Practices and Infrastructures for ML Systems – An Interview Study

Dennis Muiruri, Lucy Ellen Lwakatare, Jukka K. Nurminen and Tommi Mikkonen
Department of Computer Science, University of Helsinki
Email: dennis.muiruri, lucy.lwakatare, jukka.k.nurminen, tommi.mikkonen @helsinki.fi,

***Abstract*—The best practices and infrastructures for developing and maintaining machine learning (ML) enabled software systems are often reported by large and experienced data-driven organizations. However, little is known about the state of practice across other organizations. Using interviews, we investigated practices and toolchains for ML-enabled systems from sixteen organisations in various domains. Our study makes three broad observations related to data management practices, monitoring practices and automation practices in ML model training, and serving workflows. These have limited number of generic practices and tools applicable across organizations in different domains.**

1. Introduction

Today, artificial intelligence (AI) techniques are incorporated in many real-world software systems and services. However, research on the development, deployment and maintenance of AI-enabled systems in industrial settings report this to be a challenging task [1, 2]. Large companies, like Google [3] and Facebook [4], often report their development and infrastructure practices for AI solutions that are useful for learning. However, many organizations are yet to adopt and tailor the suggested development practices and infrastructures to narrow the gap from mere prototyping to deploying to production AI solutions [5].

Machine learning (ML) is a subset of AI and its techniques involve the use of high-quality data. In ML logic is not explicitly programmed but is rather learned from data. The development of industrial ML-enabled software systems involves ML pipelines that consist of several interlocking steps. To support the different steps, end-to-end in one environment, ML platforms like TensorFlow Extended (TFX) [3], have been proposed to ensure increased automation across the steps.

Since industrial ML pipelines can be complex, it is important to gain an understanding of their characteristics. In a large data-intensive organization, like Google, all 3000 ML pipelines comprising of over 450,000 trained ML models continuously update the models at least seven times a day [6]. The need to support regular model training and updates in production is a common requirement in most industrial ML-enabled systems because, often, data is constantly being generated and the performance of models can deteriorate overtime [1].

Most empirical literature presents development and maintenance practices of ML-enabled systems from the perspective of a single, often large and experienced online organization. In contrast, we aim to provide empirical evidence of the practices and infrastructure setups across a diverse set of companies in various domains. Through interviews, this study investigated ML workflow practices and toolchains used in the development, deployment, and maintenance of ML-enabled systems in selected multiple organizations in Finland. Our main contributions include:

- Empirical evidence of common practices in ML workflows (Section 4)
- Tool adoption in ML pipelines (Section 5) and areas to address in future research (Section 6)

2. Background and related work

2.1. Software engineering for machine learning

Consideration and adaptation of well-established software engineering (SE) methods and approaches in ML systems have been reported to be crucial [7]. This perspective shifts the focus from just ML algorithms to also include other important aspects of ML model development and operations in production, such as data management and serving infrastructures [1]. Evidence of the integration between SE approaches and ML workflow is in MLOps (machine learning operations), a term used to show the extension of DevOps philosophy of increased agility and automation to the ML workflows [8]. In support of the latter, different tools are used to provide automation in ML workflows.

Best SE practices in ML are identified and their adoption in the industry is surveyed in [5]. The identified 29 SE practices in ML are classified into six categories: (1) data (e.g., employing sanity checks for all external data sources), (2) training (e.g., use versioning for data, model, configurations and training scripts), (3) coding (e.g., using continuous integration), (4) deployment (e.g., enabling shadow deployment), (5) team (e.g., collaborating with multidisciplinary team members), and (6) governance (enforcing fairness and privacy) [5]. According to the authors [5], the least adopted practices – related to feature management, writing tests, shadow deployment and automated hyper-parameter optimization – require effort, knowledge and tool support. This interview study provides some validation and in-depth interpretation to the survey findings related to the adoption of practices [5].

2.2. ML workflow and pipeline

ML workflows describe different tasks that are performed in order to develop, deploy and operate ML models in production [7]. ML pipelines are used to express the complex input/output relationship between the different tasks/operators of an automated ML workflow [6]. Generally, ML pipelines plug together several tools when automating the ML workflow [9].

