Software Engineering Patterns for Machine Learning Applications (SEP4MLA) - Part 2

HIRONORI WASHIZAKI, Waseda University / National Institute of Informatics / System Information / eXmotion
FOUTSE KHOMH, Polytechnique Montréal
YANN-GAËL GUÉHÉNEUC, Concordia University
HIRONORI TAKEUCHI, Muashi University
SATOSHI OKUDA, Japan Advanced Institute of Science and Technology
NAOTAKE NATORI, Aisin Seiki
NAOHISA SHIOURA, Aisin Seiki

Practitioners and researchers study best practices to develop and maintain ML application systems and software to address quality and constraint problems. Such practices are often formalized as software patterns. We discovered software-engineering design patterns for machine-learning applications by doing a thorough search of available literature on the subject. From these ML patterns, we describe three ML patterns (“Different Workloads in Different Computing Environments”, “Encapsulate ML Models Within Rule-base Safeguards”, and “Data Flows Up, Model Flows Down”) in the standard pattern format so that practitioners can (re)use them in their contexts.

Categories and Subject Descriptors: I.2.6 [**Artificial Intelligence**]: Learning—*Machine learning*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

Additional Key Words and Phrases: Machine learning patterns

ACM Reference Format:

Washizaki, H. Khomh, F. Guéhéneuc, Y.-G. Takeuchi, H. Okuda, S. Natori, N. and Shioura, N. 2020.Software Engineering Patterns for Machine Learning Applications (SEP4MLA) - Part 2. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 0 (March 2020), 10 pages.

1. INTRODUCTION

Practitioners and researchers study best practices to develop and maintain ML application systems and software to address quality and constraint problems. Such practices are often formalized as software patterns. We call these software-engineering patterns for machine-learning applications (SEP4MLA) in order to distinguish them from non-software-engineering patterns for ML, such as patterns for designing ML models. As a part of various

Author’s address: H. Washizaki, 3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan; email: washizaki@waseda.jp; F. Khomh, Polytechnique Montréal, QC, Canada; email: foutse.khomh@polymtl.ca; Y.-G. Guéhéneuc, Concordia University, Montréal, QC, Canada; email: yann-gael.gueheneuc@concordia.ca; H. Takeuchi, Muashi University, Tokyo, Japan; email: h.takeuchi@cc.musashi.ac.jp; S. Okuda, Japan Advanced Institute of Science and Technology, Japan; email: okuda@jaist.ac.jp
N. Natori, Aisin Seiki, Japan; email:naotake.natori@aisin.co.jp
N. Shioura, Aisin Seiki, Japan; email:naohisa.shioura@aisin.co.jp

Table I. Identified ML Patterns

Category	ID	Pattern name	Abstract
Topology	P_1	**Different Workloads in Different Computing Environments**	Physically isolate different workloads to different machines. Then optimize the machine configurations and the network usage [Wu et al. 2019].
	P_2	Distinguish Business Logic from ML Models	Separate the business logic and the inference engine, loosely coupling the business logic and ML-specific dataflows [Yokoyama 2019; Washizaki et al. 2020].
	P_3	ML Gateway Routing Architecture	Install a gateway before a set of applications, services, or deployments and use application layer routing requests to the appropriate instance [Yokoyama 2019].
	P_4	Microservice Architecture	Define consistent input and output data and provide well-defined services to use for ML frameworks [Everett 2018; Smith 2017; Washizaki et al. 2020].
	P_5	Lambda Architecture	The batch layer keeps producing views at every set batch interval while the speed layer creates the relevant real-time/speed views. The serving layer orchestrates the query by querying both the batch and speed layer, merges it [Menon 2017; g18 ; Basak 2017].
	P_6	Kappa Architecture	Support both real-time data processing and continuous reprocessing with a single stream processing engine [g17].
Programming	P_7	Data Lake	Store data, which ranges from structured to unstructured, as “raw” as possible into a data storage [Gollapudi 2016; Menon 2017; Singh 2019; Washizaki et al. 2020].
	P_8	Separation of Concerns and Modularization of ML Components	Decouple at different levels of complexity from simplest to most complex [Rahman et al. 2019].
	P_9	**Encapsulate ML Models Within Rule-base Safeguards**	Encapsulate functionality provided by ML models and appropriately deal with the inherent uncertainty of their outcomes in the containing system by making use of deterministic and verifiable rules [Kläs and Vollmer 2018].
	P_{10}	Discard PoC Code	Discard the code created for the Proof of Concept (PoC) and rebuild maintainable code based on the findings from the PoC [Sculley et al. 2015].
Model operation	P_{11}	Parameter-Server Abstraction	Distribute both data and workloads over worker nodes, while the server nodes maintain globally shared parameters, which are represented as vectors and matrices [Sculley et al. 2015].
	P_{12}	**Data Flows Up, Model Flows Down**	Enable mobile devices to collaboratively learn a shared prediction model in the cloud while keeping all the training data on the device [Google 2017].
	P_{13}	Secure Aggregation	Encrypt data from each mobile device in collaborative learning and calculate totals and averages without individual examination [Google 2017].
	P_{14}	Deployable Canary Model	Run the explainable inference pipeline in parallel with the primary inference pipeline to monitor prediction differences [g10].
	P_{15}	ML Versioning	Record the ML model structure, training dataset, training system and analytical code to ensure a reproducible training process and an inference process [Wu et al. 2019; Amershi et al. 2019; Sculley et al. 2015; Washizaki et al. 2020].

