2019 IEEE International Conference on Software Architecture Companion (ICSA-C)

Machine Learning System Architectural Pattern for Improving Operational Stability

Haruki Yokoyama
FUJITSU LABORATORIES LTD.
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211-8588, Japan
Email: yokoyama.haruki@fujitsu.com

***Abstract*—Recently, machine learning systems with inference engines have been widely used for a variety of purposes (such as prediction and classification) in our society. While it is quite important to keep the services provided by these machine learning systems stable, maintaining stability can be difficult given the nature of machine learning systems whose behaviors can be determined by program codes and input data. Therefore, quick troubleshooting (problem localization, rollback, etc.) is necessary. However, common machine learning systems with three-layer architectural patterns complicate the troubleshooting process because of their tightly coupled functions (e.g., business logic coded from design and inference engine derived from data). To solve the problem, we propose a novel architectural pattern for machine learning systems in which components for business logic and components for machine learning are separated. This architectural pattern helps operators break down the failures into a business logic part and a ML-specific part, and they can rollback the inference engine independent of the business logic when the inference engine has some problems. Through a practical case study scenario, we will show how our architectural pattern can make troubleshooting easier than common three-layer architecture.**

1. Introduction

Machine learning (ML) techniques are algorithms such as regression, classification, and anomaly detection by learning behavior from a large amount of data. The tasks of generating media data such as text, images, and voices have not been realized because it is difficult to define their specifications clearly. However, the appearance of deep learning and the growth of the computational resource made generating them possible [17].While it is quite important to keep the services provided by these machine learning systems stable, maintaining stability can be difficult given the nature of machine learning systems whose behaviors can be determined by both program codes and input data.

The instability of input data threatens operational stability, resulting in problems such as model staleness [6] and missing values [3]. Model staleness is when inference engine performance decreases because of changes in input data trends. For example, a content recommender system learns users' preferences and recommends content to them. However, when the frequency of learning and deploying the system is too low to catch up with the latest content, the system cannot recommend the latest content and it is not useful. In addition, failures occur in the ML system because some missing values beyond the scope of the assumption appear in the input data. For example, when the schema of input data is suddenly changed, the ML system cannot recognize the data, leading to errors in the behavior of the ML system.

The aforementioned operational issues with the ML system are caused by the short-term factor brought by changing data formats, the source code, and the hyperparameters and the mid/long-term factor brought by changing data trends. Problems based on both short-term and mid/long-term factors need to be solved in order to improve the operation stability of the ML system. Conducting quick troubleshooting, such as problem localization and rollback [6], can address short-term factors. Conducting regular time series analysis [11] of input data and internal log data can address mid/long-term factors. Even when conducting either, the output of each element of the pipeline must be monitored and the alert in the system must be set up. To apply the practice that covers the entire operation process of the system to the ML system, we examine it at the architectural level.

Multilayer architectural patterns such as client/server architecture and three-layer architecture is adopted in many enterprise applications [8]. The application of the ML system has not been discussed enough. In the ML system with a three-layer architectural pattern, for instance, the business logic can infer behavior from data. Then, the data dependency of the inference in the business logic might propagate in the entire business logic. Data dependency is one of the main technical debts [19] in the ML system. For example, when data trends change, the behavior of the ML system also changes and sometimes becomes erroneous. If the entire business logic of the ML system has data dependency, it is difficult to localize the erroneous part of the ML system.

Moreover, when the inference engine is tightly coupled with the business logic, the change in the inference engine influences other parts of the business logic and might cause new trouble. This is cumbersome when only rolling back the inference engine because of its low performance. Even if the inference engine developed with only an ML library

978-1-7281-1876-5/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA-C.2019.00055

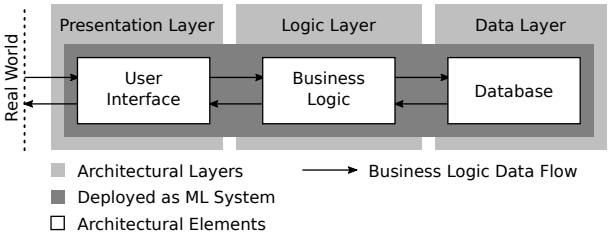

Figure 1. Three-layer Architectural Pattern

is quite loosely coupled with the business logic in early development, the coupling tightens as the development scale increases because of the addition of code for extracting new features, validating the values, and writing ML logic using some functions of business logic.

