Towards MLOps in Mobile Development with a Plug-in Architecture for Data Analytics

Rustem Dautov
SINTEF Digital
Oslo, Norway
rustem.dautov@sintef.no

Erik Johannes Husom
SINTEF Digital
Oslo, Norway
erik.husom@sintef.no

Fotis Gonidis
Gnomon Informatics SA
Thessaloniki, Greece
f.gonidis@gnomon.com.gr

***Abstract*—Smartphones are increasingly used as universal IoT gateways collecting data from connected sensors in a wide range of industrial applications. With the increasing computing capabilities, they are used not just for simple data aggregation and transferring, but have now become capable of performing advanced data analytics. As AI has become a key element in enterprise software systems, many software development teams rely on dedicated Machine Learning (ML) engineers who often follow agile development practices in their work. However, in the context of mobile app development, there is still limited tooling support for MLOps, mainly due to unsuitability of native programming languages such as Java and Kotlin to support ML-related programming tasks. This paper aims to address this gap and describes a plug-in architecture for developing, deploying and running ML modules for data analytics on the Android platform. The proposed approach advocates for modularity, extensibility, customisation, and separation of concerns, allowing ML engineers to develop their components independently from the main application in an agile and incremental manner.**

***Index Terms*—Internet of Things, Mobile Development, Android, Machine Learning, MLOps, Plug-in Architecture, Fitbit.**

I. Introduction

Smartphones have been traditionally used by end users to view and control IoT devices, either directly using some form of device-to-device communication or via IoT cloud platforms. In recent years, the role of smartphones has evolved. Thanks to the presence of multiple built-in sensors (*e.g.*, GPS, camera, accelerometer, gyroscope, proximity, *etc.*), as well as a wide range of wireless communication technologies (*e.g.*, Wi-Fi, Bluetooth, RFID, NFC, *etc.*) they now play a significant role of universally adopted edge gateways, often running increasingly advanced data processing and analytics tasks.

As more and more data analytics takes place on smartphones, mobile application development is experiencing an increased demand for data scientists and Machine Learning (ML) engineers, *i.e.* software developers with a particular focus on working with large amounts of time-series sensor data collected from downstream IoT devices. Many mobile development teams have a clear separation of roles and responsibilities within their projects. ML engineers focus on the data science lifecycle, namely data ingestion and preparation, model development, and deployment, with periodical re-training and re-deploying of the models to adjust for freshly labelled data, data drift, user feedback, or changes in model inputs, caused by evolving business requirements [1]. In parallel to this, traditional mobile developers focus on the app lifecycle – *i.e.* building, maintaining, and continuously updating the larger business application. Both parties work hand in hand to make the overall business application and data analytics perform well together to meet end-to-end performance, quality, reliability, and user experience goals.

To date, there are multiple frameworks and development environments for rapid and user-friendly mobile development (both for Android and iOS, as well as cross-platform), facilitating agile development cycles and shorter time to market. However, there is still a gap in terms of technological and tooling support for ML engineers on mobile platforms. In particular, the use of Python, the *de facto* programming language in data science, is still quite limited in mobile OSs. While it is in general possible to code stand-alone scripts and run the Python interpreter both on Android and iOS, there is a pressing challenge of seamlessly integrating ML inference modules written in Python into larger apps as part of business logic responsible for data analytics [2].

As a step forward towards addressing this limitation, this paper presents an extensible architecture for rapid development of data analytics on the Android mobile platform. The proposed architecture motivates for clear separation of concerns between app developers and ML engineers, allowing the latter to build and manage data processing modules independently from the main application logic. This is achieved by employing the prominent plug-in architectural pattern, which brings benefits in terms of application logic extensibility, flexibility, and customisation. This becomes especially important in the context of DevOps, MLOps, and other agile development practices, which depend on frequent non-blocking releases of software updates. Accordingly, the contribution of this paper is two-fold: *i)* the conceptual design of a plug-in architecture for deploying and running ML modules on the Android platform, and *ii)* a proof of concept implementation of the proposed architecture for fatigue detection using time-series sensor data from Fitbit fitness trackers.

