeric values. The preprocessed data is then again split into a training and validation sets at an 80–20 ratio [7]. In this use case, the preprocessed data is not stored to a Transformation Registry for simplicity.

During Model Building, we locally create the required trial class for *Determined* using *PyTorch*. Our basic model is 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, learning rate, and metadata about the experiment.

To start the HP Optimization, 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 asynchronous successive halving algorithm (ASHA) algorithm, which supports early stopping of low performing configurations [23]. *Determined* then presents various visualizations and metrics to find the optimal hyperparameter configuration. Since the parameters in the configuration are loaded at run-time, we can consequently use multiple configuration files for HP Optimization and (Distributed) Training with the same code for preprocessing and model definition. Thus, we write an additional configuration file for training using the results of HP Optimization, and again submit everything to the cluster.

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. Within an initial iteration of the DL 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.

Deployment Pipeline. The experiment is now approved and merged into the development branch. As this is only a demonstration use case, we do not consider model performance, and 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, therefore loads the model including its artifacts by downloading from the Model Registry within *Determined*. 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 with *Seldon*, we define a *Docker* image that copies the model wrapper and installs the required dependencies. The deployment manifest is specified using a YAML file, serving the packaged model as a representational state transfer (REST) API and specifying the Docker image as the surrounding container.

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 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.

5.2 Fashion Classification

The second use case derives a multinomial image classification model using *TensorFlow* [1] and the Fashion-MNIST dataset provided by Zalando Research [45]. Thereby, a classical computer vision example of DL is tackled.

The Fashion-MNIST dataset consists of 60,000 training and 10,000 testing images of clothing articles [45]. The grayscale 28×28 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* [47] as separate binary files representing labels and images.

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

Data Pipeline. The pipeline resembles our first use case 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.

Model Pipeline. Similar to the first use case, 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. 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 ML frameworks, *TensorFlow Keras* is used to build our image classification model, based on an example provided in the official documentation [10]. 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.

Deployment Pipeline. As in the first use case, we omit the first four steps of the deployment pipeline. Again, as DevOps Engineer, we 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.*

6 CONCLUSIONS AND FUTURE WORK

In this paper, we described a detailed step-by-step DL workflow, split into three pipelines, such that each component is independently reproducible and thereby enabling fast iterations. We provide a general guideline on the DL development process that helps to build DL 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.

Considering our abstraction of the DL workflow, 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 DL 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 the Model Registry and Code Repository, are crucial. On the one hand, VCSs do not allow storing large artifacts such as a DL model. On the other hand, a Model Registry cannot provide a history of code changes, and efficiently hold all source code of the DL workflow. An adopted VCS concept that manages ML data, model artifacts, workflow source code, etc., as a single set of dependencies in a central location would facilitate the DL, and the ML workflow in general.

Through a prototype, we showed that it is possible to translate the conceptual idea to practice using the latest technologies available on the market. It becomes clear that a large amount of different technologies is necessary to execute the complete DL lifecycle. Although there are all-in-one solutions available, these technologies often suffer from vendor lock-in and are associated with high costs, and they are not directly targeted towards DL. The large technology stack imposes the need for interfaces between tools, activities, and roles. This in turn introduces additional hazards to the implemented workflow. For example, we use a complex constellation of technologies within the data pipeline, which can become confusing and error-prone.

With the application of two distinguished use cases, we demonstrated the practicability of our DL workflow. However, the use cases do not represent the complexity of real-world challenges. Future work should therefore focus on evaluating the proposed concept in various industries to find possible alterations or inconsistencies. For instance, a field study could compare the proposed workflow to the processes within companies that have already brought DL into use. By accompanying multiple projects and workflow iterations, all performed activities are collected and mapped to our components. This procedure would ultimately highlight abundant or missing steps within the proposed workflow, inconsistencies in liabilities or more suitable technologies.

REFERENCES

[1] Martín 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, Dandelion Mané, 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 Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org

[2] Kamil Aida-Zade, Elshan Mustafayev, and Samir Rustamov. 2019. Comparison of Deep Learning in Neural Networks on Cpu and GPU-Based Frameworks. *EEE International Conference on Application of Information and Communication Technologies (AICT)* (2019), 27−30.

[3] Amazon Web Services Inc. 2021. *Data Labeling.* https://aws.amazon.com/sagemaker/data-labeling/what-is-data-labeling/

[4] Amazon Web Services Inc. 2022. *Amazon SageMaker Pricing.* https://aws.amazon.com/sagemaker/pricing

[5] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 291–300.

[6] Doğaç Asuman, Leonid Kalinichenko, Tamer Özsu, and Amit Sheth. 1998. *Workflow Management Systems and Interoperability.* Springer Berlin Heidelberg.

[7] Andriy Burkov. 2020. *Machine Learning Engineering.* True Positive Inc.

