Applicability of Machine Learning Architectural Patterns in Vehicle Architecture: A Case Study

Vasilii Mosin[1,2], Darko Durisic[1] and Miroslaw Staron[2]

[1]Division of Research & Development, Volvo Car Corporation, Sweden
[2]Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden

Abstract

Machine learning (ML) has grown in importance in the last decade and has become one of the mainstream software technologies used. Together with containerization, data-driven development and microservices, it has also been used in the automotive industry, mainly for autonomous driving functions, but also, for example, for interior driver state monitoring or voice control speech recognition. The goal of this case study is to document experiences of which of the emerging ML architectural patterns are used in modern cars already, which are planned for the future and how they are used. We study the software architecture of a modern car product line based on existing patterns and workshops with software architects. The results show that many ML-specific patterns are used or planned to be used in the near future. Only a handful of patterns are not applicable or not planned to be used. However, we have also found that the established description of the patterns is not suited for the automotive software architectures, which can jeopardize its correct broad usage in the industry. We conclude that the patterns should be described more clearly and that they are more used than we could have anticipated based on the literature.

Keywords

architectural patterns, machine learning, automotive software

1. Introduction

Software in modern cars has evolved tremendously in the last two decades, driven by the new possibilities related to connectivity, autonomous driving and electrification [1, 2]. Up until the last decade, the automotive software architectures were based on the principles of distribution, where software components were deployed on separate ECUs (Electronic Control Units) connected by the communication buses. This approach provided the benefits of separation of concerns and reuse. However, with the growing number of functions depending on the software, the number of ECUs grew, the communication overhead increased and so did the difficulty in validation of the entire system. In the modern cars, software architects use the overarching styles of centralized architectures or federated software architectures. The centralized architectures are based on the principle of using one large computing node with several supporting nodes and services and containers for the separation of executable code. In the federated architectures, the entire vehicles is divided into few separated domains with domain controllers (larger computing units) and software organized accordingly. These two

15th European Conference on Software Architecture (ECSA 2021), 13-17 September 2021
vasilii.mosin@volvocars.com (V. Mosin); darko.durisic@volvocars.com (D. Durisic); miroslaw.staron@cse.gu.se (M. Staron)

architectural styles open up new possibilities of using ML in modern cars and thus the possibility of using new architectural patterns/styles for the design of these systems.

In order to understand how this is done in practice, we conducted a case study of the modern software architecture in a car line that is intended for the market in the near future. We considered 12 architectural patterns for ML application systems and software from the systematic literature review of Washizaki et al. [3]. We set off to investigate the following research question: *How applicable are the architectural patterns for machine learning in vehicle software architecture in practice?* In this context, we operationalized the applicability as a degree to which the pattern is either applied, is considered to be applied, applicable, but not applied yet or not at all applicable. We collected the data through a workshop with 11 architects from Volvo Car Corporation (VCC) – one of the Swedish vehicle manufacturers. Finally, we also studied the software design database to understand how the patterns were represented in practice. The results show that four patterns are already applied, and one is under design. The patterns of Data Lake, Microservice Architecture, Daisy Architecture and Parameter-Server Abstraction have already been applied. At the same time, we also found that the pattern Distinguish Business Logic from ML Models is not applicable in practice in the architecture of a modern car.

The remaining of the paper is structured as follows. Section 2 presents the most important related work for our study. Section 3 describes the design of our case study, Section 4 explains the considered architectural patterns and Section 5 presents the results. Section 6 discusses the threats to validity. Finally, Section 7 presents the conclusions from our study.

2. Related Work

There is a noticeable trend in moving from decentralized architectures to centralized architectures due to the increasing complexity of software in modern vehicles. Bandur et al. [4] describe the advantages and disadvantages of decentralized and centralized architectures and report that many automotive companies have started to adopt centralized solutions to their vehicle architectures. Cvijetic and Tomazin [5] claim that centralization in the car will make it easier to integrate and update advanced software features as they are developed. The emerging use of service-oriented architectures is another trend in vehicle software associated with centralization. Kugele et al. [6] conclude that uniformity, separation of concerns, and enabling computational techniques for validation and conformity testing are the main goals of a service-oriented approach in automotive architectural work. Connected to the service-orientation trend, there is increasing popularity of microservices in automotive architectures. Lotz et al. [7] show that microservice architectures, which follow service-oriented principles, can reduce system complexity and time-consuming process steps.