The rest of the paper is organised as follows. Section II describes the motivation behind this research by looking deeper into the research context and identifying the gaps in the current state of art and practice. Section III formulates the challenges from an MLOps perspective, and then proposes a

plug-in architecture to address them. Section IV presents a proof of concept prototype, demonstrating the feasibility of the proposed approach with a real-life application scenario. Section V concludes the paper by critically evaluating the results and outlining directions for further work.

II. Research Context and Motivation

A. Technological Trends Underpinning the Research Context

Mobile software development is now experiencing the following concurrent, yet inter-connected trends:

1) *Smartphones as universal IoT gateways*: IoT-based communication is relevant to many vertical domains with a wide adoption of the ubiquitous sensor technology (healthcare, automotive, manufacturing, agritech, *etc.*). Today, the IoT gateway market is managed by proprietary hardware. An alternative, allowing connecting to the Internet without any proprietary hardware, is to have a universal gateway that can enable configuration-based easy, dynamic plug-in for any kind of IoT devices. Smartphones are increasingly used as such universal IoT gateways, thus optimising the costs and increasing the adoption of the IoT technology for common smartphone users. Furthermore, in many cases smartphones add more value by collecting data not only from downstream IoT devices, but also from built-in sensors and the user activity, thus facilitating more intelligent context-aware decisions.

2) *AI on the edge*: IoT time-series data is usually collected from multiple sensors sampling at a high frequency, often resulting in extreme amounts of collected data. Sending them to an IoT cloud for remote processing is often impractical due to limited network bandwidth and high latency. Another challenge, especially in telemedicine scenarios, is the data privacy, which restricts sending personal information externally [3]. The increased costs incurred by storing data and running 'heavy' data analytics in the cloud is another common concern. Therefore, businesses are moving AI capabilities to the edge to enable real-time actions in the field, *i.e.* as close to the original source of data as possible [4], [5].

3) *Adoption of agile development practices*: DevOps is a cultural movement containing a set of practices that facilitate the collaboration between development, testing, and IT operations [6]. *Mobile DevOps* takes these concepts further and applies them to accelerate the development of mobile apps, while maintaining the quality. Once integrated, these practices bring considerable positive effect on productivity, efficiency, client satisfaction, and eventually on revenues. In parallel to this, the increased integration of ML and AI in enterprise-level software has led to the emergence of the so-called *MLOps* [2], [7] – an adaptation of DevOps practices towards the ML domain, that aims to deploy and maintain ML models in production reliably and efficiently. To achieve this, MLOps promotes automation and monitoring at all steps of the ML system lifecycle, including integration, testing, releasing, deployment and infrastructure management.

Taken together, these three converging trends lead to **an increased demand for running advanced data analytics on mobile platforms, which in its turn calls for dedicated ML engineers to be involved in the agile development process, driven by frequent updates and continuous releases**. While this situation appears to bring considerable benefits to involved stakeholders, the existing tooling support fails to keep up with these trends, as we further discuss below.

B. State of Art and Practice

ML engineers in their work have to extract, process, define, clean, arrange and then understand the data to develop intelligent algorithms. Thanks to the rich support for all these tasks, Python has become the *de facto* programming language used in ML. This is due to its simplicity and readability, which allows ML engineers to focus on the algorithms and results, rather than on structuring code and keeping it manageable. This simplicity also allows other people to review and improve the code. Another advantage is that Python is also usually consistent across projects and platforms, allowing to seamlessly use the same few mainstream modules (*e.g.* `keras`, `tensorflow`, `pandas`, `scikit-learn`, `numpy` and several others[1]).

The wide use of Python in ML has also led to the emergence of multiple cross-platform tools that allow running Python scripts almost on any platform, often from within a non-Python execution environment. One of such platforms is Android OS, which natively supports only Java and Kotlin code. To be able to run Python, the community has come up with several frameworks[2] that provide a bridge between the native Android and the external Python code. For example, QPython and Termux[3] offer a command line interface and a simple text editor for typing and running stand-alone Python scripts. The cross-platform frameworks BeeWare and Kivy[4] can be used to package Python code as Android apps with support for user interfaces and seamless access to most Android services.

