Comparative Analysis of MLOps Implementations

Ege Atesalp

Information Systems Engineering
TU Berlin, Germany
egeatesalp@gmail.com

Abstract

In the age of big data, where more data is created and collected every day through increasing internet usage as well as improving IoT techniques, one of the biggest beneficiaries is the machine learning algorithms that remarkably become more accurate with more and more data to train on. Although many machine learning algorithms and techniques were already known and used since the 1950s, the recent data tsunami enabled the algorithms to rise to the performance levels in many tasks that compete and even dominate the average human. One crucial challenge to overcome, however, is to modify and change the existing software development culture and techniques to better fit the inherent problems and features that underlie the machine learning algorithms and pipelines.

The following paper uses the Systematic Literature Review method for researching academic papers about potential challenges and tasks of adopting DevOps principles to machine learning projects. Moreover, the strategies, techniques and technologies adopted by popular MLOps solutions are compared, analyzed and ranked with an evaluation schema based on the findings from the SLR section to provide additional insight into the MLOps paradigm.

Zusammenfassung

Im Zeitalter von Big Data, in dem durch die zunehmende Internetnutzung und die Verbesserung der IoT-Systeme täglich mehr Daten erzeugt und gesammelt werden, sind die Algorithmen des maschinellen Lernens einer der größten Nutznießer, die mit immer mehr Daten, die sie trainieren können, immer genauer werden. Obwohl viele Algorithmen und Techniken des maschinellen Lernens bereits seit den 1950er Jahren bekannt sind und verwendet werden, hat der jüngste Daten-Tsunami die Algorithmen in die Lage versetzt, bei vielen Aufgaben ein Leistungsniveau zu erreichen, das mit dem des durchschnittlichen Menschen konkurriert und ihn sogar übertrifft. Eine entscheidende Herausforderung, die es zu bewältigen gilt, besteht jedoch darin, die bestehende Softwareentwicklungskultur und -techniken zu modifizieren und zu verändern, um den inhärenten Problemen und Merkmalen, die den Algorithmen und Pipelines für maschinelles Lernen zugrunde liegen, besser gerecht zu werden.

Die folgende Abschlussarbeit nutzt die Methode der systematischen Literaturrecherche, um akademische Arbeiten über potenzielle Aufgaben bei der Übernahme von DevOps-Prinzipien für Projekte des maschinellen Lernens zu recherchieren. Darüber hinaus werden die Strategien, Techniken und Technologien, die von populäre MLOps-Lösungen eingesetzt werden, mit einem Bewertungsschema verglichen, analysiert und eingestuft, das auf den Erkenntnissen aus dem SLR-Abschnitt basiert, um einen zusätzlichen Einblick in das MLOps-Paradigma zu geben.

Table of Contents

1 Introduction

1.1 Motivation and Structure

DevOps [1] is a set of cultures, tools and practices developed and polished by software companies throughout the last two decades that help the process of creating, delivering and maintaining software services and products at high velocity. Emerged from a discussion as a potential improvement on the already existing Agile approach in 2008, it has nowadays become the de facto way to create software products or offer software services all around the world.

Machine learning algorithms, which itself have been known since the early 50's but were unable to be utilized to their full potential due to the lack of efficient and cheap processing power needed to fuel machine learning algorithms, have become much more effective and successful at solving difficult tasks because of Moore's Law as well as countless discoveries in the field. The potential of the machine learning models is obvious and many software companies, from tech giants such as Amazon, Google, etc. to smaller companies have been utilizing machine learning algorithms in their services to great success.

Naturally, there is a need to adapt the existing DevOps approach, or more specifically the CI/CD principles, to the new machine learning models and services that have inherent differences to conventional software applications. This thesis aims to identify the specific challenges that arise from the adaptation of DevOps principles to services and products that utilize machine learning algorithms, as well as explore and compare existing solutions in the field.

Structurally speaking, rest of the section 1 will give certain background information and introduce the terminology regarding DevOps (CI/CD) and machine learning algorithms. Section 2 describes the systematic literature review that took place for the paper about the challenges and tasks for a successful DevOps/ML adaptation, as well as presents the results of the systematic literature review in a categorical way by first introducing the topic broadly and then identifying the challenges for both theoretical as well as practical aspects. In section 3, proof of work, open-source and fully-managed MlOps solutions will be discussed and compared, tying back to the identified tasks that need to be resolved in section 2 when possible. Section 4 will conclude the paper with an overview of the thesis, tying the findings from the comparison back to the research question, and lastly discuss possible future work.

1.2 Background and Terminology

DevOps can be summarized as a combination of philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity and high reliability. The concept of DevOps originates to be a successor of Agile culture and is specifically designed to decrease downtime and increase communication and feedback between different steps of a software product/service lifecycle.

[1] `DevOpsDocumentation:https://devops.com`

Continuous Integration and **Continuous Delivery**, or **CI/CD**, are two distinct pipeline design principles that attempt to realize the goals of DevOps from a technical aspect.

Continuous Integration aims to develop an automated and consistent way to write, build, package and test software with minimum downtime and unnecessary code. By being allowed rapid code production and commitments, the developers are incentivized to write code in small and methodical steps, which not only increases software quality but also increases the communication and collaboration between other developers.

Continuous Delivery deals with the organization and automation of the appropriate execution environments of the packaged software. Most teams have multiple internal/external execution environments, and CD creates a streamlined delivery system to ensure rapid code push in between.

Machine Learning can be summarized as a collection of algorithms and practices that gradually "learns" from the given sets of data to optimize a given task, or for the case of unsupervised learning, learn the task itself from zero. It includes many different classes and categories ranging from simple regression models to multi-layered, deep neural networks. However, there are some functional commonalities shared across many of the algorithms and models.

Machine learning models that are neural networks, which will be the focus of this thesis since they are the most challenging types of models to maintain but also have a great capacity to perform in harder ML tasks, consists of multiple, very basic units of computation called (very aptly) **neurons**, that individually perform very elementary computations on their inputs and send out the results as their outputs. However, before their outputs are sent to the next layer of neurons, the neuron applies one of the various **non-linear activation functions** to the output. In a perhaps unintuitive way, the synthesis of non-linearity and multiple units of crude computations can enable the capacity to "learn" very complex tasks, ranging from voice recognition to self-driving cars.

A classical example to discuss would be an image detection model that aims to detect between cats and dogs. In order for a model to learn to distinguish between images of dogs and cats, the creation of the neurons and the overall model architecture is not enough. The model needs already-labeled, distinct images of cats and dogs. Through constantly updating the parameters of the individual neurons through a computation called **backpropagation**, the model on average becomes more accurate at detecting unlabeled cats and dogs, thus achieving to "learn" the given task through **training** on the labeled dataset. There are versions of machine learning algorithms that do things differently, such as algorithms that create their own labels in the training process rather than needing it from an exterior source, or algorithms that don't need data at all to train (chess games can be simulated and self-played, don't need to train on archives of past chess games). However, the classical paradigm of model architecture design and data training is important to learn two concepts relevant for the rest of this paper, **overfitting** and **underfitting**.

In order to comprehend **overfitting** and **underfitting**, it is crucial to highlight the aim of any model that learns on training datasets, which is to eventually perform the expected task on data that is not a part of the training set. If the model becomes "too" good at performing the given task on the training dataset, it might learn features that are specific for the given training dataset and not generalizable features that would apply to all possible and plausible datasets for the task, which in data scientific terms would mean that the model **overfits** on the training dataset. One reason for overfitting can be that the training datasets fail at being a good representation of the expectation for the model to perform on. Say, we want a model that can detect all breeds of dogs from all breeds of cats. But rather than training the model on the images of multiple breeds, we only train on golden retrievers. The trained model then can very possibly be exceptionally good at detecting golden retrievers, but fail to label correctly when shown a picture of a husky.

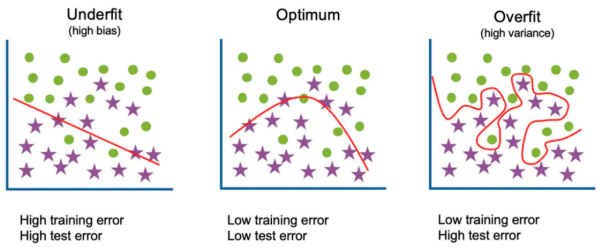

Fig. 1. Taken from https://www.ibm.com/cloud/learn/underfitting. Visualizes overfitting and underfitting given a certain regression task.

Underfitting is the opposite of overfitting in some manner since it means that the model failed to correctly extract the patterns or learnable features of the training dataset to then perform on other datasets. While overfitting models performed very well on training datasets and poorly on test datasets(or any expected dataset given the task), underfitting models perform poorly on both sides. The aim of a data scientist is to design the model architecture, the hyperparameters (configurable variables that play certain roles in the internal computations of the model) and the datasets in such a way as to find the right balance between overfitting and underfitting. Possible techniques and design principles that help against overfitting and underfitting of the model in a MLOps setting will be explored further in sections 3 and 4.

1.3 Related Works

With the rising demand for machine learning solutions, there are quite a few emerging as well as established reviews and analyses with similar intentions and

goals to this paper. In this section, existing surveys and research [38] [42] [21] [35], which analyze existing MLOps approaches available and are being utilized, are looked into to create a reference as well as provide inspiration for the papers own method and structure.

The paper "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?" by Makinen et al. is an academic survey on the importance of a MLOps implementation perceived by professionals from different industries and different expertise levels.

The questionnaire is quite extensive and meant to primarily gauge the value the experts put on various, distinct properties of a MLOps implementation, such as scalability, ease of use, version control tools, etc. By collecting the scores (between 1-10) and then aggregating over the many experts, the survey provides insight into the parts of a MLOps that might be worth focusing on, and parts that might be not important for the end-user relatively.

As a side note, the survey also measures several other, less technical problems that data scientists and other professionals in the ML domain suffer, such as group management, lack of data scientists in a team, difficulty of collaboration, etc. Potential improvements of more social and less tangible aspects of ML-specific domain by a well-established MLOps and DevOps culture could be worth exploring.

The second paper [27] from Lwakatare et al. deals with 5 different cases of software companies that have through different solutions adapted ML components to their software systems. Categorizing the cases with the "maturity" of their ML implementations, Lwakatare et al. highlight the possible software engineering problems that tend to emerge in different maturity states.

Being a relatively new topic, the intersection of ML and CI/CD lacks the clear categorization and taxonomy that most other SE branches tend to have. Lwakatare et al. create a 2-dimensional taxonomy graph with maturity (Experimentation and Prototyping, Non-critical Deployment, Critical Deployment, Cascading Deployment) on the x-axis and model life-cycle activities(assemble dataset, create model, (re)train and evaluate, deploy), with the aim of being able to handle the specific SE challenges that arise with the adaptation of ML models to commercial software systems with more precise language.

The famous paper "Hidden Technical Debt in Machine Learning Systems" by Sculley et al. [34] deals with the concept of hidden debt and its integration with ML algorithms. Building on top of the "Hidden Debt" term coined by Ward Cunningham in 1992, Sculley et al. highlight how ML models are specifically vulnerable to the hidden debt problem.

Discussing various dependencies that ML models tend to have, such as data, external libraries, etc., Sculley et al. reflect on the fact that there are many various factors, excluding the model code, that have a significant effect on the output of a model. Similar to hidden debt in a conventional software setting, these dependencies can, if not acknowledged or taken care of, have harmful long-term effects on the project since their effects are generally more challenging to keep track of.

Although the hidden debt concept will be explained further in later sections, it is worth noting that the introduction of the hidden debt as a concept is invaluable, since it provides a well-defined skeleton to the thesis in the form of an overarching "meta-problem" that MLOps implementations and frameworks aim to minimize.

2 Systematic Literature Review of MLOps Challenges

2.1 Research Method

The research method used in the first part of this study is the systematic literature review as one of the most widely used in the software engineering area. With SLR, the paper aims to have a structured, organized and reproducible way to research the academic papers about the investigated topic. Following the SLR guidelines reported in [23], the review protocol consists of research questions, study selection and assignment criteria, and lastly data extraction and synthesis. The question that is researched for the paper is:

1. RQ: Regarding the adoption of DevOps practices and cultures in ML tasks, what individual challenges have been identified from a technical point of view and what are the solutions for those challenges?

For study selection, the research has been done on the popular software publications and literature search engines, specifically Google Scholar, Springer, ACM and IEEE. The research consisted of academic papers and conference proceedings as well as references from other relevant primary studies. As search and assignment criteria, the search has been done specifically for papers and articles between the years of 2015 and 2020, and the search was done using the keywords "MLOps", "machine learning DevOps ", "CI/CD in Machine Learning", "Challenges of MLOPs". The inclusion criteria for the assessment were the following:

1. Does the paper include a specific or a wide overview on the topic of adaptation of ML services or models to the DevOps (CI/CD) approach?
2. Does the paper include a theoretical and thorough explanation of the problems it describes regarding the adaptation?

The information gathered from the SLR will be discussed later in this section, where the individual challenges and tasks of MLOps will be categorized, described and elaborated upon. Below is a table that summarizes and organizes the result of SLR.