SEP4MLA such as ML requirements engineering patterns and ML security engineering patterns, we discovered 15 software-engineering design patterns for machine-learning applications (hereafter, ML patterns) by doing a thorough search of available literature on the subject.

We grouped these ML patterns into three categories (Table 1): ML systems topology patterns that define the entire system architecture, ML system programming patterns that define the design of particular component, and ML system model operation patterns that focus on ML model operation.

Since not all of the identified ML patterns are well documented in the standard pattern format, which includes clear problem statements and corresponding solution descriptions, we are describing these ML patterns in the standard pattern format so that practitioners can (re)use them in their contexts [1]. Herein we describe three ML patterns selected from three different categories: “Different Workloads in Different Computing Environments” (P_1), “Encapsulate ML Models Within Rule-base Safeguards” (P_9), and “Data Flows Up, Model Flows Down” (P_{12}).

2. DIFFERENT WORKLOADS IN DIFFERENT COMPUTING ENVIRONMENTS (P_1)

2.1 Source

[Hazelwood et al. 2018]

2.2 Intent

To satisfy different resource requirements and constraints of different workloads by deploying them in different computing environments.

2.3 Context

The typical ML pipeline process [Amershi et al. 2019] consists of various stages including data collection, data processing (such as data cleaning and data labelling), feature engineering, model training, model deployment, and model execution (i.e., inference and prediction) and monitoring. When deploying ML at the large scale, there are various considerations since different workloads correspond to different stages impose different significant requirements on necessary computing resources.

For many machine learning models, the availability of extensive, high-quality data is crucial for prediction. The ability to rapidly process and feed these data to the training machines is important for ensuring fast and efficient offline training. For sophisticated ML applications, the amount of data to ingest for each training task can be very huge. In addition, often complex preprocessing is necessary to have cleaned and normalized data to allow efficient transfer and easy learning. These impose very high resource requirement especially on storage, network, CPU and GPU.

Quality requirements and constraints on data and software systems also impose resource requirements, and often vary depending on the workload. For example, the sensitivity and security requirements of data can be often an issue. Data for training a ML model is need to be protected within the training environment if it is personally identifiable information. In contrast, data processed from the user input at the inference environment might be exposed to the outside. Requirements on quality attributes of software systems such as flexibility and scalability can also be different by workloads.

2.4 Problem

There are four types of ML workloads having quite different characteristics: Data collection and processing, and model training and execution. These four workloads tend to have different resource requirements and constraints. Thus, it becomes very hard to optimize the machine configuration if these workloads are deployed on the same machine. Such co-existence also prevents various quality attributes such as security, flexibility and scalability.

2.5 Solution

To decouple the data collection workload, data processing workload, model training workload and model execution workload, and isolate them to different machines in order to satisfy different resource requirements and constraints, and optimize for each workload. Figure 1 shows the entire structure of those workloads deployed on different computing environments. To realize that, it is necessary to structure communication among those workloads as follows.

