Machine Learning Architecture and Design Patterns

Hironori Washizaki
Waseda University / National Institute of Informatics / SYSTEM INFORMATION / eXmotion

Hiromu Uchida
Waseda University

Foutse Khomh
Polytechnique Montréal

Yann-Gaël Guéhéneuc
Concordia University

***Abstract*—Researchers and practitioners studying best practices strive to design Machine Learning (ML) application systems and software that address software complexity and quality issues. Such design practices are often formalized as architecture and design patterns by encapsulating reusable solutions to common problems within given contexts. In this paper, software-engineering architecture and design (anti-)patterns for ML application systems are analyzed to bridge the gap between traditional software systems and ML application systems with respect to architecture and design. Specifically, a systematic literature review confirms that ML application systems are popular due to the promotion of artificial intelligence. We identified 32 scholarly documents and 48 gray documents out of which 38 documents discuss 33 patterns: 12 architecture patterns, 13 design patterns, and 8 anti-patterns. Additionally, a survey of developers reveals that there are 7 major architecture patterns and 5 major design patterns. Then the relationships among patterns are identified in a pattern map.**

■ **THE POPULARITY OF** ML techniques has increased in recent years. ML is used in many domains, including cyber security, IoT, and autonomous cars. ML techniques rely on mathematics and software engineering. The former generates algorithms, develops capabilities to learn from input data, and produces representative models. The latter is employed for implementation and performance.

Although many works investigated the mathematics and computer science on which ML techniques are built, few have examined their implementation. This situation raises many concerns. The first is software complexity of ML techniques. The second is quality of the available implementations, including performance and reliability. The third is model quality, which may be negatively impacted by software bug. These concerns could be alleviated if developers could demonstrate the software quality of their implementations. Consequently, researchers and practitioners have been studying best practices to design ML application systems to address issues with software complexity and quality of ML techniques. Such practices are often formalized as architecture patterns and design patterns. These patterns encapsulate reusable solutions to commonly occurring problems within ML application systems and software design.

Herein we report the results of a systematic

 © 2020 IEEE

literature review (SLR) of good/bad design patterns for ML. Based on the results, we also report on developers' perceptions as well as relationships among extracted ML patterns[1].

How the Literature Addresses Software Engineering ML Design Patterns

We performed a SLR of both academic and gray literature to collect SE good (bad) design patterns for ML application systems and software. For the academic literature, we chose Engineering Village, which is a search platform that provides access to 12 engineering document databases such as Ei Compendex and Inspec. Engineering Village can search in all recognized scholarly engineering journals, conferences, and workshop proceedings with a unique search query. Moreover, Engineering Village automatically detects and removes most duplicative search results. On August 14, 2019, we designed and used the following query specifying "pattern" as well as keywords related to patterns to search for documents addressing ML design practice: ((((system) OR (software)) AND (machine learning) AND ((implementation pattern) OR (pattern) OR (architecture pattern) OR (design pattern) OR (anti-pattern) OR (recipe) OR (workflow) OR (practice) OR (issue) OR (template))) WN ALL) + ((cpx OR ins OR kna) WN DB) AND ((ca OR ja OR ip OR ch) WN DT).

For the gray literature, we used a Google search on August 16, 2019. The query was the same as that for the academic literature: (system OR software) "Machine learning" (pattern OR "implementation pattern" OR "architecture pattern" OR "design pattern" OR anti-pattern OR recipe OR workflow OR practice OR issue OR template) and "machine implementation pattern" OR architecture pattern" OR "design pattern" OR anti-pattern OR recipe OR workflow OR practice OR issue OR template.

We retrieved 32 scholarly documents and 48 gray literature documents. For each document, two of the authors vetted whether it should be included in our SLR or not. The titles and abstracts were initially reviewed. Then the entire document was read to determine whether the document pertained to software-engineering practices for ML application systems. This process identified 19 scholarly documents and 19 gray documents. All the data are available on-line[2].

Figure 1 shows the trend in the number of documents related to design for ML application systems in the past decade. ML application systems have recently become popular due to the promotion of artificial intelligence. Since 2008, academic and gray documents have discussed good (bad) practices of ML application systems design.

