Tiny-MLOps: a framework for orchestrating ML applications at the far edge of IoT systems

Mattia Antonini*, Miguel Pincheira†, Massimo Vecchio‡ and Fabio Antonelli§
OpenIoT Research Unit, DI Center, Fondazione Bruno Kessler, Trento, Italy
Email: *m.antonini@fbk.eu, †mpincheiracaro@fbk.eu, ‡mvecchio@fbk.eu, §fantonelli@fbk.eu

2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS) | 978-1-6654-3706-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EAIS51927.2022.9787703

***Abstract*—Empowering the Internet of Things devices with Artificial Intelligence capabilities can transform all vertical applications domains within the next few years. Current approaches favor hosting Machine Learning (ML) models on Linux-based single-board computers. Nevertheless, these devices' cost and energy requirements limit the possible application scenarios. Conversely, today's available 32-bit microcontrollers have much lower costs and only need a few milliwatts to operate, making them an energy-efficient and cost-effective alternative. However, the latter devices, usually referred to as *far edge devices*, have stringent resource constraints and host non-Linux-based embedded real-time operating systems. Therefore, orchestrating such devices executing portions of ML applications represents a major challenge with current tools and frameworks. This paper formally introduces the Tiny-MLOps framework as the specialization of standard ML orchestration practices, including far edge devices in the loop. To this aim, we will tailor each phase of the classical ML orchestration loop to the reduced resources available onboard typical IoT devices. We will rely on the proposed framework to deliver adaptation and evolving capabilities to resource-constrained IoT sensors mounted on an industrial rotary machine to detect anomalies. As a feasibility study, We will show how to programmatically re-deploy ML-based anomaly detection models to far edge devices. Our preliminary experiments measuring the system performance in terms of deployment, loading, and inference latency of the ML models will corroborate the usefulness of our proposal.**

I. Introduction

The convergence of cloud computing with the Internet of Things (IoT) and Artificial Intelligence (AI) has progressively started a disruptive transformation of several domains [1]. In the next future, giant volumes of data will be generated by billions of connected devices to the point that it will not be possible to transfer, process, and store all such data to the cloud. As an example, connected cars generate around 25 GB of data per hour [2]. These numbers suggest dismissing traditional cloud-centric approaches in favor of more sustainable, distributed computing paradigms. One of these approaches is Edge Computing, whose main founding principle is to process the generated data as close as possible to the streaming sources, leveraging the computational capabilities offered by connected devices such as IoT sensors and gateways [3]. Edge Computing promises to open up new opportunities to create novel, "latency-sensitive" and "context-aware" applications.

However, the global adoption of Edge Computing technologies is still far from being reached since no commonly agreed solutions or approaches are available to address all its inherent challenges [4]. Among these challenges, one of the most relevant is the effective orchestration (*i.e.,* the configuration, coordination, and management) of much more heterogeneous hardware and software toolchains than a more traditional cloud context. Furthermore, while it is clear that endowing IoT devices with some Machine Learning (ML) capabilities has the potential to transform all vertical applications within the next few years, a uniform approach to effectively host such capabilities on the tiniest elements of the technology stack is still missing. On the one hand, a simple solution is to host an ML model (*e.g.,* an anomaly detection model) on cost-attractive Linux-based single-board computers (SBC), available off-the-shelf for less than 15 EUR. The biggest downside of these SBCs is that they can burn hundreds of milliwatts (mW), giving them battery-powered lifetimes on the order of hours. On the other hand, 32-bit microcontrollers (MCU) cost less than 1 EUR, and they need only a few mW to operate, meaning that a device running on a coin battery has an expected lifetime of the order of one year [5]. However, these devices have some stringent resource constraints (a few KB of SRAM and storage and clock frequencies of just a few tens of MHz) and host non-Linux-based embedded real-time operating systems. In the recent literature, they are usually referred to as *far edge* devices [6]. Unfortunately, to orchestrate such non-Linux-based, constrained devices executing (even portion of) an end-to-end ML application represents a major challenge. Furthermore, orchestrating the process in an automated way is one of the keys to evolving ML applications.

