# Software Engineering Patterns for Machine Learning Applications (SEP4MLA)

HIRONORI WASHIZAKI, Waseda University / National Institute of Informatics / System Information / eXmotion
FOUTSE KHOMH, Polytechnique Montréal
YANN-GAËL GUÉHÉNEUC, Concordia University

To grasp the landscape of software engineering patterns for machine learning (ML) applications, a systematic literature review of both academic and gray literature is conducted to collect good and bad software-engineering practices in the form of patterns and anti-patterns for ML applications. From the 32 scholarly documents and 48 gray documents identified, we extracted 12 ML architecture patterns, 13 ML design patterns, and 8 ML anti-patterns. From these 33 ML patterns, we describe three major ML architecture patterns ("Data Lake", "Distinguish Business Logic from ML Models", and "Microservice Architecture") and one ML design pattern ("ML Versioning") 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. and Guéhéneuc, Y.-G. 2020. Core Machine Learning Architecture and Design Patterns. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 0 (March 2020), 10 pages.

## 1. INTRODUCTION

Researchers and practitioners studying best practices strive to provide patterns that address software complexity and quality issues in any type of software system. Recently, software systems with machine learning (ML) algorithms have become very popular. Because these ML applications are complex, they should benefit from patterns that codify good/bad architectural and design practices during development. To help developers ensure quality, such practices are often formalized as architecture and design patterns, which encapsulate reusable solutions to common problems within a given context.

To grasp the landscape of software-engineering architecture and design patterns for machine-learning applications (SEP4MLA), we performed a systematic literature review (SLR) of both academic and gray literature to

The authors would like to thank Prof. Naoshi Uchihira, Mr. Norihiko Ishitani, Dr. Takuo Doi, Dr. Shunichiro Suenaga, Mr. Yasuhiro Watanabe, and Prof. Kazunori Sakamoto for their help. This work was supported by JST-Mirai Program Grant Number JP18077318, Japan.
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