[8] Canonical Ltd. 2021. *MicroK8s - Zero-Ops Kubernetes for Developers, Edge and IoT.* https://microk8s.io

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

[10] François Chollet. 2017. *Basic Classification: Classify Images of Clothing | TensorFlow Core.* https://www.tensorflow.org/tutorials/keras/classification

[11] Determined AI. 2021. *High-Performance Distributed Deep Learning.* https://determined.ai/product

[12] Kirill Dubovikov. 2019. *Managing Data Science: Effective Strategies to Manage Data Science Projects and Build a Sustainable Team.* Packt.

[13] R Garcia, V Sreekanti, N Yadwadkar, Daniel Crankshaw, Joseph E. Gonzalez, and Joseph M. Hellerstein. 2018. Context: The Missing Piece in the Machine Learning Lifecycle. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD).*

[14] GitLab Inc. 2021. *Open DevOps Platform.* https://about.gitlab.com

[15] Google Cloud. 2021. *Compute Engine GPU Pricing.* https://cloud.google.com/compute/gpus-pricing

[16] Derek Greene. 2020. *Insight - BBC Datasets.* http://mlg.ucd.ie/datasets/bbc.html

[17] Derek Greene and Pádraig Cunningham. 2006. Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering. In *International Conference on Machine Learning (ICML).* 377–384.

[18] Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. 2021. AI Lifecycle Models Need to Be Revised: An Exploratory Study in Fintech. *Empirical Software Engineering (EMSE)* 26, 5 (2021), 95.

[19] Heartex Inc. 2021. *Open Source Data Labeling Tool.* https://labelstud.io

[20] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated Machine Learning.* Springer.

[21] Kubernetes. 2021. *Production-Grade Container Orchestration.* https://kubernetes.io

[22] Valliappa Lakshmanan. 2020. *Machine Learning Design Patterns: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps.* O'Reilly Media, Inc.

[23] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, B. Recht, and Ameet S. Talwalkar. 2020. Massively Parallel Hyperparameter Tuning. *arXiv:1810.05934 [cs.LG]* (2020). https://arxiv.org/abs/1810.05934

[24] Giuliano Lorenzoni, P. Alencar, N. Nascimento, and D. Cowan. 2021. Machine Learning Model Development from a Software Engineering Perspective: A Systematic Literature Review. *arXiv:2102.07574 [cs.SE]* (2021). https://arxiv.org/abs/2102.07574

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

[26] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. Towards Unified Data and Lifecycle Management for Deep Learning. In *IEEE International Conference on Data Engineering (ICDE).* 571–582.

[27] MinIO Inc. 2021. *MinIO: High Performance, Kubernetes Native Object Storage.* https://min.io

[28] MLOps Community. 2021. *Best Practice Real-World Machine Learning Operations.* https://mlops.community

[29] Aiswarya Munappy, Jan Bosch, Helena Holmstrom Olsson, Anders Arpteg, and Bjorn Brinne. 2019. Data Management Challenges for Deep Learning. In *Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA).* 140–147.

[30] NVIDIA Corporation. 2021. *Inference: The Next Step in GPU-Accelerated Deep Learning | NVIDIA Developer Blog.* https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/

[31] Pachyderm Inc. 2021. *The Data Foundation for Machine Learning.* https://www.pachyderm.com

[32] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NIPS).* 8024–8035.

[33] 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. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[34] M.F. Porter. 2006. An Algorithm for Suffix Stripping. *Program* 40, 3 (2006), 211–218.

[35] PostgreSQL Global Development Group. 2022. *PostgreSQL.* https://www.postgresql.org

[36] Red Hat Inc. 2021. *Who Is a DevOps Engineer?* https://www.redhat.com/en/topics/devops/devops-engineer

[37] Khalid Salama, Jarek Kazmierczak, and Donna Schut. 2021. *Practitioners Guide to MLOps: A Framework for Continuous Delivery and Automation of Machine Learning.* https://cloud.google.com/resources/mlops-whitepaper

[38] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems* 2 (2015), 2503–2511.

[39] Seldon Technologies Ltd. 2021. *Open Source Machine Learning Deployment for Kubernetes.* https://www.seldon.io/tech/products/core/

[40] Saleh Shahinfar, Paul Meek, and Greg Falzon. 2020. "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 (2020), 101085.

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

[42] Wil van der Aalst and Kees Max van Hee. 2002. *Workflow Management: Models, Methods, and Systems.* MIT Press.

[43] Laryza Visengeriyeva, Anja Kammer, Isabel Bär, Alexander Kniesz, and Michael Plöd. 2021. *MLOps: Machine Learning Operations.* https://ml-ops.org

[44] 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. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Conference on Empirical Methods in Natural Language Processing (EMNLP).* 38–45.

[45] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747 [cs.LG]* (2017). https://arxiv.org/abs/1708.07747

[46] 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. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin* 41 (2018), 39–45.

[47] Zalando Research. 2021. *Fashion-MNIST GitHub Page.* https://github.com/zalandoresearch/fashion-mnist

[48] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *IEEE International Symposium on Software Reliability Engineering (ISSRE).* 104–115.

[49] Alice Zheng. 2015. *Evaluating Machine Learning Models.* O'Reilly Media, Inc.