In this paper, we formally introduce the Tiny-MLOps framework as the specialization of standard MLOps practices [7] when far edge devices are included in the loop, not as mere data producers but as computational elements of an ML model inference. Briefly, well-defined practices, processes, and software tools are essential to effectively managing today's ML artifacts (datasets, features, models, software code) and their life cycles in a flexible and automated way. This software engineering branch is often referred to as Machine Learning Model Operationalization Management (hence, MLOps). This paradigm is gaining much momentum in cloud environments [8] and will be thoroughly introduced in Section II. Unfortunately, the inherently limited hardware and software capabilities of far edge devices usually prevent adopting any of these MLOps tools and practices within the lowest layers of an IoT technology stack. As a result, such devices cannot be re-configured or adapted in an automated way while in operations. To alleviate this limitation, in Section III, we will

978-1-6654-3706-6/22/$31.00

tailor each phase of the classical MLOps loop to the reduced resources available onboard typical IoT devices. Then, in Section IV, we will highlight how the Tiny-MLOps framework can be used to deliver adaptation and evolving capabilities to a resource-constrained IoT monitoring device mounted on an industrial rotary machine to detect anomalies. To this aim, an initial ML-based anomaly detection model is loaded into the IoT device for local, unsupervised training. Then, based on a specific business logic defined in the IoT gateway, we will show how to deploy another anomaly detection model selected from a pool of available models, from the IoT gateway to the device, in a fully automated and programmable way. The validation scenario will be depicted in Section V, and a preliminary set of experiments measuring the deployment, loading, and inference time of an anomaly detection model will corroborate the feasibility (Section VI) of the proposed approach. Finally, Section VII will summarize the main outcomes and limitations of the proposed approach, also outlining the most attractive research and development directions.

II. Background

The recent literature portraits the MLOps paradigm as a loop comprising seven distinctive phases, namely Plan, Create, Verify, Package, Release, Configure, and Monitor [7]–[9]. For the sake of completeness, in the following, we will briefly introduce all these phases.

The first phase, *Plan*, involves a data scientist as the main actor and targets the study of the data sources, checks if a suitable *ground truth* is available, and identifies possible patterns that can be used to train one or more ML models. An important step of this phase is feature engineering, which is when data are processed using multiple approaches (*e.g.,* statistical, frequential, visual, etc.) to derive a set of processed features that later need to be selected. Next, during the *Create* phase, a possibly large set of different ML models (*e.g.,* SVMs, decision trees, artificial neural networks, etc.) can be trained and the feature set adapted, and changed with the objective of crafting effective models on the training data. After, the *Verify* phase checks how the model behaves in a production environment. This phase is essential for the subsequent ones since the data scientist understands if the developed model(s) is working properly (*e.g.,* sanity-check) and meets the target non-functional objectives. This is the last phase involving a data scientist; after this step, the main actor will be a system and operations engineer.

During the *Package* phase, first, the model is stored in a repository for future use, together with the used features and dataset, as well as the computed metrics and various experiment metadata. Then, the model and its processing pipeline are packaged into an executable binary file, according to the production runtime environment it will be executed on. In doing so, the engineer can follow two alternative approaches, namely model containerization and model embedding. The former approach requires hosting the model inside a container (*i.e.,* Docker container) and serving it using a proper set of Application Programming Interfaces (*e.g.,* a REST API). This approach follows a microservice paradigm [10] and allows the deployment of the model in an orchestrated environment like Kubernetes. The latter approach consists in embedding the model directly inside the main application code and, usually, this is done in runtime environments lacking modular programming. Next, during the *Release* phase, the engineer verifies that the model is suitable for the target production environment. In this phase, system performances (*e.g.,* memory utilization, CPU/GPU load) are extremely critical since any possible issue could make the application unresponsive. Similarly, in the *Configure* phase, some runtime models and application parameters can be fine-tuned to achieve the target objective. Finally, the last phase, *Monitor*, aims at detecting issues causing performance degradation, triggering an additional iteration of the MLOps loop. Here, on the one hand, the engineer is focused on system performances (*e.g.,* model inference time/responsiveness, system load, failures). On the other hand, the data scientist is more interested in detecting model degradation due to, for instance, input drift, outliers, biases, non-representativeness of input features.

In the recent literature, we could find sporadic works using the above described MLOps loop in the context of Edge computing [11], [12]. However, it is worth recalling that those works were using Linux-based edge devices (*e.g.,* an IoT gateway), hence relying on modern orchestration frameworks (*e.g.,* Docker and Kubernetes). On the contrary, we argue that the edge of an IoT network is typically populated by devices that could potentially exploit better the concept of locality of data. However, they can offer only limited resources in terms of energy, computing power, memory, and storage. Despite such limitations, in the next sections, we will show how these devices can host some ML capabilities in an orchestrated way, after proper model and process adaptation.

III. The proposed Tiny-MLOps paradigm

This section introduces an improved version of the classical MLOps loop, taking into account ML models life cycles on devices with limited resources. It could be considered as the natural evolution of MLOps, therefore we label such paradigm as Tiny-MLOps. To ease its introduction, Figure 1 depicts the phases of the loop, coloring them with two different shades of blue and green. The phases in blue are mainly carried out by data scientists, data engineers, and domain experts, while the green phases are mainly operated by software and operations engineers. The main differences with respect to the MLOps loop are described below.

