Bridging the Gap Between Java and Python in Mobile Software Development to Enable MLOps

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 S.A.
Thessaloniki, Greece
f.gonidis@gnomon.com.gr

Spyridon Papatzelos
Gnomon Informatics S.A.
Thessaloniki, Greece
s.papatzelos@gnomon.com.gr

Nikolaos Malamas
Gnomon Informatics S.A.
Thessaloniki, Greece
n.malamas@gnomon.com.gr

***Abstract*—The role of Machine Learning (ML) engineers in mobile development has become increasingly important in recent years, as more and more business-critical mobile applications depend on AI components. Many development teams already include dedicated ML engineers who aim to follow agile development practices in their work, as part of the larger MLOps concept. However, the availability of MLOps tools tailored specifically towards mobile platforms is scarce, often due the limited support for non-native programming languages such as Python, as well as the 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 data ingestion and processing components written in Python on the Android platform. With the possibility to pass a user-defined schema with the data format and structure, the proposed architecture ensures that time-series datasets are correctly interpreted by multiple ML modules dealing with both data ingestion and processing, . The proposed approach benefits from modularity, extensibility, customisation, and separation of concerns, which enable ML engineers to be fully involved in a mobile development lifecycle following agile MLOps practices.**

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

I. Introduction and Motivation

Smartphones are increasingly used as universal IoT gateways in the context of vertical IoT communication. With the growing computing and networking capabilities, they serve not only to collect and transfer data, but also to run advanced timely analytics locally. As AI components are integrated into more and more business-critical applications, the role of Machine Learning (ML) engineers in mobile development has become of critical importance to build intelligent AI-driven apps. Many mobile development teams have a clear separation of roles within their projects. ML engineers focus on the data lifecycle, including 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, mobile app developers focus on the lifecycle of the overall application. Thanks to this separation of concerns, both parties work well together to meet end-to-end performance, quality, reliability, and user experience requirements. This situation is underpinned by the following technological trends:

- **Smartphones as universal IoT gateways**: Vertical communication is critical to many IoT-enabled domains driven by the ubiquitous sensor technology (healthcare, automotive, manufacturing, agritech, *etc.*) [2]. To support this, smartphones allow connecting to the Internet without any proprietary hardware, and thus reducing the costs and increasing the adoption of the IoT technology for common smartphone users [3].
- **AI on the edge**: IoT time-series data is collected from multiple sensors, often sampling at a very high frequency. Transferring the resulting large datasets over a congested network to the IoT cloud for remote processing is often impractical. Another challenge, particularly in e-healthcare scenarios [4], is user privacy, which restricts sending personal data externally. Another common concern is the the increased costs for cloud-based data storage and computation. As a result, companies are motivated to push AI capabilities to the edge to enable real-time actions in the field, *i.e.* as close to the source of data as possible.
- **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 [5]. These practices are expected to bring a 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** [6], [7] – an adaptation of DevOps practices to the ML domain, that aims to deploy and maintain ML models in production reliably and efficiently. To achieve this, MLOps promotes automation at all steps of the ML system lifecycle, including integration, testing, releasing, deployment and infrastructure management.

These three converging trends are changing the mobile

978-1-6654-6975-3/22/$31.00 ©2022 IEEE

software development paradigm and lead to an increased involvement of dedicated ML engineers in the agile development of mobile applications. 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 technological gap in terms of tooling support for ML engineers when it comes to mobile platforms. The use of Python, the *de facto* programming language in data science, as well as its integration with the native code are still quite limited in Android OS.[1] While it is in general possible to code stand-alone scripts and run the Python interpreter, there is a pressing challenge of integrating ML inference modules written in Python into larger apps as part of business logic responsible for data analytics [6].

To address this limitation, this paper presents an extensible architecture for rapid development of data analytics components on the Android 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. Thanks to the prominent plug-in architectural pattern, the proposed solution benefits from application logic extensibility, flexibility, and customisation. This becomes especially important in the context of DevOps and MLOps, which rely on frequent releases of software updates.

Accordingly, the contribution of this paper is three-fold: *i)* the formulation of the existing technological gap, which, according to the best of our knowledge, has not been explored neither by the academia nor the industry, *ii)* the conceptual plug-in architecture for deploying and running Python modules on the Android platform, and *iii)* a proof of concept implementation of the proposed architecture for fatigue detection using time-series sensor data from wearable fitness trackers.

The rest of the paper is organised as follows. Section II looks into the research context and the current state of practice, and describes the research gap from an MLOps perspective. Section III describes the conceptual architecture of the proposed approach. Section IV presents the proof of concept, demonstrating the feasibility of the proposed approach with a real-life application scenario. Section V summarises the results and outlines directions for further work.