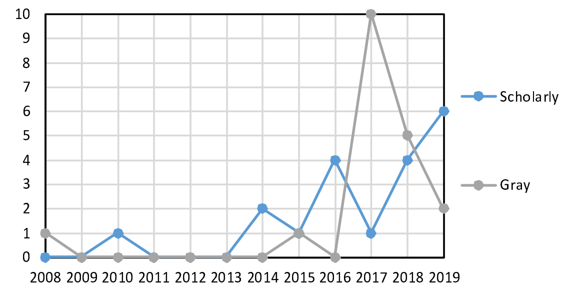

Figure 1. Numbers of Documents per Year

Overview of ML Patterns

Two of the authors read half of the documents. Each author extracted patterns independently. Then the remaining author vetted each pattern. Although 69 patterns related to the architecture and design of ML application systems were initially identified, only 33 remained after the vetting process.

In general, systems and software design processes have two major phases [2] with different abstraction levels: architectural design and detailed design. Similarly, the extracted patterns can be classified into two types: ML architecture patterns and ML design patterns. Documents describing ML architecture patterns recommend architecture designs of ML application systems and software to address recurrent architectural problems such as ensuring maintainability of ML components. In contrast, ML design patterns address recurrent detailed design problems such as enabling proper communications among specific modules.

[1]Preliminary results of our SLR is presented at [1]. In this paper, we examined all patterns and relationships among patterns in detail. We also newly studied developers' perceptions on patterns.

[2]http://www.washi.cs.waseda.ac.jp/ml-patterns/

Table 1. Extracted Architecture Patterns (NP: Number of participants who used the pattern.)

Pattern Name	Problem (excerpt)	Solution (excerpt)	NP and Source
Data Lake	We cannot foresee the kind of analyses that will be performed on the data and which frameworks will be used to perform these analyses.	The data ranging from structured data to unstructured data should be stored as "raw" as possible and the centralized data repository should allow parallel analyses of different kinds and with different frameworks.	5, http://bit.ly/33DTKTe
Distinguish Business Logic from ML Models	The overall business logic should be isolated as much as possible from the ML models so that they can be changed/overridden when necessary without impacting the rest of the business logic.	Separate the business logic and the inference engine, loosely coupling the business logic and ML-specific dataflows.	4, [3]
Microservice Architecture	ML applications may be confined to some "known" ML frameworks and miss opportunities for more appropriate frameworks.	Data scientists working with or providing ML frameworks can make these frameworks available through microservices.	4, http://bit.ly/2DyHGrV
Data-Algorithm-Serving-Evaluator	Prediction systems should connect different pieces in the data processing pipeline into one coherent system and prototyping predictive model.	Separate the following like MVC for ML: data (data source and data preparator), algorithm(s), serving, and evaluator.	2, http://bit.ly/2r6edmu
Event-driven ML Microservices	Due to frequent prototyping of ML models and constant changes, development teams must be agile to build, deploy, and maintain complex data pipelines.	Construct pipelines by chaining together multiple microservices, each of which listens for the arrival of some data and performs its designated task.	2, http://bit.ly/2OZDuXH
Lambda Architecture	Real-time data processing requires scalability, fault tolerance, predictability, and other qualities. It must be extensible.	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.	2, http://bit.ly/33DTKTe
Parameter-Server Abstraction	For distributed learning, widely accepted abstractions are lacking.	Distribute both data and workloads over worker nodes, while the server nodes maintain globally shared parameters, which are represented as vectors and matrices.	2, [4]
Daisy Architecture	The ability to scale content production processes must be acquired via the use of ML. Then the coverage of that tooling must be extended over as much of their remaining content.	Utilize Kanban, scaling, and microservice to realize pull-based, automated, on-demand, and iterative processes.	1, http://bit.ly/2DyHGrV
Gateway Routing Architecture	When a client uses multiple services, it can be difficult to set up and manage individual endpoints for each service.	Install a gateway before a set of applications, services, or deployments and use application layer routing requests to the appropriate instance.	1, [3]
Kappa Architecture	It is necessary to deal with huge amount of data with less code resource.	Support both real-time data processing and continuous reprocessing with a single stream processing engine.	1, http://bit.ly/37Xkguc
Closed-Loop Intelligence	It is necessary to address big, open-ended, time-changing or intrinsically hard problems.	Connect machine learning to the user and close the loop. Design clear interactions along with implicit and direct outputs.	0, http://bit.ly/2L8ZpdB
Federated Learning	Standard machine learning approaches require centralizing the training data on one machine or in a datacenter.	Employ Federated Learning, which enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on the device.	0, http://bit.ly/2qaRJk3