- At the *Plan* phase, the target computing platform should already be known to design a set of features that can be efficiently computed on the embedded device. For instance, some neural network layers may not be executable on memory-constrained devices;
- At the *Create* phase, the application and the ML algorithm can be partially carried out directly on the constrained device to exploit the principle of locality. This

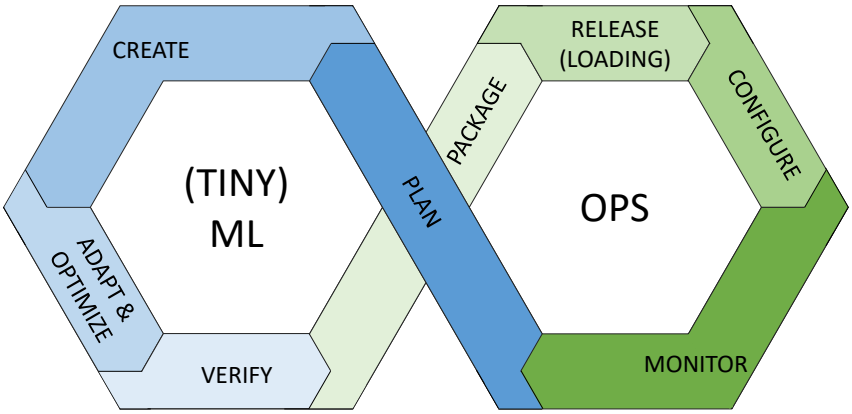


Fig. 1: The Tiny-MLOps loop derived from the MLOps loop.

allows to better learn the problem from the production environment;

- An additional phase, called *Adapt and Optimize*, is added between the *Create* and the *Verify* phases. This phase aims to optimize the model trained in the previous step to fits in the target computing platform;
- The *Verify* phase adds additional checks about the model since all the operations required to perform an inference need to be supported by the target platform;
- At the *Package* phase, the model needs to be further processed, or "compiled", to be executable on the target execution platform. The model may be subject to many conversion steps where its structure may change to fit the hardware architecture;
- The *Release* phase for Tiny-MLOps does not consider the deployment of the packaged model over the infrastructure but the loading in memory of the model itself. This operation is critical since it may saturate the device memory or take a long time, even minutes;
- The *Configure* phase may change some run-time parameters of the model (e.g., decision thresholds) to adapt to the production environment;
- Finally, the *Monitor* phase continuously checks the model health and execution (*e.g.,* latency, behavior, energy consumption, etc.) and triggers another iteration of the Tiny-MLOps loop if needed. The principle of locality can be exploited to provide additional information to the next iteration.

The Tiny-MLOps loop enables an iterative design, development, and deployment of ML applications even on constrained devices with limited resources. One of the main benefits is exploiting the locality of devices to better learn from the surrounding environment [13]. In such scenarios, one common problem is the missing ground truth.

An example is a condition monitoring application that implements an anomaly detection algorithm to detect early signs of failures [14]. In this case, the dataset used for the training may not be labeled; thus, the algorithm itself should be able to detect anomalies, even if they are rare or absent.

IV. Tiny-MLOps for anomaly detection scenarios

Our technological playground comprises a far edge device endowed with some sensing and communication capabilities and an IoT gateway able to communicate with it. Briefly, thanks to a specific networking protocol, the device and the gateway can communicate with each other, transferring information back and forth. Finally, the far edge device is installed close to a monitored asset (*e.g.,* a rotary engine, a vibrating chassis, etc.) so as to sample with its onboard sensors a physical phenomenon. The assumptions (verified in the next section) are that i) the IoT gateway can programmatically update the application logic of the far edge device using the communication channel; ii) the far edge device can host and execute a complete ML-based anomaly detection algorithm.

With these assumptions, we can describe the proposed anomaly detection system, depicted in Figure 2. When the digital samples reach the embedded device from the sensors (*e.g.,* using an SPI or an I2C hardware protocol), they are managed by the *Data In manager* component, which is the entry point of the Tiny-MLOps loop.