[1]In [Washizaki et al. 2020], we described other four ML patterns: “Data Lake” (P_7), “Distinguish Business Logic from ML Models” (P_2), “Microservice Architecture” (P_4) and “ML Versioning” (P_5).

—Collector: The data collection machine(s) collects the data from data sources and store it into storage.

—Reader: The data processing machine reads the data from the storage, process and condense them, and then send to the trainers.

—Trainer: The training machine solely focuses on executing the training options rapidly and efficiently. The trained model is deployed into the predictor.

—Predictor: The prediction machine focuses on execution of the deployed model to conduct inference and prediction according to the given input.

Machine configurations can be highly optimized for each different workload. And, high flexibility and scalability can be achieved by distribution over the network. Since the resource requirements for training ML models especially deep learning models are often much larger than the requirement for executing that model, the trainer (and the reader) is often implemented in cloud computing environments. In contrast, the predictor can be deployed at closer to users and more resource-limited environments such as IoT edge and mobile devices.

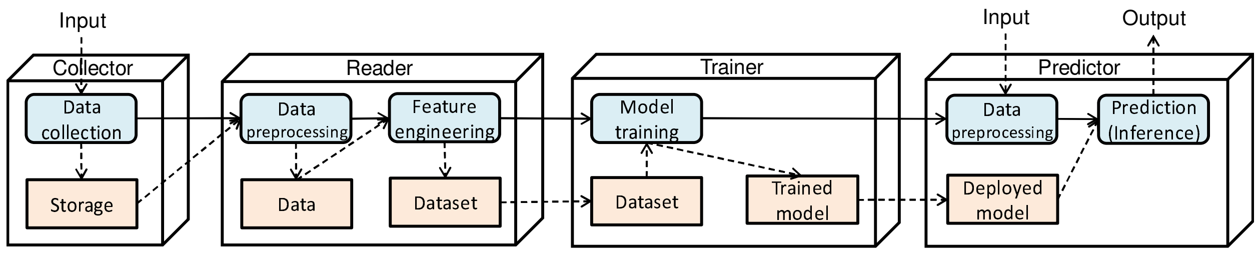

Fig. 1. Structure of the “Different Workloads in Different Computing Environments” pattern

2.6 Example

At Facebook, most of the ML training is run through the FBLearner platform [Hazelwood et al. 2018]. FBLearner is a suite of three tools: FBLearner Feature Store as reader, FBLearner Flow as trainer, and FBLearner Predictor as predictor. Each tool focuses on different parts of the ML pipeline as shown in Figure 2. FBLearner has an internal job scheduler to allocate resources and schedule jobs on a shared pool of GPUs and CPUs to meet with different resource requirements of the reader, trainer and predictor.

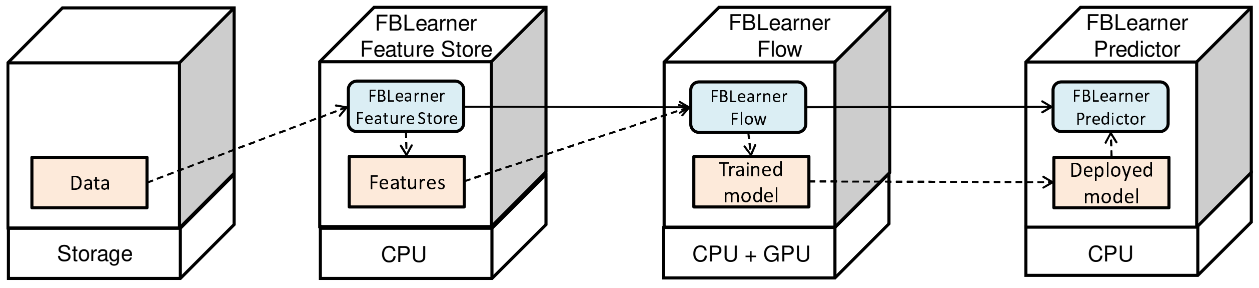

Fig. 2. Facebook’s Machine Learning Flow and Infrastructure

2.7 Discussion

The data traffic generated by training can become significant. If not handled intelligently, this can easily saturate network devices and even disrupt other services. Thus, network optimization is important such as compression, scheduling algorithms and data/compute placement.