In addition to the ML architecture patterns and design patterns, we also identified ML anti-patterns. Similar to general anti-patterns [5], ML anti-patterns (including code smells and technical debts [6]) describe commonly occurring situations that generate negative consequences in ML application systems and software design.

Tables 1, 2 and 3 list the extracted ML patterns. Of these, 18 (55%) were extracted from the scholarly documents, while 15 (45%) were from the gray documents. Twelve (36%) are ML architecture patterns, and thirteen (39%) are ML design patterns, suggesting that the unique nature of design in ML application systems and software appears at both of the architecture level and the detailed design level. The remaining are 8 ML anti-patterns (24%).

How Engineers' Perceived Software Engineering ML Design Patterns

ML techniques are concrete solutions to practical problems. Hence, ML developers may already have built a body of knowledge on the good (bad) design practices of ML development. To clarify how ML developers perceive existing ML patterns, we asked 19 developers and researchers

Table 2. Extracted ML Design Patterns (NP: Number of participants who used the pattern.)

Pattern Name	Problem (excerpt)	Solution (excerpt)	NP and Source
ML Versioning	ML models and their several versions may change the behaviour of the overall ML applications.	Record the ML model structure, training data, and training system to ensure a reproducible training process.	4, [7]
Wrap Black-Box Packages into Common APIs	Using generic, independent ML frameworks often results in different glue code for each framework, for which a massive amount of supporting code is written to get data into and out of the framework from and to the rest of the application.	Wrap black-box packages into common APIs to make supporting infrastructure more reusable and to reduce the cost of changing packages.	4, [4]
Test Infrastructure Independently from ML	It is difficult to identify errors when infrastructure and machine learning are mixed.	Ensure that the infrastructure is testable and the learning parts of the system are encapsulated so that everything around it can be tested.	3, http://bit.ly/34zt2wx
Handshake (Hand Buzzer)	A ML system depends on inputs delivered outside of the normal release process.	Create a handshake normalization process, regularly check for significant changes, and send ALERTS.	2, http://bit.ly/2qdsWvG
Isolate and Validate Output of Model	Machine learning models are known to be unstable and vulnerable to adversarial attacks and to noise in data and data drift overtime.	Encapsulate ML models within rule-base safeguards and use redundant and diverse architecture that mitigates and absorbs the low robustness of ML models.	2, [8]
Canary Model	A surrogate ML that approximates the behavior of the best ML model must be built to provide explainability.	Run the canary inference pipeline in parallel with the primary inference pipeline to monitor prediction differences.	1, http://bit.ly/35U0C0i
Decouple Training Pipeline from Production Pipeline	It is necessary to separate and quickly change the ML data workload and stabilize the training workload to maximize efficiency.	Physically isolate different workloads to different machines. Then optimize the machine configurations and the network usage.	1, [7]
Descriptive Data Type for Rich Information	The rich information used and produced by ML systems is often encoded with plain data types like raw floats and integers.	Design a robust system, where the model parameter knows if it is a log-odds multiplier or a decision threshold, and a prediction knows information about the model.	1, [4]
Design Holistically about Data Collection and Feature Extraction	The system to prepare data in an ML-friendly format may become a pipeline jungle. Managing these pipelines is difficult and costly.	Avoid pipeline jungles by thinking holistically about data collection and feature extraction that can dramatically reduce ongoing costs.	1, [4]
Reexamine Experimental Branches Periodically	The code-paths accumulated by individual changes can create a growing debt due to the increasing difficulties of maintaining backward compatibility.	Reexamine each experimental branch periodically to see what can be removed to eliminate glue code and pipeline jungles.	1, [4]
Reuse Code between Training Pipeline and Serving Pipeline	Training-serving skew can be caused by a discrepancy between how data in the training and serving pipelines are handled.	Reuse code between training pipeline and serving pipeline by preparing objects that store results in an understandable way for humans.	0, http://bit.ly/34zt2wx
Separation of Concerns and Modularization of ML Components	ML applications must accommodate regular and frequent changes to their ML components.	Decouple at different levels of complexity from simplest to most complex.	0, [9]
Secure Aggregation	The system needs to communicate and aggregate model updates in a secure, efficient, scalable, and fault-tolerant way.	Encrypt data from each mobile device in Federated learning and calculate totals and averages without individual examination.	0, http://bit.ly/2qaRJk3