Typical lifecycle phases of ML workflow include model requirements, data collection, data cleaning, data labelling, feature engineering, model training, model evaluation, model deployment and model monitoring [7]. Studies show that end-to-end automation of ML workflow improves both the development time and rate of deploying ML models [9, 6]. Furthermore, it allows organizations to (1) automate the orchestration of workflows steps, (2) track and reproduce the different outputs of ML workflow, and (3) reuse common steps of ML workflow across multiple ML-enabled systems [3, 9].

Few studies report in detail the characteristics of ML pipelines, in terms of their components and architectures [9, 6]. Different from our qualitative analysis, Xin et al [6] quantitatively analysed over 3000 ML pipelines at Google and presented their high-level characteristic in terms of pipeline lifespan, complexity and resource consumption. For the complexity of ML pipelines, the authors analyzed typical input data shape, feature transformation and model diversity. Model diversity showed that a large portion used neural networks (NN) (64%). The latter is informs the characteristic of ML pipeline since the choice of model type and architecture has an influence on ML pipeline steps. From the analysis, the authors [6] identified areas for optimizing the ML pipelines, that were mostly related to data management.

3. Research Method

We conducted an exploratory multiple-case study [10] between March and August 2021. The main research questions (RQ) of our study were:

- RQ1. What practices are applied in the development, deployment and maintenance of industrial ML-enabled software systems?
- RQ2. What tools are used to support the development, deployment and maintenance of industrial ML-enabled software systems?

3.1. Research design and case selection

The main goal of our study is to understand the state-of-practice of ML-enabled systems' development and toolchain within the Finnish context. In this study, a case (Table 1) is an organization in Finland with experience in developing, deploying and maintaining ML-enabled software systems. The main criterion for case selection is that the ML-enabled software system needs to be operational in a production environment.

We first identified relevant practitioners from different organizations to take part in the study and adapted an interview guide used in an earlier study [2]. The identified practitioners (or their organizations) were primarily known to be working on ML solutions by the researchers and others were gathered from LinkedIn.

We reached out to 37 organizations via e-mail out of which 16 agreed to participate in the study. Generally, practitioners were free to choose whether (and who) to participate in the study, but researchers purposefully ensured that the organizations were varied in terms of sector and size.

Interviewed practitioners had varying roles: Chief Machine Learning Engineer (2), Chief Scientific Officer, Head of Natural Language Understanding, Machine Learning Engineer (Founder), Solution Architect(2), Director, Data Science Manager, Chief Architect, Data Scientist (5), AI Specialist, Director of Consulting, Machine Learning Engineer, Computational Biologist, AI Engineer, Chief Data Architect, Principle Data Scientist. Academically, ten (43%) practitioners hold a PhD degree, ten (43%) hold a Master's degree and two (9%) hold a Bachelor's degree. Ten cases are large organizations and five constitute small or medium-sized organizations with revenues below €50 million based on 2020 financial reports (https://bit.ly/3BtbFgN).

3.2. Data collection

Research data was primarily collected through semi-structured interviews conducted by two researchers. A total of 23 practitioners from the 16 organizations were interviewed between 24^{th} March and 27^{th} April 2021. All interviews were conducted virtually due to COVID-19. Each interview session took on average 80 minutes.

During the interview session, the researchers presented details of the study and requested consent to record the interview. One researcher asked the question outlined in the interview guide that contained five broad categories: data management, model training, model deployment, model monitoring and general challenges. The interview guide was not strictly followed to allow probing questions depending on the interviewees' responses and their experience with the topic. All recorded interviews were automatically transcribed using Otter.ai and errors in the transcriptions were manually corrected by the researchers.

3.3. Data analysis

Analysis of the interview transcripts mainly consisted of two coding steps and a session to discuss and harmonize the codes [10]. A deductive approach formed the first stage of our coding process in which main themes informed by the structure of our interviews were outlined. The themes were constituted high-level codes organized into six groups: role and responsibility, organisation, ML usecase, Practices, Challenges and Tools. The actual coding of data was done in an iterative manner using both deductive and inductive [10] approaches within each group and applied broadly at a paragraph or statement level. The sub-groups were further refined during researcher meetings.

4. Practices in ML workflow (RQ1)

4.1. Data management

Data collection and storage typically begins either by batch loading data from internal systems, streaming from devices/sensors, extracting from other third-party vendors through APIs or from open-source repositories. The training data is then commonly stored in cloud platforms, as shown from data sources in Table 1.