Name	Authors	Publishing Year	Number of Citations	Relevant Findings
Applying DevOps Practices of Continuous Automation for Machine Learning	Karamitsos, Ioannis and Albarhami, Saeed and Apostolopoulos, Charalampos	2020	8	Generalized overview of the MLOps as the integration of ML and DevOps with an emphasis on continuous deployment and different release strategies
Building Continuous Integration Services for Machine Learning	Bojan and Interlandi, Matteo and Renggli, Cedric and Wu, Wentao and Zhang	2020	7	Discusses overfitting problem in automated continuous integration pipelines, introduces a solution with randomized, continuously reshuffled test sets
A taxonomy of software engineering challenges for machine learning systems: An empirical investigation (Pages 227 -243)	Lwakatare, Lucy Ellen and Raj, Aiswarya and Bosch, Jan and Olsson	2019	67	Analysis of six different MLOps use cases to identify common challenges, attempts to create a unique taxonomy of different maturity levels of automation
Hidden technical debt in machine learning systems	Sculley, David and Holt, Gary and Golovin, Daniel and Davydov	2015	553	Discusses concept of hidden debt in ML setting, introduces hidden and explicit dependencies of ML models
A Data Quality-Driven View of MLOps	Cedric Renggli and Luka Rimanic and Nezihe Merve Gürel and Bojan Karlaš	2021	2	Bayes Error rate, Data Quality-Model Quality relationship
What is MLOps? (Pages 79-124)	Alla, Sridhar and Adari, Suman Kalyan	2021	2	Overall view of MLOps, creating necessary terminology around MLOps, identifying challenges
The problem of concept drift: definitions and related work	Tsymbal, Alexey	2004	1098	Definition and analysis of concept drift
Activeclean: Interactive data cleaning for statistical modeling	Krishnan, Sanjay and Wang, Jiannan and Wu	2016	133	Data Preprocessing, runtime Data Cleaning that can have strong impact on model accuracy
Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment	Renggli, Cedric and Karla, Bojan and Ding, Bolin and Liu, Feng and Schawinski, Kevin and Wu, Wentao and Zhang, Ce	2019	25	Proposal of a CI system specifically for ML Pipelines, identification of various MLOps challenges regarding data and testing
An Approach to DevOps and Microservices	Klein, Brandon Thorin and Giese, Gerald and Lane, Jayson and Miner, John Gifford and Jones, John J and Venezuela, Otoniel	2020	4	Separation of various DevOps terminologies, Microservices
Bayes Error Estimation Using Parzen and k-NN Procedures	Fukunaga, Keinosuke and Hummels, Donald M.	1987	186	Bayes Error description and analysis, possible ways to calculate pre-operation
Is machine learning software just software: A maintainability view	Mikkonen, Tommi and Nurminen, Jukka K and Raatikainen, Mikko and Fronza	2021	4	Identification of possible challenges of MLOps regarding maintenance
Designing an open-source cloud-native MLOps pipeline	Maekinen, Sasu	2021	2	Complete MLOps solution including step by step description of the pipeline
MLOps Challenges in Multi-Organization Setup: Experiences from Two Real-World Cases	Granlund, Tuomas and Kopponen, Aleksi and Stirbu	2021	1	Scaling in MLOps, using two real-life examples to illustrate different emerging challenges in MLOps pipelines with growing load
Challenges in the deployment and operation of machine learning in practice	Baier, Lucas and Johren, Fabian and Seebacher, Stefan	2019	21	Industry questionnaire regarding MLOps implementations
Continuous Deployment of Machine Learning Pipelines	Derakhshan, Behrouz and Mahdiraji, Alireza Rezaei and Rabl, Tilmann	2019	5	Pre-processing, data management(dataset separation)

2.2 Results

Hidden Debt One of the critical differences between classical software services and ML services is the concept of Hidden Technical Debt, coined by Ward Cunningham in 1992 as a means to conceptualize the issues that come up when a

software is written rapidly without consideration to specific factors, which can in future cause more harm than the benefit of the rapid production of code[14]. Hidden Debt, in general, can be applied and discussed for every software product or service, however as Sculley et al. [34] argue that machine learning models create an amplified and more severe version of hidden debt that is more problematic to solve

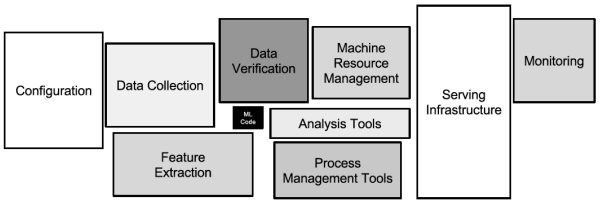

Fig. 2. Taken from [14]. Illustrates the different parts of a ML model.

As the figure above shows, only a very small portion of ML models is the actual code, with the majority being pre- and post-processes as well as glue code such as configuration for existing libraries, frameworks, etc. All of these different aspects of a model create inherent dependencies, that need to be resolved for each additional change, which can seriously damage the growth of a software product or service if the dependencies are allowed to grow out of proportion. A successful MLOps implementation is naturally one that acknowledges the hidden debt caused by ML implementations and attempts to minimize it by automation of the various parts of the pipeline. The following section is a synthesis of various academic papers and articles reviewed in the systematic literature review that deals with the concepts, challenges and tasks that need to be solved for a successful implementation MLOps. Topics with similar underlying causes have been grouped up to create a more structured discussion.

Data Dependency Machine learning algorithms are distinctively different than other software algorithms in their relationship with data. Unlike conventional algorithms that have a certain, ordered functionality specified in code by a developer, machine learning codes act like a "compiler" [32]that only gets compiled given the data to train on. The nature of this interplay between data and model means that the quality machine learning model, or an application/service that uses that model, is a reflection of the quality that the underlying dataset possesses.

In a simulated world where the data is always constant and uniform, this wouldn't be a fundamental challenge for the adaptation of MLOps. However, since many applications of MLOps use real-life data that is messy and ever-changing[36], data quality is a crucial task to solve. In this subsection, some relevant topics and steps related to data quality will be introduced and discussed.

Concept Drift, or sometimes called Concept Shift, is a phenomenon of data collected from an observation, which occurs when an underlying, hidden factor creates an unexpected change in the distribution of the known, observed data[36]. This can happen suddenly or can happen gradually over time. An example would be global warming that gradually increases the average temperature of locations all around the globe. The average temperature of a location should theoretically be used as an indicator of expected temperature in normal circumstances, but the existence of global warming causes the average temperature to be not reliable in creating an expectancy. A machine learning model that attempts to estimate the temperature can only give accurate results if there is a constant feed of information that enables the algorithm to learn the degree of average temperature change happening in real-time.

This has two important implications for a ML model and a MLOps system. First of all, identifying the differences between noise, which is expected, and constant fluctuations in data from concept drift becomes a task that must be resolved[32], which makes a trade-off to emerge. If a model reacts too strongly to every slight change in data, it can be very costly to constantly re-train and can be detrimental to the accuracy of the model. But the system is unable to distinguish the concept drifts happening, then the model can degrade over time. The advantage of MLOps in this circumstance is that by making training, building and deploying cheaper in comparison to manual implementations, the automation can incentivize the developers to look for concept shifts and act accordingly if necessary.

Secondly, the existence of concept shift in real-life data causes the data quality aspect of an MLOps to be a continuous process as well. Similar to model building or deployment, data collection and processing is necessarily a part of the lifecycle of the application, and new data should constantly be collected to make sure the performance of the application doesn't degrade over time.

To keep up with the principles of DevOps that emphasize rapid development and deployment, the data cleaning and processing step must be an automated part of the ML Pipeline that minimizes the human interaction needed. The data gathered for the models, which can include the feedback metrics and data from the deployed products, should be continuously processed, cleaned and organized in specified storage units for future usage.There are various proposed solutions for this step with great success[25][32]. A key insight of these researches is that the data processing can be done iteratively alongside model training for better results, rather than being done only once before the model training.

Another concept that must be taken care of is the division of training, test and validation datasets in an automated environment. Generally speaking, datasets in machine learning context are divided into three separate groups to counteract the overfitting problem that might occur when a model is tested or validated with the same data it trained on, which would mean that the test or the validation is biased towards the model to perform better than it actually can for an entirely unknown dataset. Managing overfitting vs underfitting is one of the main challenges of every ML task and the data separation is a crucial step in this

paradigm. The training/testing/validation datasets technically need to belong to the same data distribution, thus many MLOps systems perform the separation automatically in the data register. There are, however, some theoretical and technical challenges that arise from the separation.

The inherent noise that is existent in many datasets collected means that the testing step is more tricky to accomplish in comparison to a fully deterministic environment. Tests, conventionally, have often binary results of (passed/failed). This is perfectly fine when the testing is done on a function with a perfectly deterministic outcome. If a multiplier function is tested with inputs 2 and 2, 4 is the only reasonable and passing outcome. Thus 4 is passing, and every other possible outcome fails the test. This becomes messier when the testing is done with a test set that is randomly selected from a pool of data. Model A, which actually would on average perform better than Model B, can randomly get tested with a set of data that happens to anomalously not fit with the data Model A trained on, which would cause Model B to perform better on that specific randomly selected test set. This phenomenon can cause the automated pipeline to automatically prefer Model B over Model A, which makes the testing very unreliable. There are various solutions to overcome this such as having a confidence interval to counteract the inherent probabilistic nature or having a specific automated software module inside the pipeline that specifically looks for anomalies such as this [32][22].

Also related to the data separation paradigm, using the same testing data set multiple times for any specific ML model has the detrimental effect of the model incrementally overfitting on the given test set. This is because the testing set "leaks" information about the underlying distribution of its data by giving a result, which in the next iteration the model specifically trains for to perform artificially better. To overcome this, the obvious solution is to create a new dataset each time a new testing pipeline is created, which would incur very high costs in terms of new, unused data. This paradigm again can be represented as a tradeoff. N can be the number of times a new test set can be used before it gets replaced by a new randomly generated dataset. The higher the N, the more vulnerable the model is to potentially overfit to the test set, and lower N demands a very constant influx of new data to create new test sets, which can be unmanageable.

Various articles and papers have provided valuable suggestions for the overfitting problem, such as choosing an N arbitrary and then tuning it as a hyperparameter [22], or developing a "sample size estimator" that automatically estimates an N for each model during development[32]. Another solution is to prevent the "information leakage" by changing the test scores from an accuracy metric to a simple binary (passes/failed) [31] so that during the training the model cant optimize for marginal test score increases. The common property in all the suggested techniques is for the pipeline to acknowledge the exposure of the model to potential overfitting and implement an automated way to restrict the number of same test set usages.

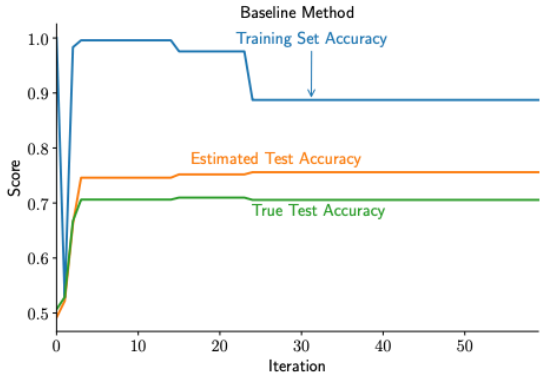

Fig. 3. Taken from [22] Page 8. Illustrates how with each iteration of testing with same test set, the estimated test results diverge from the true test result and converge to the training set.

To summarize, the influx of data with an underlying probabilistic nature, as well as potential concept shifts over time, creates various challenges for any MLOps systems to find a solution to. As the suggested solutions often create a tradeoff situation with a tuning parameter, the best solution is often an optimization task with monitoring of the model for overfitting. This paradigm still has no perfect answer and the papers agree on it being an open area for more research and discussion [31][32][22].

Feasibility DevOps culture and approach in the industry generally includes an exploratory phase in the beginning [24], where the feasibility of the proposed service or application is tested from a business and engineering perspective with various types of analysis. There are various research and articles that propose a new type of exploratory analysis in regards to data quality that can potentially lower the risk of investing further in unattainable goals[31][32].

Bayes Error Rate, also called irreducible error, quantitates the minimum error rate a model can achieve given a certain data distribution and training, test and validation sets, independent of the properties and the structure of the model. The reason this inherent non-zero error rate exists is related to the noise that is observed during the measurement process of many real-life data acquisitions. Estimating Bayes Error Rate analytically in a short period is known to be a notoriously hard problem [18] [31], as it requires very high computational power in high dimensional feature spaces for it to be logical to use it as a part of proof-of-concept analysis. That being said, the research is still ongoing for this process, and having a feasibility analysis from a data science-orientated perspective is vital for the exploratory phase of MLOps implementation.

Continuous Deployment Continuity of deployment is one of the two pillars of the CI/CD paradigm. It is crucial for being able to operate a rapid production of software, however, it can be considered even more important in an MLOps setting, mostly because of the "concept drift" phenomena discussed earlier. Generally speaking, ML models and ML services that utilize these models degrade over time, if not constantly retrained and developed with the current datasets. This act of constant retraining can only be realized with a constant, or even better a continuous, procedure of automated deployment of the model and its dependencies with each of its iteration. In this section, specific challenges and concepts related to the Continuous Deployment aspect of the MLOps that have been reviewed in the systematic literature review will be introduced and analyzed.

