Practices and Infrastructures for ML Systems – An Interview Study in Finnish Organizations

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

Abstract—
Using interviews, we investigated the practices and toolchains for machine learning (ML)-enabled systems from 16 organizations across various domains in Finland. We observed some well- established artificial intelligence engineering approaches, but practices and tools are still needed for the testing and monitoring of ML-enabled systems.

1. Introduction

Today, artificial intelligence (AI) is 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 practices and infrastructure 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. 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, understanding their characteristics is essential. In a large organization like Google, some 3000 ML pipelines comprising 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 the performance of models deteriorates over time [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 that are used in the development, deployment, and maintenance of ML-enabled systems in selected organizations in Finland. Our main contributions include:

- Empirical evidence of the enacted practices in ML workflows (Section 4),
- Tool adoption in ML pipelines (Section 5) and

areas of future research (Section 6).

2. Background and related work

2.1. Software engineering (SE) for ML

Adaptation and incorporation of well-established SE methods in the development of ML systems are crucial [7]because they emphasize other important engineering aspects beyond ML algorithms [1]. With the practices, organizations can address several challenges reported at the different stages of the taxonomy that depict evolution of ML components in software-intensive systems (experimentation, non-critical deployment, critical deployment, cascading deployment, and autonomous ML components) [2].

Serban and van der Blom [5] developed a catalog of 29 SE practices for ML applications based on literature and later measured their adoption rate through a survey with 313 practitioners. The catalog includes SE practices about data (e.g., employing sanity checks for all external data sources), training (e.g., use versioning for data, model, and training scripts), coding (e.g., using continuous integration), deployment (e.g., automate model deployment), team (e.g., collaborating with multidisciplinary team members), and governance (enforcing fairness and privacy). Compared to our study, some of the reported SE practices are too general. The lack of details gives room to multiple interpretations, for example, the named practice to use continuous integration. In addition, we study practice enactment also by investigating the toolchains, for which the authors [5] had only speculated to influence the adoption rate of specific practices.

2.2. ML workflow and pipelines

ML workflows describe different tasks performed to develop, deploy, and operate ML models [7]. ML pipelines express complex input/output relationships between different tasks/operators of an automated ML workflow [6]. Generally, ML pipelines plug together several tools to automate ML workflows [8].

A typical ML workflow life cycle includes model requirements, data collection, cleaning, labeling, feature engineering, model training, evaluation, deployment, and monitoring [7]. Studies show that end-to-end automation of ML workflows improves ML models' quality, traceability, development time, and deployment rate [6, 8]. Furthermore, it allows organizations to reuse common workflow steps across multiple ML systems [3, 8].

Few studies report the characteristics of ML pipelines in terms of their components, architectures, and tools [8, 6]. Unlike our qualitative analysis, Xin et al [6] quantitatively analyzed over 3000 ML pipelines at Google and presented their high-level characteristic concerning 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 was neural networks (NN) (64%) and model type and architecture influence the characteristics of the resulting ML pipelines. The authors [6] identified data management-related areas as key for optimizing ML pipelines.

3. Methodology

An exploratory multiple-case study [9] was conducted between March and August 2021. The research method was selected to gain a deep understanding of the enacted practices and tool support for ML systems in real-world settings. The main research questions (RQ) include:

- 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 must be operational in a production environment.

We first identified relevant practitioners from different organizations and adapted an interview guide used in earlier research [2]. For the current study, we modified questions under the project background and characteristics section of the interview guide to only inquire about operational ML systems rather than practitioners' general experience in ML projects. This was done to exclude ML systems in the experimental stage because they often have immature practices and toolchains [2]. In addition, we added new questions to the interview guide that inquired about the infrastructure and tools used. 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, but researchers purposefully ensured that the organizations were varied in terms of sector and size. Interviewed practitioners had varying roles, as seen in Table 1. Academically, ten (43%) practitioners hold a Ph.D. degree, ten (43%) hold a Master's degree, and two (9%) have 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.

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 expertise. All recorded interviews were automatically transcribed using Otter.ai, and the researchers manually corrected errors in the transcriptions.