1) The *Data In manager* buffers and windows the incoming data, then sends them to the *Local Training Engine* running on the embedded device. This module applies a pre-processing pipeline to extract a pre-defined set of features and trains a local AD model, which we call the Champion model (CM), using data samples coming from the real environment.
2) Once the model is trained, it is forwarded to the *Model Deployment Agent* that locally packages the model for the local deployment together with the pre-processing pipeline, which performs the feature extraction.
3) Following, the *Model Deployment Agent* deploys the model and the pipeline in execution within the *Local Inference Engine*. Moreover, an intermediate step may be necessary to configure some parameters of the model, *e.g.,* the decision threshold over the model outcome.
4) Then, the Champion model is placed and executed by the *Local Inference Engine*.

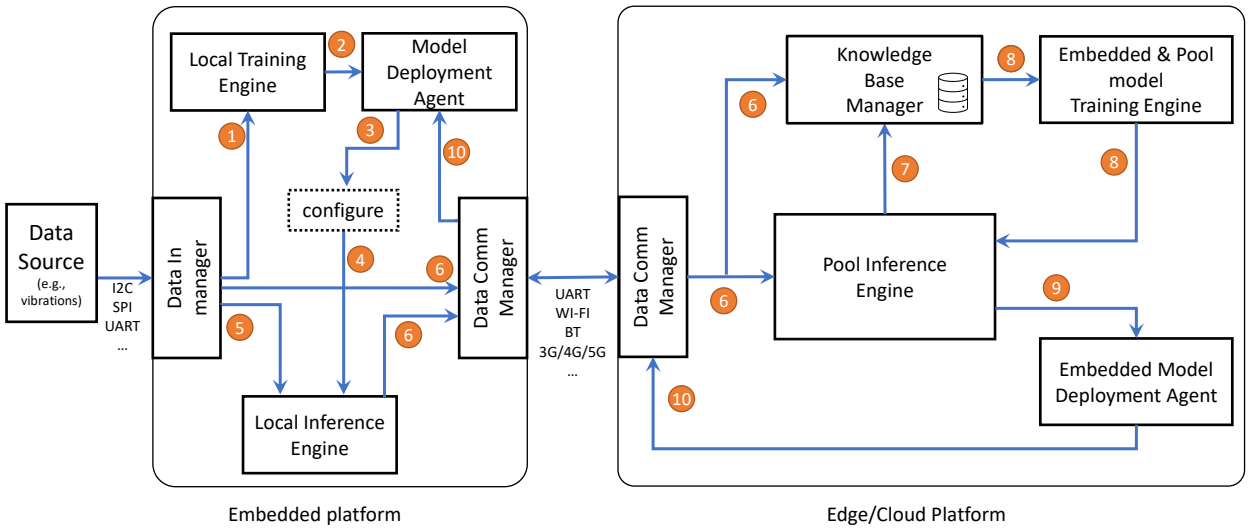


Fig. 2: The anomaly detection system proposed and supported by the Tiny-MLOps loop. Numbers inside balloons refer to the steps described in Section IV.

5) At this point, the *Data In manager* forwards incoming data to the *Local Inference Engine*. The Champion model is inferred and returns if the processed data are associated with an anomalous or normal behavior of the monitored machine.
6) Periodically, besides the inference of data with respect to the AD model, data associated with the normal behavior of the machine (the weak labeling is given by the Champion model) are forwarded to the edge gateway (*i.e.,* the edge/cloud platform) and stored in the *Knowledge Base manager*. These data can be used for further analyses by data scientists. Data are moved from one computing entity to the other via the *Data Comm managers*, which handle more complex networking protocols. Moreover, every time the Champion model detects an anomaly, the data are sent to the edge gateway and inferred by a pool of models executed by the *Pool Inference Engine*. This model pool, which may execute complex models and algorithms, has to decide if the incoming weakly-labeled anomalous data are associated with an anomaly or not. The pool can adopt different decision strategies to declare the anomaly. A simple strategy can be that the majority of models have detected an anomaly. However, the pool can declare that the anomaly identified by the Champion model on the embedded device is a misdetection, or even better, a false positive.
7) The outcome of the decision taken by *Pool Inference Engine* is associated with the instance of data coming from the embedded device and stored by the *Knowledge Base manager* for future analysis. Moreover, the *Pool Inference Engine* assesses the candidate embedded models stored in the repository within the *Knowledge Base manager*. Here, a rewarding strategy is used to rank the models, *i.e.,* a score is assigned to every model and is increased by +1 when the model detection is aligned with the *Pool Inference Engine* main outcome or decreased by -1 when it is not aligned. Then, models are sorted by their ranking.
8) The *Knowledge Base manager* stores the data associated with the normal and anomalous behaviors, but it also stores the models trained for the *Pool Inference Engine* and the embedded device. Indeed, the data stored can be transferred to the *Embedded and Pool model Training Engine* to train (or retrain) models that can be deployed in the *Pool Inference Engine* or sent to the embedded device. The *Embedded and Pool model Training Engine* module can also optimize the model's hyperparameters against different objective functions, perform feature selection, and feature engineering (a data scientist can interact with this module to push new models or features algorithms). Once the model is ready, it is packaged and be put in production.
9) The *Pool Inference Engine* keeps monitoring the detection performance of the Champion model deployed in the embedded device. If the Champion model starts to fail (*e.g.,* the number of false-positive detection in a time window is above a threshold), the *Pool Inference Engine* can trigger the replacement of the Champion model to improve the system's performance. Thus, the model is selected by choosing the model with the highest ranking (see description in point 7) and forwarded to the *Embedded Model Deployment Agent*.
10) Once the *Embedded Model Deployment Agent* receives the model to deploy over the embedded platform, it packages the model together with the feature extraction pipeline and forwards the package to the *Data Comm manager*, which sends it to the embedded device. The *Data Comm manager* on the side of the embedded device receives the model package, verifies if the communication was error-free, and in case asks for