2.8 Related Patterns

—Data Lake [Washizaki et al. 2020] (P_1): The storage associated with the data collection workload is often implemented as "Data Lake", which stored data ranging from structured to unstructured.

—Data Flows Up, Model Flows Down (P_{12}): The model training workloads can be distributed over clouds and mobile devices. In the pattern "Data Flows Up, Model Flows Down", if the collected data at mobile devices is the user's privacy data, the deployed model is re-trained locally by using the data, and the difference between the original model and the re-trained model is uploaded to the cloud.

3. ENCAPSULATE ML MODELS WITHIN RULE-BASE SAFEGUARDS (P_9)

3.1 Source

[Kläs and Vollmer 2018]

3.2 Intent

To mitigate and absorb the low robustness of ML models by encapsulating ML models within rule-base safeguards.

3.3 Context

ML models are becoming a part of highly autonomous software systems, where a high risk exists that humans, businesses or the environment may be harmed in the case of a failure.

The Sheridan-Verplanck taxonomy defines 10 levels of automation, which the AI application aims at or achieves [Feldt et al. 2018]. The higher the level of automation, the more autonomous the AI technology becomes in making decisions, and the higher the risk involved. Especially at the level 7 or higher, computers make and implement decisions without getting approval from humans.

3.4 Problem

Because of the complexity and empirical nature of ML models, no guarantee can be provided for their correctness. Thus, it is difficult to rely on ML in highly autonomous systems, especially in the case of safety-critical systems as well as highly individualized or adaptive systems since ML models are known to be unstable and vulnerable to adversarial attacks and to noise in data and data drift overtime.

3.5 Solution

To encapsulate functionality provided by ML models and appropriately deal with the inherent uncertainty of their outcomes inside the implementation of the business logic API by making use of deterministic and verifiable rules through the safeguards. Figure 3 shows the entire structure. The business logic API, the safeguard and the model are responsible for the followings:

—The business logic API is a Facade that wraps the corresponding ML model.

—The safeguard encapsulating the ML model is responsible for adequate risk management, taking into account the likelihood that the outcome of the model might be wrong, as well as the risky consequences of every decision made. The safeguard could, for example, decide to consider further information sources (as applied in sensor fusion) or adapt its behavior in order to handle the remaining uncertainty adequately.

—The encapsulated ML model has to deliver its service together with information about outcome-related uncertainty that can justifiably be trusted. It allows the system to conduct informed decisions via the business logic API.

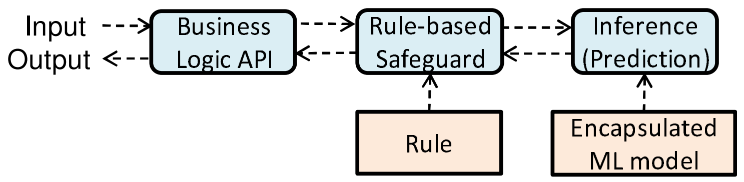

Fig. 3. Structure of the “Encapsulate ML models within rule-base safeguards” pattern

3.6 Example

In a scenario of autonomous intersection crossing by ML-based traffic sign recognition, the containing system could, for example, use GPS localization as an additional information source or slow down the vehicle, thereby buying time to analyze further images taken of the traffic situation [Kläs and Vollmer 2018].

Another scenario is a health recommender system that is individualized and adaptable to enable early detection of individual health problems [Mellin 2017]. If the change in the health is slow enough, then the system may adapt to a situation that is risky and consider it to be normal. Thus, a rule-based safeguard that checks boundaries based on knowledge would be necessary to avoid risky recommendation based on such excessive individual adaptation.

3.7 Discussion

Considering sources of uncertainty such as model fit, data quality and scope compliance would lead to more systematic development and testing of the model and the system.

3.8 Related Patterns

—ML Gateway Routing Architecture (P_3) [Yokoyama 2019]: A gateway can be installed in front of business logic APIs so that clients can utilize multiple ML-based services with safeguards while avoiding difficult setting-up and management of individual endpoints.