These existing tools are good for quick prototyping, but have limited support for integration with native Java/Kotlin code, especially in the context of a large enterprise-level project that follows DevOps practices. A notable exception is Chaquopy Python SDK[5] that allows running Python scripts from Android-native code. It can be integrated using standard build automation tools such as Gradle and Maven. This proves especially useful in a team of Android developers coding in Java/Kotlin and ML engineers coding in Python.

In parallel to this, the research community has been trying to develop solutions for deploying and running ML models on resource-constrained platforms (including smartphones), typically investigated under the umbrella term *Edge AI* [8], [9]. The main focus, however, has primarily been on creating more light-weight, yet accurate enough models, with very few works focusing on the integration aspects of MLOps aspects. A relevant approach is described in [10], where the

[1] https://www.upgrad.com/blog/top-python-libraries-for-machine-learning
[2] https://wiki.python.org/moin/Android
[3] https://www.qpython.com, https://termux.com
[4] https://beeware.org, https://kivy.org
[5] https://chaquo.com

authors propose their solution for building light-weight ML models, deploying and running them on almost any platform, including Android. Although the approach enables loosely-coupled interaction between multiple software components, it depends on the additional containerisation middleware, which is not usually present on most users' smartphones.

There are also several enterprise-level MLOps tools that automate the lifecycle of AI/ML components on mobile platforms. Two prominent examples are TensorFlow Lite and PyTorch Mobile,[6] which allow training light-weight models, deploying and running them on Android and iOS. The provided SDKs are available in several mobile-native languages, but are limited to mobile-oriented scenarios, such as text, image, audio and video analysis.

The lack of Python support forces mobile developers to code in native programming languages, which are not well-suited for such tasks due to complexity and rigidity, resulting in more time spent on code structuring and management and leading to slower development pace. Companies are often forced to completely drop the promising idea of running AI-driven business logic on the mobile edge and roll back to keeping everything in a centralised cloud (or less commonly – in a fog environment). In these circumstances, the universal adoption of MLOps for mobile software development to achieve agile and rapid integration of ML features is hindered.

With this paper, we aim to address these challenges and allow ML engineers to be fully involved in mobile development following agile development practices. The main goal of this research effort is to bridge the gap between DevOps and MLOps in the context of mobile software development. Both disciplines are mature enough on their own, with commercially-available products on the market. However, their parallel use for mobile development by a team of Android developers and ML engineers, remains an open question.

III. Proposed Approach: Plug-in Architecture

Business requirements for timely data processing on the mobile edge tend to continuously evolve [11]. These changes may be triggered by multiple factors, such as, for example, newly introduced or updated application scenarios or new hardware sensors integrated into the system. This naturally calls for a loosely-coupled architecture, wherein frequently updated components, *i.e.* the ones containing the data processing logic, can be seamlessly added or removed, with minimum disruption to the rest of the running system. From an MLOps perspective, this overall challenge has the following aspects:

1) **Modularity to enable separation of concerns**: A modular architecture will reduce complexity and allow developers to deploy new features independently from each other.
2) **Agile support for frequent updates**: New features added to the system should have minimum effect on the rest of the project, meaning that incremental code updates are applied in a safe, isolated and non-blocking manner.

[6]https://www.tensorflow.org/lite, https://pytorch.org/mobile

3) **Multi-language support**: ML engineers can continue coding their components in a programming language and using a technology stack they are most proficient with.

The described challenges have traditionally been addressed by researchers and practitioners by employing various decomposition techniques, such as component-based software engineering [12] or the service-oriented architecture [13]. A particularly relevant pattern for loosely-coupled software systems composed of two main components – namely, a relatively static core part and multiple dynamically evolving extensions – is the *plug-in architecture*. The main design principle here is to allow adding new features as plug-ins to the core application, providing extensibility, flexibility, customisation, and isolation of application logic.