a re-transmission. Finally, the package is passed to the *Model Deployment Agent* that will replace the Champion model in the *Local Inference Engine* by following the procedure described starting from step 3.

Following this process, and thanks to the proposed Tiny-MLOps framework, the system can self-adapt to the specific conditions of the monitored asset.

V. Validation scenario

To validate the proposed system design, we consider a real-world industrial scenario consisting of a wastewater management plant. Multiple pumps are deployed in different pools to mix and move water across the different cleaning stages. Each pump is composed of a fan designed for the target water density and connected to an electric engine with a shaft. Then, on top of each engine, there is a sealed chamber isolating the terminal block (that is where the power line is connected) from the water.

The trial's goal is to demonstrate the applicability of the proposed system design and, more specifically, the ability to deliver an ML-based condition monitoring application as near as possible to the submersible pump, in compliance with the locality principle imposed by the Edge Computing paradigm. Therefore, a far edge device hosting the pre-processing pipeline and the anomaly detection model algorithm is placed inside the terminal block chamber. It is important to recall that the operational conditions of the monitoring device are harsh, if not extreme: the access to the monitored asset is difficult after deployment, while the communications and power supply impose strong engineering challenges. Indeed, only a high-power cable is connected to the pump; therefore, device powering and communications happen along the same cable, using Power Line Communication (PLC) characterized by limited bandwidth (*i.e.,* 2.4 kbaud). Furthermore, an external IoT industrial gateway hosts the software components and services required to enable the Tiny-MLOps functionalities. We will demonstrate that, by leveraging the Tiny-MLOps paradigm, the anomaly detection system can self-adapt to the specific operating conditions of the monitored asset. In particular, i) the far edge device can autonomously train the anomaly detection model using data from its sensors; ii) the gateway provides continuous monitoring of the system's detection capabilities, triggering model re-deployment based on specific business logic, in a fully programmable way.

A. Dataset preparation

The target scenario makes the experimentation not cost-effective. Indeed, periodic inspections of the pumps are expensive since each time a pump is inspected, it is extracted from the pool, opened, inspected, bearings and gaskets are replaced (regardless of their status), closed, and plunged back. This maintenance process prevents us from having a preliminary sensor data acquisition campaign. With the objective of minimizing operational costs without loss of generality, we emulate sensor data production. Specifically, instead of generating our own dataset, we resort to the NASA Bearing Dataset [15], [16]. The latter is a well-known dataset made publicly available by the University of Cincinnati, containing three sets of vibrational recordings saved in 1-second length files describing different test-to-failure experiments. Each file contains 20,480 rows sampled at 20kHz. For our validation, we consider only the first set composed of 2,156 files, which have been created by sampling 1 second of signal every 10 minutes (except the first 43 files that were recorded every 5 minutes). For our analysis, we sub-sampled the dataset with three different sampling frequencies (f_s): 500 Hz, 1 kHz, and 2 kHz. Sub-sampling was performed by applying the FFT to get the frequency spectrum of the original signal, then filtering with a rectangular low-pass filter with a cut-off frequency at $f_s/2$, then transforming back the signal in the time domain. For every file, we have the vibrations data related to four different bearings over two orthogonal axes, *i.e.,* x and y.