4. DATA FLOWS UP, MODEL FLOWS DOWN (P_{12})

4.1 Source

[Google 2017]

4.2 Intent

Decoupling the ability to predict and re-train from the need to store the training data and the trained ML model in the cloud.

4.3 Context

Some ML applications are running on local devices such as smart phones, cameras and IoT devices. Such applications predict human behavior and supports their activities timely. The ML models are trained by human behavior data collected through the devices.

4.4 Problem

The ML application on the local devices should return the prediction results in real time and this means that the ML model execution should be performed on the edge and the model training is performed on the cloud. Because the ML applications running on the local devices sometimes return the prediction results customized, the ML models need to be re-trained by the data collected through the device. Though the ML model on each device can be re-trained locally, we need to store the data collected by each device into the cloud for re-training the ML model more effectively. In this situation, there is a case that we need to protect the user's privacy data to satisfy the confidentiality.

4.5 Solution

To solve the problem described above, we introduce "Data Flows Up, Model Flows Down" pattern. Figure 4 illustrates the structure of this pattern. In this pattern, the ML model is trained on the cloud and deployed to each local device (Model flow down). By using the ML application, the data for re-training is collected in each device and these data or the model re-trained locally will be sent to the cloud (Data flows up).

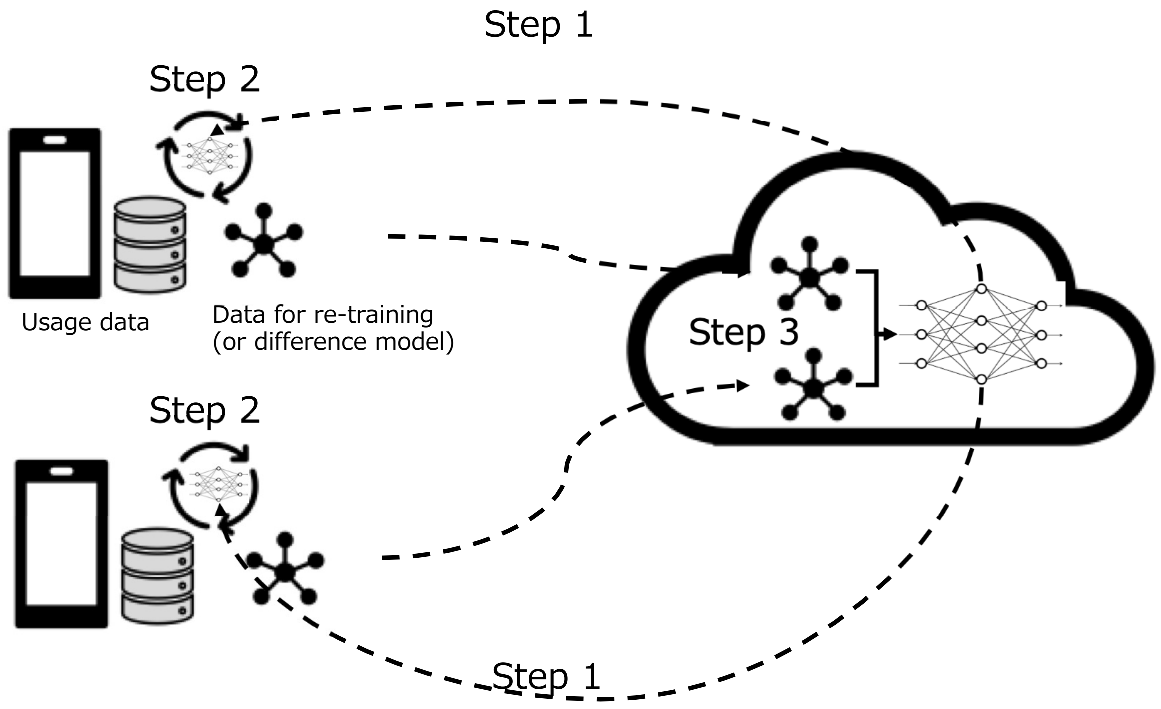

Fig. 4. Structure of the "Data Flows Up, Model Flows Down" pattern

Detailed steps of this pattern are as follows:

- The initial training is performed on the cloud environment and the trained ML model is deployed to each mobile device (Step 1).
- In each local mobile device, usage data is collected through the device (Step 2).
  - If the collected data does not contain any privacy data, the device flows the real time data up to the cloud environment.
  - If the collected data is the user's privacy data, the ML model is re-trained locally by using the collected data. After that the difference between the original ML model and the re-trained model is uploaded to the cloud.
- In the cloud, by using the uploaded data, the ML model is retrained (Step 3). When the difference models provided from each local device, they are averaged and updated into the ML model. Updating the ML model

from difference models provided from each device is a special case of this pattern and is called the Federated Learning. The updated ML model will be re-deployed to each local device.

4.6 Example

“Data Flows Up, Model Flows Down” is implemented in the ML system using the edge devices such as mobile phones, cameras, and IoT devices.

A major retail company in US introduced the ML system to identify traffic problems in their store parking lots [IBM 2020]. They train an ML model from the image data collected by the parking-lot security cameras and the trained model is deployed at each store (Model flow down). In the store, the model is executed by real time image data from the security cameras and the notification is sent to the staffs whenever a traffic jam occurs so that they can rectify the problem. Whenever the store encountered an example that was adversarial (e.g. a false positive or one where it missed a jam somehow) they would flag that to the application which would then send that set of images to the cloud for model re-training (Data flows up).

Federated Learning, the special case of this pattern, is implemented in the Google Keyboard on Android called Gboard [Google 2017]. When the user types texts, the smart phone stores information about the current contewhether the user clicked the suggested query provided by Gboard. Gboard contains the ML model locally and re-trains the model by the stored usage data. As the results, the Gboard reflects the user’s specific behavior and the suggested query will be improved. The difference between the original model and the re-trained model in each local device is uploaded to the Google Cloud. At fixed intervals, the base ML model in the cloud will be updated by using these uploaded difference models and re-deployed to each local device.

This Federated Learning approach is also implemented in medical imaging system [Wen and Rieke 2020]. To identify the disease from a medical image by using a ML system, we need huge amount of medical images. However, such image data required to train a reliable and robust ML model algorithm may not be available in a single institution. At the same time, it is often not feasible to share patient data between some institutions in a common data center due to patient privacy concerns. In this case, the initial ML models trained from the small training data are deployed to each institution and the model is re-trained locally by using the institutions own patient data. After that, the difference models from the institutions are collected at the data center and averaged to update the ML model.

4.7 Discussion

When applying this “Data Flows Up, Model Flows Down” pattern, we need to assess the characteristics of the data collected at the local devices. If the data contains the user’s privacy data, we need consider whether we can send such data to the cloud directly. If we cannot store the user’s privacy data in the cloud, we need to consider the Federated Learning approach. Furthermore, before applying this Federated Learning, we need to assess whether the following conditions are satisfied. First, the local device such as the smart phone has enough computer resources to store the trained ML model and re-train the model. Next, we need to select the ML model that can be updated by averaging the difference model. Third, there are not any negative effects by the updated ML model that is locally re-trained by using the user’s own behavior data.

4.8 Related Patterns

—Different Workloads in Different Computing Environments (P_1)

—Secure Aggregation (P_{13})

5. CONCLUSION

Herein three patterns (“Different Workloads in Different Computing Environments”, “Encapsulate ML models within rule-base safeguards”, and “Data Flows Up, Model Flows Down”) are described. These patterns are from a set of ML patterns identified through a thorough search of available literature. In the future, we plan to write all ML patterns in the standard pattern format to help developers adopt good practices described by these patterns.

Acknowledgement

We are grateful to shepherd Kyle Brown for his careful and valuable reviews that significantly improved this paper. This work was supported by JSPS Bilateral Program JPJSBP120209936, JST-Mirai Program Grant Numbers JP18077318 and JP20319852, and enPiT-Pro Smart SE.

Received July 2020; revised August 2020; accepted

REFERENCES

A Design Pattern for Explainability and Reproducibility in Production ML. `https://www.parallelm.com/a-design-pattern-for-explainability-and-reproducibility-in-production-ml/`. (????).