Organizations often use cloud storage providers with data centres either in Finland, close proximity to Finland or within the European Union geographic area following customer preferences or regulatory constraints. In very strict circumstances, data is stored in private infrastructure.

Additionally, low-level metrics, such as IOPS (I/O Operations Per Second), are considered when choosing a storage architecture as data fetching can constitute a sizeable amount of the overall model training time. Case E uses mounted discs solution instead of a network drive accessed via a web interface.

Table 1. Summary of ML usecase, frameworks, data sources and storage platforms across cases. (*of the ML usecase)

Case	ML usecase (Type)	Domain*	Data Source	Storage	ML-Framework
A	Object Detection (NN)	IoT	Camera	Google Cloud	Tensorflow
B	Form data extraction (NN)	Finance	Internal Systems	AWS	Tensorflow
C	Form data extraction (NN)	Public Services	Internal Systems	On-premise	Tensorflow, PyTorch
D	Transcription (NN)	Healthcare	Internal Systems	Google Cloud	Kaldi ASR framework
E	Speech based UI (NN)	E-Commerce	Open-source, End-users	Google Cloud	PyTorch
F	MLOps (NN, Non-NN)	IT Services	Camera	-	Multiple frameworks
G	MLOps (NN, Non-NN)	Energy	Internal systems	AWS	Tensorflow, Scikit-Learn
H	Risk management (Non-NN)	Finance	Internal systems	AWS	Scikit-Learn, Heuristics
I	Predictive maintenance (Non-NN)	Engineering	Sensor and Technicians	AWS	Spark Analytics, Heuristics/Rules
J	Predictive maintenance (Non-NN)	Biochemicals	Sensor	Azure Cloud	-
K	Anomaly detection (Non-NN)	Real Estate	Meters	Azure Cloud	Scikit-learn, XGBoost
L	Data analysis (Non-NN)	Pharmaceutical	Device, Genome	Azure Cloud	R
M	Report/Document classification (Non-NN)	Healthcare	Internal systems	Azure Cloud	PyTorch, Scikit-learn (Classification)
N	Chatbots, Profiling (NN)	Finance	Internal systems	AWS	Watson(IBM), Tensorflow
O	ML pipeline automation (Non-NN)	IT services	Internal Systems	Azure Cloud	-
P	Marketing/campaign management (NN)	Media	Internal systems	AWS	Scikit-learn, Tensorflow, fastText

Data storage formats are also important architectural decisions when considering scalability of data processing pipelines, portability of data between computing environments, support of different ML frameworks and programming languages. In this regard, two data storage formats are presented: the Apache Parquet (https://bit.ly/3kGFaVI) and the NetCDF (https://bit.ly/3zy287R) file formats.

Case H uses Apache Parquet in favour of CSV (Comma Separated Values) or TSV (Tab Separated Values) file formats commonly used to store structured data for analytics purposes. Case G uses NetCDF as a solution to implement a generic data interface to abstract data across ML frameworks and computing platforms. Data scientists then ensure models can process NetCDF input and produce NetCDF output.

Data discoverability and accessibility affects the rate of experimentation and development of ML features. Discoverability tends to be a concern in setups that feature a data lake or where different types of data are collected. Case O describes a solution to this problem based on maintaining a *data catalogue* where data and its value are described.

Data access is a concern whenever an organization handles personal data or requires collaboration with third parties e.g. in consultancy settings. The process to grant data access can be lengthy and can result in data governance anti-patterns.

Data validation techniques are commonly applied as a means of controlling data quality. However, data types influence the type of validation approaches applied. Image/video, speech and text tend to require human actors supported by custom tools to validate and ensure data meets aspired quality thresholds. E.g, in an object detection setting, a human validator ensures that objects fall within the annotated bounding boxes. With speech, validation ensures that recorded utterances are coherent and consistent with corresponding text. Case D uses additional heuristics for detecting anomalies between generated texts and submitted utterances. Numerical data types normally easier to automatically validate.

Data validation in Case O is done at a schema and data level. The schema is maintained by dedicated *data stewards* team that ensures the schema reflects the required data. Delegating quality control ensures a team managing the data lake ingests data indiscriminately. When data is sourced from third-party vendors, the vendor is expected to maintain quality controls (Case P).