Therefore, we propose a novel architectural pattern that improves the operational stability of machine learning systems. Our proposed architectural pattern can separate the business logic and the inference engine, loosely coupling the business logic and ML-specific dataflows. This architectural pattern helps the operators break down the failures into a business logic part and a ML-specific part, and they can rollback the inference engine independent of the business logic when the inference engine has some problems. Through a practical case study scenario, we will show how our architectural pattern can make problem localization and rollback easier than common three-layer architecture.

Our potential contributions are the following.

- We propose a novel architectural pattern for machine learning systems and show how it eases troubleshooting and improves operational stability.
- We show an example of ML architecture that facilitates deployment of the inference engine and enables continuous deployment [9] of the inference engine that has the best performance at runtime.

The rest of the paper is structured as follows. We describe potential problems with the existing multi-layer architectural patterns when developers apply them to the ML system in Section 2. We describe our architectural pattern in Section 3 and show how well it troubleshoots the ML system in Section 4. We describe our research agenda in Section 5. We describe related work and clarify how our research differs in Section 6. Finally, we conclude our research and describe our future work in Section 7.

2. Multi-Layer Architectural Pattern

Multi-layer architectural patterns are some of the most frequently used architectural patterns [8]. In this section, we consider three-layer architectural patterns as an instance of multi-layer architectural patterns. We show a three-layer architectural pattern in Figure 1. It has the following three layers.

- Presentation layer: interacts with users in the real world (e.g., user interface).
- Logic layer: processes the core logic of the system (e.g., business logic, application, etc.).
- Data layer: provides data access and persistence (e.g., database).

Multi-layer architectural patterns have been applied in client/server systems, web systems, etc. However, this architectural pattern has problems when developers apply it to the ML system. We will describe the problems each layer faces.

Presentation layer. Some data collection for the inference engine is for purposes other than interacting with the users or the external systems. However, the data collection is actually an unused part of the system at runtime. If data collection and the user interaction is tightly coupled, the deploy cost increases unnecessarily because data collection that does not need to deployed is deployed at the same time.

Logic layer. The behavior in the logic layer is not only predetermined but also inferred from data. We call the latter behavior the inference engine. The inference engine in the ML system is learned from data. This is dramatically different from the development of usual business logic. So, the business logic may have dependencies on data from the input data source when the business logic and the inference engine are treated as the same element. In such architecture, problems derived from code and data are too mixed to debug the logic layer. It is necessary to divide the logic layer into the inference engine that is actually inferred from data and the data processing part and the business logic. To perform analysis, data processing should arrange the data of the input data source independently of the business logic. This is because the procedure that processes collected data and stores it in the data lake (as stated below) strongly depends on the input data source. Similarly, the data processing should be independent of the inference engine because the inference engine learns from data, unlike the data processing that is developed by coding based on the software design.

Data layer. A part provides data access and persistence. In the ML system, the developer also needs various types of data to develop the inference engine. Therefore, the developer prepares a data lake that is used to develop the inference engine beside the common database that is accessed at runtime. The data lake is a repository mainly for data that is analyzed, and it contains both structured data and semi/unstructured data such as access logs and form data of UI. The data lake helps developers comprehend features and structures of data via exploratory data analysis and helps them construct datasets to generate an inference engine. On the other hand, if the inference engine can be developed, the data lake is an unused part of the ML system at runtime. Therefore, as in the case of the presentation layer, the deploy cost will increase unnecessarily unless the data lake is separated from the database.

Based on the above discussion, the requirements for ML system architecture are as follows.