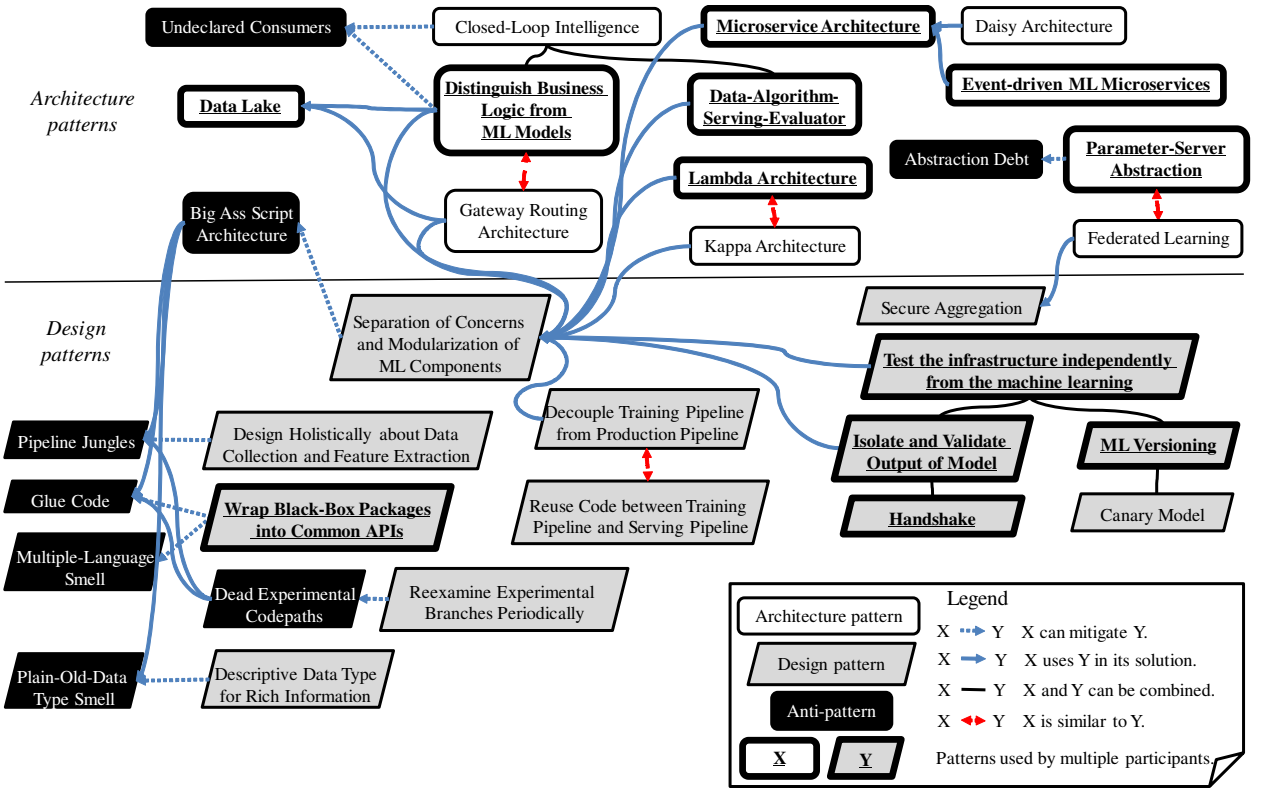


Fig. 1. Map to classify and relate patterns.

collect SE good (bad) patterns for ML application systems and software [Washizaki et al. 2019]. From the 32 scholarly documents and 48 gray documents identified in the SLR, 12 ML architecture patterns, 13 ML design patterns, 3 ML anti-architecture patterns, and 5 ML anti-design patterns were extracted. Figure 1 shows a pattern map of these patterns and their relationships [Washizaki et al. 2020].

A survey of software-engineering and machine-learning developers at a workshop revealed that there are 7 major ML architectural patterns and 5 major ML design patterns among these 33 SEP4MLA. Multiple developers answered that they used the following ML architecture patterns: “Data-Algorithm-Serving-Evaluator”, “Data Lake“, “Distinguish Business Logic from ML Models”, “Microservice Architecture”, “Event-driven ML Microservices”, “Lambda Architecture”, and “Parameter-Server Abstraction” [Washizaki et al. 2020]. They also used the following ML design patterns: “Handshake”, “Isolate and Validate Output of Model”, “ML Versioning”, “Test Infrastructure Independently from ML”, and “Wrap Black-box Packages into Common APIs” [Washizaki et al. 2020].

Since not all of the identified patterns are well documented in the standard pattern format, which includes clear problem statements and corresponding solution descriptions, herein we describe three major ML architecture patterns and one ML design pattern in the standard pattern format so that practitioners can (re)use them in their contexts.

## 2. DATA LAKE

### 2.1 Source

[Gollapudi 2016; Menon 2017; Singh 2019]

### 2.2 Intent

Collect raw data for as long as possible and offer access to analytic algorithms, like ML algorithms.

### 2.3 Context

A lot of real-world data/information cannot be expressed easily through relational schemas. This data can be documents (e.g., images), key/value pairs (e.g., users' encrypted passwords) graphs (e.g., UML class diagrams), or time series (e.g., temperatures from sensors). Although this data is not necessarily "big" data, it must be stored to be analyzed asynchronously.

### 2.4 Problem

In the traditional Data Warehouse architecture, all data is read from various sources by an ETL (Extract/Transform/Load) tool, and is well structured based on a simplified data model to gain all the advantages of a data warehouse such as fast and efficient query processing [Daehn 2017]. In other words, semantics first and content later [Daehn 2017]. In practice, neither the types of analyses that will be performed nor the adopted frameworks can be foreseen.

### 2.5 Solution

Figure 2 shows the entire structure to solve the problem. Data, which ranges from structured to unstructured, should be stored as "raw" as possible into a data storage called a "Data Lake". A data lake should allow parallel analyses of different types of data with various frameworks. Thus, a data lake should facilitate efficient (time, space) writing of data from the data sources (e.g., sensors) and efficient, parallel reading of the data for ML frameworks. To realize these features, the data lake must contain historical data and support the insert functionality.

The approach of the data lake is the opposite of a data Warehouse. That is, content first and semantics later [Daehn 2017]. By giving up the update and delete functionalities, the overhead of an active database can be avoided without sacrificing anything else [Daehn 2017].