Data integrity controls ensure data is not changed unexpectedly. Case D and F apply hashing as part of data processing pipelines this ensures training data is verifiable and traceable with respect to a model's lineage. Additionally, this practice ensures that attempts to overwrite data are flagged appropriately.

Generally, when hashing is not a suitable approach for example when dealing with image files, other custom tooling and heuristics are used to perform anomaly detection, Cases B and I make use of this approach.

Data labelling and annotation tends to be undertaken manually using custom tools developed to standardize the process. Inconsistent labels are time to time encountered due to subjective interpretations therefore resulting in poor data quality. To overcome such issues Case B implements a standardized way of normalizing and giving common meaning to concepts.

Data understanding requires domain knowledge for teams to generate valuable insights from data in specialised domains. Domain knowledge is cited as a necessity in the entire life cycle of the data. E.g when handling data from chemical processes or mechanical parts of large systems represented in cases I, J and L.

In general, challenges in data management practices are mainly attributed to data quality aspects. E.g. sensor problems, inconsistent labelling, programming errors in data handling software etc.

4.2. Model training

ML algorithm selection and transfer learning are commonly occurring practices. Selection of ML algorithms is largely influenced by training data type and formulation of the learning problem during requirement elicitation. Heuristics are used in cases H and I to complement ML algorithms, in both cases, an explicable decision based on heuristic algorithms is highly regarded compared to an ML solution with high accuracy but largely inexplicable. The trade-offs arise either due to regulatory constraints or where a heuristic based approach provides a much simpler solution compared to a complex ML with a closely similar result.

Transfer learning is indicated as the main approach to train NN efficiently since model parameters can take a long time to converge and require significant computing resources. Transfer learning is based either on publicly available models or proprietary models.

Computer vision systems in cases A and F make use of transfer learning by applying state of the art models available on a wide range of CNN architectures. Case M's NLP solution was also trained using transfer learning mainly to overcome data insufficiency challenges. Case B applies transfer learning based on proprietary models as a cost management strategy.

Training NN without using transfer learning can also be motivated by several factors we observe in case D and E. (1) The amount of data is considered sufficient for training a model to convergence. (2) Availability of required computing resources. (3) Limited availability of relevant open-source models in a domain.

ML frameworks used across the cases can be broadly categorized as either Neural Network (NN) or classical (non-NN) ML solutions. Tensorflow (https://bit.ly/38sgWc4) and PyTorch (https://bit.ly/3gMHnxG) are the two commonly used frameworks for developing DL models as summarised in Table 1. Practitioners who used TensorFlow tended to make use of the Keras (https://keras.io/) framework which abstracts the low-level syntax found in the native TensorFlow framework.

Although NN frameworks provide similar core features, a few factors can affect the choice of framework. (1) a framework's usability, (2) a framework's underlying efficiency in utilizing computing resources, (3) a framework's flexibility. A case in point, cases D and E develop an ASR solutions but make use of Kaldi and PyTorch frameworks respectively. Frameworks can mature into certain domains much later and therefore teams might seemingly use different frameworks out of such historical reasons. Specialised analytics frameworks such as Spark (https://bit.ly/3gLVtQ7) also feature in case I. We generally note that team members freely adopt frameworks suitable for accomplishing tasks efficiently.

Overall, challenges in model training relate to infrastructure costs, complexities of tuning and identifying explainable factors about a model's performance.

4.3. Model evaluation and experiment management

Model training is considerably an iterative process involving (1) determining suitability of data and algorithms, (2) parameter and hyperparameter optimization and (3) model evaluation. Managing metadata from these experiments makes the training process traceable and reproducible.

We note three unique approaches used to evaluate models: (1) Data is stored such that it can be stratified by quality allowing composition of training and validation data to include different quality (Cases D and E), (2) Use of ensemble of models each trained on a unique subset of the data (Case B) and (3) Use of a configurable inference algorithm where each configuration makes use of a unique adaptation of the model (Case E).

To manage model evaluation results from these kinds of setups, case organizations either use dedicated experiment tracking tools case (G, I, N, O and P), logging process metadata (case B, E, F) or generating hashes (case D). These approaches are summarised in Table 2.

Case D uses hashing such that a hash is computed from a given version of the data combined with a model's parameters. The resulting hash is stored for later reference. Obtaining a previously stored hash implies that a model if similar characteristics already exists.