II. Research Context and Problem Formulation

In recent years, the community has been actively developing 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 has primarily been on finding the right balance between available computing resources and the performance of the ML models. There are 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,[2] which allow training, deploying and running light-weight models on Android and iOS. The available SDKs support several mobile-native languages, but are tailored to smartphone-oriented scenarios, such as text, image, audio and video recognition, meaning that support for time-series sensor data processing is somewhat limited.

A. Use of Python in ML for time-series data analytics

The existing Android SDKs are not suitable for scenarios where a smartphone acts as an IoT gateway, because when dealing with time-series numeric data, an important task of an ML engineer is 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 use the same few mainstream modules (*e.g.* keras, tensorflow, pandas, scikit-learn, numpy and several others[3]).

There have been developed multiple cross-platform tools that allow running Python scripts almost anywhere, often within a non-Python execution environment, such as, for example, Android, which natively supports only Java and Kotlin code. To be able to run Python, the community has come up with several frameworks[4] that provide a bridge between the native Android and the external Python code. For example, QPython and Termux[5] offer a command line interface and a simple text editor for typing and running stand-alone Python code. The cross-platform frameworks BeeWare and Kivy[6] can be used to package Python code as Android apps with support for user interfaces and access to most Android services and hardware interfaces. These existing tools are good for quick prototyping of Python-only apps, but have limited support for integration with native Java/Kotlin code, especially in the context of enterprise-level projects that follow DevOps practices. A notable exception in this context is the Chaquopy library described in Section IV. Another possible solution to a smooth Java-Python interaction [10] describes an approach 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 containerisation middleware, which is not usually present on most users' smartphones.

Limited Python support forces mobile developers to code in native programming languages. Albeit compiled native code in Java is known to demonstrate better performance compared

[1] While the main focus of this paper is on Android OS, many of the described issues and concepts are also applicable to iOS.

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

[3] https://www.upgrad.com/blog/top-python-libraries-for-machine-learning

[4] https://wiki.python.org/moin/Android

[5] https://termux.com, https://www.qpython.com

[6] https://kivy.org, https://beeware.org

to an interpreted code in Python [11], native languages are not well-suited for ML-oriented tasks due to complexity and rigidity, resulting in more time spent on code structuring and management and leading to slower development pace. As a result, the universal adoption of MLOps in mobile software development to achieve agile and rapid integration of ML features is hindered.

This paper aims to address these challenges by allowing ML engineers to be fully involved in agile mobile development. The main goal of this research effort is to bridge the gap between Java-driven DevOps and Python-driven MLOps in the context of mobile software development. Both disciplines are mature enough on their own, with multiple commercially-available products on the market. However, their parallel use within a mobile development project by a diverse team of Android developers and ML engineers is yet to be explored.

B. Characterising the Problem from an MLOps Perspective

Business requirements for local and timely data processing on the mobile edge tend to continuously evolve, triggered by, for example, updated user requirements or newly-added hardware sensors. This naturally calls for a loosely-coupled architecture, where frequently updated components handling the actual data processing can be modified 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. The separation of concerns also assumes that ML engineers can continue coding their components in a programming language they are most proficient with.
2) **Agile support for frequent updates**: New features added by team members should have minimum effect on the rest of the running system, meaning that incremental code updates are applied in a safe, isolated and non-blocking manner.
3) **Understanding of the data on both ends**: it is important that the data ingested into the application on one end is correctly and unambiguously interpreted and fed into an ML model on the other end. Therefore, it is required to enable end-to-end communication of the data format and structure.

III. Proposed Approach: Plug-in Architecture

The described challenges have traditionally been addressed by applying various software decomposition techniques, such as the service-oriented architecture (SOA) [12] or component-based software engineering [13]. Another prominent pattern, particularly useful for loosely-coupled systems composed of two main elements – *i.e.*, a relatively stable core part and multiple dynamically evolving extensions – is the **plug-in architecture**. It relies on the principle of allowing adding features as plug-ins to the core application, providing extensibility, flexibility, customisation, and isolation of application features. This way, the specific application logic of separate software modules is separated from the core system. At any given point, including run-time, plug-ins can be added, removed, and changed with little or no effect on the rest of the core system or other plug-in modules.

The **core system** is often defined as the general business logic which provides the bare minimum for the application to function. The specific implementations of that functionality are up to individual plug-ins. The core also often 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** are stand-alone, independent components that contain specialised processing code and additional features, which are meant to enhance or extend the core system to produce additional capabilities. The core system declares extension points, usually in the form of a well-defined API, that plug-ins can hook into. The core also keeps track of loaded plug-ins through some form of registry, which includes information about available plug-ins and the protocols for accessing them.