3.3. Data analysis

Analysis of the interview transcripts mainly consisted of two coding steps and a session to discuss and harmonize the codes [9]. 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 constituted the high-level codes in our analysis, and these included: role and responsibility, organization, ML usecase, Practices, Challenges, and Tools. The actual coding of data was done in an iterative manner using both deductive and inductive [9] 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 are handled in various ways: batch loading data from internal systems, streaming from devices/sensors, extracting from third-party vendors APIs or open-source repositories. The training data is then commonly stored in cloud platforms, as shown from data sources in Table 1.

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

Data storage formats are factored in when considering the scalability of data processing pipelines, data portability between computing environments, and support of different ML frameworks. Case H uses Apache Parquet instead of CSV (Comma Separated Values) or TSV (Tab Separated Values) file formats commonly used to store structured data for analytics purposes. Case G uses network common data form (NetCDF) to implement a generic data interface to abstract data across ML frameworks and computing platforms.

Data discoverability and accessibility is emphasized in setups that feature a data lake or where different data types are collected. Case O describes a solution to this problem based on maintaining a *data catalog* where data and its

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

(Cases often use cloud storage providers with data centers in Finland or within the European Union, following customer preferences or regulatory constraints)

GCP: Google Cloup Platform, AWS: Amazon Web Services, AC: Azure Cloud

Case	Interviewee Role	ML usecase (Type)	Domain*	Data Source	Storage	ML-Framework
A	Chief ML Engineer, Founder	Object Detection (NN)	IoT	Camera	GCP	Tensorflow
B	Chief ML Eng.	Form data extraction (NN)	Finance	Internal Systems	AWS	Tensorflow
C	Architects (2), data scientists (2)	Form data extraction (NN)	Public Services	Internal Systems	On-premise	Tensorflow, PyTorch
D	Chief Scientific Officer	Transcription (NN)	Healthcare	Internal Systems	GCP	Kaldi ASR framework
E	Head of Natural Language Understanding	Speech based UI (NN)	E-Commerce	Open-source, End-users	GCP	PyTorch
F	ML Eng, Founder	ML Operations (NN, Non-NN)	IT Services	Camera	-	Multiple frameworks
G	ML Eng.	ML Operations (NN, Non-NN)	Energy	Internal systems	AWS	Tensorflow, Scikit-Learn
H	Solution Architect	Risk management (Non-NN)	Finance	Internal systems	AWS	Scikit-Learn, Heuristics
I	Data Science Mgr	Predictive maintenance (Non-NN)	Engineering	Sensor and Technicians	AWS	Spark Analytics, Heuristics/Rules
J	Data Architect	Predictive maintenance (Non-NN)	Biochemicals	Sensor	AC	-
K	Data Scientist	Anomaly detection (Non-NN)	Real Estate	Meters	AC	Scikit-learn, XGBoost
L	Computational Biologist	Data analysis (Non-NN)	Pharmaceutical	Device, Genome	AC	R
M	Data scientists (2), Director of Consulting Business	Report/Document classification (Non-NN)	Healthcare	Internal systems	AC	PyTorch, Scikit-learn (Classification)
N	Principal Data Scientist	Chatbots, Profiling (NN)	Finance	Internal systems	AWS	Watson(IBM), Tensorflow
O	Solution Architects (2)	ML pipeline automation (Non-NN)	IT services	Internal Systems	AC	-
P	Data Scientist	Marketing/campaign management (NN)	Media	Internal systems	AWS	Scikit-learn, Tensorflow, fastText

value are described. Data governance and related processes can limit the use and scope of data accessible for ML purposes.

Data validation techniques are commonly applied as a means of controlling data quality. However, data types influence the type of validation approaches used. Validation of Image/video, speech, and text tend to require human actors supported by custom tools. For example, a human validator ensures that objects fall within the annotated bounding boxes in an object detection setting. A human speech validator ensures 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 are usually easier to validate automatically.