Case E and F utilize the generation and collection of metadata which includes metadata collected from tools such as git hashes. Case E stores metadata in a data warehouse which can be queried to produce spreadsheets reports. Case F's platform generates metadata at each step of the pipeline and resulting data is visualized on a web tool. Systematic management of experiments facilitates workflow automation.

4.4. Model monitoring

Training data drift. Data level monitoring is instrumented to check for drift within the data. Identifying drift in numeric data types is be achieved by using visual tools such as graphs or descriptive statistics (cases G, H, I), image-based data makes use of histograms (case F). Speech and text based data is also susceptible to drift but can be more challenging to monitor. For example, case D mentioned the emergence of the word Covid-19 in the medical sphere recently but the word is not available in any historical corpus. Typically, heuristics are used to monitor drift in these speech or NLP settings.

Model drift Model monitoring involves ensuring a model's accuracy and error metrics are maintained at a certain threshold. Observing key metrics such as accuracy and error rates were observed as the common ways production models are monitored. For example, in a transcription setting, measuring the character and word edits required after inference were used (Case D) as error metrics used to monitor already deployed models and to characterise any drift in the model.

Infrastructure utilization Infrastructure monitoring is commonly applied to ensure resources (GPU/CPU, memory, disk, network, etc.) are utilized reasonably or to flag unnoticed technical problems such as scaling designs, IO bottlenecks and how inference endpoints utilize backend resources. Inference endpoints often require further low-level monitoring to characterize the latency of the solution. Cases D, E and G were particularly keen to monitor endpoint latency as this formed an important requirement of the ML solution.

5. Tools in ML pipelines (RQ2)

5.1. Version management

ML training code, often written in notebooks, as well as other project artifacts are version controlled using tools like Git, Gitlab and Bitbucket. Data versioning is done by generating and versioning metadata or using specialized tools such as DVC. For many cases, training of ML models generally happens in a public cloud or on-premise servers. To quickly and consistently provision the execution environment for the training workflows, 'infrastructure-as-code' practices, using tools like Terraform (https://bit.ly/2WIt724), are adopted in Cases A and E. Serving of inferences based on the trained models is either done in a batch format or online.

5.2. ML training workflow

We observe that cases either containerize individual workflow steps or encapsulate all workflow steps to run in a single container where the former is the preferred setup. Docker is the main applied container technology. Containerization is

Case	Version Management	Container Platform	ML Training Workflow	ML Experiment Tracking	Model Repository	ML Deployment, Serving	Monitoring
A	Github, Gitlab, Bitbucket	Kubernetes	Apache Airflow	Tensorboard	GC Container Registry	Embedded with over the air updates	Logging, Grafana
B	Git	Kubernetes	Custom	Metadata	-	API endpoint using Bamboo	-
C	Github	OpenShift	Custom	None	Nexus, S3 object storage	REST API endpoint on OpenShift,	(Prometheus, Grafana)*
D	Git	Kubernetes	Argo	Hashing	-	API	Prometheus, Grafana
E	GitHub	Kubernetes	Custom	Metadata	PostgreSQL	-	Prometheus, Grafana
F	-	Kubernetes	-	Metadata	-	natively supports REST API endpoint on Kubernetes	Logging, Elastic Search, tool's Web UI
G	GitHub	Kubernetes	Custom, Metaflow	MLflow,	Docker registry	gRPC API endpoint on Kubernetes, Kafka	-
H	Git	AWS Elastic Container	Apache Airflow	-	S3 storage	-	AWS CloudWatch, Splunk
I	GitLab	-	AWS SageMaker	AWS SageMaker		-	
J	-	Kubernetes	AzureML	-	Azure container registry	Edge Server	Azure Monitor
K	Git	-	Azure ML		-	Streamlit	-
L	-	-	Azure ML	-	-	R-Shiny apps	-
M	-	-	-	-	Nexus repository	Batch prediction	-
N	Git	AWS lamda	AWS SageMaker	AWS SageMaker	S3 storage, Databricks model registry*	Batch prediction, Java apps	AWS CloudWatch
O	Git, DVC	Kubernetes	Apache Airflow, Kubeflow	MLflow, kubeflow	MLFlow model registry	REST API on Kubernetes	Logging, Grafana
P	-	-	Databricks	MLflow	S3, MLFlow model registry	API, batch, embedded	-

Table 2. Tools (*planned, - tool information not provided)

appealing because (1) it allows decoupling from the execution environment, (2) different workflow tasks/steps can be isolated and (3) it makes the workflow traceable and reproducible. Data transfers across workflow steps during training is done using standard persistent volumes. However, large datasets may require using network mounts (Case F). We further note that containers are commonly orchestrated using Kubernetes which allows model training to be executed in any environment that supports Kubernetes whether on-premise or in a public cloud environment.