at Japanese companies and research organizations to complete a survey during a workshop in October 2019. After a brief introduction of all patterns, we inquired on whether or not they used any of the ML architecture and design patterns.

As shown in Table 1, multiple participants used the seven major ML architecture patterns (out of 12): “Data-Algorithm-Serving-Evaluator”, “Data Lake“, “Distinguish Business Logic from ML Models”, “Microservice Architecture”, “Event-driven ML Microservices”, “Lambda Architecture”, and “Parameter-Server Abstraction”. And, all ML architecture patterns except for “Federated Learning” and “Closed-Loop Intelligence” are used at least by one participant.

Table 3. Extracted ML Anti-Patterns

Pattern Name	Problem (excerpt)	Source
Big Ass Script Architecture	When all code is placed in one big ass script, it becomes difficult to reuse in future analysis, understand how it works, and debug.	http://bit.ly/35QPb9N
Abstraction Debt	For distributed learning, widely accepted abstractions are lacking.	[4]
Dead Experimental Codepaths	The code-paths accumulated by individual changing can create a growing debt due to the increasing difficulties of maintaining backward compatibility.	[4]
Glue Code	Glue code is costly in the long term because it tends to freeze a system to the peculiarities of a specific package.	[4]
Multiple-Language Smell	Using multiple languages increases the cost of effective testing and can increase the difficulty of transferring ownership to other individuals.	[4]
Pipeline Jungles	The system to prepare data in an ML-friendly format may become a pipeline jungle, and managing these pipelines is difficult and costly.	[4]
Plain-Old-Data Type Smell	The rich information used and produced by ML systems is often encoded with plain data types like raw floats and integers.	[4]
Undeclared Consumers	Undeclared consumers are dangerous because they create a hidden tight coupling of model MA to other parts of the stack.	[4]

In contrast, multiple participants used the five major ML design patterns (out of 13 in Table 2): "Handshake", "Isolate and Validate Output of Model", "ML Versioning", "Test Infrastructure Independently from ML", and "Wrap Black-Box Packages into Common APIs". In addition, none of the participants reported using the three ML design patterns, suggesting that ML design patterns are less accepted in comparison to ML architecture patterns.

Example of Major Software Engineering ML Pattern

Here, we describe one major ML architecture pattern and its usage. We selected "Distinguish Business Logic from ML Model" since it was popular among our survey participants. Moreover, it provides a clear decomposition of a ML system in multiple layers and components. For brevity, participants, collaborations, implementation, and known uses are omitted.

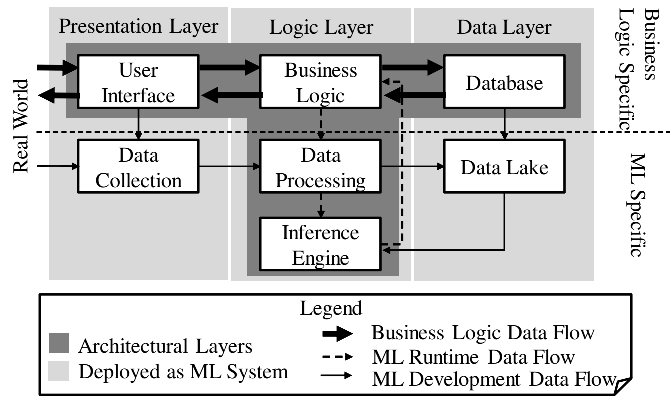

Figure 2. Structure of Distinguish Business Logic from ML Model pattern [3]

Pattern Name
Distinguish Business Logic from ML Model (originally named as "Multi-Layer Architectural Pattern" [3])

Intent
Isolate failures between business logic and ML learning layer to help developers debug ML application systems easily.