- **Core system**: the core defines how the system operates and the basic business logic. It is often defined as the general business logic or bare minimum for the application to function. The specific implementations of that functionality is up to individual plug-ins. The core needs some way to know what plug-ins are connected and how to use them, which is usually done through a plug-in registry. The core also contains common utility functionality to be used by plug-ins as a way to reduce duplicated and redundant code, and have one single source of truth. Examples of such common functionality include logging, database access, versioning, caching, security mechanisms and other standard re-usable software components.
- **Plug-ins**: plug-ins are stand-alone, independent components that contain specialised processing, additional features, and custom code that is meant to enhance or extend the core system to produce additional capabilities. Generally, plug-in modules should be independent of other plug-in modules, meaning that adding or removing a plug-in, even at run-time, does not affect the other plug-ins. The core system declares extension points, usually in the form of a well-defined API, that plug-ins can hook into. A plug-in normally implements some custom functionality, and the core keeps track of attached plug-ins through some form of registry, which includes information about available plug-ins and the protocols for accessing them.

A. Service Provider Interface to Implement the Plug-in Architecture

Service Provider Interface (SPI) naturally implements the modular plug-in architecture (both in traditional J2EE and Android), including the support for dynamic discovery and loading at run-time. The SPI mechanism was introduced to make applications more extensible, as it allows third parties to enhance specific parts of a main product without modifying the core application. Using the SPI mechanism, the application will load the newly added implementation and seamlessly work with it. There are three main elements underpinning the SPI mechanism:

- Service represents a well-known set of interfaces and abstract classes available to the core application. Speaking in terms of plug-in terminology, a Service is a well-defined interface that allows the core system to interact with plug-ins.

• Service Provider is a specific implementation of a Service. Broadly speaking, it is the actual plug-in, which hooks into the extension points provided by the core system. Each Service Provider implementation must be placed on the application class path (typically in the form of a jar file) to be discovered and loaded, both at compile- and run-time.
• Service Loader is the mechanism for discovering and loading available plug-in implementations (*i.e.* Service Providers). As a pre-requisite for discovery and loading, each Service Provider needs to be accompanied by a configuration file that associates it to a specific Service. The Service Loader also acts as the plug-in registry by keeping track and caching already loaded Service implementations.

IV. Proof of Concept

A. Health Monitoring Using Wearable Fitness Trackers

The rapid technological advances in physiological sensors, low-power integrated circuits, and wireless communication has enabled a new generation of wireless sensor networks, widely used for purposes of continuous health and well-being monitoring. This continuously collected data can then be analysed and used for early detection of medical conditions and assisted rehabilitation. Traditionally, IoT cloud platforms have been the primary location for deploying and running such ML-driven analytics. However, more recently, there has been an increased demand for a more time-critical and autonomous operation, where ML inference is placed as close to the data source as possible, *i.e.* on a smartphone gateway.

This is especially important for remote working conditions with limited network connectivity, such as the maritime sector, where workers on ships or oil platforms are exposed to hostile working conditions such as laborious work, homesickness and rough weathers, *i.e.* factors often leading to increased fatigue and stress levels among seafarers [14]. Off-shore accidents resulting from poor physical conditions and/or mental health can easily escalate to life-threatening situations, given that medical assistance is not as accessible as it would be on land.

Accordingly, the reported research has been conducted in the context of an R&D project on autonomous and time-critical fatigue detection among patients by deploying and running ML models on Android smartphones, acting as IoT gateways for physiological sensor data (*e.g.* sleep, physical activity, and heart rate) coming from Fitbit wearable fitness trackers. Data collection, feature engineering and model training, albeit fundamental parts of the whole implementation process, go beyond the scope of this paper, and below we only focus on the architectural aspects allowing deployment and integration of ML features into the core mobile app.