Perhaps the most defining and fundamental decision to be made regarding Continuous Deployment is the choice of deployment approach, which can be generalized into three categories: periodical, continuous and online[17]. All three of the categories are concerned with the training of an already deployed, fully functional ML model. Training in an *periodical* approach is simple, models are further trained with the new datasets in certain time intervals. In an *Online* approach to deployment, models that are released and are currently used are constantly get trained with the help of specialized learning algorithms, using the influx of data being collected "online" in the ML service the model is used in. Online training has a few implications: there must be an influx of data that is reliably collected "online" through different channels. Also, since every data point is used in training once and isolated from the other data points, unlike batch training as an example, the models generally have a harder time in being able to successfully distinguish noise in data distributions or recognize the concept drifts applying over time.

Continuous development approach is a newly proposed category by Derakshan et al. [17] that attempts to merge the benefits of the periodical and online deployment approaches while minimizing the disadvantages. Briefly explained, the continuous approach utilizes the fast and robust pipeline creation capabilities of a MLOps to create rapid iterations of retraining with batches of data, but without any downtime and constant use of new data, similar to an offline approach. Online Statistics Computation and Dynamic Materialization feature allow the ML pipelines to pre-process, store statistics about and transform the data "on the run". And with the addition of Proactive training, the computational ability of the model to be trained while running, and being able to answer prediction queries during training, the models receive a combination of the historical and incoming training data to train on. Two features combined can allow the models to reliably distinguish noise and concept shifts while also being "continuous" without incurring any downtime or loss of service.

There are, however, other ways of implementing continuity in deployment. *Shadow Deployment* [5] [28], also called shadow testing, is an interesting software "trick" where a ML model service is created inside a special cluster, that can mirror all the input that is channeled to the real, functioning ML service.

However, the cluster is designed to not propagate any output, thus being a safe zone for any testing or experimentation. The model which is ready for deployment is thus first deployed in the shadow to be further monitored, validated and compared to the existing models.

This process can be further modified to be more similar to a *Canary Release* in conventional DevOps setting[28]. The shadow cluster, rather than having complete output isolation, can be implemented in a way to accept a certain percentage of the real application/server traffic. Then, that percentage can be gradually increased as the shadow model is tested and validated, up to 100 percent which would indicate a complete deployment of the next version. This method of deployment, in comparison to the previous way of shadow deployment, still incurs a non-zero risk of negatively affecting the end-user, however, has the added benefit of reducing the computational cost of having a duplicated, simulated environment just for testing(100 percent vs 200 percent of the necessary computational power needed). In addition, a canary release style deployment can theoretically be a more reliable way to test and validate models in comparison to a shadow style, simulated testing environment, thus decreasing the risk of releasing a faulty version of the model.

Related to deployment,*Artifacts* can be thought of as "snapshots of code"[28] that is built during the deployment of a product or service in a traditional software pipeline. In the context of machine learning, the idea of an artifact is still crucial, so that the performance of a distinct, specific model can be successfully kept track of. However, since so little portion of a functioning ML pipeline is the ML code that describes the model (google estimates around 5 percent), a ML Model, or an artifact that describes it is naturally dependent on more factors, such as datasets, hyperparameters, etc.

The added complexity of artifacts in the context of machine learning ties in with the versioning of models and their pipelines. Since ML pipelines can take days to complete, *versioning* is of paramount importance for the developers to be able to restore the pipeline to a safe checkpoint in the case of a malfunction, or an unexpected decrease in performance due to unknown factors. Although we have already clarified that model training is not a fully deterministic process and can small performance differences might come up, two artifacts that store the same state of the parameters, hyperparameters, datasets and the same architecture of a model can be reliably expected to perform similarly[29] [28]. Storing all the different versions of data and parameters for each individual pipeline can be overwhelming in terms of storage space or for network performance, thus a distribution of data into different spaces such as cloud for maximum efficiency can be recommended[28]. Moreover, depending on the application, there could be specific restrictions for the access of specific data points[9]. Generally speaking, all papers [9][28][29] seem to suggest a specialized, application-specific model versioning and storage system that stores models, their related datasets and parameters, as well as any additional metadata regarding its performance, etc. in an organized and accessible way.

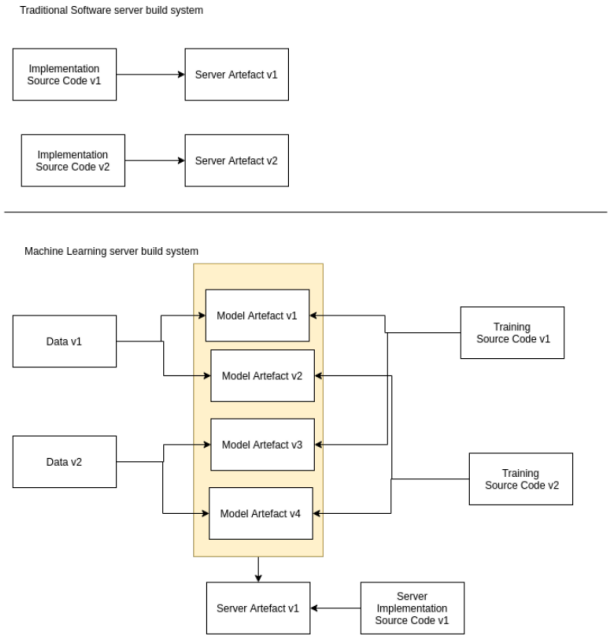

Fig. 4. Taken from [28] Page 24. Comparison of conventional software versus ML artifacts.

Maintenance and Monitoring Regarding maintenance and debugging, ML models and especially deep neural networks are notoriously difficult to inspect, mainly due to the "black box" nature of the networks. Although there are relatively successful attempts to "open" the boxes through various visualizations[33] [41] and similar observational mechanics, pinpointing an exact error in a layer or a data point in a deep network is still widely considered to be very difficult. This effect is only magnified with concept drifts that happen over time, meaning the degradation of performance for a ML service might be caused by a bug, or might just be the expected time-dependent concept shift, or even a combination of both factors. The difficulty of distinguishing between two given factors can potentially incur unnecessary losses or might hide more severe problems in the system that ideally requires immediate attention[19].

In the paper written by Mikkonen et al. [29], multiple key attributes are introduced that are meant to make a MLOps Pipeline easier to maintain. *Modularity* of a pipeline can be considered to be the most impactful in terms of representing a practical goal that can be implemented within the pipeline. By having the pipeline consist of many individual pipeline modules with specified expected outputs and inputs, various unit/module tests that enforce these expected output/inputs as well as well-defined interfaces that allow for clear communication and synchronization between the modules, errors in the pipeline can

become dramatically easier to search for. Since a successful implementation of a modularized pipeline abstracts the inner functionality of the individual module, the inner complexities of the module do not need to be fully understood to be utilized and maintained.

Reusability, which shares many commonalities with Modularity in terms of implementation, represents the ability of the written code, or modules, in a given pipeline to be further used in another pipeline in the future. Reusability can be obtained with correctly abstracted and interfaced modules, as well as code that is well written. *Reusability* supports maintenance by decreasing the number of modules that would need to be inspected in the case of a possible bug since a large portion of the modules that are being implemented in a pipeline could have past uses in other pipelines, which would provide information and reference in terms of error proneness for those modules.

Lastly, the attributes *Analysability*, *Testability* and *Modifiability* can be contextually grouped up together, since they all address the degree of maintainability the modules have and the qualities that support it. *Analysability* represents how well a given module can be understood by a human given the code and its structure, which is for ML modules relatively poor in comparison to classical software modules. One can perfectly know the architecture of a given deep network while having no idea what to expect from it as output. *Testability* quantifies the portion of the functionality in a module that can be reliably tested. We have clarified beforehand while the ML models are much more difficult to test and provide inherently probabilistic results. *Modifiability* is the ease of modification or change in code (or datasets in a ML context) for a module. Tied with the reasoning behind the other two attributes, the unexpected, dramatic change in output from slight changes in network design or datasets also makes the ML modules less modifiable and more volatile. Generally speaking, it is clear that ML modules are naturally unfavored regarding each of the three maintenance attributes in comparison to a conventional software module. That being said, modules can be made easier to maintain by rigorous testing for the functionalities, data formats, edge cases, etc. for the module, as well as constant monitoring of key metrics that could apply for the given module. Various frameworks attempt to solve the challenges proposed here that will be analyzed further in the comparison section.

Automation Maturity One crucial step in understanding the task of MLOps implementation is to recognize the differences between different ML pipelines in terms of automation and general maturity of CI/CD implementation[8]. Generally speaking, these different stages of maturity [8] are widely chosen to be divided into three specific levels: manual implementation, continuous model delivery, and continuous integration/continuous delivery of pipelines. Although it must be noted that this distinction of three levels is somewhat arbitrary, such classification provides very valuable information about the requirements and the goals of the system that is being built and maintained.

Manual Implementation is the least advanced setup of all three, where none of the MLOps principles and techniques are applied to the pipeline. In a manually implemented setup, there are distinctly different teams that consist of different specializations. There is a "model development team" that includes data scientists and machine learning engineers, who are responsible for manually performing data analysis and building, training, testing, as well as validating the models. In addition to the model development team, there is an "operational team" consisting of software engineers and operators, who integrate the model, which is pushed into a code repository by the model development team, to the rest of the system or application. After the integration, they build and deploy the system, maintain functionality and provide feedback back to the model development team based on the monitoring.

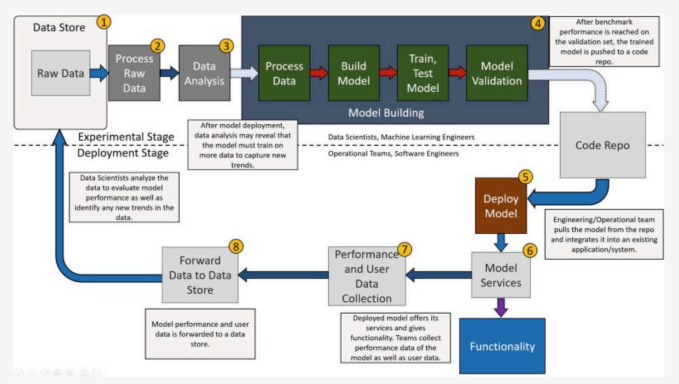

Fig. 5. Taken from [8]. Maps different components of a potential Manual Implementation system.

The development cycle with manual implementation setup is, hence its name, entirely manual and has obvious flaws and shortcomings[8]. First of all, since the entire model building is manual, the model development team has to rebuild the model each time there is a change in the training dataset or model hyperparameters. In an environment where the data input changes over time through trends and fluctuations, it is easy to see how manual model building can become very cumbersome and expensive. In addition, ML is a software branch that is constantly evolving, and the expensive and slow nature of the model building that cant catch up with the latest technology and knowledge can harm the performance of the end product in the long run. From an operational perspective, having two different teams can make the communication in between harder, which would make the integration task more difficult to realize. A change in the model can create different needs in terms of configuration, or even different metrics to look for and monitor in the deployed product. All these modifications must be done communicated and done manually, which can be hard to rely on.

Ultimately, the manual implementation has obvious flaws, and through DevOps principles can be improved upon to make the entire ML pipeline more dynamic, faster and cheaper.

Continuous Model Delivery is the level of maturity that first introduces the DevOps approach to the ML development and operation cycle. To be able to speed up and automate the pipeline, there are various new tools and techniques introduced to the system.

Feature Store can be considered as a data storage system that provides the necessary data feed in different steps of the various pipelines. Having a unified data system that automates the data cleaning and processing and provides the same data to different pipelines allows potential data inconstancy problems to be eliminated.

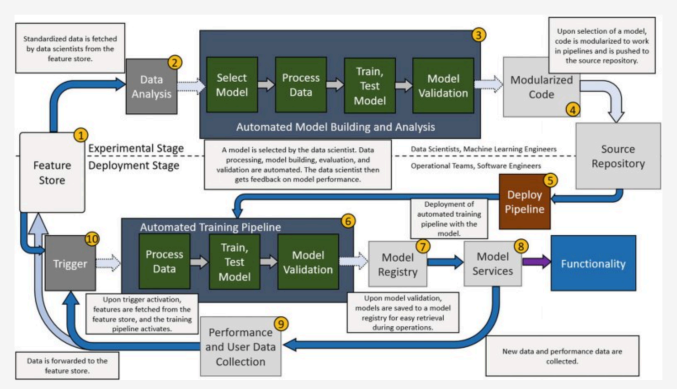

Fig. 6. Taken from [8]. Maps different components of a potential Continuous Model Delivery system.

Automated Model Building step streamlines the main responsibility of the Model Building Team beforehand, and is responsible for the automated building, training, evaluation and validation of the model. The description and the parameters of the given model are created by data scientists and machine learning engineers and given to the pipeline, which outputs a finished model that is ready for deployment to a code repository.

After manually deploying a specific model from the code repository, Automated Training Pipeline, which can be triggered either manually or through some automated response to some exterior factor (once every week, accuracy below a certain threshold), automatically pulls clean data from the feature store and trains the model with the data. After the training, the model and its updated parameters are stored in a specialized Model Registry. The operators of the application can access the models through the Model Registry and choose, for example, the best-performing model to use and integrate with the rest of the application.

The system can also send information about the performance or various metrics back to the developers of the model through the Performance and User Data Collection step, which automatically creates a feedback loop from Model Registry to the Feature Store.