Data validation in Case O is done at a schema and data level where dedicated data stewards maintain the schema. Delegating quality control ensures the 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 their 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 tend to be undertaken manually using custom tools developed to standardize the process. Inconsistent labels are sometimes encountered due to subjective interpretations, resulting in poor data quality. Case B implements a standardized way of normalizing and giving common meaning to concepts to overcome such issues.

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

In general, challenges in data management practices are mainly attributed to data quality aspects. For example, sensor related problems, inconsistent labeling, programming errors in data handling software, etc.

4.2. Model training

ML algorithm selection and transfer learning The choice of ML algorithms is influenced by training data type and formulation of the learning problem during requirements elicitation. Heuristics are used in cases H and I to complement ML algorithms; in both cases, an explainable decision based on heuristics is preferred compared to an ML solution with high prediction accuracy but inexplicable. The ML-heuristic trade-off tends to arise due to business sector regulatory constraints.

Transfer learning is typically used to train large NN efficiently, for example, in speech recognition and computer vision settings. This is mainly because model convergence can be a prolonged process that requires significant computing resources. Transfer learning is based on publicly available models or proprietary models.

In cases A and F, computer vision systems utilize transfer learning to test different Convolutional Neural Networks architectures. Case M's Natural Language Processing (NLP) solution is trained using transfer learning to overcome data insufficiency challenges. Case B applies transfer learning based on proprietary models as a cost management strategy.

Training NN without transfer learning can be motivated by two factors observed in cases D and E. One, there is sufficient data and computing resources for training a model to convergence. Two, there is limited availability of suitable open-source models.

ML frameworks used across the cases can be broadly categorized as either NN framework or classical (non-NN) frameworks. Tensorflow-Keras and PyTorch are the two commonly used frameworks for developing NN models, as summarized in Table 1.

Although ML frameworks may provide similar core features, the choice of the framework can be based on a framework's usability, flexibility, or underlying efficiency in utilizing computing resources. For example, both cases D and E develop automatic speech recognition (ASR) models but use Kaldi and PyTorch frameworks, respectively. Frameworks can mature into specific domains at varying rates, and therefore teams might adopt different frameworks for such historical reasons. Analytics frameworks such as Spark also feature in case I.

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 an iterative process with distinct stages; determining the suitability of data and algorithms, parameter and hyper-parameter optimization, and model evaluation. Managing metadata from these stages makes the ML workflow traceable and reproducible.

We note three unique approaches used to evaluate models. One, data is stored according to its quality, which enables composing datasets with different levels of quality for training and validation purposes (Cases D and E). The second approach uses model ensembles, where each model is trained on a unique subset of the data (Case B). The third approach applies a configurable inference algorithm where each configuration uses a unique adaptation of the model (Case E).

To manage model evaluation results, case organizations either use dedicated experiment tracking tools (case G, I, N, O, and P), log process metadata (case B, E, F) or generate hashes (case D). Hashing involves computing hash values on given combinations of ML artifacts (data, configurations, model) following the execution of an ML pipeline. Case E and F generate and collect

Table 2. Summary of practices and challenges

Workflow Stage	Practices	Challenges
Data management	• Batch or stream data loads largely from internal systems, third party vendors or devices and sensors • Co-location of data and compute to reduce I/O latency in data transfer • 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 requirements 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 and 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.

metadata (e.g., Git hashes), which are used to produce custom reports. These approaches are summarised in Table 3. Systematic management of experiments facilitates workflow automation and further increases traceability and reproducibility of ML workflows.

4.4. Model monitoring

Training data drift commonly occurs due to structural changes in the data generating process. Identifying drift in numeric data types is 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 can occur due to data or concept drift, and it is often manifested by a model's gradual or sudden loss of accuracy. Metrics such as accuracy and error rates are commonly used to monitor production models. For example, in a transcription setting, measuring the character and word edits required after inference were used (Case D) as error metrics to monitor production models and characterize any model drift.

Infrastructure monitoring is applied to ensure models efficiently utilize resources (GPU/CPU, memory, disk, network, etc.) or to flag technical problems such as scaling designs and I/O bottlenecks during training or inference. Cases D, E, and G closely monitor endpoint latency since it forms an important requirement of the entire ML solution.