Automotive software architecture nowadays also needs to accommodate ML components requiring changes in the design patterns. Kläs et al. [8] show that one of the main challenges of introducing ML into the software is bringing inherent uncertainty from ML to the system, that can jeopardize the system's quality attributes. Rahman et al. [9] claim that the overall system should be built by the integration of interacting modules since ML applications are expected to be modular in design. The design of ML components needs to be lightly coupled because of their fast evolvement, and the design must accommodate the appropriate data handling mechanism

in the system as long as a large volume of data is required for ML application. Castanyer et al. [10] show that the trade-off between the accuracy and complexity of ML system is an important aspect to consider in the context of low computational power.

Due to the challenges described above, it is important to understand how to design the systems with ML components in the right way. Washizaki et al. [3] provide the systematic review of ML application system design identifying 12 architectural patterns, 13 design patterns, and 8 anti-patterns used in ML systems. Similarly, Serban and Visser [11] have done the systematic literature review demonstrating that traditional software architecture challenges (e.g., component coupling) still play an important role when using ML components. They also claim that ML-related attributes, such as privacy, are considered marginally important comparing to traditional attributes, such as scalability or interoperability, when designing a system software. Yokoyama [12] presents a new architectural pattern for ML systems that aims to separate components for application business logic and components for ML, which is helpful for quick troubleshooting. Kugele et al. [13] outline the importance of comparing ML patterns in terms of their training effort, hardware and safety requirements, and explainability in the context of critical applications such as automotive.

3. Research Methodology

We address the following research question: "How applicable are the architectural patterns for machine learning in vehicle software architecture?". To answer this research question we conduct a case study accessing the adaptation of ML architectural patterns to the automotive software. We focus on architectural patterns for ML application systems and software from the systematic literature review by Washizaki et al. [3], where they have identified 12 architectural patterns as practically used and published in either academic or gray literature. For data collection we have organized a workshop in VCC. To mitigate the bias related to the selection of the participants, we have identified and invited 3 architects who, as we know, are working with ML, and we have asked these architects to invite more people who, as they think, are relevant for the discussion. In total, 11 people out of 26 invited were able to participate in the workshop.

During the workshop, one author was leading the discussion, one author was presenting the architectural patterns, and one author was taking notes. We have been discussing each architectural pattern at a time. Firstly, a pattern was presented to the architects to make sure that everybody understands the pattern and how the pattern works. Then, we've asked the architects to rate the pattern's applicability in vehicle software architecture according to the following scale: 1 - not applicable, 2 - applicable in the future with major modifications, 3 - applicable in the future without modifications, 4 - applicable directly, 5 - already implemented. We've tried to facilitate the discussion between the architects by asking them to provide the reasoning behind their ratings and give examples from the automotive domain where possible.

For each pattern, we have recorded its applicability score provided by the architects, which was actually a consensus score that the architects agreed on together. Additionally, we have kept track of the discussion by taking notes during the workshop, helping us understand the architects' opinions better. We have collected both the quantitative ratings of the patterns' applicability and their qualitative assessment summarized from the workshop notes.

4. Considered ML Architectural Patterns

Washizaki et al. [3] have identified the following 12 architectural patterns used in ML application systems: (1) Data Lake, (2) Distinguish Business Logic from ML Models, (3) Gateway Routing Architecture, (4) Microservice Architecture, (5) Event-Driven ML Microservice, (6) Lambda Architecture, (7) Kappa Architecture, (8) Parameter-Server Abstraction, (9) Federated Learning, (10) Data-Algorithm-Service-Evaluator, (11) Closed-Loop Intelligence, (12) Daisy Architecture.

(1) A Data Lake is a centralized data repository that can store a multitude of data ranging from structured or semi-structured data to completely unstructured data [14]. Contrary to more traditional data warehouses, where the data is stored in an already preprocessed format, the data in Data Lakes should be stored in ”raw” format. The motivation behind using this pattern is that it is not always possible to know what kind of analysis will be performed on the data. Therefore, the ”raw” format is preferred allowing to apply different frameworks for the analysis.

(2) Yokoyama [12] proposes an architectural pattern to Distinguish Business Logic from ML Models. It is based on multi-layer architectural patterns, and each layer is divided into elements that have different responsibilities. Using this pattern facilitates the simpler changing of ML models by separating the business logic from the ML pipeline.

(3) Gateway Routing Architecture has the same goal, but the business logic and ML pipeline are implemented as services [12]. It is difficult to manage individual services if there are many in the system. Gateway installed before the services allows to use application layer routing requests to the appropriate instance.