Overall the Continuous Model Delivery setup creates the foundations of the MLOps principles. Through the automation of model building and model training steps, the pipeline allows the rapid creation of new models and experimentation. Both the data and the models are stored in specialized repositories that allow more organized maintenance as well as easier access. Data scientists and machine learning engineers receive feedback from the models they create for faster response to change in data or trends. That being said, there are still some underlying flaws in this setup that partially slows down the pipelines. The two most glaring ones are the manual deployment and pipeline testing steps. Manual deployment means that the pipelines need to be re-deployed manually any time a structural change is made anywhere within the source code of the model. Manual testing of the pipelines causes the experimenting with many iterations of pipelines rapidly to be impossible. To summarize, although greatly improved from the manual version, there is still room for improvement in terms of automation in this type of setup.

Continuous Integration/Continuous Delivery of Pipelines builds on the existing framework of Continuous Model Delivery and improves/automates various steps in the model to allow rapid creation and deployment of pipelines and their respective models.

Pipeline testing, which is a crucial addition for achieving Continuous Integration in the system, is primarily an automated testing environment for proposed models and their pipelines. After the models are built and tested, the pipelines themselves are tested to ensure their outputs are correct and fit to the structure of the application. The tests can be done to test the functionality of the whole pipeline or any individual component, for example, to check if the data files that are being outputted are the correct type and version or if the data processing step correctly cleans the data. Automatically testing all these processes that are done automatically ensures that if there are no failed tests, minimum human interaction is needed for a functioning pipeline ready to be deployed.

The pipelines that pass the tests then are stored in the package store, which is the specialized unit of storage for pipelines ready for deployment. The pipelines developed by data scientists and machine learning engineers can then be deployed by the operator team and software engineers. This feature is crucial for the Continuous Deployment principle because it increases the speed of the deployment process through automation as well as decreases the dependency of operators of the system for the developers.

A well-structured implementation of Continuous Integration/Continuous Delivery of Pipelines as an architecture achieves to minimize the human interaction needed by not only automating the development and deployment of models but also their pipelines. This level of automation maturity theoretically gives the

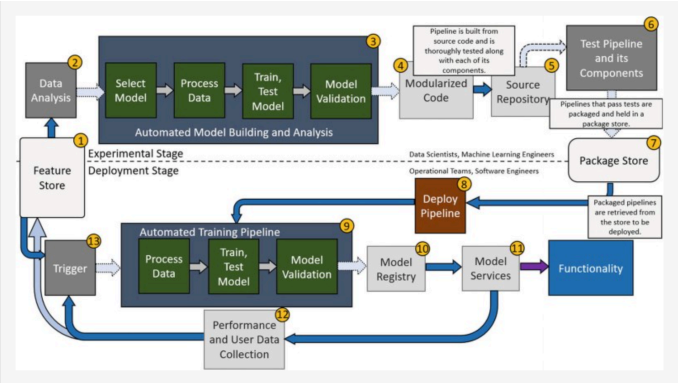

Fig. 7. Taken from [8]. Maps different components of a potential Continuous Integration/Continuous Delivery of Pipelines system.

greatest potential for experimentation and thus improvement of performance by decreasing the time spent and cost of ML pipeline creation.

It must be noted, however, that the given classifications of automation maturity levels do not always need to fit exactly the descriptions. In the comparison section, we analyze tools that have different solutions for the problems or belong somewhere in between two classifications. Also, a higher level of maturity might not always be the right way for the given circumstance. As an example, a startup might benefit from having a prototype pipeline developed and deployed manually in a fast manner rather than investing money and time to create a fully automated system beforehand just to build the same prototype. That being said, levels of maturity as a concept is encountered throughout the research, and many frameworks and MLOps solutions have different systems specifically for the level of automation maturity that is needed in that application.

Fig. 8. Diagram of different challenges, concepts and ideas discussed in Section 4.

3 Framework Introduction and Analysis

3.1 Introduction

Being a relatively new topic of research and experimentation, there are various practical implementations of MLOps coming from academia with distinct goals and methodologies. In addition, there are a number of commercialized "MLOps as a Service" products provided by many big tech companies that offer more customizable solutions, which can be modified for the exact need of the customer. In the following section, we will introduce and analyze various MLOps frameworks and platforms from a technical and operational point of view, referring to the topics discussed in the previous section when appropriate. After individual analysis, there will be an evaluation and comparison section, where frameworks will be ranked based on specific evaluation criteria that are based on the findings from Section 2. The MLOps frameworks and platforms are sorted based on whether they are proof of concept, open-source projects, or fully-managed MLOps services, in that order. This distinction is important for comparison, as the research has made clear that the ownership of framework/platform is the most influential aspect in the thoroughness and completeness of the automation in regards to the pipeline creation, and consequentially the number of tasks identified in Section 2 that are solved.

3.2 Proof of Work Frameworks

ease.ml/CI ease.ml/CI [32] [7], is a part of the project [2] developed by partly by DS3Lab at ETH Zürich. Coauthored by many other teams(Snorkel, Label box, Holocleans, Kubeflow, etc.), ease.ml is an open project to create a fully automated AutoML system consisting of modular steps, including the ease.ml/CI, which is the Continuous Integration and Testing piece of the puzzle.

Before the ML modeling starts, the ease.ml package includes pre-ML subprocesses such as automatic data injection and augmentation (Ease.ML/ DataMagic) and feasibility study (Ease.ML/Snoopy) that performs various technical analysis and groundwork for the modelling step. Ease.ML/DataMagic provides aims to standardize the different possible data formats the datasets might be in into a flexible, low-overhead template called Document. Ease.ML/Snoopy can also be utilized as a way to perform a statistical feasibility study on the given datasets before modeling to avoid any unnecessary labor and time investment on ML tasks with impossible accuracy goals.

The continuous integration system then operates on a step by step basis with a feedback loop on a model, where a script (.yml file) describes various properties of the testing sequence, examples including the passing condition, the decision between adaptive or non-adaptive testing and steps, which represents the number of times the dataset can be used to test before a new dataset is expected. The user also as input presents the dataset with N samples to the

[2] `https://github.com/DS3Lab/easeml`

system, N being determined by the system beforehand given a certain accuracy threshold. ease.ml/CI then automatically builds and tests the model with the given N samples as test set, and presents a binary (pass/fail) output. If the output is passing, then the system can be developed in two ways. Either the system can throw a signal to the developer regarding the result of the testing, or the new ML code, parameters and the artifacts can be automatically committed and deployed.

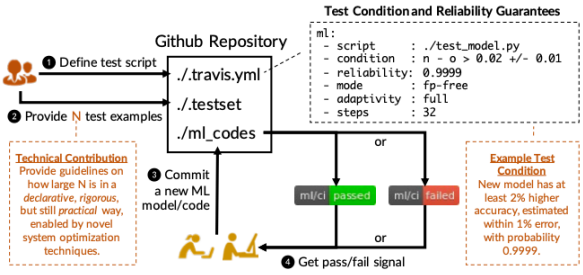

Figure 1. The workflow of `ease.ml/ci`.

Fig. 9. Taken from [14]. Illustrates the different parts of a ML model.

It is crucial to pass only a binary (passed/failed) result rather than a test set accuracy score as feedback to avoid "information leak" back to the system. Generally speaking, a ML model should train on the training dataset to optimize its performance on the real-life data it will work on once in runtime. The test/validation sets are meant to be faithful representations of data that can be expected in runtime, thus its inherent distribution or features should be unknown by the model or the developer during training. If the score of the test set is "leaked" to the model, the next training iteration can use this "rumor" to change the parameters specifically to score higher in the test score, which would harm the integrity of the test set/training set division. This phenomenon is called "adaptivity" and most of the ML/CI frameworks aim to keep the adaptivity below a certain threshold to avoid excessive overfitting.

mltest mltest is a project from Karlas et al. [22], who initially base mltest on the theoretical foundations of the ease.ml project to create a more practical implementation of continuous integration to machine learning model adaptation with a few new features. Similar to the ease.ml, the emphasis is to create an automated testing pipeline that is configurable and automated.

Multiple amounts of testing with the same or similar test sets increase the risk of the adaptivity problem, which was discussed in the subsection above.

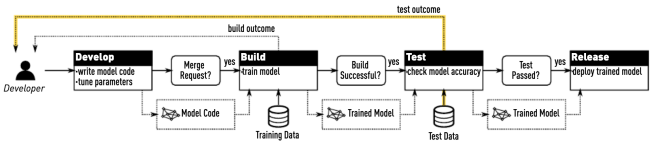

Fig. 10. Taken from [14]. Illustrates the different parts of a ML model.

Building on top of ase.ML/CS, mltest project aims to reduce the amount of overfitting by cycling through test sets in certain intervals. There is an obvious trade-off between overfitting and sample complexity, since creating a new test dataset for every iteration of testing is also not feasible, as that would require cycling through countless amounts of cleaned datasets, which in most of the realistic ML application cases simply don't exist. mltest attempts to solve it by using probabilistic semantics.

Assume that there is n number of data samples ready to be used for testing. Rather than using it all, or dividing it into x batches, simply keep it all in a test dataset pool. With each testing iteration, pull m samples randomly and score the accuracy of the model. This score is obviously random since a different randomly pulled m samples from the test dataset pool can score differently. To "overcome" the randomness, attach a certain confidence interval to the test condition n - o ¿ 0.01 +/- 0.005 , and return only a binary (passed/failed). By configuring the random batch sample size m(what matters is m in comparison to pool size), a testing iteration on a random batch can statistically achieve the preferred confidence to perform similarly to the testing done on the entire pool size, without leaking any information about the rest of the pool.

Alongside probabilistic testing semantics, mltest has the benefit of being "seamlessly" integrable with existing, popular CI tools such as TravisCI, Microsoft Azure DevOps and GitHub Actions, and is available to public.

3.3 Open-Source Frameworks

Kubeflow Kubeflow[13] is a framework that is tailored towards simplifying and supporting the development of machine learning pipelines in the Kubernetes infrastructure. Originally an internal MLOps framework in Google[13], the tool was changed to be an open-source project in 2017. Kubeflow leverages the already existing and popular Kubernetes paradigm of container/cluster to create efficient and scalable partitioning of several steps in a ML Pipeline. With this line of thought in mind, individual components(or operational steps, such as pre-processing data, training, etc.) in a pipeline are described and abstracted as docker images, which include self-contained lines of code, metadata information as well as output/input which link them to other docker images(or components). Describing docker images with their output/input enables a faster development process of a pipeline since the individual components have explicit descriptions of where they slot in a workflow. The input/output definitions also tend to make

bug fixing easier by creating isolated and self-contained bodies of code, which means an error in a faulty component is relatively easier to expose than in an uncompartmentalized pipeline.

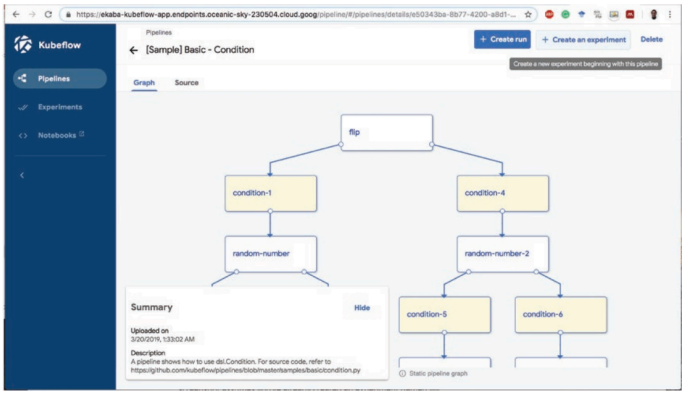

Fig. 11. Taken from page 683 of [13]. Illustrates the pipeline section of Kubeflow UI.

Kubeflow offers a very clean interface and organized interface that enables many of the operations regarding the pipelines to be made without the usage of Kubernetes specific kubectl CLI, which is crucial since many ML Developers don't have experience with Kubernetes or DevOps engineering in general. Individual components can be mixed and matched to create pipelines, which can then be "experimented" in the experiment section to test different models as well as tune hyperparameters.

Since many popular tools for ML implementations (Jupyter Notebook, Pytorch, Tensorflow, Katip, etc.) have native support, Kubeflow is very customizable in terms of the technologies used in individual components. That being said, there is a very clear Kubernetes and indirectly Google Cloud Platform(GCP) dependency that might be problematic depending on the specific implementation.

MLflow MLflow[39][15], which is an open-source MLOps project initiated by the Databricks team, offers a unique approach to the problem by being divided into four distinct components that can be utilized together or separately depending on the functionality required. The components, MLflow Tracking, Mlflow Models, MLflow Projects and MLflow Model Registry, all can be deployed separately and be mixed with other MLOps frameworks so that the strengths of multiple frameworks can be benefited at the same time.

MLflow Tracking is the API package used for logging and querying experimental runs, which outputs relevant data such as metrics, parameters, versions in customizable output files called artifacts. These artifacts can then be stored locally or in the cloud(for projects that require centralized tracking to compare

models of multiple developers), and can be accessed through its customizable UI or its API. In addition, one very interesting implementation is the Experiments section, which allows multiple developers to "experiment" on the same database and share the chosen metrics of the results in a leaderboard to compare the performances of specific models.