Also Known As
Machine Learning System Architectural Pattern for Improving Operational Stability.

Problem
ML application systems are complex because their ML components must be (re)trained regularly and have an intrinsic non-deterministic behavior. Similar to other systems, the business requirements for these systems and the ML algorithms change over time.

Solution
Define clear APIs between the traditional and ML components. Place the business and ML components with different responsibilities into three layers (Fig. 2). Divide data flows into three.

Applicability
It is applicable to any ML application system with outputs that depend on ML techniques.

Consequences
Decoupling "traditional" business and ML components allows the ML components to be monitored and adjusted to meet users' requirements and changing inputs.

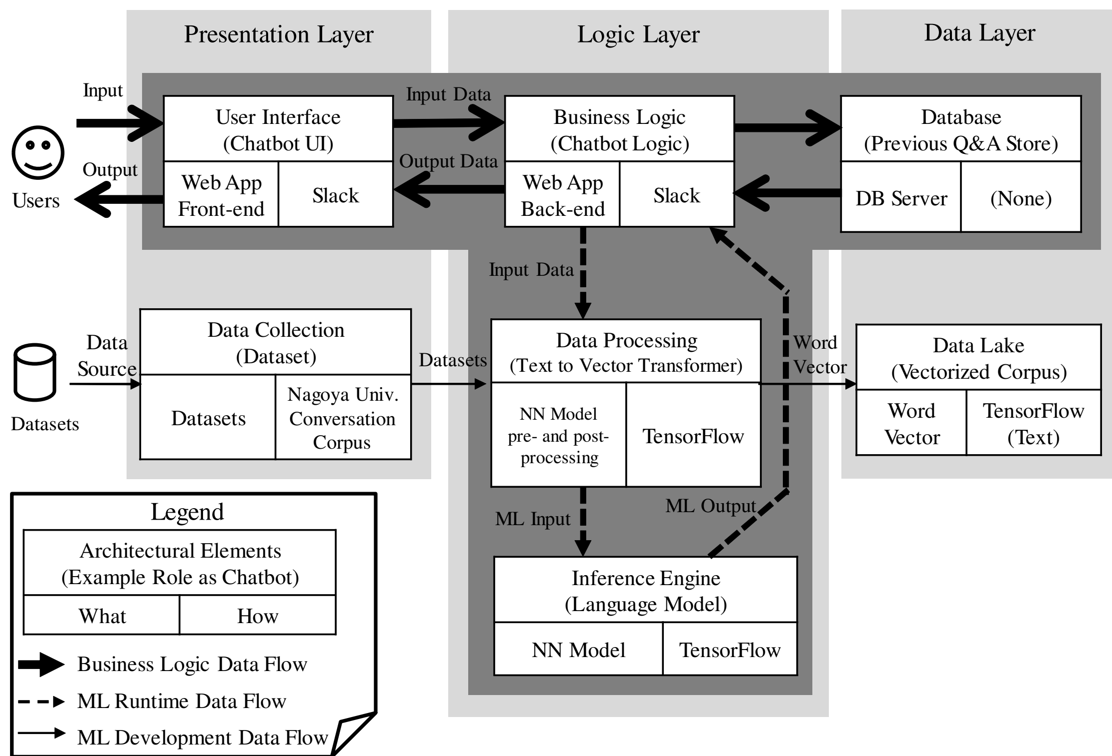

Figure 3. Example of Chatbot System Architecture by applying "Distinguish Business Logic from ML Model"

Usage Example
Figure 3 presents an example of implementation of the pattern "Distinguish Business Logic from ML Model" in a Slack-based Chatbot system. By referring to the architecture pattern, we easily specified necessary elements as well as relationships among them while having clear separation between the Chatbot service (as the business logic) and the underlying ML components.