### 2.6 Example

Apache Hadoop[1] and Azure Data Lake Storage Gen2[2] implement data lakes. In addition, Amazon Simple Storage Service (S3)[3] can be used to implement a data lake, and Amazon provides a service called "AWS Lake Formation"[4] to set up a secure data lake in days.

### 2.7 Discussion

Data lakes must have clear access controls to allow certain users to write data and other users to read data. For example, sensors can write (not read), while a control system can read (not write). Analyses applying on a data lake must first transfer (while transforming) the relevant data from the data lake into a data warehouse. Then the data warehouse can be completely rebuilt using a transformation tool such as Apache Spark[5].

### 2.8 Related Patterns

"Distinguish Business Logic from ML Models" (see Section 3) and Gateway Routing Architecture (not described here).

---

[1] https://hadoop.apache.org/
[2] https://azure.microsoft.com/solutions/data-lake/
[3] https://aws.amazon.com/s3/
[4] https://aws.amazon.com/lake-formation/
[5] https://spark.apache.org/

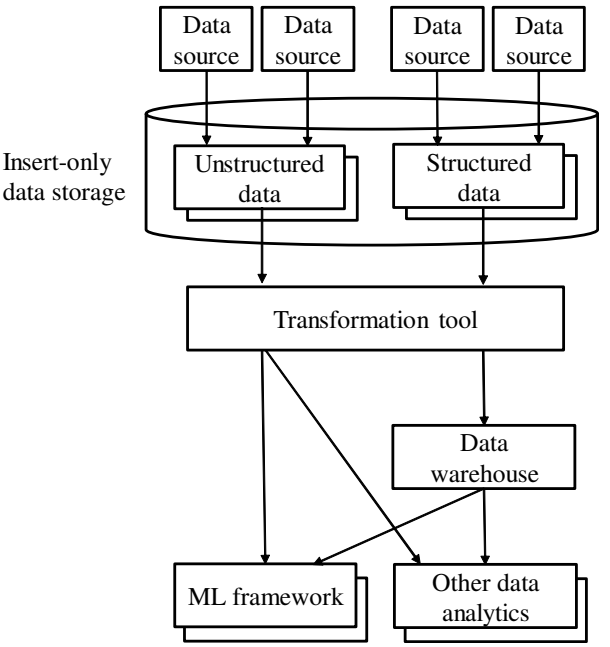


Fig. 2. Structure of the Data Lake pattern

## 3. DISTINGUISH BUSINESS LOGIC FROM ML MODELS

### 3.1 Source

[Yokoyama 2019]

### 3.2 Intent

Separate the overall business logic of the ML application from the actual used ML models.

### 3.3 Context

ML applications often use different ML frameworks and associated models (trained in previous, off-line phases). The frameworks and models play roles in the wider business logic.

### 3.4 Problem

The business logic depends on the results of the models, which may fail for a variety of reasons (as the rest of the business logic can too). Hence, the overall business logic should be isolated as much as possible from the ML models. This way, they can be changed/overridden when necessary without impacting the rest of the business logic.

### 3.5 Solution

Figure 3 shows the entire structure to solve the problem.

(1) Separate the business logic and the inference engine, loosely coupling the business logic and ML-specific dataflows.

(2) Use loggers to monitor the behavior of the ML models.

(3) Set-up alarms for changes in the data distributions that may invalidate the ML models throughout the execution of the ML application.

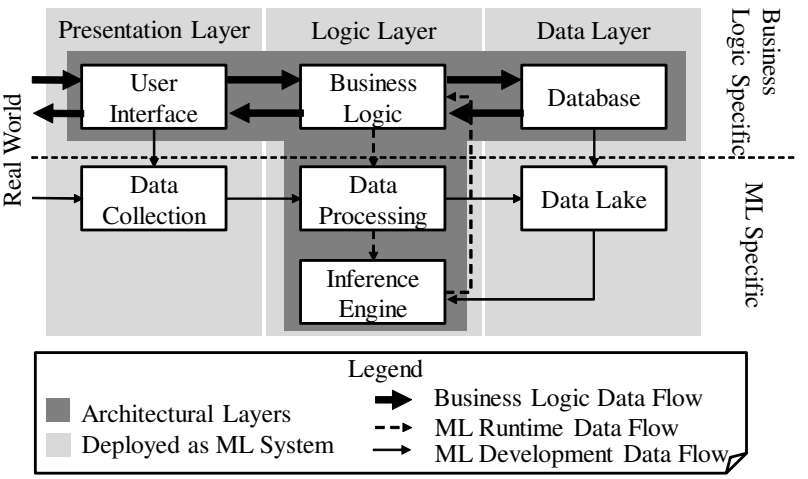


Fig. 3. Structure of the Distinguish Business Logic from ML Model pattern.