B. The embedded device

The first component of the prototype is the far edge device, hosting the pre-processing pipeline and the anomaly detection model. We chose an ESP32-ROVER-E DevKit V4 from Espressif, available off-the-shelf for less than 10 EUR yet provides a dual-core microcontroller unit (MCU) with 4 MBytes of RAM and 4 MBytes of flash memory. This device was flashed with a MicroPython firmware[1], a lean and optimized open-source implementation of Python 3.4 for MCUs. Briefly, it provides a runtime environment in pure Python. Therefore, we codified the components and the pre-processing pipeline depicted on the left-hand side of Figure 2 in Micropython. More in detail, the MCU interprets the Python code stored in the device flash memory, generating the bytecode that is executed by the ESP32. Moreover, this firmware supports multiple features of the Python languages like modules. In this way, we could integrate the *ulab* package[2], a NumPY-like library to manipulate arrays in MicroPython and implement a subset of methods of NumPY [17] and SciPy [18] modules. Additionally, we adapted and integrated a pure-Python implementation[3] of the Isolation Forest algorithm [19] and the pySerialTransfer[4] library. The former module implements code to train and infer Isolation Forest models, while the latter module implements methods to send data over UART links using a predefined structure (frame). We use this module to communicate over two different UART channels: the first is used to emulate the vibration data source by receiving the sub-sampled data; the second is necessary for the communication between the board and the gateway, emulating a PLC channel at 2.4 kbaud.

With this developing environment, we could design, test, and then package and deploy the anomaly detection model together with the feature extraction script as a Python module,

[1]https://micropython.org/
[2]https://github.com/v923z/micropython-ulab
[3]https://github.com/msimms/LibIsolationForest
[4]https://github.com/PowerBroker2/pySerialTransfer

loading it at runtime without the need to flash the firmware of the MCU.

C. The edge gateway

The other component of the prototype is the gateway that hosts the services to support the anomaly detection application, as depicted on the right-hand side of Figure 2. We implemented the gateway using a Raspberry Pi 4B board with 8 GB of RAM running RaspbianOS derived from Debian 10 (buster). More in detail, the gateway hosts the *Knowledge Base manager*, which stores in a series of files the sampled data associated with the normality and anomalous labels, the repository and the training engine for models of the *Pool Inference Engine* and for the embedded device, and the *Embedded Model Deployment Agent*. The training engine for embedded devices trains Isolation Forest models using the same library used for the embedded device.

Regarding the training of models for the pool, for validation purposes, the *Pool Inference Engine* (PIE) executes a set of models selected from the model repository for the embedded device. Selected models share the same training set with the model deployed on the embedded board but different hyper-parameters (*e.g.,* number of trees, subsampling size, etc.). Consequently, we have only one model repository in our validation scenario.
Every time the PIE receives an instance of data, it infers the models then apply a majority strategy over models' outcome to declare if an anomaly actually happened. Moreover, the PIE uses the pool outcome to keep track of the performance of the *Champion Model* (CM) deployed in the ESP32 embedded board. Internally, it implements an assessment and ranking strategy to identify the best candidate model within the repository by assigning a score from candidate's result with respect to the pool outcome. If the outcomes are aligned, the candidate model score is increased by one unit, otherwise it is decreased by one unit. The model with the highest positive score is selected as the next CM when the deployment condition happens. In our application, the CM is replaced once the pool has detected N_{fp} false positives; that is when the CM has identified N_{fp} anomalies in contrast with the pool. In our implementation, we set $N_{fp} = 5$. Then, the *Embedded Model Deployment Agent* (EMDA) packages the model for the embedded device by creating a Python module fully compatible with the MicroPython runtime. Such package contains four files: the model structure (*e.g.,* parameters and weights) in JSON format, the Python script for feature extraction, a JSON configuration file with information about the deployment (*e.g.,* model name, the threshold for the decision over the IF output, the size of the time window used for the sample, etc.), and the *__init__.py* file that is needed to load the module in the Python runtime.

VI. Preliminary results

The anomaly detection system and the TinyMLOps methodology presented in this paper are at their preliminary stages; thus, the performance evaluation of the system in terms of accuracy is out of scope. However, we provide the system metrics that describe the behavior of the system at runtime.
First, we train a set of Isolation Forest models using the first 60% of the useful life of The Nasa dataset sub-sampled at 500 Hz; thus, this does not correspond with the effective 60% of data since there are some time windows where the data were not sampled. We computed the bearing life from the timestamps associated with every instance. Moreover, from literature, faults are more likely after 70% of the useful life [20]. We train five models for each bearing by setting the subsampling size to 100 and the number of trees of the ensemble equal to 10, 20, 30, 40, and 50. Overall, we have a model repository of 20 models. Models are fed using the features (*i.e.,* kurtosis, skewness, and mean) computed over continuous non-overlapping time windows of 500 ms. These models are used for the PIE and, eventually, to deploy a new *Champion Model* on the embedded device. Finally, given that the Isolation Forest algorithm returns a normalized score between 0.0 and 1.0 associated with the likelihood of an anomaly, we set a threshold at 0.6; thus, we declare that the model has detected an anomaly if the score is greater than the threshold.