- The data collection should be separated from the user interface.

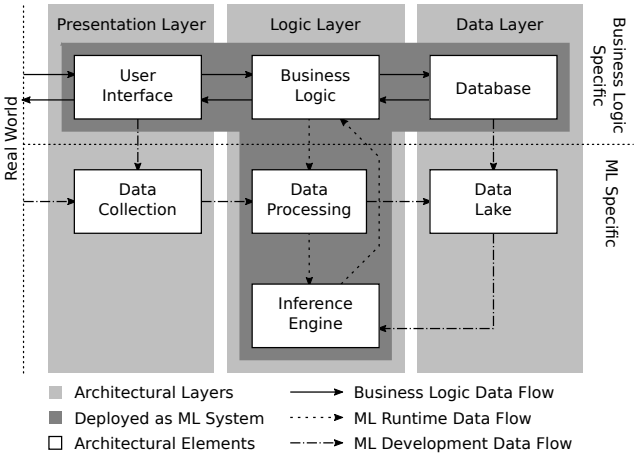

Figure 2. Proposed ML Architectural Pattern

- The data processing and the inference engine should be separated from the business logic.
- The data lake should be separated from the database.

3. Architectural Pattern for ML System

In this section, first, we will show a novel architectural pattern for an ML system. Next, we will show an alternative architectural pattern. Then, we will discuss which pattern is suitable for the ML system in terms of team building, coupling, and maintenance costs. Then, we will show a concrete example of the ML system instantiated by our architectural pattern.

3.1. Proposed ML Architectural Pattern

To address the aforementioned problems, we propose the ML architectural pattern shown in Figure 2. This architectural pattern has each architectural element contained in a layer that is divided into elements that have different responsibilities. This division makes four new elements as follows.

- Data collection: collecting data from the input data source
- Data processing: converting collected data into data available for machine learning and data analysis
- Inference engine: inferring behavior from processed data
- Data lake: continuously storing preprocessed data and providing data access for machine learning and data analysis

The ML system has a more complicated data flow than a traditional one. The data flow in this architecture can be divided into the following three data flows.

- Business logic data flow
- ML runtime data flow
- ML development data flow

In the business logic data flow, first, a user inputs data via the user interface. Next, the business logic receives and processes the input data (referring and updating the database if necessary), and it returns the output data. Then, the user views the output data via the user interface. This is the same data flow structure as three-layer architecture.

In the ML runtime data flow, first, the business logic provides the user input to the data processing. Next, the data processing converts the user input into a specific format that the inference engine can load. Then, the inference engine returns the results to the business logic.

In the ML development data flow, first, the data collection receives input via the user interface or input data sources (such as web crawlers, logs, and datasets) and provides them to the data processing. Next, the data processing converts them into a specific format and stores the processed data in the data lake. Then, the developers generate an inference engine based on the data accumulated in the data lake—namely machine learning.

3.2. Discussion of ML Architectural Design

Which architectural pattern is the most suitable for ML system architecture? In this section, we discuss the pros and cons of the following designs.

- Three-layer architecture shown in Figure 1.
- Our architecture shown in Figure 2.
- Gateway-routing architecture [14] shown in Figure 3.

In the three-layer architecture shown in Figure 1, the ML specific code is included in the business logic. This simple architecture is useful when the overall logic of the system is only ML code or all the ML code is written only with ML libraries. However, as business logic becomes large scale, the inference engine is tightly coupled with the business logic. As mentioned in Section 1, when the inference engine is tightly coupled with the business logic, the change in the inference engine influences other parts of the business logic and can cause new problems.

In our architecture shown in Figure 2, the inference engine and data processing are separated with the business logic. This architecture helps the developers avoid tight coupling among components.

Another possible ML system architectural pattern is the gateway-routing architectural pattern shown in Figure 3. We assume that a typical ML system has both business logic and an inference engine. In this pattern, the business logic and the inference engine with the data processing are implemented as services. This pattern can avoid tight coupling between the business logic and the inference engine. However, this pattern makes extra overhead between the user interface and the business logic, and the latency performance of the system might decrease.