ML Pattern Map

To help developers navigate the patterns, we identified the following four types of relationships among the patterns using basic relation types [10]: X is similar to Y but has different objectives, X can use Y in its solution, X and Y can be combined to solve larger problems, and, X can mitigate the problem of Y.

Figure 4 shows a result of identification of the relationships among ML patterns as a pattern map. For example, "Closed-Loop Intelligence" is an architecture pattern to have clear interactions with users; it would mitigate the problem of "Undeclared Customers". "Closed-Loop Intelligence" can be combined with other high-level architecture patterns that address business logic and user interactions such as "Distinguish Business Logic from ML Models" and "Data-Algorithm-Serving-Evaluator". "Gateway Routing Architecture" is similar to "Distinguish Business Logic from ML Models" since both use "Data Lake" in their solutions to handle variety of data; however, objectives of these two architecture patterns are quite different.

According to the number of connected relationships, the fundamental patterns are "Big Ass Script Architecture" and "Separation of Concerns and Modularization of ML Components", suggesting that developers should initially identify the corresponding problems in their design and solve them by referring to these patterns as well as connected related patterns.

In terms of the anti-patterns, we suggest that developers should refactor their code as soon as these symptoms appear by applying corresponding design patterns connected in the pattern map.

CONCLUSION

To bridge the gap between traditional software systems and ML application systems with respect

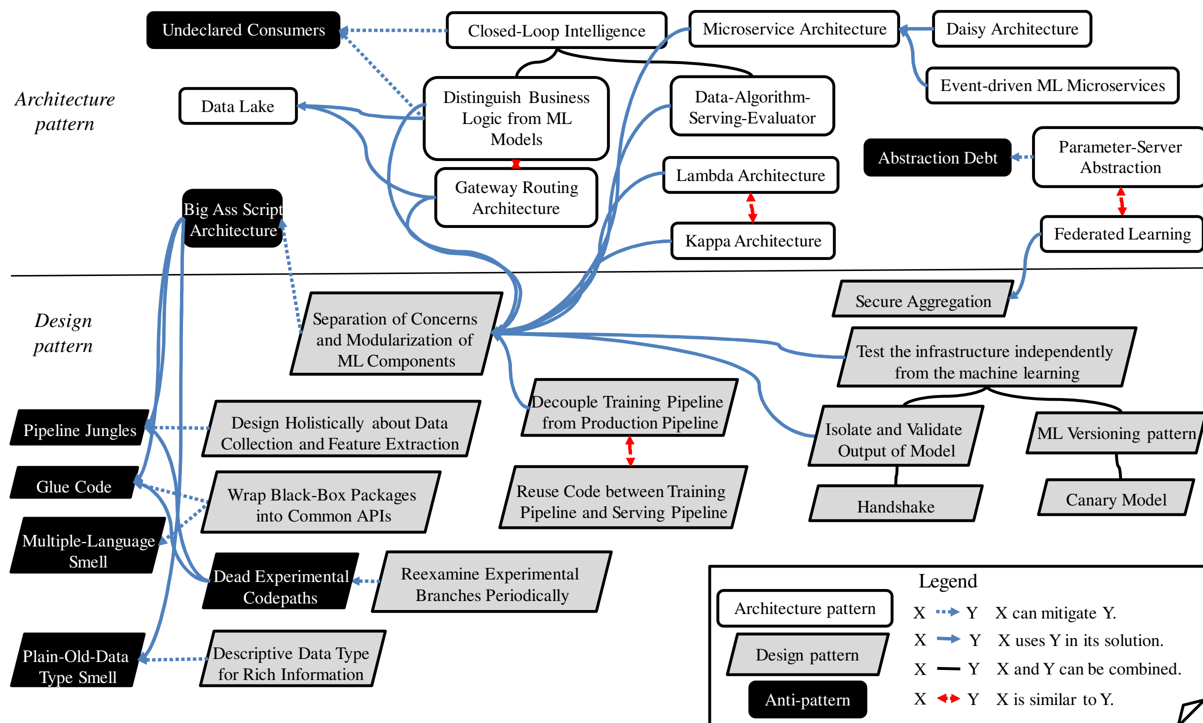

Figure 4. Pattern Map showing classifications and relationships among ML patterns