5. Tools in ML pipelines (RQ2)

This section presents common tools observed across the cases as summarized in Table 3.

5.1. Version management

Model training code, often written in notebooks, and other project artifacts are version controlled using tools like Git, Gitlab, and Bitbucket. Data versioning is done by generating and versioning metadata using specialized tools, such as data version control (DVC). Model training is conducted in public cloud settings for most cases, while a few cases train on-premises. To consistently provision training environments, ’infrastructure-as-code’ practices, tools such as Terraform are used (Cases A and E).

5.2. ML training workflow

We observe that most case organizations containerize (using docker) individual workflow steps instead of encapsulating all workflow steps in a single container. In ML, containerization facilitates the isolation of different workflow tasks/steps, making the workflow modular, traceable, and reproducible. We further note that containers are commonly orchestrated using Kubernetes, allowing easier migration of pipelines (or parts of it) across infrastructure vendors. Data transfers across workflow steps during training are done using standard persistent volumes. However, large datasets may require using network mounts (Case F).

ML workflows may include steps specifying feature extraction, model training, and validation. The complexity involved in these steps can vary depending on the ML domain. Workflows can be managed using a custom configuration tool for example, Yet Another Markup Language (YAML-based) or a dedicated workflow toolkit. In complex ML setups, frameworks such as Argo (Case D) and Metaflow (Case G) are preferred. We note that although high-level ML workflow platforms, such as AWS SageMaker, provide an end-to-end integration advantage, they are also challenging to use when developing complex models due to inflexibility (Cases B, G).

Those in support of custom tooling appreciate the flexibility to add different tools to the workflow. For example, a tool such as an explainer dashboard that facilitates a model’s explainability is added as part of an ML workflow (Case A). An alternative workflow setting can have a single step containing multiple containers (data access data and model training). Customized components that provide access to these containers can be created to monitor independent utilization of computing resources at the container level (Cases C, G).

One overall advantage of using ML workflow tools is that event-based training queues can be orchestrated, for example, based on the continuous arrival of training data. Tools like Apache airflow provide the functionality to schedule model training based on given triggers.

ML experiments can be tracked using custom web-based user interface tools; this facilitates the evaluation of results and model performance comparison during the development process (Cases B and F). To their advantage, custom platforms can freely include any metadata the team considers relevant (Case F). Plugins can also be developed to integrate with existing open-source solutions such as MLflow (Case G). Low-level training metrics are observed with Tensorboard (Case A).

5.3. Continuous integration (CI) and testing

CI tools, such as Jenkins, are used to run tests and build docker images based on model artifacts resulting from the training workflow (Cases A, D, G). Static code analysis and other tests check general container functionality during the image building process (Case A, G). Domain-specific tests are also executed to ensure the scope of a model’s inputs and outputs is unchanged. These tests generally extend testing to the entire pipeline using small amounts of input data (Case D, F, G).

Docker images created from the CI system

Table 3. Tools (*planned, - tool information not provided)

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	Google Cloud 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	REST API endpoint	(Prometheus, Grafana)*
D	Git	Kubernetes	Argo	Hashing	-	API	Prometheus, Grafana
E	GitHub	Kubernetes	Custom	Metadata	PostgreSQL	-	Prometheus, Grafana
F	-	Kubernetes	-	Metadata	-	REST API endpoint	Logging, Elastic Search, tool's Web UI
G	GitHub	Kubernetes	Custom, Metaflow	MLflow,	Docker registry	gRPC API endpoint, Kafka	-
H	Git	AWS Elastic Container	Apache Airflow	-	S3	-	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, Databricks model registry*	Batch prediction, Java apps	AWS CloudWatch
O	Git, DVC	Kubernetes	Apache Airflow, Kubeflow	MLflow, kubeflow	MLFlow model registry	REST API	Logging, Grafana
P	-	-	Databricks	MLflow	S3, MLFlow model registry	API, batch, embedded	-

S3 storage: Amazon Simple Storage Service
*: Planned tool adoption
-: Tool information not provided

are (automatically) deployed to a staging environment for additional tests before deployment to production. Typically, this may include testing the model API's data type (Case A, D), ensuring a model makes sound predictions (Case E) and also ensuring that the deployment procedure loads a serialized model into the relevant API endpoints.