### 3.6 Example

Figure 4 presents an example of the pattern in a Slack-based Chatbot system [Washizaki et al. 2020]. In the system, there is clear separation between the Chatbot service (as the business logic) and the underlying ML components.

### 3.7 Discussion

This pattern distinguishes failures from the business logic and the ML models so that ML models can be changed without impacting the rest of the logic.

### 3.8 Related Patterns

“Gateway Routing Architecture” and “Closed-loop Intelligence” (not described here).

## 4. MICROSERVICE ARCHITECTURE

### 4.1 Source

[Everett 2018; Smith 2017]

### 4.2 Intent

Define consistent input and output data and provide well-defined services to use for ML frameworks.

### 4.3 Context

ML applications often use different ML frameworks, which are developed independently by ML researchers. ML researchers are experts in ML but not in the development of frameworks. Moreover, each team of researchers has a different idea about their working frameworks. Thus, ML frameworks are heterogeneous in terms of forms (APIs) and contents (data structures, algorithms).

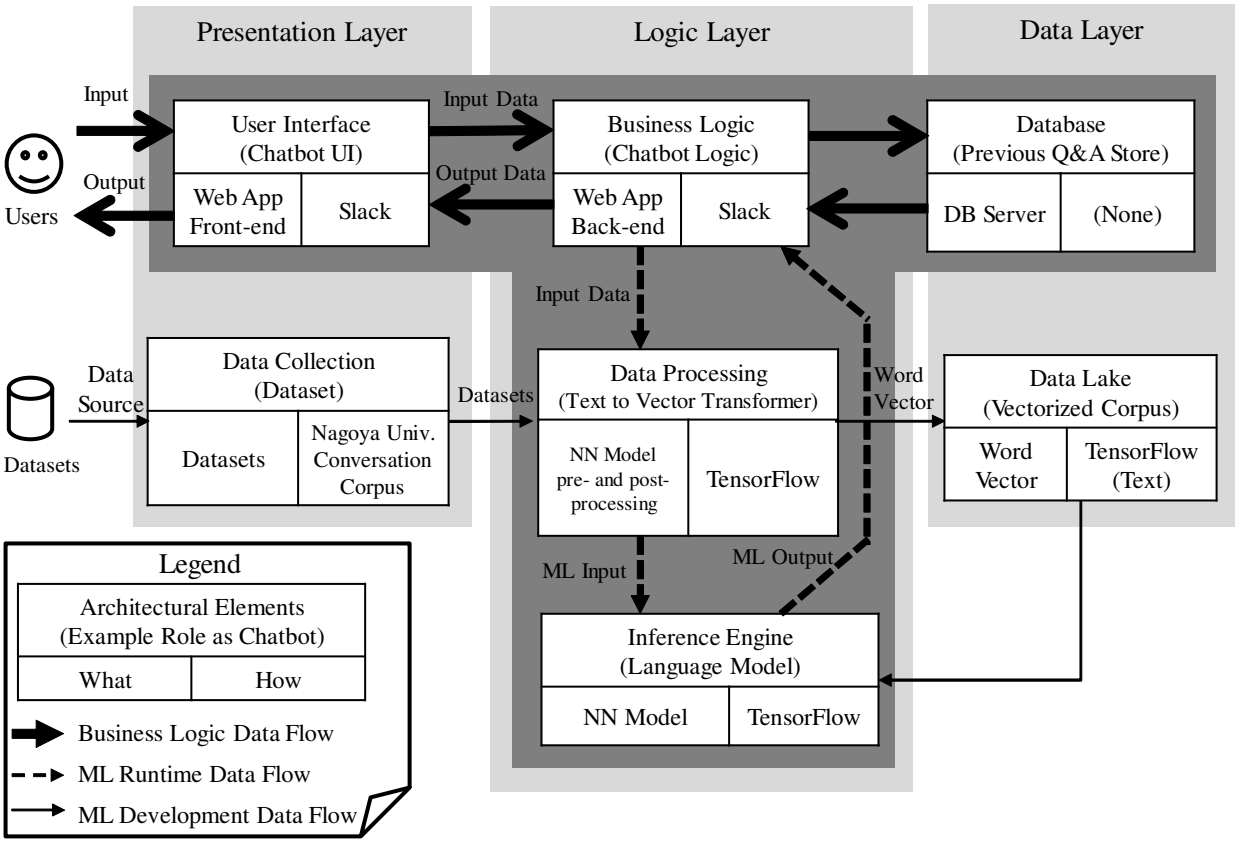


Fig. 4. Example of a Chatbot System Architecture by applying the “Distinguish Business Logic from ML Model” pattern.

A wide variety of these frameworks are available in open-source at places like `mloss.org`, or from in-house code, proprietary packages, and cloud-based platforms. ML applications may use one framework but later change for another framework.

### 4.4 Problem

It is impossible to be familiar with all ML frameworks. In addition, data scientists may be unable to install and run these frameworks locally. Thus, ML applications may be confined to “known” ML frameworks, and opportunities for more appropriate frameworks may be missed.