Model building steps are managed using a configuration tool (e.g YAML based) or a workflow toolkit to depict various workflow tasks/steps. A workflow may include steps specifying feature extraction, model training and validation. The complexity involved in these steps can vary depending on the ML domain. For complex ML models, low-level ML training workflow frameworks such as Argo (Case D) and Metaflow (Case G) are preferred mainly because of the tool's flexibility. We note that although high-level ML workflow platforms such as AWS SageMaker are available in some organizations, such platforms were not preferred when developing complex models (Cases B, G).

Proponents of dedicated ML training workflow tools prefer the end-to-end integration provided by such tools while those in support of custom tooling prefer the ability to add different tools to the workflow. Typically, when a single task contains multiple containers, custom solutions involve implementing components or agents that provide an interface to a container for accessing data and compute resources during training (Cases C, G). In addition, components such as explainer dashboard (https://bit.ly/3Byj1js) (Case A) used to facilitate a model's explainability can be added as part of the workflow.

After adding model training workflow tools, data scientists can orchestrate event based training queues e.g., based on continuous arrival of training data. Tools like Apache airflow provide the functionality to schedule model training based on certain triggers.

To track model experiments, custom tools such as web-based UI tools in Cases B and F are developed to facilitate evaluation of results and compare model predictions with ground truth. To their advantage, custom platforms can include any data the team deems important (Case F). In other cases, plugins can be developed to integrate with existing open-source solutions like MLflow (Case G). Low-level training metrics are observed with Tensorboard (Case A).

5.3. Continuous integration and testing

Continuous integration or build tools, such as Jenkins, are used to run tests and build docker images based on model artifacts from the training workflow (Cases A, D, G). Static code analysis and other tests to check whether a container works are performed when building the container images (Case A, G). In addition, other domain-specific tests are executed to make certain that the inputs and outputs of the model are still correct, thus extending the tests to the whole pipeline by performing tests on small amounts of input data (Case D, F, G). Docker images created from the CI system are (automatically) deployed to another (test/staging) environment for additional tests prior to deployment. For Case C, the latter environment contains a copy of production data which due to restrictions was not accessible in other environments. Prior to deployment, the tests are performed to verify the type of data that the model API accepts (Case A, D), the models make predictions (Case E) and ensure that the deployment procedure loads a serialized model into the relevant API.

Trained models are stored in different ways, including general data storage and container image registries. For the trained models, Case E stores the model and metadata about the model, including name and version number in a PostgreSQL data warehouse. Built container images of trained models from the build system or test environment are uploaded to Nexus in Case C, and to the docker registry in Case G prior to production deployment.

5.4. ML deployment and serving

In many cases, trained models are deployed as REST (Representational State Transfer) APIs on public cloud or on-premise servers. Other deployment targets include embedding the model on the actual application such as a mobile application (Case P) or deploying to IoT devices through over-the-air deployments.

For applications with strict inference latency requirements, a gRPC (Google Remote Procedure Call) based API is used in Case C because it supports streaming interfaces. For ML solutions that need to serve inference in (or near) real-time, different strategies are continuously evaluated both at models level and services level (Case E, G).

Most cases implement custom serving infrastructure although emerging model serving systems like KFServing (https://bit.ly/3DBF1eZ) and Seldon (https://bit.ly/2Ybwuix) are being tested in Cases C and O respectively.

Finally we note that deployment is often not performed by data scientists but by other dedicated team members or platform teams (Case G) because it requires considerably low-level knowledge of Kubernetes.

5.5. Monitoring

After model deployment, monitoring is performed at different levels of granularity. Most common monitoring is done for infrastructure where logging, monitoring and alerting services and tools like Prometheuse and Kubernetes logging (stackdriver) are used to collect a cluster's performance metrics which are visualised on dashboards using tools like Grafana, Tableau or any other business analytics tools. For models deployed as API, model logging (e.g., model predictions) integration with services like Elasticsearch and BigQuery can be used to perform model health and quality checks in production e.g., average accuracy on sampled log (Cases A, F and O).