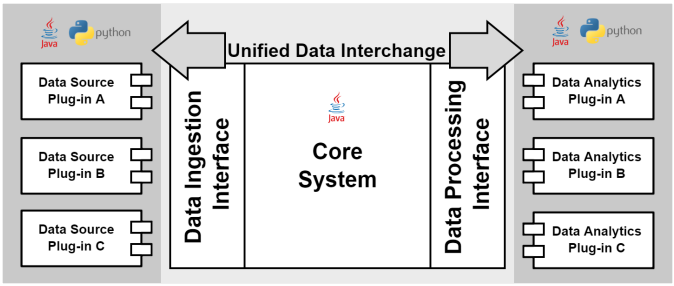

Fig. 1. Conceptual architecture for data analytics on an Android gateway.

In the context of time-series data analytics, the envisioned plug-in architecture (depicted in Fig. 1) should address the following threefold functionality:

- **Data Ingestion**: this covers various heterogeneous sources of data that will be handled in a mobile app. The core system defines a data ingestion interface, which custom plug-ins are expected to implement. On a smartphone gateway, these may include, for example, streaming data from wearable devices or built-in sensors, SQL results from the Android-native SQLite DB, CSV data from a local file, *etc.*
- **Data Processing**: this covers the actual data analytics logic applied to input data in the context of the given mobile app. The core defines a data processing interface to be implemented by custom plug-ins. Such custom data processing may range from simple arithmetical operations to advanced ML using deep learning techniques.
- **Unified Data Interchange**: while it is sometimes possible to infer the data schema (attributes, types, column order, *etc.*) from the data itself, it is still important to communicate this explicitly [14]. Therefore, a key pre-requisite for correct data processing is to ensure that input values provide by the data ingestion plug-ins is properly 'understood' on the other end by the data processing plug-ins.

IV. Proof of Concept

The described plug-in architecture was implemented as the **Android Time-Series** (AnTS) prototype framework,[7] depicted in Fig. 2. The implementation follows the functional and non-functional requirements described earlier. As a proof of concept demonstration, we will now explain how it is used for loading and processing time-series sensor data, collected by a wearable fitness tracker, while the Android smartphone acts as an IoT gateway.

A. Service Provider Interface for Plug-in Architecture

Service Provider Interface (SPI) was introduced in Java to make applications more extensible, as it allows third parties to enhance specific parts of a main product without modifying the core application. It naturally implements the modular plug-in architecture, including the support for dynamic discovery and loading of plug-in modules at run-time. There are three main elements underpinning the SPI mechanism:

- Service: in terms of the plug-in terminology, it is a well-defined interface that allows the core system to interact with plug-ins.
- Service Provider: it is a specific implementation of a Service, which hooks into the extension points provided by the core system. Each Service Provider implementation is placed on the application class path (typically as a JAR file) to be discovered and loaded, both at compile- and run-time.
- Service Loader: it is the mechanism for discovering and loading available plug-in implementations (*i.e.* Service Providers). As a pre-requisite, each Service Provider needs to be accompanied by a self-declaration file that associates it to a specific Service. Service Loader also acts as the plug-in registry by caching and keeping track of already loaded Service implementations.

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

B. Chaquopy for Bridging Java and Python

A promising library **Chaquopy Python SDK**[8] allows directly invoking Python scripts from within Android-native code, thus enabling co-operation between Java code and Python scripts. Distributed as a JAR library, it can be integrated using standard build automation tools such as Gradle and Maven. This proves especially useful within a team of Android developers coding in Java and ML engineers coding in Python. Accordingly, in this work we used the Chaquopy SDK to enable user-customised Python extensions responsible for various ML-related data management tasks, on the sides of both data ingestion and data processing.

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

[8]https://chaquo.com/

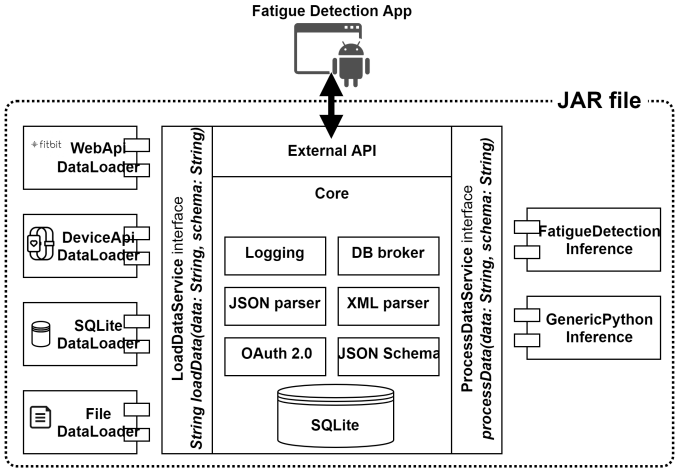

Fig. 2. AnTS framework for fatigue detection using fitness trackers.