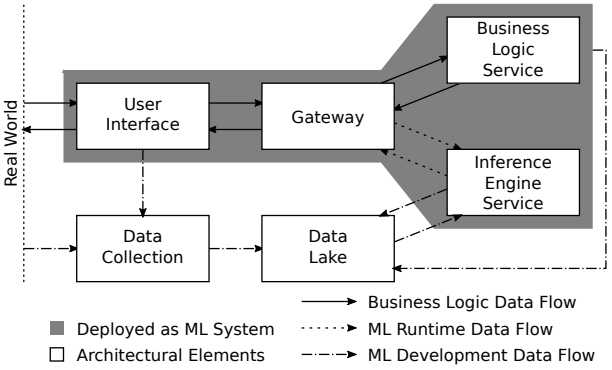

Figure 3. Gateway-Routing Architectural Pattern

3.3. Example of ML System Architecture Implementation

Whether it is better to first make a tightly coupled system and later divide it or to first make a loosely coupled system and later unite some parts to make them suitable for constructing the ML system depends on the effort required for the rough tight coupling analysis for division compared to that required for the bottleneck analysis for uniting. Therefore, the developers should decide which architectural pattern to adopt for the ML system based on its scale, complexity, and non-functional requirements such as performance and operational stability. Here, we use the ML system architecture shown in Figure 4 as an example of our architectural pattern that will construct the chatbot system. First, we show architectural elements; then, we describe data flows of the chatbot system. This chatbot has the following architectural elements.

- User interface (UI): It presents some pre-defined questions expected by the users. The users who could not find questions that they wanted to ask in the list of questions were also provided an input form to input their own questions using natural language. When the users input their questions via the UI, the chatbot displayed an answer. It was developed as a front-end web application written in HTML5 and JavaScript.
- Business logic: It interacts with the user interface and generates answers to the questions. If the users' questions were known questions that were displayed as the choices in the UI, the business logic got the answers from the database. If the users' questions used natural language, it generated answers using the inference engine. It was developed as a back-end web application written in various languages and web frameworks such as JavaScript (Node.js) and Express, respectively.
- Database (DB): It stores frequently asked questions and provides answers to the business logic. It was deployed as a DB server such as MySQL and MongoDB.
- Data collection: It collects various statements. It was developed as a web crawler of the natural language corpus and a log crawler of the questions and user feedback via monitoring. It may be datasets of the natural language corpus. The crawler is written in languages such as Python and Ruby. We can also use Goods [10], which organizes datasets, logs, source code repositories, and databases for data collection.
- Data processing: It converts natural language text into word vectors and puts them into the inference engine. It also converts various statements, the question sentence, and user feedback from the data collection into the word vectors and stores them in the data lake. This is implemented as an API Server written in languages commonly used in ML-related processes (such as Python). We can also use TFX [4], which supports data pre-processing and post-processing for the inference engine generated by the TensorFlow [1] ML software library.
- Data lake: It stores various statements, questions, and user feedback, and it provides a data analysis environment in which the developer can do exploratory data analysis (EDA), such as summarize, visualize and find new knowledge. For example, Cassandra is used as data storage and Spark is used for data analysis. We can also use DataHub [5], which provides an EDA environment for multiple users.
- Inference engine: It infers the answer statement from the question word vector. This is an application of the language model using neural networks such as seq2seq [21]. ML software libraries such as TensorFlow [1] provide many ML algorithms that are used to generate the inference engine with learning datasets from the data lake.
- Comparative evaluation: It deploys the inference engine. It is typically realized with continuous integration or a continuous delivery environment (CI/CD) and a version control system (VCS) such as TravisCI and Git/GitHub, respectively. We also use DVC [16] to support version control of data, hyperparameters, and the source code.
- Monitoring and alert: It monitors each data flow in the ML system. It alerts the operator of the system when an unexpected value is detected. Each monitor element associates with each architectural element as a cross-cutting concern. If the architectural elements are shipped as containers, a sidecar pattern [7] can be applied, and the pattern can collect the monitoring results from each architectural element. It is called "distributed monitoring" supported by Zabbix [15], Prometheus [2], etc.