Maintaining the collective set tools used across a pipeline can develop into a complex task especially when dealing with NN architectures.

Generally, we observe that practices vary across organizations based on factors such as the type of data being used for training, availability of computing resources and type of ML solution being developed. We also note there are two primary ways case organizations develop their pipelines, (1) compose a variety of tools to orchestrate the pipeline or (2) apply a more encompassing framework such as Sagemaker which contains inbuilt tools for different parts of a pipeline. We note that most teams prefer flexibility and the ability to extract low level information provided by independent tools, i.e. the first approach. While the few teams that use the second approach

Table 3. Summary of practices and challenges

	Practices	**Challenges**
ML workflow		
Data management	• Batch or stream data loads largely from internal systems, third party vendors or devices and sensors • Training data stored often in cloud platform or in close proximity to computing units (CPUs or GPUs) • Selecting data storage formats (e.g., Apache Parquet) with great consideration of scalability, portability, ML frameworks • Data documentation (e.g., data catalogue) for fast data identification • Employing data validation approaches (e.g., descriptive statistics and schema) that are tailored to the types of data • Maintaining data quality by a dedicated team or third party vendor • Determining data quality metrics from domain knowledge especially in highly specialized settings	• Determining ownership of data quality aspects especially in large organizations or when data collection is outsourced • IoT related factors such as sensor outage, network latency or low traffic priority, sensor quality etc. • Programming defaults in data collection components can lead to poor data quality through subtle hard to notice errors. • Lack of standardized annotation formats across DL networks especially in computer vision reduces interoperability across network architectures.
Model training & evaluation	• Selecting ML algorithm based on available data and learning problem formulation during requirement elicitation and exploratory experiments • Using heuristics to compliment or over ML algorithm when constrained by regulations or the complexity of models • Employing transfer learning to effectively and accurately train DL models. • Flexibility to choose standard ML frameworks e.g. Tensorflow and PyTorch as popular in DL, and Scikit-Learn and XGBoost in non-DL • Using ML frameworks that offer great flexibility, efficiency and usability • Employing multiple approaches to evaluate quality of ML models e.g., using validation dataset stratified by quality • Managing and tracking model evaluation results using experiment tracking tools, or metadata and hash-based approaches.	• The cost of training deep learning models from a clean start can be prohibitively high • Determining model explainability • Feature extraction and hyper parameter tuning can be a time consuming activity especially in organization with different types of data. • Model benchmarking was highlighted as an inherently difficult task given that it is challenging to replicate publicly available state of the art models and related results.
Model Deployment & Monitoring	• Inference serving through REST based API endpoints deployed in public cloud environments • Inference serving with strict latency requirements through gRPC endpoints as opposed to REST endpoints. • Model deployment for either batch inference or online inference purposes • Monitoring at different parts of the pipeline, to ensure data quality, model quality and performance and infrastructure utilization	• Deploying models within organizations that do not use the cloud environment can be a lengthy process due to relevant data governance protocols. • Monitoring model or data drift in deployed systems can be a challenge due to lack of visibility especially in scenarios where input data cannot be saved due to GDPR related constraints.
ML Pipeline	• Version control code and all pipeline related artifacts e.g. in git, and provision execution environment using infrastructure-as-code frameworks e.g. Terraform • Encapsulating ML training workflows in docker containers to increase portability • Using common container orchestration platforms e.g., Kubernetes to build scalable containerised pipelines • Using ML workflow automation tools e.g. Argo and kubeflow to execute schedule ML training pipelines and queues • Tracking ML training experiments largely in custom ways e.g. hashing and custom web tools but also with ML workflow automation tools. • Employing continuous integration tools e.g., Jenkins to test and build docker images prior to deployment	• Maintaining an up to date stack of tools frameworks requires rigorous testing to avoid regression errors and dependency breaks across tool chains. • Pipelines can become quite complex especially when dealing with complex DL architectures where multiple models are maintained. • Skills required to run end-to-end automated ML pipelines are not easily available.

prefer the instant integration of tools provided by their cloud provider. We summarize all general practices in Table 3.