(4) Microservice is an architectural pattern structuring an application as a collection of services that should be loosely-coupled. It is emerging as the data science architectural pattern that focuses on creating standard interfaces for all data extraction, manipulation, and visualization operations [15]. Moreover, ML applications may be confined to already existing ML frameworks and miss opportunities for more appropriate frameworks. Therefore, data scientists working with, or providing, ML frameworks can make these frameworks available through microservices.

(5) Event-Driven ML Microservice architectural pattern is based on the microservice pattern. Each service publishes an event whenever it updates its data, and other services subscribe to these events. This pattern aims to build ML pipelines by chaining together multiple microservices, each of which listens for the arrival of some data and performs its designated task. Event-driven AI can lead to immediate insights by reducing the time to under a second between when an event occurs and the data needed to drive action [16].

(6) Lambda is a data processing architectural pattern. It consists of three layers [17]: the batch layer, the speed layer and the serving layer. The batch layer receives arriving data, combines it with historical data, and processes the entire combined dataset. The batch layer produces the most accurate processing results since it operates on the full data. The speed layer works only on arriving data and produces real-time results. The serving layer enables various queries of the results sent from the batch and speed layers.

(7) Kappa is also a data processing architectural pattern. It has only the stream and the serving layers. This pattern was invented to avoid maintaining two separate code bases for the batch and speed layers as in Lambda architecture [17]. The key idea is to handle both real-time data processing and continuous data reprocessing using a single stream processing engine.

(8) Parameter-Server Abstraction architectural pattern consists of server groups to facilitate

running of multiple algorithms in the system [18]. This pattern is used for distributed learning. Both data and workloads are distributed across worker nodes, while the server nodes maintain globally shared parameters.

(9) Another pattern for distributed learning is Federated Learning [19]. Standard ML approaches require centralizing the training data on one machine or in a datacenter. Federated learning decentralizes ML by removing the need to pool data into a single location. Instead, the model is trained in multiple iterations at different sites.

(10) Data-Algorithm-Service-Evaluator architectural pattern originates from Model-View-Controller (MVC) pattern. MVC pattern is based on separation of concerns principle [20]. It separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. Similarly, Data-Algorithm-Service-Evaluator pattern separates the data, the algorithm, the serving, and the evaluator.

(11) Closed-Loop Intelligence architectural pattern is about creating a virtuous cycle between the intelligence of a system and the usage of the system [21]. It is designed to improve the learning of the system after its deployment. As the intelligence gets better, users get more benefit from the system (and presumably use it more), and as more users use the system, they generate more data to make the intelligence better. It is needed to establish the interactions between the users and ML components, so they produce useful training data.

(12) Daisy Architecture is an architectural pattern for which an application architecture resembles a flower. All the product streams, both manual and automated with ML, are connected through some central repository. These product streams pull and push the data from and to the central repository, so they are connected in this way. Daisy architecture evolves as organizations acquire the ability to scale their premium content production processes via the use of ML, text analytics, and annotation techniques, and then extend the coverage of that tooling over as much of their remaining content geography as possible [22]. It is based on utilizing Kanban, scaling, and microservices to realize pull-based, automated, on-demand, and iterative processes.

5. Results

Figure 1 presents the summary of the results from the workshop. Each of the patterns was discussed, and below, we present how these patterns are applied, could be applied or why they cannot be applied in the architecture of modern cars. The patterns are presented in the figure according to their applicability, while we describe them in the order in which they were presented and discussed during the workshop.

(1) Data Lakes are already used in vehicle software for data collection using a customer fleet, which is a pool of cars the customers drive, and also test cars. Engine control, emission monitoring, and road condition together with images and videos for autonomous driving are the examples of different types of data that are collected in vehicles. The data from sensors, such as cameras, radar, LiDAR, and various tracking sensors, is stored in a raw format in Data Lakes, but some additional preprocessing and enrichment can be done. Data Lakes usually constitute the central part of the software, but prioritizing what data should be saved is required due to cost reasons. Therefore, there is still some structure on top of the data in the form of