B. Implementation

Following the design principles, described in Section III, the proposed plug-in architecture was implemented as the **Android Time-Series** (AnTS) framework,[7] depicted in Fig. 1. As a proof of concept demonstration, we will now explain how the AnTS framework, deployed on an Android platform, is used for loading and processing time-series data collected from a wearable fitness tracker.

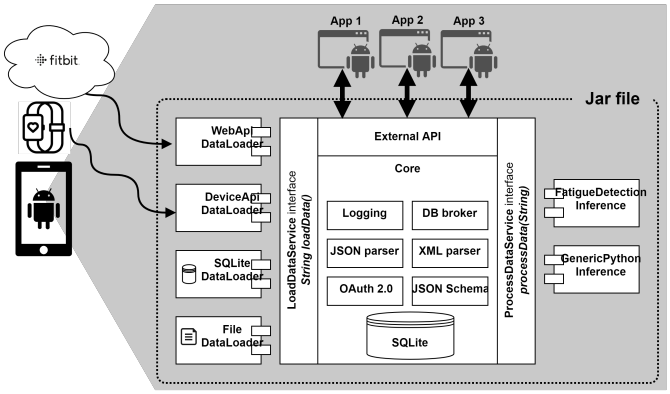

Fig. 1. Proof of concept: AnTS framework for fatigue detection using fitness trackers.

The core of the framework is implemented in Java. It contains code with generic utility functionality (*e.g.* SQLite database broker, logging, JSON and XML parsers, *etc.*), as well as defines several entry-points to be used by third-party apps when imported as a jar file. The core also implements the plug-in loading functionality using SPI's ServiceLoader, which scans for available Service implementations on the classpath and keeps track of them at run-time.

Currently, there are two abstract Services exposed by the core system, whereas the actual implementations are contained in several plug-in extensions:

1) LoadDataService: this abstract class represents possible sources of data to be further used as input for processing and analytics. Apart from some other auxiliary functionality, it declares the main method loadData() to be implemented by any ServiceProvider inheriting this class. For the purposes of the fatigue detection scenario, the following ServiceProvider implementations were developed:

• WebApiDataLoader: this plug-in is responsible for fetching data from the Fitbit Web API,[8] which offers multiple entry-points for parameterised querying across a wide range of collected biomarkers.
• DeviceApiDataLoader: although the support for third-party mobile apps directly accessing live Fitbit data (*i.e.* without transferring it first to the cloud) is limited due to the proprietary commercial nature, we also implemented a mock-up plug-in for querying real-time biomarker data from trackers. This plug-in plays an important role when the symptoms of a growing fatigue need to be detected in an autonomous offline manner.
• SQLiteDataLoader: this plug-in is responsible for querying data from Android's native relational database SQLite. Being extremely light-weight, this DB does not have rich built-in data types (*e.g.* Timestamp for time-series data), but is, nevertheless, able to store sensor data using string timestamps

[7]https://github.com/SINTEF-9012/ants

[8]https://dev.fitbit.com/build/reference/web-api/

– a simple, yet highly effective solution. Therefore, it was natural to use SQLite for storing and further accessing locally-cached data.

• FileDataLoader: this plug-in implementation is actively used for testing purposes, when we loaded previously recorded data from locally stored CSV and JSON files.

2) ProcessDataService: this abstract class declares an interface for various data processing and analytics components. It declares the main method processData(String) to be extended by child implementations. The data processing functionality can range from a simple mathematical operation (*e.g.* calculating an average value over some time window for some sensor measurements) to running advanced ML inference. For the purposes of the fatigue detection scenario, the following plug-in implementations were developed:[9]

• FatigueDetectionInference: a collection of fatigue detection inference modules, which vary in terms of the underlying ML algorithm (*i.e.* different regression and classification ML models), the data format and specific data features used, the granularity and time window of time-series data, *etc.* The plug-in implementations themselves are coded in Java, whereas for invoking the ML models we have relied on the ChaquoPy library, enabling seamless interplay between Java and Python, as well as provisioning of the required Python libraries (*e.g.* `tensorflow`, `pandas`, `numpy`).