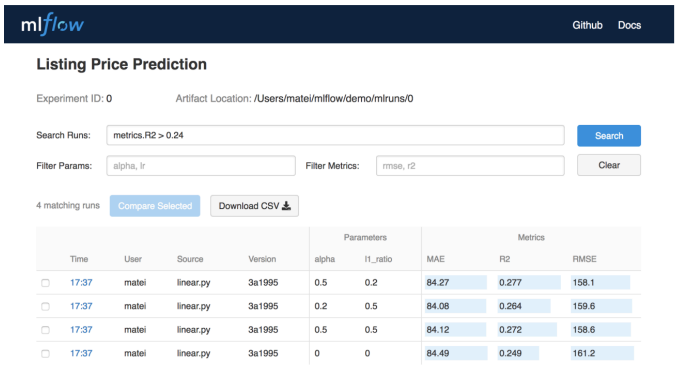

Fig. 12. Taken from page 42 of [15]. Illustrates the experiments section of the MLflow Tracking component.

Mlflow Models is a lightweight model packaging component that enables the same ML models to be used in a variety of environments. A model is simply a directory of various code files, as well as a YAML file that contains metadata and instructions. Unique to MLflow Models is the "flavor" feature, which is appended in the YAML file and enables the same model to be deployed in different execution environments seamlessly. As an example, a YAML file can instruct to deploy a sklearn model if possible, but also work without it (using more low-level python functions) by featuring two flavors, which increases the flexibility of the code and reduces unnecessary repetition.

MLflow Projects is MLflow's chosen format to create reproducible, packaged code to run in diverse programming environments. Similar to other MLOps frameworks, the code is packaged with a YAML file that describes its dependencies, parameters and other relevant metadata so that the project can be deployed optimally in local computer or the cloud.

Lastly, MLflow Model Registry is the latest component added to the project and is the CI/CD aspect of the MLflow framework. Mlflow model registry functions as a collaborative hub to monitor and manage the model deployment lifecycle. There are 4 pre-defined logical stages: Development, Staging, Production and Archived. Developers and deployers can request state transitions and organizations can restrict access to the models on an individual or team basis. More mature projects that rely on automated tests can integrate model registries with their preferred CI/CD tools by providing API to fetch specific models.

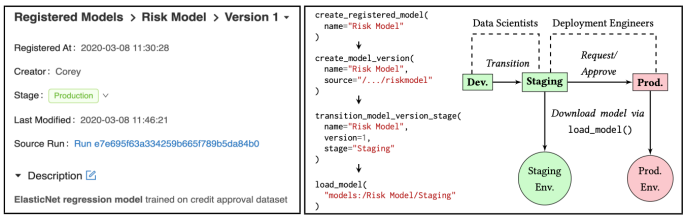

Fig. 13. Taken from page 3 of [39]. User Interface for the MLflow model registry component.

Both papers[39][15] state a very clear lack of data management tool within the MLflow project, one that pre-processes, manages, divides and versions input data for the models. Such data management tools are generally considered to be very crucial for any complete MLOps implementation, which is highlighted by the data subsection in the SLR. Utilizing only the components in the MLflow project to create a completely automated workflow might be a challenging task, that being said, the fact that the components are separately usable makes MLflow one of the more popular MLOps solutions in the market, especially when combined with other tools to overcome its shortcomings.

Google TFX TensorFlow Extended is a specialized machine learning platform created by Google[12] that although can serve as a general MLOps framework, has a set of features that specifically serve to enhance the continuous training and deployment aspect of the process. TFX was originally deployed as a way to train the personalized recommender model in the Google App store[11], an online shop app store with new additions and updates every hour. To eliminate the need to manually retrain the model in certain time intervals to include the new additions and updates, TFX was initially deployed with an automated and online trainer that continuously trained the models in a seamless fashion. TFX after its deployment evolved to be a more general and complete framework with an end-to-end approach, while still keeping the continuous deployment aspect as its core strength relative to other competitors.

The first step in the framework that is worth highlighting is Data Preprocessing, which includes Data Analysis, Data Transformation and Data Validation. In the scale of big data such as the google example, where input influx is generated by multiple different pipelines, systems or logging mechanisms, having bugs or errors is most often than not the norm. It has been observed throughout the industry that it is often more efficient to implement a proper data validation step to fix incomplete or faulty data, rather than to individually fix each data entry point.

Aligned with that line of thought, the Data Analysis step first automatically analyses the datasets fed to the system and creates useful metrics for the next steps, such as distribution of features, in the dataset, potential groups of datasets with shared features, etc. Data Transformation mostly deals with data

wrangling, where input data of many different sources are "wrangled" into existing, formalized feature space that eliminates the need to create new model trainers for new datasets. For this purpose, "vocabulary generation" is a very popular method of wrangling, which involves different features, often in string, to be transformed into a set of integers, with a string-to-int map orchestrating the transformation process. Integers are generally preferred because they use less memory, a factor that can potentially reduce very significant amounts in cost, especially in the context of big data. Lastly, data validation checks the transformed data for signals of unhealthy data distribution. This is achieved by cross-examining the features of the data with a schema that provides a versioned description of the expected values of the feature distribution, which signals for potential errors if the comparison is off by an arbitrary amount, defined by the user. The flagged datasets then can be depreciated with no further use or can be manually inspected by an operator.

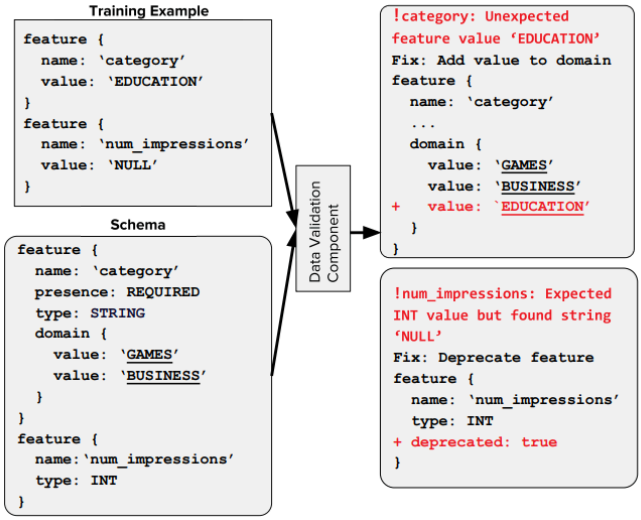

Fig. 14. Taken from page 1390 of [39]. An example case for the comparison of input and its feature schema.

Model Training step can be viewed as the second differentiating part of the TFX framework and includes state-of-the-art techniques[11] to create a seamlessly continuous and online training experience. Warm-Starting, which is TFX's implementation of the widely known transfer learning technique, uses sets of models, that are either pre-trained with prior datasets or use model parameters similar to the prior models, in order to expedite the training of the models in the pipeline. This is an effective method as many of the features used within the models are shared, and the "knowledge" of the models can be taught faster

by sharing model parameters, rather than learning it from scratch by extraction from training datasets. It should be noted, however, that the warm-starting approach is effective to the extent that the features are shared between the models themselves, or are generalizable to features with similar properties or distributions.

The other component in the paradigm of continuous training is an automated model validation process, one that can ensure the models are safe to serve and have the desired prediction quality. Generally speaking, models that are ready to be deployed in the pipeline are initially tested offline using specialized, computable metrics such as AUC or cost-weighted error that are configured to approximate relevant business metrics. The models that pass the initial offline testing are then validated online in a canary deployment process, in which they are compared either to a fixed performance metric or the prior model. As discussed before in Section 2, it is very hard to differentiate the unhealthy signals from safe fluctuations in model behavior and performance, especially when continuous training and canary deployment processes are involved, both of which create additional performance variations. Speaking from experience, Baylor et al.[11] argue that creating loose thresholds is generally more appropriate in complex systems and pipelines, but also comment that loose thresholds might let minor bugs to emerge in the models without being exposed or fixed.

Ultimately, TFX[12][11] is a great tool for specific purposes such as continuous training, continuous deployment and big data scenarios, purposes that reflect its origin in Google. One feature, or flaw depending on implementation, is its very obvious and tight dependence on TensorFlow, a very popular machine learning library. Most of the API's that are used in the framework are all high-level TensorFlow API's, such as its model specification API to describe models in pipelines. This dependency can be highly problematic in more complex systems where many frameworks and components are mixed and matched, which is why it is safe to assume that TFX is not as flexible as some of its competitors, albeit being very remarkably good at its more specialized use cases.

MetaFlow Initially developed by Netflix, MetaFlow is an open-source MLOps framework with a unique approach to managing scalable, enterprise-level data science projects, with its time-based architecture consisting of three different states in a pipeline.

The first state, Development-Time, is defined by a "flow", a business logic that needs to be executed within the pipeline. It consists of many individual "steps", the smallest units of computation that take an input and give an output. These steps are further customizable by "decorators", specific external code that can be applied to the steps to give additional functionality or description to a step, rather than writing new steps repeatedly. Decorators can be anything from exceptions catchers to resource requirement descriptors, and a step can feature an arbitrary amount of decorators. A "graph" is a mapping of the logical flow of all individual steps in a pipeline, that are linked with their respective input/outputs. Optionally, checkpoints can be implemented after every step in the graph, which

not only protects the state of the current flow within a data artifact stored persistently in an artifact store, but also enables the flow to resume from the specific checkpoint location within the graph. This is invaluable for large-scale projects, where the ability to conserve the state of a pipeline before an error can save enormous amounts of time and resources.

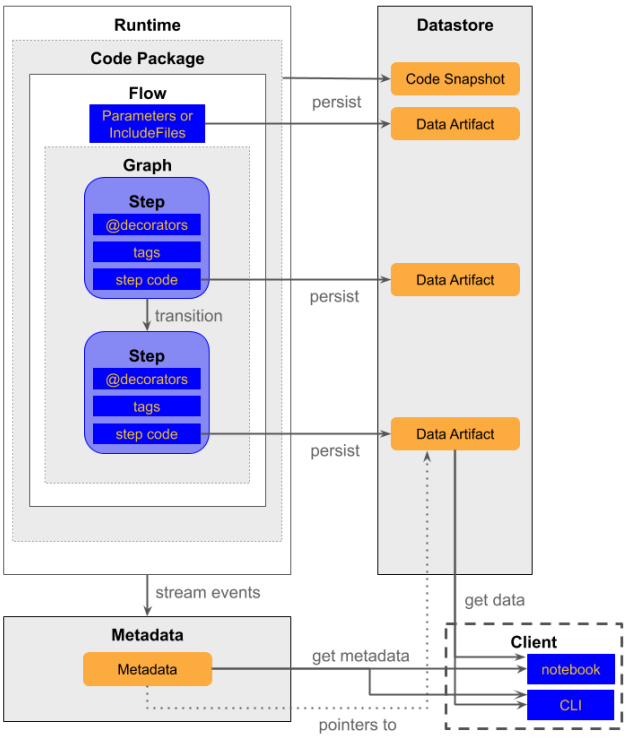

Fig. 15. Taken from [37]. Visualization of the unique MetaFlow architecture.

The second state is the Runtime, which emphasizes scalability, adaptability and flexibility in different possible execution environments. Tasks are the primary units of the Runtime, and each one represents an executable operation described by a step in the Development-Time. One step can have many tasks for it, however, such as additional tasks that are spawned by the decorators of the step. Code package is a snapshot of all the relevant flow code in the working directory, which contains all the information regarding the execution of individual tasks in an environment-agnostic way. The environment is all the dependencies as well as parameters specific to the execution environment, which encapsulates the code package to create reproducible and scalable runtime execution. Although MetaFlow currently provides a lightweight runtime orchestration tool, it is also

possible to use a more complex runtime orchestration tool that also supports retries, logging, etc, which might get especially handy during production.

Third and the last step in the system is Result-Time, which consists of internal and possibly external tools to gather, analyze and visualize the outputs after runtime. Currently, MetaFlow provides the metaflow.client library to expose the internal state of a run in a Jupyter Notebook, as well as its data artifacts collected throughout its runtime, however, any 3rd party data visualization tool can be used in parallel to further enhance the process of monitoring.

In short, MetaFlow manages to bring a unique approach to the MLOps ecosystem by creating a new pipeline architecture and terminology, which attempts to optimize scaling and reproducibility with its clear 3 layer system. It is generally quite flexible in terms of its ease of integration with other similar tools, and the ability to automatically create checkpoints that both save the pipeline state and enable the possibility to restart from in case of an error further in the flow is an invaluable asset for the framework that can potentially save hours of unnecessary computation. Moreover, its ability to integrate seamlessly with cloud solutions such as AWS makes MetaFlow a popular choice for many data scientists.

MLReef MLReef is an open-source MLOps platform that integrates many classical aspects of a MLOps framework with a collaboration-emphasized, public model networking and sharing medium. The framework architecture somehow represents this added social aspect, with the classical Create, Build and Manage steps combined with the new optional Share step.

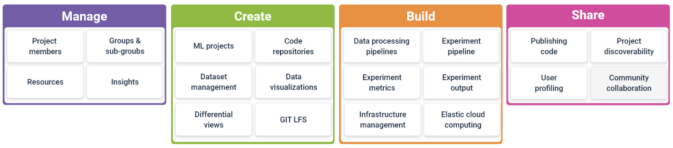

Fig. 16. Taken from MLReef documentation[2]. Four components of MLReef framework.