From Insights to Value - Building a Modern Logical Data Lake to Drive User Adoption and Business Value. `https://www.slideshare.net/Hadoop_Summit/from-insights-to-value-building-a-modern-logical-data-lake-to-drive-user-adoption-and-business-value`. (????).

Lambda Architecture Pattern. `https://hub.packtpub.com/lambdaarchitecture-pattern/`. (????).

Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. 291–300. `DOI:http://dx.doi.org/10.1109/ICSE-SEIP.2019.00042`

Anindita Basak. 2017. *Stream Analytics with Microsoft Azure: Real-time data processing for quick insights using Azure Stream Analytics*. Packt Publishing.

Julian Everett. 2018. Daisy Architecture. `https://datalanguage.com/features/daisy-architecture`. (July 2018).

Robert Feldt, Francisco Gomes de Oliveira Neto, and Richard Torkar. 2018. Ways of Applying Artificial Intelligence in Software Engineering. In *6th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2018, Gothenburg, Sweden, May 27, 2018*. ACM, 35–41.

Sunila Gollapudi. 2016. *Practical Machine Learning*. Packt Publishing, Birmingham, UK. `https://books.google.ca/books?id=3ywhjwEACAAJ`

Google. 2017. Federated Learning: Collaborative Machine Learning without Centralized Training Data. `https://ai.googleblog.com/2017/04/federated-learning-collaborative.html`. (2017).

Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 620–629. `DOI:http://dx.doi.org/10.1109/HPCA.2018.00059`

IBM. 2020. IBM Video Analytics with IBM Visual Insights. `https://www.ibm.com/downloads/cas/GJBJQM4Y`. (June 2020).

Michael Kläs and Anna Maria Vollmer. 2018. Uncertainty in Machine Learning Applications: A Practice-Driven Classification of Uncertainty. In *Computer Safety, Reliability, and Security - SAFECOMP 2018 Workshops, ASSURE, DECSoS, SASSUR, STRIVE, and WAISE, Västerås, Sweden, September 18, 2018, Proceedings*. 431–438. `DOI:http://dx.doi.org/10.1007/978-3-319-99229-7_36`

Jonas Mellin. 2017. Can a machine learning model completely replace a rules based system? `https://www.quora.com/Can-a-machine-learning-model-completely-replace-a-rules-based-system`. (April 2017).

Pradeep Menon. 2017. Demystifying Data Lake Architecture. `https://www.datasciencecentral.com/profiles/blogs/demystifying-data-lake-architecture`. (August 2017).

Md Saidur Rahman, Emilio Rivera, Foutse Khomh, Yann-Gaël Guéhéneuc, and Bernd Lehnert. 2019. Machine Learning Software Engineering in Practice: An Industrial Case Study. *CoRR* abs/1906.07154 (2019). `http://arxiv.org/abs/1906.07154`

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Annual Conference on Neural Information Processing Systems*. Neural Information Processing Systems Foundation, Montréal, QC, Canada, 2503–2511. `http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems`

Ajit Singh. 2019. Architecture of Data Lake. `https://datascience.foundation/sciencewhitepaper/architecture-of-data-lake`. (April 2019).

Daniel Smith. 2017. Exploring Development Patterns in Data Science. `https://www.theorylane.com/2017/10/20/some-development-patterns-in-data-science/`. (October 2017).

Hironori Washizaki, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2020. Software Engineering Patterns for Machine Learning Applications (SEP4MLA). In *9th Asian Conference on Pattern Languages of Programs (AsianPLoP 2020)*. Hillside, Inc., 1–10.

Y. Wen and N. Rieke. 2020. Federated Learning for Medical Imaging: Collaborative AI without Sharing Patient Data. `https://developer.nvidia.com/gtc/2020/video/s21536-vid`. (June 2020).

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim M. Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *25th International Symposium on High Performance Computer Architecture.* IEEE CS Press, Washington, DC, USA, 331–344. `DOI:http://dx.doi.org/10.1109/HPCA.2019.00048`

Haruki Yokoyama. 2019. Machine Learning System Architectural Pattern for Improving Operational Stability. In *International Conference on Software Architecture Companion*. IEEE CS Press, Hamburg, Germany, 267–274. `DOI:http://dx.doi.org/10.1109/ICSA-C.2019.00055`