• PythonInference: the capabilities of ChaquoPy to act as a seamless Java-Python bridge to invoke ML inference scripts also made us implement a general-purpose ProcessDataService implementation to be used as a default generic way of running Python code out of the box. The ML engineer is only required to upload the updated ML artefacts (*i.e.* Python script, trained model, data labels) without touching the Java part of the plug-in. To a certain extent, this can be even seen as a second-level plug-in architecture, wherein multiple Python scripts can be dropped onto the plug-in classpath to be loaded and executed at run-time.

C. ML Engineer's Perspective

One of the main goals of the proposed system was to actively involve ML engineers in the agile app development process, allowing them to iterate on their part independently from the Java developers. The described plug-in architecture allows them doing so by developing LoadDataService and ProcessDataService implementations and placing them on the project classpath along with a declaration file pointing to a specific parent Service class. The SPI mechanism will then detect the provided implementations to make them available to the core system. This comes particularly handy in an application system with continuously evolving business requirements. For example, in the context of the described fatigue detection scenario, an update to the data analytics components can be caused by the newly-introduced support for a chest heart rate monitor. There will be an emerging requirement for new biomarker data and inference models, which can be seamlessly added using the proposed approach.

Even further opportunities for agile MLOps are provided by the generic PythonInference implementation, with which the ML engineer can enjoy the second-level separation of concerns and solely focus on the Python code and ML models. Assuming that the data source has been established and is relatively stable, *i.e.* data format, labels, and granularity are known, the ML engineer is free to fine-tune and test the ML models, completely independently from the Java code.

[9]Please note that we group several similar implementations together for text clarity and simplicity. In fact, for each ML model there is a separate ProcessDataService implementation.

V. Discussion and Conclusion

The presented research work started as a small-scale prototype in the context of an R&D project focusing on remote patient monitoring. The main challenge was to allow team members responsible for data analytics (*i.e.* ML engineers mainly specialising in Python) to continuously modify and test their components, independent of the core Android application, which remained relatively stable. As the work matured, it became clear that the addressed challenges are common across a wider range of application scenarios, wherein heterogeneous sensor data needs to be processed locally on a mobile gateway, in the presence of frequent modifications modifications to the business logic. As a result, we have tried to generalise and extend this work, aiming to make it usable not only by a small group of researchers, but serving a wider community of practitioners, including mobile developers and ML engineers. The proposed architecture and implementation have both their benefits and shortcomings, as well as room for further work, as discussed below.

A. Benefits and Shortcomings

The proposed system comes with several advantages due to the nature of the plug-in architecture, which gives the agility to rapidly change, remove, and add data sources and processing modules. Each plug-in can be deployed and tested independently, which opens up promising opportunities for MLOps in the context of mobile development. More specifically, we highlight the following main advantages of the proposed system:

• **Modularity**: because plug-ins are separate modules with well-defined interfaces, it is easier to quickly track down, isolate and fix emerging issues.

• **Extensibility**: the application can be dynamically extended to include new data processing and ML features, even at run-time thanks to the dynamic discovery and loading of plug-ins.

• **Customisation**: creating custom versions of an application without modifying the core system is very important for ML-driven software, where iterative fine-tuning and optimisation of ML models is required.

• **Separation of concerns and parallel development**: since various app features can be implemented as separate components, they can be developed in parallel by different teams. One of the teams is ML engineers, who are not necessarily

proficient in native app development, but are still fully involved in the ML-related activities.

On the other hand, by implementing the plug-in architecture, the proposed framework inherits several shortcomings (some of which are planned to be addressed in the future):

- **The core being the bottleneck and the single point of failure**: changing the core system might break or alter the behaviour of the dependent plug-ins. It requires careful design with support for backward-compatibility in mind. Another related issue is the separation of functionality, which requires deciding what belongs to the core and what should be delegated to plug-in implementations.
- **Limited integration testing**: even if a plug-in is successfully tested alone or against the core system, some issues may emerge only in combination with other plug-ins. This slows down the testing process, especially if multiple independent parties develop their own plug-ins in parallel.
- **Reduced performance**: too many loaded plug-ins can slow down the overall system. Therefore, it is required to carefully design the plug-in loading logic, so that only relevant and non-conflicting functionality is loaded at a time. On the other hand, if properly implemented, targeted loading of plug-ins can actually make the system more light-weight and increase the performance.