The graphs Figure 3 depict the time performance of the Isolation Forest algorithm running in the MicroPython environment over an ESP32 board. More in detail, Figure 3a and Figure 3b show the average time required to perform one inference using the data coming from bearing 1 and bearing 3, respectively. We performed 200 inferences per forest size, then we computed the average. For instance, the inference time over the bearing 1 set is between, on average, from 8.7 ms (std. 1.3 ms) when the model has 10 trees to 34.4 ms (std. 5.0 ms) when the model has 50 trees. On the bearing 3 set, the inference time is, on average, between 10.44 ms (std. 2.57 ms, 10 trees) and 38.88 ms (std. 6.11 ms, 50 trees). Moreover, these plots highlight the behavior of the Isolation Forest algorithm; indeed, the average time required to perform an inference when data are anomalous is lower than the time required for one inference over normal data. This is due to the average path length along with the trees that is shorter when there is an anomaly.
Figure 3c shows that the time required to load a model from the flash storage to the main memory has a linear trend with respect to the size of the forest. Differences among different bearings are related to the different structure of trees inside forests.

As we presented above, the PIE may decide that the deployed CM is not valid anymore; thus, we need to replace it. However, the communication channel between the embedded device and the gateway offers a baudrate of 2400 baud in half-duplex mode since it emulates the PLC channel between the devices. Given these constraints, the new CM needs to be fragmented before sending. Figure 4a shows the estimation of the PLC channel occupation for the different models trained over the different datasets and with different forest sizes, which