The build step is generally the first step in the workflow and consists of four different possible types of repositories that are managed and distributed in a git style version-control-system.

The model repository is the code repository for the model, generally written in Python, which contains the architecture and the learner for the model. DataOps repositories are a collection of python scripts that take inputs and parameters, execute some data-procession operations depending on the parameters to the input, and ultimately give an output. Data Visualisations are also python scripts that create arbitrary visualizations from the input data, either before or

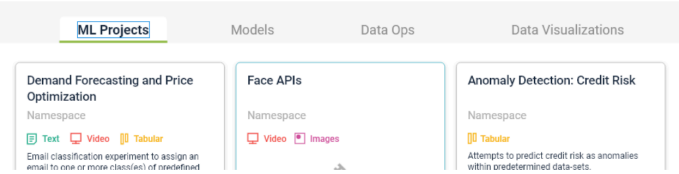

Fig. 17. Taken from [2]. Interface of the four code repositories.

after the DataOps scripts. And lastly, ML Projects are either single or a collection of ML pipelines that orchestrate the execution of code from the other three repositories, as well as additional relevant data and parameters, making them both data and code repositories.

Build step, which is at the time of this paper a work-in-progress, is based around the concept of experiments, which are distributed deployment and execution of the given ML Projects that output a model, and optionally, artifacts that contain relevant statistics and log files based on the output model.

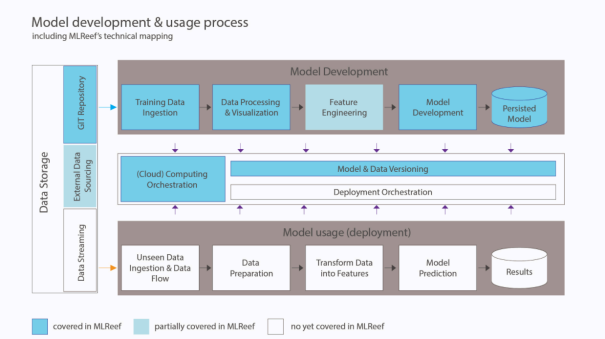

Fig. 18. Taken from MLFlow github page[3]. Visualization of the MLFlow workflow, colored based on the completeness of the implementation.

The unique feature of the MLReef framework in comparison to other MLOps tools are the Manage and Share steps, which enables the possibility of the models and their datasets to be shared between the team or the public through a simple role-based sharing platform. Manage Step includes users that can be given specific roles and authorizations for viewing and pushing the ML Projects, similar to a git-based distributed version control system. Share enables teams to share their repositories to either the rest of the team or to the public. Visibility of the data and code files can be changed both for single files through the visibility parameter, as well as ML Projects which apply the given visibility broadly to all sub repositories. The capability to share model architecture and hyperparameters while keeping the right to protect datasets is very crucial for different

teams to distribute their know-how and knowledge without leaking valuable or confidential data.

As stated by Zhao et al. [40], the current data science community and sector can potentially benefit enormously from a shared model system, especially since many of the models developed throughout the field attempt to accomplish very similar goals. A platform that enables model and parameter sharing while respecting data confidentiality can save hours of unnecessary computation and labor, making implementing new projects much easier for especially the non-expert machine learning projects. Although not every feature of the framework is fully implemented yet, MLReef has the unique position to solve two problems with one solution by being a simple, intuitive MLOps platform that enables public sharing and collaboration of ML model architecture and parameters.

3.4 Fully-Managed MLOps Platforms

SageMaker Amazon SageMaker, which is the fully-managed MLOps platform offered by the Amazon Web Services(AWS), is a very polished and highly compartmentalized MLOps solution with many different components for specific needs[30]. Being a paid service provided by the world's leading web service company, most of the more sophisticated services are provided as external packages that can be integrated seamlessly with the default SageMaker service. Customers have the freedom to pick the packages they require and fit the bill accordingly.

For this section, the focus is mostly on the default or "naked" Amazon SageMaker service. This is mostly because most of the additional services provided are functions or supporting tools that can be implemented relatively easily within a developer team, and are mostly suited for use cases that require machine learning algorithms without a dev/op team devoted to maintaining them. Just to give an example for the statement above, SageMaker Clarify [20] is Amazon's specialized solution package for bias detection and computing feature importance, metrics that somehow "translate" the knowledge of algorithms back to the human users. Detecting bias or understanding how certain features affect the output is obviously very important in certain use cases such as social media or politics, where some of the harmful aspects of ML algorithms are becoming more clear, however, these are metrics that professional data scientists care about and are taught to inspect anyways, and use of a paid, external product for this specific purpose may not be necessary for a team of experts.

Regarding SageMaker without any additional components, the system is a managed, end-to-end approach to the MLOps frameworks that can be fully realized in the cloud. Although most of the frameworks that are analyzed in Section 3 offer cloud computation in some capacity, SageMaker provides very clearly a more sophisticated approach to cloud computation and scalability with unique solutions. One of such solutions is the ability to predict costs for doing cloud computations, and pausing and dividing the workload using elastic and scalable training schemes. As stated by Liberty et al.[26], large-scale ML operations generally have imbalanced compute workloads, which includes perhaps days without any new model training followed by days of concurrent training for many models

in hundreds of different machines. The costs for computation in the cloud benefit from monotonous and continuous jobs, rather than spikes in compute power required. The elasticity and the ability to distribute computation over time result in significantly lower cloud costs, which becomes increasingly important as the scale of operations gets bigger.

Parallel to computation elasticity, SageMaker also offers the feature to maintain and correctly dispose of "ephemeral data", streams of input data that generally take relatively big amounts of storage space and can be discarded after utilized, video streams or network traffic can be given as examples. Ephemeral data are streamed to the pipeline and only stored in data batches of fixed length and fixed duration, significantly lowering cloud storage costs of otherwise very expansive data formats. SageMaker also offers a variety of ready-to-use machine learning algorithms(mostly simpler, non-deep learning algorithms that are evidently [26] easier to stream and more elastic), such as linear regression and classification models, K-Means clustering algorithms, principal component analysis models, etc.

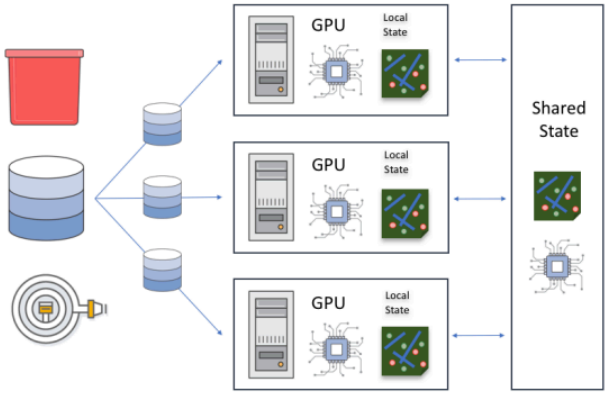

Fig. 19. Taken from page 4 of [26]. Visualization of 3 learners being fed one stream of data with the support of MXNet library .

Another, a more technical remark is the use of MXNet [16] library across all of the ML algorithm implementations, which acts as an interface between code and the underlying hardware by acting as a graph of tensor operations for maximum efficiency in both CPU's and more specialized GPU's. It has been noted that the GPU's specifically benefit greatly from the use of the MXNet library, by leveraging their ability to compute parallel operations.

To summarize, Amazon SageMaker is a fully-managed MLOps platform with various supplementary packages for less mature machine learning projects. SageMaker offers very clear advantages, perhaps the most important being its ability

to operate parallel to other AWS services, such as Amazon Cloud Services. In addition to ease of integration, SageMaker has tools, such as parallel learning and workload distribution that reduce clouding costs and increase the efficiency of the learners, which becomes more important as the scale of the project increases. However, its dependence on Amazon Cloud and its price tag, which scales with the resources taken for cloud computations and storage in addition to the fixed rates, can be seen as its major flaws.

Azure Machine Learning Azure Machine Learning platform, developed under the Microsoft Azure Web Services, is a very comprehensive, fully-managed MLOps platform that achieves an end-to-end approach by featuring an ecosystem of different MLOps components and tools that the users might benefit from in their machine learning projects.

Similar to SageMaker by AWS, being part of a bigger web service platform with a network of integrated service components is a huge advantage for Azure Machine Learning in comparison to its other competitors, with many possible tools to benefit from with seamless integration. One distinct example is the Azure Synapse Analytics, with built-in PySpark, a very popular data-procession tool, especially for big data.

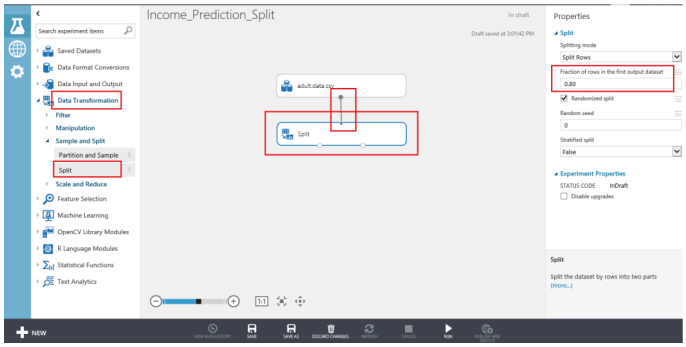

Fig. 20. Taken from page 62 of [10]. Training/Test split done within the Azure ML Studio interface.

Another unique component of the Azure Machine Learning platform is the Azure ML Studio, which is Azure's IDE/UI hybrid application that helps to increase the ease of access through abstraction and visualizations of common data-scientific operations and activities. The abstractions range from simplification of common ML pipeline orchestration through a drag-and-drop interface, to customizable data cleaning and procession operation chains that can be created and then stored in the ML Studio for future use.

After the models are created through pipelines, they can be automatically exposed as web services for future use in different external or internal use cases.

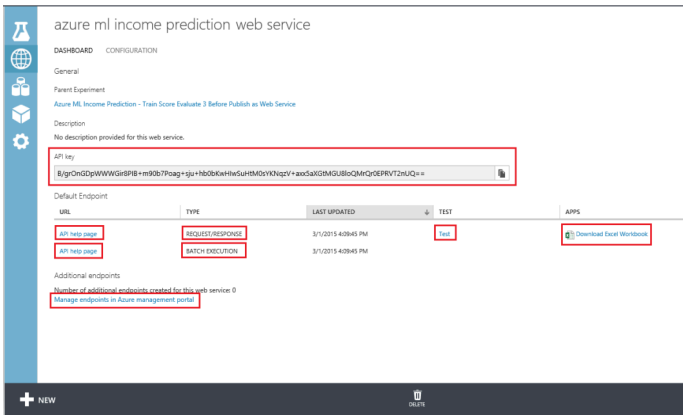

Fig. 21. Taken from page 80 of [10]. Web services dashboard for endpoint creation and management.

The automated nature of the complete web service creation process, including the management of the endpoints, is somewhat unique even compared to other fully-managed services.

Being one of the components of a bigger web service package, Azure Machine Learning benefits from the ecosystem it is positioned in and is consequently one of the most complete MLOps platforms included in the comparison. The abstractions for model creation, ease of use due to the advanced interface and visualizations make the Azure ML one of the better fitting MLOps systems for teams with lower project maturity, a property that is shared across most of the other fully-managed MLOps platforms as well.

Valohai Valohai is a fully managed, fully automated MLOps platform with a clear end-to-end approach. It is distinctive in the degree the non-data science aspects of a MLOps paradigm such as DevOps, cloud management, data storage, etc. are all fully automated or abstracted from the user.

Orchestrating any Valohai project is a special Valohai.yaml file, which defines executed computations as well as the REST endpoints generated by the project in a given order. Steps, specified in the Valohai.yaml file, are any modular type of execution, such as training, data loading, etc. Each step is defined by a name, an operation or operations(mostly scripts or executable files), a docker image and any relevant input/output/parameters for the given operations. The steps are then positioned in their respective positions in pipelines, which are defined by their name, nodes(steps) listed by their order in the workflow and edges that define the data flow between nodes. Endpoints are special, auto-scaling REST endpoints that are technically groups of Docker containers running HTTP servers that get automatically deployed to a Kubernetes Cluster. Results are very flexible, automated REST endpoints that make communication between different

```yaml
- step:
    name: generate-dataset
    image: python:3.6
    command: python preprocess.py
- step:
    name: train-model
    image: tensorflow/tensorflow:2.2.0-gpu
    command: python train.py
    inputs:
      - name: dataset-images
        default: http://...
      - name: dataset-labels
        default: http://...
- pipeline:
    name: simple-pipeline
    nodes:
      - name: generate-node
        type: execution
        step: generate-dataset
      - name: train-node
        type: execution
        step: train-model
      - name: deploy-node
        type: deployment
        deployment: mydeployment
        endpoints:
          - predict-digit
    edges:
      - [generate-node.output.images*, train-node.input.dataset-images]
      - [generate-node.output.labels*, train-node.input.dataset-labels]
      - [train-node.output.model*, deploy-node.file.predict-digit.model]
- endpoint:
    name: predict-digit
    description: predict digits from image inputs ("file" parameter)
    image: tensorflow/tensorflow:1.13.1-py3
    wsgi: predict_wsgi:predict_wsgi
    files:
      - name: model
        description: Model output file from TensorFlow
        path: model.pb
```

Fig. 22. Taken from Valohai documentation page[4]. Sample Valohai.yaml file, illustrating different elements and their parameters.

projects, or between teams working on the same project significantly easier and error-proof.