6. Discussion and conclusion

In relation to existing literature, our findings provide explanations to the low adoption rates of the important best practices in ML [5]. In addition, compared to experienced online data-intensive organizations, our study show that majority of organizations have not achieved the capability required to enable continuous deployment and operations of ML models in production. However, there is great awareness of these amongst the practitioners as evidenced by their continued efforts to improve their practices and tools. We make the following observations concerning the current state of practice in many organizations. These areas would greatly benefit from future research contributions.

First, streamlining data handling practices in order to get good quality data is an arduous task given increasing amounts of data, challenging data types and widening regulatory oversight around personal data. As a result, data handling workflows tend to induce long ML production cycles. Establishing efficient data handling procedures would shorten ML production cycles and increase experimentation for R&D purposes.

Second, maintaining a ML system involves monitoring and evaluating procedures to ensure high availability and to control for model staleness respectively. Data distributions may change over time meaning that models may have an implicit viability lifespan. Due to the availability of tools, teams tend to focus on monitoring infrastructure while model evaluation tends to be conducted on a case by case basis. Issues such as model explainability, control for bias, feature attribution continues to be an open challenges from a tooling perspective and developing practice.

Third, we observe that a significant number of organizations use a wide collection of tools to support their ML development as opposed to using an integrated MLOps platform. This is mainly due to desired flexibility and access to low-level features when necessary. Some topical areas with defacto tooling include version management, containerization and monitoring. Other areas continue to have multiple tooling options. We further note that there are industry wide efforts to standardize model serving through open development of a serving API which is realized by products such as NVIDIA's Triton Inference Server (https://bit.ly/3Brm3Wv).

A main threat to the validity of our study relates to external validity and generalization of the findings. These were mitigate by the inclusion of diverse set of organizations from multiple domains and sizes. Most of the participating companies were global organizations While we cannot generalize the findings across the entire population, they do provide insight into the state-of-practice of ML especially in contexts having similar organisational profiles.

7. Biographies

Dennis Muiruri is a member of the empirical software engineering research group at the University of Helsinki. His current research interests include deployment and operations of machine learning systems.

Lucy Ellen Lwakatare is a postdoc at the University of Helsinki. She received her PhD from University of Oulu in 2013. Her research interests include: Agile, DevOps and ML engineering.

Jukka K. Nurminen is a professor at the University of Helsinki. He has worked extensively on software research in industry at Nokia Research Center, in academia at Aalto University, and in applied research at VTT. He received PhD from Helsinki University of Technology (now Aalto University). His main interests are on tools and techniques for developing data-intensive software systems including testing of AI solutions, computational moral and software development for quantum computers.

Tommi Mikkonen received his Ph.D. from Tampere University of Technology year 1999. He is a full professor in University of Helsinki of the empirical software engineering research group. His research interests include IoT, software architectures, and software engineering for AI.

REFERENCES

1. D. Sculley et al, "Hidden technical debt in machine learning systems," in *Advances in neural information processing systems*. Curran Associates Inc., 2015, pp. 2503–2511.
2. L. E. Lwakatare et al, "A taxonomy of software engineering challenges for machine learning systems: An empirical investigation," in *Extreme Programming Conference*. Springer, 2019, pp. 227–243.
3. D. Baylor et al, "Tfx: A tensorflow-based production-scale machine learning platform," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1387–1395.
4. K. Hazelwood et al, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *International Symposium on High Performance Computer Architecture*. IEEE, 2018, pp. 620–629.
5. A. Serban et al, "Adoption and effects of software engineering best practices in machine learning," in *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2020.
6. D. Xin et al, "Production machine learning pipelines: Empirical analysis and optimization opportunities," in *International Conference on Management of Data*. ACM, 2021, p. 2639–2652.
7. S. Amershi et al, "Software engineering for machine learning: A case study," in *International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2019, pp. 291–300.
8. Y. Zhou, Y. Yu, and B. Ding, "Towards mlops: A case study of ml pipeline platform," in *International Conference on Artificial Intelligence and Computer Engineering*, 2020, pp. 494–500.
9. W. Hummer et al, "Modelops: Cloud-based lifecycle management for reliable and trusted ai," in *International Conference on Cloud Engineering*, 2019, pp. 113–120.
10. P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, 2008.