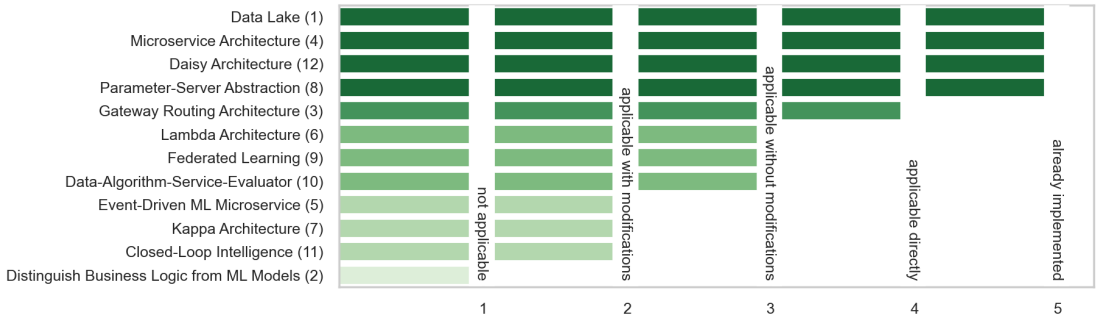

Figure 1: Architectural patterns according to their applicability scores.

catalogs and extracted meta-data for deciding on which data should be kept further. Data Lakes are also already used on servers for training and validation of in-vehicle algorithms.

(2) Distinguish Business Logic from ML Models pattern is not used in the current vehicle architecture and is not considered. The architects have understood that this pattern has a similar goal of modularization as the next Gateway Routing architectural pattern, which is more interpretable according to their opinion. Therefore they attributed Distinguish Business Logic from ML Models architectural pattern as not applicable.

(3) Gateway Routing architectural pattern is not used today but can be directly applied in vehicle software for perception-related functions implementation. It is considered as one of the approaches to modularization where perception is a part of a component. As a practical example, in the rear emergency brake function, the auto-brake feature will be separated from different sensors pipelines in the perception module through the gateway routing. This makes managing ML algorithms in the perception module easier since they will be separated from the auto-brake feature logic.

(4) Microservice Architecture is already applied in vehicle software. In general, automotive software development is moving towards more centralized service-oriented solutions that employ microservices as we have discussed in Section 2 ([4], [5], [6], [7]). For example, a vision stack is implemented using this pattern where communication between different services is done through messages. There are separate services, e. g. for cameras, videoframes, videoservices, and object detection, communicating with each other. In practice, due to the limited resources in the car, a hybrid approach is used as a trade-off between monolithic and microservice architectures. For example, a neural network with multiple heads as a multi-task algorithm for object detection and semantic segmentation may contain a shared monolithic backbone structure but different prediction layers providing separate services.

(5) Event-Driven ML Microservice architectural pattern is applicable in the future with modifications. Chaining different microservices is possible, and, to some extent, it is already done today, but the current implementations are based on working with the constant data flow, which does not contain any events. The area where this pattern is considered for application is camera pipelines, which are used for perception. The camera pipelines could be event-driven as they are based on periodically taken images. It becomes the event-driven microservice once

the events from the camera service are incorporated into the pipeline. For example, this could be useful when some anomalous object is detected on the road by the camera service, and this event is sent to the trajectory planning service to keep a safe distance or maybe even stop the car.

(6) Lambda Architecture is applicable in the future without modifications. Batch data processing is already used in vehicle software. However, it is rather implicit batching in time, so the historical batch information is accumulated inside an algorithm (for example, recurrent neural networks). ML inference will be performed on a continuous stream of data, then combining the streaming processing with the batch processing will result in Lambda data processing architecture. A potential use-case could be a combination of object detection based on the continuous streaming data and trajectory prediction based on the historical batch data.

(7) Kappa Architecture is less applicable than Lambda Architecture according to the architects' opinion. It will require modifications if it is used in the future. The reason is that data processing pipelines in vehicle software will still need to have historical information for few past timesteps to perform inference, therefore the batch layer will still be necessary. However, adding a batch layer to Kappa Architecture will result in Lambda Architecture. So, the architects agree that having a single stream layer for data processing in vehicle software will not be enough. Therefore Lambda Architecture has more potential than Kappa in future vehicle architecture.

(8) Parameter-Server Abstraction architectural pattern is already in use, but outside of the vehicle's architecture. It is used for distributed offline training of deep learning algorithms. The data and the workload are distributed across available resources, and the training results are aggregated on the central computer. This allows reducing the training time for computationally consuming models having several compute nodes.

(9) The usage of Federated Learning is currently being investigated. It will be applicable in the future without modifications for a fleet of cars and on data centers when it is impossible to take data between the countries. The legislature, like privacy rules, will cause more processing on the edge (i. e. in vehicle computers) because it will allow decreasing the amount of data transferred from cars to servers and from servers to servers or eliminate it at all. An example application can be an end-to-end ML model learning to drive from the recorded images, steering angle, and acceleration. It is possible to federatively train the model in vehicles without transferring the recorded data and then aggregate the results by sharing only the trained model weights.