The last two architectural elements described above (the comparative evaluation and the monitoring and alert) are key elements for the operational stability of the ML system. They are also strongly related to operating scenarios. Therefore, we describe their details in Section 4.

Next, we describe the data flows of the chatbot system

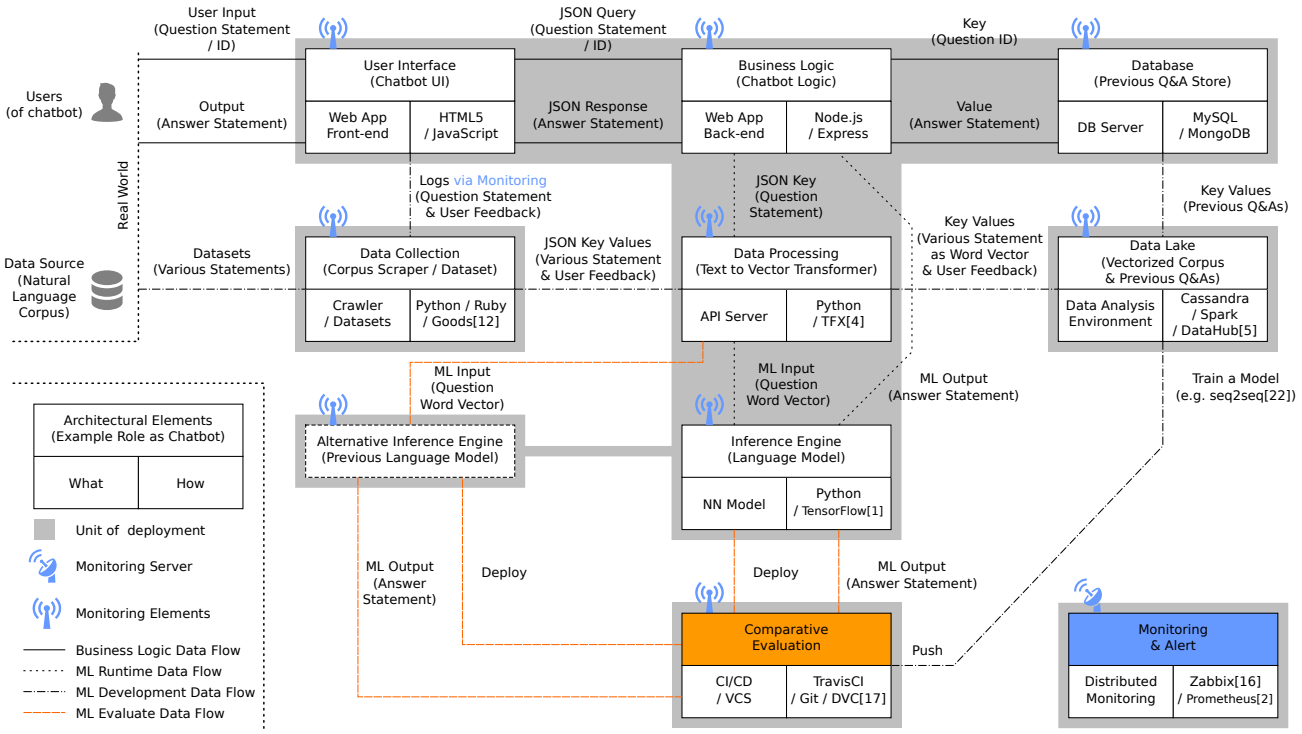

Figure 4. Example of ML Architecture

in the developing and operating phase.