Machine Learning workloads are then encapsulated in entities called "Executions", which are defined by the steps in the project's Valohai.yaml file. Executions can be created by API calls, CLI or the web app. Execution environments and images define the specific execution space, any dependencies for the execution can be installed by either external scripts (pip install) or in Valohai.yaml file. Individual executions in a pipeline can leverage different execution environments since the operations in a pipeline often require different hardware/software systems to be optimized. As an example, while training large deep networks require many tensor/matrix multiplications and is best done by specialized GPU's, preprocession of data coming from stream is best done with very large data storage and agile CPU's that are more agile than GPU's with data transportation.

Valohai Executions are archived in a way that stores not only its particular code, execution environment, costs related to the execution, etc. Further metadata from the execution, which can be anything from basic statistics to high-level visualizations, can be extracted from past and ongoing executions. Input data and output data connected to the execution can be traced in the system, and pinpointing the exact location and time of the data set can offer insight into the context or the distribution of data. If needed, the exact execution can also be re-run easily with the same environment, parameters and data.

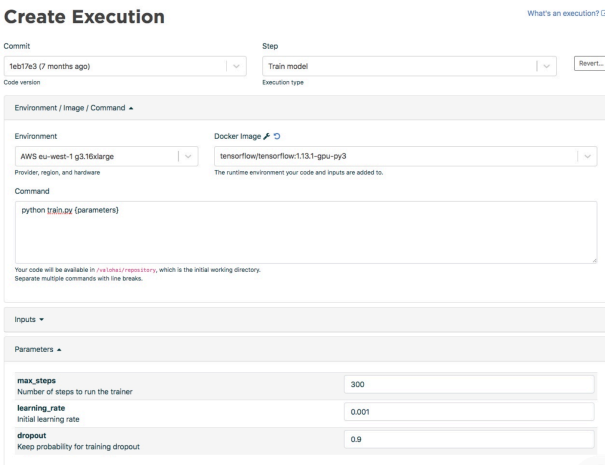

Fig. 23. Taken from Valohai documentation page[4]. Web user interface for creating executions.

The worker nodes can be deployed in different ways, with options for various privacy and hardware requirements. Valohai cloud installation is the easiest, out-of-the-box functioning solution where the workloads are run under Valohai-owned AWS or Google Cloud accounts. The customers are billed depending on the resources used, however, the data confidentially is not guaranteed. A more secure option is the private cloud installation, with user data being contained in the user-owned cloud domain. The third option is local-hosted installation, where the worker nodes are deployed to on-premises, persistent hardware.

To sum up, the Valohai platform attempts to create an end-to-end, fully automated MLOps framework that can be run and utilized without DevOps expertise. Unlike other, more compartmentalized fully-managed solutions(such as AWS Sagemaker or Microsoft Azure ML), Valohai follows a centralized control scheme while giving the user the flexibility to choose the specific execution/data storage domains, which makes it unique to the extent it achieves cloud integration without specific dependencies, in comparison to similar fully-managed services.

Clear.ML Clear.ML is a MLOps framework that uniquely provides both a free open-source, as well as a paid, managed MLOps service that has additional features on top of the free version. The additional features are not trivial, especially "Hyper-Datasets" and job scheduling components for complex execution sequences, which is why the fully-managed option is chosen for the following comparison section.

One of Clear.Ml's distinct feature is Clear.ML Agent, which is a virtual environment and execution manager that achieves to abstract the differences between cloud and on-premise computation for the user. For most of the frame-

works, changing the execution environment of a task requires a change in code or configuration, while with Clear.ML Agent all the configuration is done automatically and any change can be done seamlessly. The agent can be controlled either programmatically or by its Web UI, by enqueueing tasks to the listener of the agent.

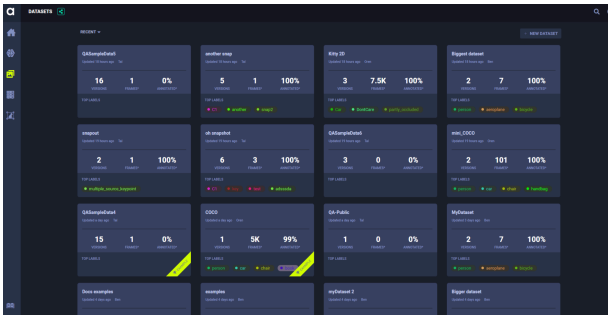

Fig. 24. Taken from Clear.ML documentation page[1]. Web user interface of Hyper-Datasets

Hyper-Datasets is Clear.ML's answer to the data management problem, which is a 3-element data abstraction that aims to reduce memory storage usage while increasing traceability and flexibility of data throughout different pipelines. The first element of a hyper-dataset is a data frame, which is the basic unit of data storage, basically a conventional dataset. Dataset Versions are the second element, which acts as archives of the past states of the hyper-dataset and the associated data frames. The Dataset Version also includes information of the pipelines that use the hyper-dataset, as well as the version that it uses to guarantee robust reproducibility. Lastly, and most noteworthy, Dataviews are special queries defined by users that create subsets of the frames, which can be used in pipelines instead of frames. The Dataviews feature heavily decreases storage costs, since the conventional alternative to Dataviews is creating new sub-datasets by filtering the already existing datasets, thus using multiple amounts of data storage for duplicates of the same data. Dataviews do not only eliminate unnecessary data usage but also extract and illustrate data-hierarchy separating the main frame and the sub-frames and making the relationship visible to the user.

Clear.ML, in addition to the unique features stated above, also features most of the MLOps components expected from a fully-managed, paid service. It is observed throughout the analysis, that in general the fully-managed services all provide the toolset required to automate the creation of MLOps pipelines, and differ mostly in the approach they take rather than the goal.

4 Framework Evaluation and Comparison

4.1 Evaluation Criteria

For the following comparison section, the evaluation and ranking criteria are based on the tasks and challenges identified in Section 2. During the comparison, a higher ranking will be given to the framework that achieves to solve an identified challenge in a more comprehensive way, or better integrate the solution to the framework approach. After disclosing every ranking individually, there will be a commentary section per evaluation criterion on the justification for the given rankings, followed lastly by a complete overview summarizing all the rankings.

Data Management The efficiency in which the input data for the machine learning models are processed, labeled, divided(training/test) and stored. This evaluation criterion encompasses the topics discussed in the Data Dependency subsection of the systematic literature review, such as Concept Drift, Noise Recognition, Data Cleaning and Pre-Processing, as well as Feasibility Analysis. Typical solutions are data stores and more advanced Data Process Pipelines with both storing and processing capabilities.

Deployment The ease of deployment in various environments, such as on-premise, cloud, etc, as well as the ability to perform online training, or seamless re-deployment of models for training purposes with the new sets of data. Based primarily on the Continuous Deployment subsection from the systematic literature review, the Deployment criterion evaluates the various strategies the frameworks include to make the deployment of models more scalable, fast, secure and seamless.

Automation The extent in which the framework/platforms achieves automation in pipeline creation. This criterion specifically refers to the level of automation that is achievable within the given framework, with a level of automation describing in other words the percentage of ML pipeline operations that can be automatically executed once configured without any human intervention, as discussed in the Automation Maturity subsection.

Modularity The degree in which the framework/platform's approach achieves modularity in code, which heavily increases reproducibility and maintenance, as well as makes the pipelines easier to understand for the users through abstraction. The topics discussed in the Maintainability subsection, especially Modularity and Reusability heavily influence this evaluation criterion.

Visibility The use of different tools and techniques such as logging, tracing, metrics and visualizations to help the developers and operators examine the code for bug fixing purposes, as well as data scientists to improve model performance. This criterion is contextually described and supported both in the Data Dependency subsection of the systematic literature review, where data visualizations are discussed as one of the ways of identifying unhealthy data, as well as the Testability and Analysability sub-topics of the Maintainability subsection.

Collaboration A qualitative measure of the support a framework features for collaboration of different users working in the same project. Although not specifically identified as a MLOps challenge in the systematic literature review, Collaboration is included as part of the comparison criteria as it reflects the social aspect of the more general DevOps paradigm. Classic examples are role-based systems that attempt to reflect the real-life work hierarchies in the framework for privacy reasons and Git-based Version Control Systems for models and code to enable teams of developers to work simultaneously.

4.2 Frameworks Overview

Name	Ownership	GitHub
ease.ml	PoW	https://github.com/easeml
mltest	PoW	https://aka.ms/gsl-ml-test
Kubeflow	OS	https://github.com/kubeflow/kubeflow
MLFlow	OS	https://github.com/mlflow/mlflow
Google TFX	OS	https://github.com/tensorflow/tfx
MetaFlow	OS	https://github.com/Netflix/metaflow
MLReef	OS	https://github.com/MLReef/mlreef
SageMaker	FM	-
Azure Machine Learning	FM	-
Valohai	FM	https://github.com/valohai
Clear.ML	FM/OS	https://github.com/allegroai/clearml

Name	Data Management	Deployment	Automation	Modularization	Visibility	Collaboration
ease.ml	☑	☐	☑	☐	☑	☐
mltest	☑	☐	☑	☐	☑	☐
Kubeflow	☐	☑	☑	☑	☑	☐
MLFlow	☐	☑	☐	☑	☑	☑
Google TFX	☑	☑	☑	☑	☑	☐
MetaFlow	☑	☑	☑	☑	☑	☐
MLReef	☑	☑	☑	☑	☑	☑
SageMaker	☑	☑	☑	☑	☑	☑
Azure Machine Learning	☑	☑	☑	☑	☑	☑
Valohai	☑	☑	☑	☑	☑	☑
Clear.ML	☑	☑	☑	☑	☑	☑

Overview of the frameworks. The first table lists the introduced frameworks based on their ownership (PoW = Proof of Work, OS = Open-Source, FM = Fully-Managed MLOps Service) and their GitHub pages if it exists. Checklist refers to the criteria introduced above that are addressed by the frameworks.

4.3 Rankings

.

Data Management

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	4	3	-	-	2	8.	8.	5.	5.	5.	1

For the ranking of the frameworks regarding data management, the Hyper-Datasets abstraction within the Clear.ML framework is the most complete data management system since it enables thorough traceability of the data within the pipelines and has the Dataviews option creating subsets of data with no additional storage cost. Training/Test Set separation can be done automatically, as well as any arbitrary filtering/shuffle by creating a task in the pipeline.

Google TFX's data management is especially geared towards large-scale data and stream data with advanced data cleaning and pre-procession tools. Google TFX's Data Analysis step attempts to accomplish the use of multiple sources of data by using data cleaning and feature extraction, which is crucial for complex systems.

mltest proof of work project builds on the ease.ml framework to create a systemic approach for overcoming the probability and concept drift within the input data, while many of the other open-source and managed frameworks have no distinct tools associated with those issues and generally expect the data scientist to find their own solutions.

Deployment

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	-	-	3.	7.	6	7.	9	1.	1.	3.	3.

Being a component of bigger web service ecosystems with leading cloud solutions, Azure Machine Learning and Amazon SageMaker both benefit strongly from a pre-existing cloud-based infrastructure, with options of transforming a trained model to a micro-service with one button. That being said, it must be stated that both options also have enforced dependency on the cloud service to their respective service providers, which can make the framework inflexible depending on the project case.

The other three frameworks in the 3. spots, Kubeflow, Valohai and Clear.ML all include cloud systems and automatic scaling in the system without any external service/component. Google TFX is unique in its seamless, continuous

training option for in-service models, despite not being as flexible as the three frameworks it trails in the rankings in terms of cloud technology.

The last frameworks are able to create technology-agnostic deployment scripts in some manner, but require additional tools and likely some DevOps expertise for configuration and operation.

Automation

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	10.	10.	7.	7.	5	6	7.	1.	1.	1.	1.

In terms of Automation, there is a very distinct advantage the fully-managed services have over other frameworks, especially when it comes to the degree of automation they offer in terms of general pipeline creation. Being "managed", most of the noncritical configurations or common steps are automated and abstracted away from the user.

Following the fully-managed platforms are the open-source frameworks with automation options for parts of the general pipeline creation process. Google TFX can remarkably automatically retrain, redeploy and republish individual models seamlessly with the influx of new data. Metaflow allows the individual "steps" to be programmed to be automated depending on certain conditionals etc. Other three OS frameworks, KubeFlow, MLFlow and MLReef offer options that automate some parts of the process, but don't cover the entire process. Ease.ml and MLReef both offer automatic, preventative actions against identified, mostly data-related issues regarding training and test sets.

Modularization

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	-	-	4.	1.	8.	1.	8.	4.	4.	1.	4.

The three highest-ranking frameworks, MetaFlow, MLflow and Valohai, all have systems that fully leverage modularity in different aspects, such as "decorators" or "flavors" that can be used to modify the individual computation or logic units. Both also include graph-like, multi-branch pipelines with multiple input/output sockets with more complex pipeline creation.

The frameworks and platforms in the 3. ranks also feature ways of creating modular steps that can be used repetitively and mixed and matched with other steps, however, these steps are mostly for avoiding repetition and visualizing

code within abstractions and don't contain additional modifiers for the modules similar to first two frameworks. The rest of the framework systems overall do not leverage the modularity property but may allow for partial modularity for some specific operations, declining the lower the rank is for that particular framework.