Trained models are generally stored in classic data storage solutions or dedicated container image registries. For example, Case E stores a model and metadata about the model to a PostgreSQL data warehouse. Case C stores trained models in Nexus while case G uses docker registry to store the models before deployment to production.

5.4. ML deployment and serving

Generally, inference serving is either done in batch or online format. Majority of the models are deployed as REST (REpresentational State Transfer) API endpoints 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 or onsite installations.

Models with strict inference latency requirements are deployed as gRPC (Google Remote Procedure Call) API endpoints. For example, case C business application has strict latency requirements and therefore uses the gRPC, which supports streaming.

Most cases implement custom serving infrastructure, although emerging model serving systems like KFServing and Seldon are tested in Cases C and O, respectively.

Finally, we note that data scientists often do not undertake deployment-related tasks, but these are done by other dedicated teams with specialist knowledge, such as Kubernetes configuration (Case G).

5.5. Monitoring

After model deployment, monitoring is performed at different levels of granularity. The most common form of monitoring is undertaken for infrastructure management. Logging, monitoring, and alerting tools, like Prometheus and Kubernetes logging (stackdriver), are used to collect a cluster's performance metrics. These metrics are then visualized on dashboards using tools like Grafana, Tableau, or other business analytics tools. For models deployed as APIs, model metrics are logged and stored in Elasticsearch and BigQuery where the logs are used to perform model health and quality checks in production, e.g., monitoring average accuracy levels (Cases A, F, and O).

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

6. Discussion and conclusion

As the AI engineering field is still making progress in defining well-established processes, there is a need for details on how such systems are engineered [10]. Compared to existing literature, our findings of practices in ML workflows (Section 4) and tools in ML pipelines (Section 5) present the 'how' knowledge, in contrast to cataloging the practices or tools used in ML applications. This information can be used to objectively analyze practices applied across organizations and identify areas to guide future research efforts that seeks to improve knowledge in the field of AI engineering.

Following the taxonomy and challenges described in [2], we consider the presented cases to be at the critical deployment stage where ML components need to co-exist with other general software components in production systems. We observed that practitioners were implementing practices that also address all of the challenges associated with the critical deployment stage [2]. In particular, tools and practices adopted in ML workflows that are summarised in Table 3 and Table 2 respectively are indicative of these solutions. However, the challenge related to data management remains an active research area given the increasing amount of data and disparity in data types. Similarly, the challenges in subsequent stages also need to be addressed, particularly new techniques for monitoring the final models observed in both studies.

Similar to [5], our results reveal a low adoption of SE best practices in the deployment category and a medium-to-high adoption in other categories (data, training, team, and coding). We also observed that smaller and younger organizations found it easier to adopt SE practices and emerging tools than older organizations with larger and distributed teams. We attribute this disparity to legacy systems and processes in the older organizations.

While the survey [5] indicated that teams with low experience have low adoption of the SE practices, we observed the contrary that there was a high adoption of the SE practices in teams with limited experience. We attribute this to the field of AI engineering having clearly defined the specific practices (e.g., tracking of model experiments) with supporting tools (MLflow, Kubeflow) in academia and industry. There are several areas in the AI engineering field where certain practices, such as data versioning, are considered valid but lack well-established knowledge of how to enact the practice and thus are least adopted in the industry.

6.1. Implications to research

Practices and tools for data discoverability. Obtaining good quality training data for ML purposes is an arduous task more so due to the increasing amount of data being produced and the variability of data handling procedures across data types. Establishing efficient data discoverability procedures would shorten ML production cycles and increase experimentation of ML models for R&D purposes.

Although feature stores have emerged as an intermediate solution for managing data in ML settings, there lacks empirical research on their adoption rates, benefits, and applicability across various data types and business settings.