In the developing phase, the data flow of the ML system consists of the ML development data flow and the ML evaluation data flow. In the ML development data flow, first, the data collection gets various statements by using the datasets and the web crawler. At the same time, the monitoring elements collect the questions and the feedback logs at runtime and provides them to the data collection. Next, the data process converts the natural language texts into the word vectors and stores them in the data lake. Then, the developers analyze the collected data and construct the inference engine using machine learning. In the ML evaluate data flow, the comparative evaluation deploys and evaluates various inference engines constructed from the data lake. When the comparative evaluation assesses a new inference engine for certain period of time and it performs better than the previous one, the developers switch the older inference engine to the newer one. To perform a runtime evaluation, it is necessary to expose a new inference engine to the production environment to get users' feedback. At this time, the developers apply Canary Release [18] in the CI/CD environment to minimize the runtime risks. Canary Release is a technique that provides the inference engine to a small number of users for verification. When a problem occurs at runtime, the developers can rollback the inference engine immediately.

In the operating phase, the ML system's data flow consists of the business logic data flow and the ML runtime data flow. We assume the user will choose one of the pre-defined questions or input a natural language question statement in the chatbot system. In the business logic data flow, first, the UI gives the user's question to the business logic. Next, the business logic judges whether the question is a pre-defined question or a natural language question statement. When one of pre-defined questions is selected, the business logic converts the selected question into ID, inquires of the database, and returns the answer via UI. This case has been completed only by the business logic data flow. On the other hand, when the input is a natural language question statement, the data flow forks to the ML runtime data flow. In the ML runtime data flow, first, the business logic gives it to the data processing. Next, the data processing converts the statement to a word vector and inquires of the inference engine. Then, the inference engine returns the answer to the business logic. Finally, the business logic checks whether prohibited words are included in the answer and returns the answer to the user via UI.

4. Scenario of Troubleshooting

In this section, we discuss how to realize problem localization and the rollback at the failure in the ML system that applies our architectural pattern.

4.1. Problem Localization

Failure in the ML system is based on a server load and the network or unexpected input data and changes in input data at runtime. An example problem is followings. First, the inference engine outputs an erroneous value caused by the unexpected input at runtime. Next, the business logic cannot handle the exception. Then, the whole ML system stops.

To deal with such a problem, the operators monitor each data flow and detect unexpected data flows. Distributed monitoring helps tackle the problem. Distributed monitoring is implemented as the monitoring and alerts, as shown in Figure 4. The monitoring and alert provides the monitoring results to the operators and alerts them when an unexpected value is detected.

In the business logic data flow, the operator should monitor both performance (the load of the UI, the business logic, and the database) and security (whether or not the system is illegally accessed). The operator should also monitor users' behavior and evaluate the business value of the system.

In the ML runtime data flow, in addition to the above, the operator should monitor the changes of the data trends and the inference results at runtime. In case of the data processing, for instance, the operators should monitor the following data.

- Unexpected values due to a lack of data and changes in the data format
- Changes in the data trends for detecting "model staleness" by time series analysis of various periods
- Difference in data between learning and serving time for detecting "training/serving skew" by statistical analysis of their distribution

4.2. Rollback

In the ML system, the operators rollback to a past inference engine when the new inference engine does not function correctly. In addition, before the operators deploy the new inference engine, they may want to evaluate whether its performance is better than the present one. At this time, the comparative evaluation in Figure 2 helps evaluate both the new inference engine's performance and the old one's. It also helps judge if the operators should conduct a workaround task such as rollback or deployment of new inference engine. The comparative evaluation is typically a combination of the CI/CD environment and the VCS. Automatic deployment of the inference engine whose performance is the best at runtime is an instance of continuous deployment [9].

Reproducibility of the inference engine is necessary to realize continuous deployment. Unlike a usual program, the inference engine depends on various data, algorithms, and hyperparameters. Therefore, if the depending element cannot be prepared accurately, a past inference engine might not be able to be reproduced. To address this problem, version control is necessary to manage the data, algorithms, and hyperparameters. DVC [16] and ModelHub [12] [13] support such tasks.