B. Future Work

We see the proposed approach as a step forward towards implementing MLOps in the context of mobile app development. Therefore, an immediate next step for us is to integrate our solution with the existing tools in order to cover the whole automated MLOps cycle, *i.e.* from data collection, preprocessing and model training to deployment and operation, repeated in an iterative incremental manner. This will also underpin the empirical evaluation of the developed approach, since one of its main advantages is the practical usability by ML engineers in the context of agile mobile development. This is something that can only be validated in real-life settings within a diverse team of mobile developers and ML engineers.

One of the assumptions of the current implementation is that all LoadDataService and ProcessDataService plug-ins rely on some known data format and structure when passing time-series data sets from the source to analytics. This works fine in a relatively small development team, where one and the same ML engineer has full understanding of the data and will most probably implement both types of plug-ins. With more people involved, it becomes a pressing concern to implement a mechanism for communicating the data schema across all involved modules. We are currently implementing overloaded methods in both LoadDataService and ProcessDataService, which provide the data schema as a string argument along with the transferred data sets.

While plug-in discovery and loading are already natively supported by the SPI mechanism, another important direction for future work is the thorough design and implementation of the selection mechanism at run-time. Since the plug-in architecture in general assumes the availability of multiple, often conflicting implementations, it is important to ensure that only the required correct plug-ins are loaded at a time. We are planning to achieve this by designing a classification taxonomy shared between the core system and all the plug-ins, which will allow to uniquely describe plug-ins using a combination of tags and select only the most relevant one at run-time in a context-aware manner.

ACKNOWLEDGEMENT

The research leading to these results has been supported by a grant from Iceland, Liechtenstein and Norway through the EEA Grants Greece 2014-2021, in the frame of the "Business Innovation Greece" programme. This work was also partly supported by the Research Council of Norway through the BIA-IPN programme, project no. 309700.

REFERENCES

[1] M. A. Waller and S. E. Fawcett, "Data science, predictive analytics, and big data: a revolution that will transform supply chain design and management," *Journal of Business Logistics*, vol. 34, no. 2, pp. 77–84, 2013.

[2] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[3] R. Dautov, S. Distefano, and R. Buyya, "Hierarchical data fusion for smart healthcare," *Journal of Big Data*, vol. 6, no. 1, pp. 1–23, 2019.

[4] R. Dautov and S. Distefano, "Three-level hierarchical data fusion through the IoT, edge, and cloud computing," in *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*, pp. 1–5, 2017.

[5] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Pushing intelligence to the edge with a stream processing architecture," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 792–799, IEEE, 2017.

[6] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[7] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?," in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, pp. 109–112, IEEE, 2021.

[8] Y.-L. Lee, P.-K. Tsung, and M. Wu, "Techology trend of edge AI," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–2, IEEE, 2018.

[9] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief, "Communication-efficient edge AI: Algorithms and systems," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2167–2191, 2020.

[10] M. Lootus, K. Thakore, S. Leroux, G. Trooskens, A. Sharma, and H. Ly, "A VM/containerized approach for scaling tinyML applications," *arXiv preprint arXiv:2202.05057*, 2022.

[11] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Data agility through clustered edge computing and stream processing," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 7, pp. 1–1, 2021.

[12] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the pieces together, addison-westley*, vol. 5, 2001.

[13] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization, New York, 2004.

[14] J. R. Jepsen, Z. Zhao, and W. M. van Leeuwen, "Seafarer fatigue: a review of risk factors, consequences for seafarers' health and safety and options for mitigation," *International maritime health*, vol. 66, no. 2, pp. 106–117, 2015.