### 4.5 Solution

Figure 5 shows the entire structure to solve the problem. Data scientists working with or providing ML frameworks can make these frameworks available through micro-services. Micro-services are fine-grained and accessible through simple protocols. Thus, data scientists can offer access to their ML frameworks through micro-services so that any ML application can benefit from their frameworks without having to (1) install them and (2) maintain them when new versions are available.

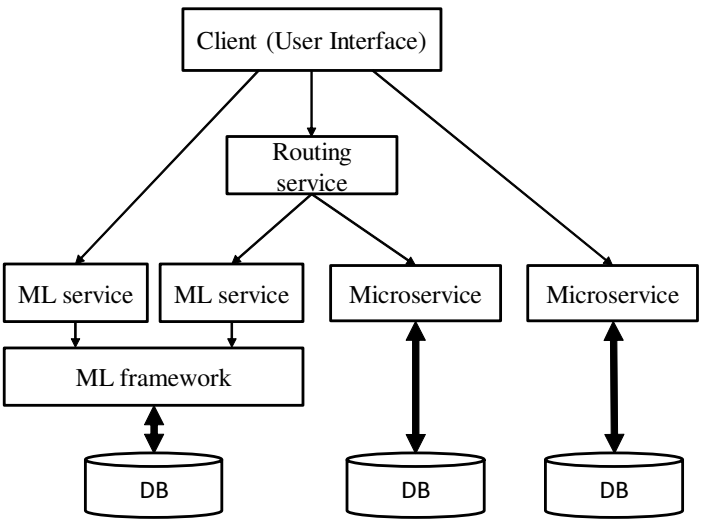


Fig. 5. Structure of the Microservices Architecture pattern.

### 4.6 Example

Amazon SageMaker[6] (see also AWS DeepRacer[7])

### 4.7 Discussion

The benefit of microservices is that data scientists, who may not be experts in software engineering, do not have to worry about installing/maintaining ML frameworks. They can also use the same set of microservices in different ML applications. Similarly, microservices can be used by different, competing ML applications. Hence, microservice providers can capitalize on their investment.

Unlike “Wrap Black-Box Packages into Common APIs”, ML applications depend on microservices and cannot immediately replace one ML framework with another (unless the microservice provider offers different frameworks). Thus, ML applications become dependent upon one service provider.

### 4.8 Related Patterns

“Daisy Architecture”, “Event-driven ML Microservices”, “Wrap Black-box Packages into Common APIs”, and ‘Separation of Concerns and Modularization of ML Components” (not described here).

## 5. ML VERSIONING

### 5.1 Source

[Wu et al. 2019; Amershi et al. 2019; Sculley et al. 2015]

### 5.2 Intent

Version ML models like any other software artifacts are tested for regressions and replication ease.

### 5.3 Context

ML applications may use (1) several models that (2) can change over time (new training).