Here we show examples of the performance evaluation at runtime. In the chatbot, feedback from the users is necessary to evaluate the system. The users' answer to the question "Did I answer your question?" is data for the evaluation. Comparative evaluation becomes difficult because inference engines cannot get user feedback for answers to the same question simultaneously during the evaluation. To address this problem, the comparative evaluation calculates a percentage of correct answers from each inference engine. In the system forecasting future, the correct answer will be obtained in the future. In this case, we cannot conduct a performance evaluation in real time.

5. Research Agenda

To validate our proposed architectural pattern, we will test the following hypothesis.

Hypothesis

An ML system that is based on our architectural pattern has higher operational stability than an ML system that is based on three-layer architecture pattern.

To improve operational stability, the operators identify the cause of failure quickly and rollback the inference engine independent of the business logic quickly when the inference engine has some problems. Therefore, we will answer the following research questions to test the hypothesis.

RQ1. Do the operators identify the cause of failure in our architectural-pattern-based ML system faster than in the three-layer architectural-pattern-based one?

RQ2. Do the operators rollback the inference engine in our architectural-pattern-based ML system faster than in the three-layer architectural-pattern-based one?

For RQ1, if the operators can identify the cause of failure in the former one faster than in the latter one, our architectural pattern reduces the downtime of the ML system and improves operational stability. For RQ2, if the operators can rollback the inference engine in the former one faster than in the latter one, our architectural pattern reduces the downtime of the ML system and improves operational stability.

To answer the research questions, we conduct the following experiments. To prepare the experiment, first, we enumerate functional and non-functional requirements, collect datasets for ML, and define use case scenarios and scenario tests based on the requirements of each type of ML system (e.g. classification, regression, and clustering). Next, we order all types of combinations of ML systems and types of architectural patterns (such as the three-layer one and ours) with developer teams. We prepare two developer teams per ML system to avoid an information bias. When a team develope two systems that have the same features used in two different ways, they could develop the latter system

better than the former one. Therefore, one team develop ML systems in the order of our architectural-pattern-based one and the three-layer architectural-pattern-based one and the other team develop them in the inverse order. Then, when all the deployed systems pass our defined scenario tests, preparation for the experiment is finished.

In the experiment for RQ1, first, we give the following two input sets including anomalous input to each ML system.

- input sets that change the trends from the middle of the data timeline.
- input sets that change the schema from the middle of the data timeline.

Next, the developer team identify the anomalous input and determine how anomalous it is. Then, we measure how long the developer team identify the anomalous input and judge whether the identified input is anomalous or not.

In the experiment for RQ2, first, we give them a new learning dataset that has a different trend and schema for the initial data that we give them to prepare the experiment. Next, they redeploy the system to pass the scenario tests using the data. Then, we have them rollback the system to the previous version to pass the scenario tests using the initial data. Finally, we measure how long the developer team rollback the system to the previous version and judge whether the rollback is successful or not.

After these experiments, the developers answer the following questions.

- Does our architectural pattern help you identify the failure?
- Does our architectural pattern help you rollback the system?
- If you have the authority to decide the architecture, will you adopt our architectural pattern?

If the experiments show that our architectural pattern helps developers identify the failure and rollback the system, our architectural pattern improves the operational stability of various ML systems, which is the potential contribution of this study.

6. Related Work

Model Versioning. DVC is a model versioning tool used with the Git [16] version control tool. DVC focuses on the reproducibility of machine learning and centrally manages the kind of output that can be obtained from the input (including hyperparameters). DVC can be used to switch the version and preserve the results of an experiment. Miao et al. proposed a DL pipeline management system that integrates a model versioning system, a hyperparameter description DSL and model hosting system [12]. The main difference from DVC is that various combinations of hyperparameters can be described in DSL, they can be evaluated all at once, and the results can be compared simultaneously on the hosting system. In addition, they focused on how the experiment generates a model with several combinations of hyperparameters that fill up the storage capacity, and they conducted research on saving space [13]. The main difference between these model versioning techniques and our architectural pattern is that they mainly target reproducibility of the inference engine, while our architectural pattern helps rollback that is supported by reproducibility. Then, they become a part of the comparative evaluation in our architectural pattern.