(10) Data-Algorithm-Service-Evaluator architectural pattern is considered to be used (score 3 in Figure 1) to design ML components in vehicle software, but it is not yet fully implemented at the studied company. It is considered in the autonomous driving functions area, as it combines safety, separation, and redundancy. The plan is that each component in the autonomous drive stack would have its own evaluator to increase safety. In particular, perception, planning, and decision-making will have their own input data pipelines, processing algorithms, ML inference engines, and separate evaluators. These evaluators should control the execution of each component in the stack instead of monitoring the autonomous driving function as a whole.

(11) Closed-Loop Intelligence architectural pattern can be applied in the future to improve the car's functionality by, post-deployment, learning driver's personal preference. For example, the car's software can check if everything in the environment is detected correctly, and if not, it would require triggering a data recording. This process can be considered as a probe sourcing. To implement Closed-Loop Intelligence pattern, it will be required to add an oracle to the system,

which will be used for collecting the feedback from the driving.

(12) Daisy Architecture is currently being used offline outside of vehicle architecture for training perception algorithms. Different perception microservices would require the same data for the training but with different annotations. Then, they can pull this data from the shared repository and then push the results back there. Such architecture is used for developing perception pipelines consisting of both traditional and ML-based algorithms.

6. Threats to Validity

The fact that the set of discussed architectural patterns was limited á priori could affect construct validity. There is a risk that the selection does not cover all applicable patterns, but we intentionally limit ourselves to the most known patterns according to the published systematic literature review to facilitate deeper discussions with the architects.

We have organized the workshop only in one company, which affects external validity of our study. The selection of the company for the workshop is motivated by the authors' affiliation and our ability to study the application of the patterns in a real car's architecture (post-workshop). This meant that we increased the construct validity (ability to study the patterns deeper) instead of studying more companies that can affect the ability to generalize our results to the whole industry. However, we believe that VCC being a big well-known automotive company is representative enough to judge about the industry state in software architecture area to some extent. The selection of people inside the company for the workshop is also concerned by external validity. We have invited all the company architects working with ML. The size of the group itself is dealing with significance of the results and conclusion validity. We had 11 people in the group with 3 people most actively participating in the discussion. So, the group was small, but it was dictated by the number of architects working with ML in the company.

The used applicability score for presenting our results can be subjected to question its reliability and conclusion validity as a consequence. We tried to make it as clear as possible for the architects to eliminate any potential bias due to misunderstanding the metric. However, it is still based on the knowledge of the architects participating in the workshop. In our future studies, we plan to count how many times a particular pattern is used in the car's architecture to understand how widely spread the pattern is.

Internal validity of our study is affected by maturation, history, and instrumentation threats. The workshop was long and took about 2 hours, without any breaks, that leads to maturation threat. On the one hand, people could be tired and bored by the end of the workshop, which could negatively affect the results. On the other hand, the architects could start to understand the idea of the workshop better as the workshop was going further, that could bring a positive effect. There is also a risk that the opinions of the architects could be affected by the previous answers, which is a history threat. We tried to eliminate this threat by abstracting and discussing each architectural pattern separately. Instrumentation threat is related to the architects' understanding of the patterns from our presentation and explanation. Imprecise descriptions could negatively affect the architects' ability to identify the patterns. However, even if the architects did not understand some patterns, then it means that they do not use them explicitly in their work, so we just do not count them.

7. Conclusions

In this work, we have studied the applicability of ML architectural patterns in vehicle software architecture. Based on the results of the organized workshop in VCC, we have found that four patterns (Data Lake, Microservice Architecture, Daisy Architecture and Parameter-Server Abstraction) are already implemented in some way in vehicle software, and one pattern (Distinguish Business Logic from ML Models) is not seen as applicable. The rest of the patterns are applicable directly or in the future vehicle architecture according to the architects. We have also observed that the description of the studied patterns is not well suited for use in the automotive domain that can be potentially addressed and improved in further work. Nevertheless, the provided case study results with ML architectural patterns description and examples from the automotive company's perspective is a good basis for further research in the area of ML automotive software architecture.