Visibility

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	10.	10.	3.	7.	6	7.	7.	1.	1.	3.	3.

Although Azure Machine Learning and AWS SageMaker again benefit from their integrated, industry-grade data visualization and analytics frameworks within their web service ecosystems, the frameworks in the 3. ranks all have the capability to create customizable model artifacts that expose any relevant information about a specific point in the pipeline, such as execution environment, hyperparameters, datasets versions, etc. Having model artifacts enable precise model reproducing capabilities within the framework, which can have significant contributions to general maintenance and bug fixing of the models.

Other frameworks in the list don't include an internal solution to a complete model artifact system, however can be modified and supported with external tools. The research has shown that generally speaking, open-source frameworks include very elementary data visualization capabilities mostly for data experimentation and most of the practical implementations of those frameworks depend on other, external data visualization tools.

Collaboration

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Ranking	-	-	-	3	-	-	1	3	3	3	2

MLReef offers the ability to share models and compare performances with other users/teams in the community while having access to privacy options meanwhile. This option is unique among the frameworks, as it creates the potential for machine learning models to benefit from the general public's expertise, similar to how many open-source projects work.

All other frameworks listed in the rankings feature some kind of role-based collaboration tool, with varying access to models and data based on the role of the user in the system. Generally speaking, the roles themselves are customizable by the highest-ranked role (admin), thus the role system can be fine-tuned to the

specific structure of the team. Clear.ML has the additional property of using the same dataset for different purposes through the Dataviews option, which allows different teams to work on the same data without potentially interfering with each other.

Summary

Framework	ease.ml	mltest	Kubeflow	MLFlow	Google TFX	MetaFlow	MLReef	SageMaker	Azure	Valohai	Clear.ML
Data Management	4	3	-	-	2	8.	8.	5.	5.	5.	1
Deployment	-	-	3.	7.	6	7.	9	1.	1.	3.	3.
Automation	10.	10.	7.	7.	5	6	7.	1.	1.	1.	1.
Modularization	-	-	4.	1.	8.	1.	8.	4.	4.	1.	4.
Visibility	10.	10.	3.	7.	6	7.	7.	1.	1.	3.	3.
Collaboration	-	-	-	3.	-	-	1	3.	3.	3.	2

It is safe to claim that the research, evaluation and comparison of the frameworks have shown the diverse ecosystem of the MLOps platforms and tools in the industry. One intuitive notion about the tools that is reinforced by the results is that the ownership of the framework has a direct impact on its general rank among its competitors, with fully-managed services performing better on average across all the evaluation criteria. The other, more interesting observation is that for open-source frameworks, which are generally initialized by a team/company and then made open source for the community, the design choices for the framework heavily impact its performance for specific aspects and those choices generally reflect the team's identity. One example of that could be Kubeflow's deployment structure that emphasizes cloud and container solutions, which is expected as the team behind the project have direct ties to the Kubernetes team, both originating within Google.

It should also be stated, that the decision to create a ranking-based comparison schema for the paper can be a point of contention, one that we acknowledge as well. A ranking-based comparison was preferred over a point-based system to avoid the necessity of handing out somewhat arbitrary points without very specific, technical and detailed justifications, which is not in the scope of this thesis. However, we decided to create an ordinal, grade-based comparison schema, one that is between 1-11, rather than not comparing the frameworks among each other at all, to give the evaluation the freedom to properly illustrate that certain frameworks have distinct advantages over others in certain aspects, based on their design choices and infrastructures.

5 Conclusion

The goal of this paper was to research the potential challenges, and their solutions, that arise from the adaptation of DevOps culture and practices into the machine learning lifecycle. In order to research the challenges and tasks of MLOps implementations, a systematic literature review was conducted and the results of the review are introduced and discussed in Section 2. One remark that can be made about the systematic literature review is the difficulty of finding academic papers about the general subject of MLOps or DevOps in Machine Learning, which is mostly justified with its rather recent emergence in the industry.

After the systematic literature review, a number of popular MLOps frameworks and platforms are briefly introduced, evaluated and compared in Section 3. The role of this section is to investigate the different, possible solutions to the challenges identified within a more practical context reinforced with concrete examples. For the comparison of frameworks, an evaluation schema is created based on criteria that represent the findings from the systematic literature review. Regarding the comparison process, it must be stated the criteria, as well as the rankings, are a highly subjective matter, and different versions of comparisons can have significantly different results. It is not the intention or the claim of this paper to provide an objective, technical comparison of the frameworks, but rather use the frameworks and their solutions to the relevant tasks as examples to delve deeper into the research question.

The findings from the comparison present a rich ecosystem of tools and frameworks with different goals, emphases and strengths. One intuitive pattern that is supported by the results is the fact that paid, fully-managed services generally perform better on most aspects of the evaluation. That being said, some of the challenges identified have single or very similar solutions across all the frameworks, indicating the existence of established, popular solutions for some aspects of MLOps.

I would also like to hereby declare that I completed the work summarized above independently in accordance with Section 46 (8) AllgStuPO.

Regarding future work, it is fair to claim that the MLOps ecosystem is relatively new with many different dimensions to improve. As the machine learning algorithms and the underlying technology become more and more popular, which is almost undeniable at this point in time, it is also expected for the demand for practical and effective MLOps tools to rise.

One slightly tangential topic of interest, which has yet to be fully realized in a commercial use case, is the concept of AutoML, which aims to fully automate the entire operation of machine learning model building, assigning even the creation of model architecture and algorithm choice to the machine, essentially being code that writes code. AutoML appears to be the next logical step after MLOps, as it further reduces the necessity of human intervention in the process of creating and maintaining ML services and applications. There are current services that provide AutoML solutions[6], and it can be expected for more AutoML services to emerge and the existing solutions to improve.

References

1. Clear.ml documentation (2021), https://clear.ml/docs/latest/docs, last accessed 21 December 2021
2. Mlreef documentation (2021), https://docs.mlreef.com/, last accessed 14 December 2021
3. Mlreef github page (2021), https://github.com/MLReef/mlreef, last accessed 14 December 2021
4. Valohai documentation (2021), https://docs.valohai.com/, last accessed 14 December 2021
5. Application deployment and testing strategies (2022), https://cloud.google.com/architecture/application-deployment-and-testing-strategies, last accessed 1 January 2022
6. Google automl (2022), https://cloud.google.com/automl, last accessed 3 January 2022
7. Aguilar Melgar, L., Dao, D., Gan, S., Gürel, N.M., Hollenstein, N., Jiang, J., Karlaš, B., Lemmin, T., Li, T., Li, Y., et al.: Ease. ml: A lifecycle management system for machine learning. In: 11th Annual Conference on Innovative Data Systems Research (CIDR 2021)(virtual). CIDR (2021)
8. Alla, S., Adari, S.K.: What Is MLOps?, pp. 79–124. Apress, Berkeley, CA (2021). https://doi.org/10.1007/978-1-4842-6549-9$_3$, https://doi.org/10.1007/978-1-4842-6549-9_3
9. Baier, L., Jöhren, F., Seebacher, S.: Challenges in the deployment and operation of machine learning in practice (2019)
10. Barnes, J.: Azure machine learning. Microsoft Azure Essentials. 1st ed, Microsoft (2015)
11. Baylor, D., Breck, E., Cheng, H.T., Fiedel, N., Foo, C.Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al.: Tfx: A tensorflow-based production-scale machine learning platform. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1387–1395 (2017)
12. Baylor, D., Haas, K., Katsiapis, K., Leong, S., Liu, R., Menwald, C., Miao, H., Polyzotis, N., Trott, M., Zinkevich, M.: Continuous training for production {ML} in the tensorflow extended ({TFX}) platform. In: 2019 {USENIX} Conference on Operational Machine Learning (OpML 19). pp. 51–53 (2019)
13. Bisong, E.: Kubeflow and Kubeflow Pipelines, pp. 671–685. Apress, Berkeley, CA (2019). https://doi.org/10.1007/978-1-4842-4470-8$_4$6, https://doi.org/10.1007/978-1-4842-4470-8_46
14. Buschmann, F.: To pay or not to pay technical debt. IEEE Software **28**(6), 29–31 (2011). https://doi.org/10.1109/MS.2011.150
15. Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S.A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., et al.: Developments in mlflow: A system to accelerate the machine learning lifecycle. In: Proceedings of the fourth international workshop on data management for end-to-end machine learning. pp. 1–4 (2020)
16. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
17. Derakhshan, B., Mahdiraji, A.R., Rabl, T., Markl, V.: Continuous deployment of machine learning pipelines. In: EDBT. pp. 397–408 (2019)

18. Fukunaga, K., Hummels, D.M.: Bayes error estimation using parzen and k-nn procedures. IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-9**(5), 634–643 (1987). https://doi.org/10.1109/TPAMI.1987.4767958
19. Granlund, T., Kopponen, A., Stirbu, V., Myllyaho, L., Mikkonen, T.: Mlops challenges in multi-organization setup: Experiences from two real-world cases. arXiv preprint arXiv:2103.08937 (2021)
20. Hardt, M., Chen, X., Cheng, X., Donini, M., Gelman, J., Gollaprolu, S., He, J., Larroy, P., Liu, X., McCarthy, N., et al.: Amazon sagemaker clarify: Machine learning bias detection and explainability in the cloud. arXiv preprint arXiv:2109.03285 (2021)
21. Karamitsos, I., Albarhami, S., Apostolopoulos, C.: Applying devops practices of continuous automation for machine learning. Information **11**(7) (2020). https://doi.org/10.3390/info11070363, `https://www.mdpi.com/2078-2489/11/7/363`
22. Karlaš, B., Interlandi, M., Renggli, C., Wu, W., Zhang, C., Mukunthu Iyappan Babu, D., Edwards, J., Lauren, C., Xu, A., Weimer, M.: Building continuous integration services for machine learning. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining. p. 2407–2415. KDD '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3394486.3403290, `https://doi.org/10.1145/3394486.3403290`
23. Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering–a systematic literature review. Information and software technology **51**(1), 7–15 (2009)
24. Klein, B.T., Giese, G., Lane, J., Miner, J.G., Jones, J.J., Venezuela, O.: An approach to devops and microservices. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2020)
25. Krishnan, S., Wang, J., Wu, E., Franklin, M.J., Goldberg, K.: Activeclean: Interactive data cleaning for statistical modeling. Proceedings of the VLDB Endowment **9**(12), 948–959 (2016)
26. Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., et al.: Elastic machine learning algorithms in amazon sagemaker. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 731–737 (2020)
27. Lwakatare, L.E., Raj, A., Bosch, J., Olsson, H.H., Crnkovic, I.: A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In: International Conference on Agile Software Development. pp. 227–243. Springer, Cham (2019)
28. Mäkinen, S., et al.: Designing an open-source cloud-native mlops pipeline (2021)
29. Mikkonen, T., Nurminen, J.K., Raatikainen, M., Fronza, I., Mäkitalo, N., Männistö, T.: Is machine learning software just software: A maintainability view. In: International Conference on Software Quality. pp. 94–105. Springer (2021)
30. Perrone, V., Shen, H., Zolic, A., Shcherbatyi, I., Ahmed, A., Bansal, T., Donini, M., Winkelmolen, F., Jenatton, R., Faddoul, J.B., et al.: Amazon sagemaker automatic model tuning: Scalable black-box optimization. arXiv preprint arXiv:2012.08489 (2020)
31. Renggli, C., Karlaš, B., Ding, B., Liu, F., Schawinski, K., Wu, W., Zhang, C.: Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment. arXiv preprint arXiv:1903.00278 (2019)
32. Renggli, C., Rimanic, L., Gürel, N.M., Karlaš, B., Wu, W., Zhang, C.: A data quality-driven view of mlops (2021)

33. Samek, W., Binder, A., Montavon, G., Lapuschkin, S., Müller, K.R.: Evaluating the visualization of what a deep neural network has learned. IEEE transactions on neural networks and learning systems **28**(11), 2660–2673 (2016)
34. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. Advances in neural information processing systems **28**, 2503–2511 (2015)
35. Shawi, R.E., Maher, M., Sakr, S.: Automated machine learning: State-of-the-art and open challenges. CoRR **abs/1906.02287** (2019), `http://arxiv.org/abs/1906.02287`
36. Tsymbal, A.: The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin **106**(2), 58 (2004)
37. W3Techs: internals-of-metaflow/technical-overview (2021), `https://docs.metaflow.org/internals-of-metaflow/technical-overview`, last accessed 16 December 2021
38. Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y.F., Tu, W.W., Yang, Q., Yu, Y.: Taking human out of learning applications: A survey on automated machine learning (2019)
39. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S.A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al.: Accelerating the machine learning lifecycle with mlflow. IEEE Data Eng. Bull. **41**(4), 39–45 (2018)
40. Zhao, S., Talasila, M., Jacobson, G., Borcea, C., Aftab, S.A., Murray, J.F.: Packaging and sharing machine learning models via the acumos ai open platform. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 841–846. IEEE (2018)
41. Zintgraf, L.M., Cohen, T.S., Adel, T., Welling, M.: Visualizing deep neural network decisions: Prediction difference analysis. arXiv preprint arXiv:1702.04595 (2017)
42. Zöller, M.A., Huber, M.F.: Benchmark and survey of automated machine learning frameworks (2021)