C. JSON Schema for Unified Data Interchange

In a relatively small development team, an ML engineer has full understanding of the data and will solely implement both data ingestion and data processing plug-ins. With more people involved, it becomes a pressing concern to implement a mechanism for communicating the data schema from data sources to ML modules. To address this, we used JSON Schema[9] – a modelling vocabulary for annotating and validating JSON documents.[10] It provides clear human- and machine-readable documentation and is widely used for all kinds of automated validation of input JSON data. The use of this standard formalism in our work has two main benefits. First, it allows the data processing plug-ins to understand the input data format. Second, it can also be used to automatically check that input data is indeed correct and complies with the schema, using standard software libraries. Standard validation checks may include data type, format, range, uniqueness, presence, cardinality and many others.

D. Running Example: Fatigue Detection Using Fitness Trackers

The proof of concept was validated in the context of a fatigue detection mobile app. The app deploys and runs ML models on an Android smartphone, which acts as an IoT gateway for physiological data (*e.g.* sleep activity, physical activity, and heart rate) collected by Fitbit fitness trackers.[11] Currently, there are two abstract Services exposed by the core system, whereas the actual Service Provider implementations are contained in the following plug-ins:

1) LoadDataService represents various sources of data to be ingested into the analytics plug-ins. Apart from some other

[9]https://json-schema.org/

[10]In the context of this paper we primarily focus on the JSON format. It is also possible to use similar standard tools for structuring and validating CSV input data, albeit it is less challenging.

[11]Data collection, feature engineering and model training, albeit fundamental parts of the overall implementation, 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.

auxiliary functionality, it declares the main method loadData() to be implemented by any Service Provider inheriting this class. Each implementation returns both the ingested data and the corresponding data schema. For the purposes of the fatigue detection scenario, the following Service Provider implementations were developed:

- WebApiDataLoader is responsible for fetching data from Fitbit Web API,[12] which offers multiple entry-points for parameterised querying of collected biomarkers.
- DeviceApiDataLoader provides real-time biomarker data from Fitbit trackers.[13] This plug-in has an importnat practical application when the symptoms of a growing fatigue need to be detected in an autonomous offline manner.
- SQLiteDataLoader is responsible for querying data from Android's native relational database SQLite. Being extremely light-weight, SQLite does not offer rich built-in data types (*e.g.* Timestamp for time-series data), but is, nevertheless, able to effectively store time-series data using string timestamps – a simple, yet highly efficient 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 loading previously recorded data from locally stored CSV and JSON files.

2) ProcessDataService defines an interface for various data processing and analytics components. It declares the main method processData(data: String, schema: String) to be extended by child implementations. For the purposes of the fatigue detection scenario, the following plug-in implementations were developed:[14]

- FatigueDetectionInference: a collection of fatigue detection inference modules were developed, varying in the underlying ML algorithms (*i.e.* regression and classification) and the input data (*e.g.* specific data features, the granularity of time-series data, *etc.*). The plug-in implementations themselves are coded in Java, whereas for invoking the time-series data processing and the actual ML models we relied on the Chaquopy SDK, enabling interplay between Java and Python and loading of the required Python libraries (*e.g.* tensorflow, pandas, numpy).
- PythonInference: using the Java-Python bridging capabilities of Chaquopy, we also implemented a general-purpose ProcessDataService plug-in for running any Python code out of the box. The ML engineer is only required to upload the updated ML artefacts (*i.e.* Python script, trained model, scalers) without touching the Java part of the plug-in. To a certain extent, this can be seen as a second-level plug-in architecture, where Python scripts are dropped on the plug-in classpath, and then loaded and executed at run-time.

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

[13]A mock-up implementation was used because third-party mobile apps are not allowed to directly access live data from a tracker without transferring it first to the cloud.

[14]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.

While we tested multiple algorithms and data features for fatigue detection, the best correlation was observed with users' physiological data related to heart rate, sleep and physical activity. Accordingly, the ingested data is accompanied by a corresponding JSON schema (a simplified version is depicted in Listing 1), which defines it as an array of user objects containing the static (*i.e.* age, weight, gender) and dynamic (*i.e.* time-series Fitbit measurements: heart rate, sleep, and physical activity) data. The data processing plug-ins are able to parse the schema using standard tools to correctly handle and validate the input data and run the fatigue detection inference.