[6] https://aws.amazon.com/sagemaker/
[7] https://aws.amazon.com/deepracer/

### 5.4 Problem

ML models and their different versions may change the behavior of the overall ML application.

### 5.5 Solution

Figure 6 shows the entire structure to solve the problem. Record the ML model structure, training dataset, training system and analytical code to ensure a reproducible training process and an inference process. Similar to other software artifacts, ML models should be versioned so that they can be tested and rolled back in case of regression.

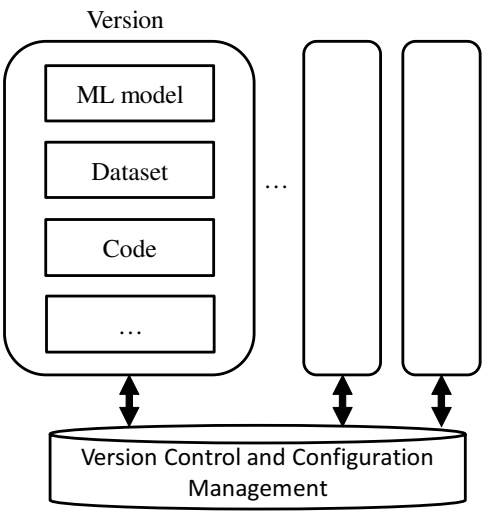


Fig. 6. Structure of the ML Versioning pattern.

### 5.6 Example

In the Eclipse Foundation library DP4J, models can be saved and loaded from the file system using the `org.deeplearning4j.rl4j.policy.DQNPolicy.save(String)` and `org.deeplearning4j.rl4j.policy.DQNPolicy.load(String)` methods. These methods can be used not only during training/production but also during versioning to store models and version them as any other software artifacts.

### 5.7 Discussions

To the best of our knowledge, previous works have not studied the performance and the sustainability of serialized ML models. It is unknown whether saving/loading models has a negative impact on the performances of ML applications. In addition, the long-term abilities to load and use ML models with new versions of (different) ML frameworks are unknown.

Although versioning the control dataset, training system, analytical code, and corresponding ML models is recommended to ensure the reproducibility, versioning all of them in a sophisticated and systematic manner can be difficult due to the size of the dataset and complicated configurations among the hyperparameters, training/inference code, etc. Two layers of version control can mitigate the issue by handling code and metadata

(of dataset and ML models) in a code storage such as Git[8], and handling large dataset and ML models in a data storage such as DVC (Data Version Control)[9] [Wu 2019].

### 5.8 Related Patterns

"Test the Infrastructure Independently from the Machine Learning" and "Canary Model" (not described here).

## 6. CONCLUSION

Herein three major ML architecture patterns ("Data Lake", "Distinguish Business Logic from ML Models", and "Microservice Architecture") and one major ML design pattern ("ML Versioning") are described. These pattern are from a set of 33 patterns identified through a SLR. In the future, we plan to write all 33 patterns in the standard pattern format to help developers adopt good practices and avoid bad practices described by these patterns.


## Acknowledgement

We are grateful to shepherd Prof. Nien Lin Hsueh for the careful review that improve this paper. We thank Mr. Hiromu Uchida, Prof. Naoshi Uchihira, Mr. Norihiko Ishitani, Dr. Takuo Doi, Dr. Shunichiro Suenaga, Mr. Yasuhiro Watanabe and Prof. Kazunori Sakamoto for their help. This work was supported by JST-Mirai Program Grant Number JP18077318, Japan.

Received December 2019; revised ; accepted


REFERENCES


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

Werner Daehn. 2017. A future-proof Big Data Architecture. https://blogs.sap.com/2017/10/10/future-proof-big-data-architecture/. (October 2017).

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

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

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

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, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Studying Software Engineering Patterns for Designing Machine Learning Systems. In *10th International Workshop on Empirical Software Engineering in Practice*. IEEE CS Press, Montréal, QC, Canada, 1–6.

Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2020. Machine Learning Architecture and Design Patterns (under review). (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


[8] https://git-scm.com/

[9] https://dvc.org/

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

Tianchen Wu. 2019. Version Control ML Model. https://towardsdatascience.com/version-control-ml-model-4adb2db5f87c. (August 2019).

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