ML Pipeline Management. KeystoneML is a machine learning framework that uses Apache Spark [20]. It constructs end-to-end machine learning pipelines and enables large-scale distributed learning. In ML development, the developer attempts optimization to improve the reasoning ability and ends up building a complicated, error-prone pipeline. They solve this problem by providing a high-level pipeline description API, concealing processing related to computational resources, and automatically optimizing computing resources. Their research is one way to realize the part that constructs the inference engine from the data lake in the ML development data flow that we showed. TFX provides a platform that covers the analysis, conversion, validation, and evaluation of the inference engine of data [4]. TFX also conducts data pre-processing and post-processing for the inference engine generated by TensorFlow [1]. The main difference between these ML pipeline management techniques and our architectural pattern is that they cover ML development and runtime data flow such as data processing, learning on the data lake, and generating the inference engine, while our architectural pattern covers ML development, runtime data flows, and business logic data flows.

Data Lake. DataHub is a collaborative data analytics tool [5] that enables many individuals to analyze datasets simultaneously. The main difference between DataHub and our architectural pattern is that DataHub provides versioning of datasets and an API to control versions, while our architectural pattern uses it as the data lake.

Data Correction. Goods [10] organizes datasets, logs, source code repositories, and databases in Google LLC and provides a search and monitoring engine. The main difference between Goods and our architectural pattern is that Goods organizes heterogeneous data, while our architectural pattern uses it as the data collection or directly as the data lake.

7. Conclusion

In this paper, we proposed a novel software architectural pattern that improves the operational stability of machine learning (ML) systems. We also used a concrete example of ML system architecture to discuss how to deal with operational problems such as problem localization and rollback at failure.

In our future research, we will implement and evaluate a concrete ML system using our architectural pattern based on the research agenda in Section 5 to verify if our architectural pattern improves operational stability in ML systems.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[2] P. Authors, "Prometheus - Monitoring system & time series database," 2018. [Online]. Available: https://prometheus.io/

[3] G. E. A. P. A. Batista and M. C. Monard, "An analysis of four missing data treatment methods for supervised learning," *Applied Artificial Intelligence*, vol. 17, no. 5-6, pp. 519–533, 2003.

[4] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1387–1395.

[5] A. Bhardwaj, Deshp, A. e, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang, "Collaborative Data Analytics with DataHub," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1916–1919, 2015.

[6] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction," in *Proceedings of IEEE Big Data*, 2017.

[7] B. Burns and D. Oppenheimer, "Design Patterns for Container-based Distributed Systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[8] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] M. Fowler, "ContinuousDelivery," 2013. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html

[10] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang, "Goods: Organizing Google's Datasets," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 795–806.

[11] J. D. Hamilton, *Time series analysis*. Princeton university press Princeton, NJ, 1994, vol. 2.

[12] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "ModelHub: Deep Learning Lifecycle Management," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 1393–1394.

[13] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Towards Unified Data and Lifecycle Management for Deep Learning," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 571–582.

[14] M. Narumoto, "Gateway Routing pattern - Cloud Design Patterns," 2017. [Online]. Available: https://docs.microsoft.com/azure/architecture/patterns/gateway-routing

[15] R. Olups, *Zabbix 1.8 network monitoring*. Packt Publishing Ltd., 2010.

[16] D. Petrov, "Data Science Version Control System," 2018. [Online]. Available: https://dvc.org/

[17] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A Survey on Deep Learning: Algorithms, Techniques, and Applications," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 92:1–92:36, 2018.

[18] D. Sato, "CanaryRelease," 2014. [Online]. Available: https://martinfowler.com/bliki/CanaryRelease.html

[19] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine Learning: The High Interest Credit Card of Technical Debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[20] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 535–546.

[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, 2014, pp. 3104–3112.