Listing 1. A simplified JSON data schema for fatigue detection inference.

```
{
 "type": "array",
 "items": {
   "type": "object",
   "properties":
     {
        "userid": { "type":"int" },
        "age": { "type": "int", minimum: 0 },
        "weight": { "type":"number", "minimum":0 },
        "gender": { "type":"string" },
        "heart-rate": { <...> },
        "sleep": { <...> },
        "activity": { <...> }
     }
     "required": [ "userid", "age", "weight", "gender", "
         heart-rate", "sleep", "activity" ]
 },
 "minItems": 1
}
```

E. ML Engineer's Perspective

The target audience of the described approach is a team of Java developers and ML engineers working together on an AI-driven mobile app. Accordingly, one of the main challenges was to actively involve ML engineers in the agile app development process, allowing them to iterate on their Python components independently from the Java developers (this is outlined as non-functional requirements in Section II-B). The described plug-in architecture enables this 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 make the provided implementations 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 ingestion and 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 added using the proposed approach. In this context, creating a shared JSON schema defining the structure and the format of the ingested data becomes an important task. Depending on the application requirements, the schema can be quite expressive and complex, supporting various validation constraints. This becomes even more critical when several ML engineers work on various data ingestion and processing plug-ins, where there is a pressing need for unified data representation across all involved modules.

TABLE I
BENEFITS AND SHORTCOMINGS

Benefits	Shortcomings
• **Modularity**: because plug-ins are separate modules with well-defined interfaces, it is easier to quickly detect, isolate and fix emerging issues. • **Extensibility**: the application can be dynamically extended to include new data ingestion and processing 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 applications, which rely on iterative fine-tuning and optimisation of data sources and ML models. • **Separation of concerns and parallel development**: various app features can be coded in parallel as separate components, which allow ML engineers to be fully involved in the agile development cycle. Furthermore, by communicating the format of ingested data, several ML engineers can independently work on their respective modules in parallel.	• **The core is 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 thorough design with support for backward-compatibility in mind. • **Limited integration testing**: even if a plug-in is tested alone or against the core system, some issues may emerge only in combination with some other plug-ins. This slows down the testing process, especially if multiple independent parties develop their own plug-ins in parallel. • **Reduced performance**: it is required to carefully design the plug-in loading logic, so that only relevant and non-conflicting functionality is used at a time. On the other hand, if properly implemented, this can make the system more lightweight and increase the performance. It becomes especially important given that loading and executing non-native code in Android is usually associated with considerable performance downgrades [11].

V. DISCUSSION AND CONCLUSION

The described research effort is part of an R&D project dealing with remote patient monitoring and diagnosing, using data collected from wearable fitness trackers. While it has its own benefits and shortcomings, as summarised in Table I, the main addressed challenge was to allow data scientists responsible for ML modules to work on their part independently from the core Android app. As the work matured, it became clear that the addressed challenges are common across a wider range of application scenarios, where time-series sensor data needs to be locally processed on a mobile gateway in the presence of frequently changing business logic.

As far as future work is concerned, an immediate next step for us is to integrate our proposed solution with the existing MLOps tools in order to cover the whole automated lifecycle, *i.e.* from data collection, pre-processing and model training to deployment and operation, repeated in an iterative manner. This will also underpin the empirical evaluation, since one of the main advantages of the presented approach 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.

The SPI mechanism already natively supports dynamic plug-in discovery and loading. What is still missing is the targeted selection of a specific plug-in among several available alternatives. In the presence of multiple, often conflicting implementations, it is important to ensure that only correct plug-ins are loaded at a time. We are addressing this challenge by designing a classification taxonomy shared between the core system and plug-ins. Using a combination of taxonomy tags, it will be possible to uniquely annotate plug-ins and resolve their suitability at run-time in a context-aware manner. More complex constraint solving techniques [15], [16] will also be explored.

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] R. Dautov and S. Distefano, "Distributed data fusion for the Internet of Things," in *International Conference on Parallel Computing Technologies*, pp. 427–432, Springer, 2017.
[3] 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.
[4] 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.
[5] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.
[6] 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.
[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] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.
[12] 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.
[13] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the pieces together, addison-westley*, vol. 5, 2001.
[14] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich, "Data Validation for Machine Learning," in *MLSys*, 2019.
[15] H. Song, R. Dautov, N. Ferry, A. Solberg, and F. Fleurey, "Model-based fleet deployment of edge computing applications," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 132–142, 2020.
[16] R. Dautov, H. Song, and N. Ferry, "A light-weight approach to software assignment at the edge," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing*, pp. 380–385, IEEE, 2020.