Despite that the considered patterns are positioned as ML architectural patterns, they actually appear as an adaptation of the existing architectural patterns to the use with ML components. In essence, there are no really core ML concepts presented in these patterns. However, there are many important aspects to consider when describing the usage of ML in a car, such as data preprocessing, model selection and training, parameter tuning, probability nature of the algorithms, etc. Therefore, in our future work, we will focus on bringing these aspects to vehicle software architecture based on the needs of pure ML components. Additionally, it is worth analyzing the quality attributes associated with ML architectural patterns. We should also consider traditional software architectures, such as, for instance, layered and multi-tier patterns, and investigate how they can be combined with the principles of ML in the automotive domain.

Acknowledgments

This work was carried out within DeVeLop project financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number 2018-02725.

References

[1] M. Staron, Automotive software architectures, An Introduction, 2nd edition, Springer, 2021.

[2] C. Ebert, J. Favaro, Automotive software, IEEE Annals of the History of Computing 34 (2017) 33–39.

[3] H. Washizaki, H. Uchida, F. Khomh, Y.-G. Guéhéneuc, Machine learning architecture and design patterns, 2020.

[4] V. Bandur, G. Selim, V. Pantelic, M. Lawford, Making the case for centralized automotive e/e architectures, IEEE Transactions on Vehicular Technology 70 (2021) 1230–1245. doi:10.1109/TVT.2021.3054934.

[5] N. Cvijetic, T. Tomazin, Developing a centralized compute architecture for autonomous vehicles, ATZelectronics worldwide 16 (2021) 10–15. URL: https://doi.org/10.1007/s38314-020-0573-8. doi:10.1007/s38314-020-0573-8.

[6] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, W. Hopfensitz, On service-orientation for automotive software, in: 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 193–202. doi:10.1109/ICSA.2017.20.

[7] J. Lotz, A. Vogelsang, O. Benderius, C. Berger, Microservice architectures for advanced driver assistance systems: A case-study, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019, pp. 45–52. doi:10.1109/ICSA-C.2019.00016.

[8] M. Kläs, A. M. Vollmer, Uncertainty in machine learning applications: A practice-driven classification of uncertainty, in: B. Gallina, A. Skavhaug, E. Schoitsch, F. Bitsch (Eds.), Computer Safety, Reliability, and Security, Springer International Publishing, Cham, 2018, pp. 431–438.

[9] M. S. Rahman, E. Rivera, F. Khomh, Y.-G. Guéhéneuc, B. Lehnert, Machine learning software engineering in practice: An industrial case study, 2019. arXiv:1906.07154.

[10] R. C. Castanyer, S. Martínez-Fernández, X. Franch, Integration of convolutional neural networks in mobile applications, 2021. arXiv:2103.07286.

[11] A. Serban, J. Visser, An empirical study of software architecture for machine learning, 2021. arXiv:2105.12422.

[12] H. Yokoyama, Machine learning system architectural pattern for improving operational stability, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019, pp. 267–274. doi:10.1109/ICSA-C.2019.00055.

[13] S. Kugele, C. Segler, T. Hubregtsen, Architectural patterns for cross-domain personalised automotive functions, in: 2020 IEEE International Conference on Software Architecture (ICSA), 2020, pp. 191–201. doi:10.1109/ICSA47634.2020.00026.

[14] A. Singh, Architecture of data lake, https://datascience.foundation/sciencewhitepaper/architecture-of-data-lake, 2019.

[15] D. Smith, Exploring development patterns in data science, https://www.theorylane.com/2017/10/20/some-development-patterns-in-data-science, 2017.

[16] How serverless platforms could power an event-driven ai pipeline, https://thenewstack.io/how-serverless-platforms-could-power-an-event-driven-ai-pipeline/, 2019.

[17] J. Forgeat, Data processing architectures – lambda and kappa, https://www.ericsson.com/en/blog/2015/11/data-processing-architectures--lambda-and-kappa, 2015.

[18] Parameter server for distributed machine learning, https://medium.com/coinmonks/parameter-server-for-distributed-machine-learning-fd79d99f84c3, 2018.

[19] N. Rieke, What is federated learning?, https://blogs.nvidia.com/blog/2019/10/13/what-is-federated-learning/, 2019.

[20] R. D. Hernandez, The model view controller pattern – mvc architecture and frameworks explained, https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/, 2021.

[21] G. Hulten, Closed-loop intelligence: A design pattern for machine learning, https://docs.microsoft.com/en-us/archive/msdn-magazine/2019/april/machine-learning-closed-loop-intelligence-a-design-pattern-for-machine-learning, 2019.

[22] G. Everett, An architecture pattern for pull-based, automated content enrichment in media organisations., https://datalanguage.com/blog/daisy-architecture, 2018.