to architecture and design, software-engineering architectural and design (anti-)patterns for ML application systems were analyzed via an SLR and a survey of developers. ML application systems are quite popular due to the recent promotion of artificial intelligence. From the 32 scholarly documents and 48 gray documents identified in the SLR, 12 ML architecture patterns, 13 ML design patterns, and 8 ML anti-patterns were identified. A survey of developers reveals that there are 7 major ML architecture patterns and 5 major ML design patterns. The relationships among the patterns were elucidated.

In the future, we plan to write all patterns into a standardized format because not all the identified patterns are well written. Additionally, we intend to investigate the impact of SE patterns on the quality attributes of ML application systems. These patterns will be validated by not only sharing these patterns to participants at the Writers' Workshop at the PLoP conference series but also by contacting the original designer. We also plan to involve more practitioners from the community by continuing the survey and the SLR.

ACKNOWLEDGMENT

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 helps. This work was supported by JST-Mirai Program Grant Number JP18077318, Japan.

REFERENCES

1. H. Washizaki, H. Uchida, F. Khomh, and Y.-G. Guéhéneuc, "Studying software engineering patterns for designing machine learning systems," in *The 10th International Workshop on Empirical Software Engineering in Practice (IWESEP 2019), Tokyo, Japan*, 2019, pp. 1–6.
2. International Organization for Standardization, "ISO/IEC 12207:2008 Information technology – Software life cycle processes, institution = International Organization for Standardization," ISO/IEC, Tech. Rep., 2017.
3. H. Yokoyama, "Machine learning system architectural pattern for improving operational stability," in *IEEE International Conference on Software Architecture Companion, ICSA Companion 2019, Hamburg, Ger-*

many, March 25-26, 2019, 2019, pp. 267–274. [Online]. Available: https://doi.org/10.1109/ICSA-C.2019.00055

4. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015, pp. 2503–2511. [Online]. Available: http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems
5. W. J. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley & Sons, 1998.
6. W. Cunningham, "The wycash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993. [Online]. Available: https://doi.org/10.1145/157710.157715
7. C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine learning at facebook: Understanding inference at the edge," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, 2019, pp. 331–344. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00048
8. M. Kläs and A. M. Vollmer, "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*, 2018, pp. 431–438. [Online]. Available: https://doi.org/10.1007/978-3-319-99229-7_36
9. M. S. Rahman, E. Rivera, F. Khomh, Y. Guéhéneuc, and B. Lehnert, "Machine learning software engineering in practice: An industrial case study," *CoRR*, vol. abs/1906.07154, 2019. [Online]. Available: http://arxiv.org/abs/1906.07154
10. W. Zimmer, "Pattern languages of program design," J. O. Coplien and D. C. Schmidt, Eds. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, ch. Relationships Between Design Patterns, pp. 345–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=218662.218687

Hironori Washizaki is the Director and a Professor with the Global Software Engineering Laboratory, Waseda University. He is also a Visiting Professor with the National Institute of Informatics, Tokyo as well as an Outside Director of System Information, Tokyo and eXmotion, Tokyo.

Hiromu Uchida is with Waseda University, Tokyo, Japan. Contact him at eagle_h.21@toki.waseda.jp.

Foutse Khomh is with Polytechnique Montréal, Canada. He is an Associate Professor with the Polytechnique Montréal, where he heads the SWAT Laboratory, and is involved with software analytics and cloud engineering research. Contact him at foutse.khomh@polymtl.ca.

Yann-Gaël Guéhéneuc is with Concordia University, Canada. He has been a Full Professor with the Department of Computer Science and Software Engineering, Concordia University since 2017, where he leads the Ptidej Team on evaluating and enhancing the quality of software systems, focusing on the Internet of Things and researching new theories, methods, and tools to understand, evaluate, and improve the development, release, testing, and security of such systems. Contact him at yann-gael.gueheneuc@concordia.ca.