Practices and tools for testing ML models and monitoring them in production. We observed a lack of extensive end-to-end testing of ML pipelines in the studied cases. In some cases though, static analyses were performed on the ML training code and the resulting container images were tested.

We further note that monitoring of ML-enabled systems needs to extend beyond general infrastructure monitoring practices. A model's utility can be reduced by degrading accuracy levels as a result of model drift. Although data drift can be detected during model development, concept drift is more challenging to control in production settings. The empirical effects of concept drift and control practices are yet to be explored in literature.

6.2. Implication to practice

Platforms vs independent tools in ML. Generally, we make a similar observation to [6] that practices in ML workflows and pipelines vary based on factors, such as the type of data being used in model training, availability of computing resources, and the type of ML solutions being developed. However, we also note two primary ways organizations developed their ML pipelines. One, they can compose a variety of tools to orchestrate a pipeline. Two, teams can use integrated frameworks/platforms such as SageMaker, which contain built-in tools for various stages of an ML pipeline. Most of the studied teams preferred the first approach because it offers flexibility and the ability to extract low-level information provided by independent tools. However, a common challenge when using separate tools is the required high maintenance efforts. The few teams that use the second approach preferred the instant integration and support offered by platform providers.

Dominant tools. Some well-established tools in SE remain useful when engineering AI systems. Such tools relate to version management (Git), containerization (Docker), and monitoring (Prometheus). However, some of these tools are arguably insufficient for other AI artifacts. For example, code version management tools are not suitable for data version management. Alternative tools dedicated to ML settings are emerging to address some of the inefficiencies encountered by practitioners.

Model inference infrastructure. Following general SE practices, ML API endpoints are based on custom webservers. However, we note there are initiatives to standardize model serving servers through open development of a serving API, which is realized by frameworks and tools like NVIDIA's Triton Inference Server, TensorFlow Serving, Torchserve etc.

6.3. Validity Threats

Construct validity. Considers whether the constructs discussed in the interview questions were interpreted in the same way by the researchers and the interviewees. This was mitigated by sharing the study objective and an outline of the interview guide to practitioners before the interview. During the interviews, a brief presentation was given by researchers to communicate the interview framework, and later the discussion was tailored to practitioners' expertise.

External validity. Concerns generalization of the findings and other threats that can cause incorrect conclusions to be drawn from the study. Despite having a global presence, the involved organizations are from one geographical location (Finland). This means the conclusions drawn about the state of practice and tools for ML may not be generalized for the whole SE industry population.

Reliability. Concerns the extent to which data and analysis are dependent on specific researchers. This threat to validity was mitigated by having at least two researchers throughout the research process. Furthermore, the results were shared with practitioners to review before submission for publication.

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. W. Hummer et al, "ModelOps: Cloud-based lifecycle management for reliable and trusted AI," in *International Conference on Cloud Engineering*, 2019, pp. 113–120.
9. P. Runeson and M. Höst, "Guidelines for conducting and

reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, 2008.
10. I. Ozkaya, "What is really different in engineering AI-enabled systems?" *IEEE Software*, vol. 37, no. 4, pp. 3–6, 2020.

Biographies

Dennis Muiruri is a member of the empirical software engineering research group and holds a Masters of Science degree in Data Science from the University of Helsinki, Helsinki, 04100, Finland. His research interests include deployment and operations of ML systems.

Lucy Ellen Lwakatare is a postdoc at the University of Helsinki, Helsinki, 04100, Finland. She received her PhD from University of Oulu in Software Engineering. Her research interests include agile, DevOps and ML engineering.

Jukka K. Nurminen is a professor at the University of Helsinki, Helsinki, 04100, Finland. He received PhD from Helsinki University of Technology in Systems and Operations Research. His main interests are on tools and techniques for data-intensive software systems, testing of AI solutions, and software development for quantum computers.

Tommi Mikkonen is a professor of software engineering at University of Jyväskylä, Jyväskylä, 40014, Finland. He received his Ph.D in Software Engineering from Tampere University of Technology. His interests include IoT, software architectures, and software engineering for AI.