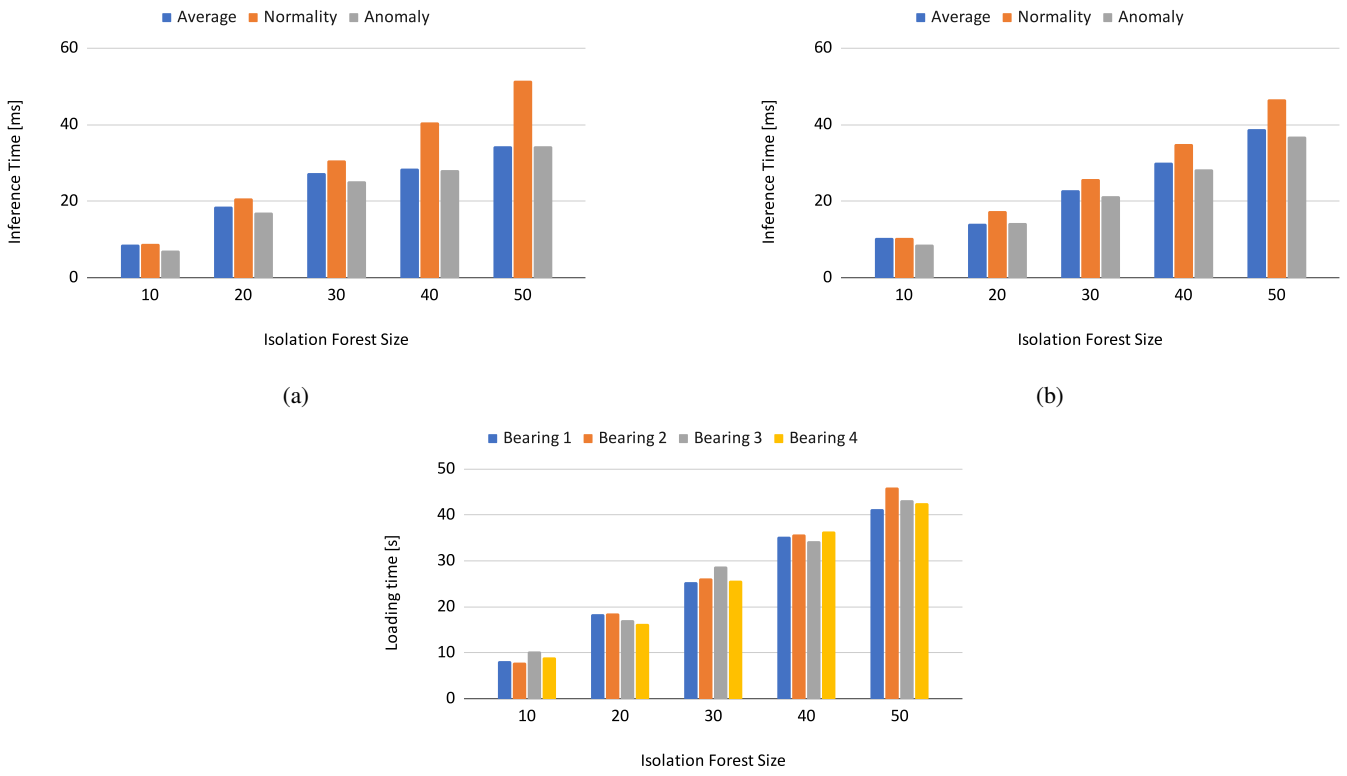


Fig. 3: Average inference time of IF over Bearing 1 data (a), over Bearing 3 data (b), and average loading time (from flash) of IF models over the ESP32 platform.

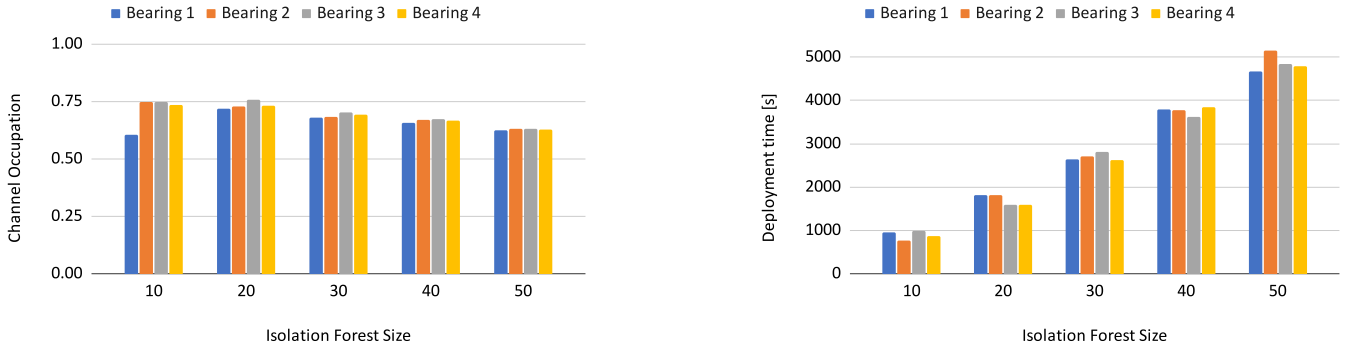
is around 0.6-0.7 for every model. We note that the estimated channel occupation decreases with the increase of the number of trees in the forest. We conjecture that this is due to the higher number of frames since, every time a frame is sent from the gateway, the ESP32-based device has to reply with an acknowledgment packet to ensure that transmission was error-free; otherwise, the packet is resent. On the other side, Figure 4b highlights that the time required to deploy a model increases when the forest size increases, and this is an expected behavior since the model is bigger. The deployment time is, on average, between 759 s (bearing 2, 10 trees) and 5147 s (bearing 2, 50 trees). As a side effect, it is important to consider how much time is required to deploy the model since it plays a fundamental role in the application's responsiveness. However, in some scenarios, a deployment time of one hour or more cannot be suitable.

VII. Conclusions and Future works

The combination of the Internet of Things and Artificial Intelligence, with the rising of novel computing paradigms (*i.e.,* Edge Computing), leads the next generation of intelligent applications. Indeed, novel applications follow well-established methodologies to deliver effective features continuously. A similar approach has been borrowed to machine learning, where the delivery of models follows an iterative process, called MLOps, that combines the work of data scientists and operational engineers. This has been adopted for applications with models typically hosted by Linux-based devices, with a consequent high energy footprint that limits their adoption in multiple scenarios.

A natural evolution of MLOps is its adaptation to comprise the class of constrained embedded devices that typically populate the very far edge of the network. Despite their limited resources in the order of a few hundreds of KB of RAM/flash, these devices could indeed offer enough capabilities to execute purposely adapted ML models.

In this paper, we formalized the Tiny-MLOps approach that has the goal to deliver and orchestrate ML models at the far edge of the network. We described the different phases that have to be tackled in order to design and deliver an application with our approach. Then, we applied it to the design of an anomaly detection application in the far edge. Finally, we implemented a working prototype using the framework and evaluated some system metrics, such as deployment, loading, and inference latency of some ML models. These promising results seem to corroborate our challenging vision: if correctly used, the proposed Tiny-MLOps approach can support the delivery of AI capabilities at the far edge of the IoT systems. The implemented system is at its early stages of development; therefore, it suffers from a few known limitations. First, the



(a) (b)

Fig. 4: Estimated of channel occupation during deployment (a) and time required to deploy a model (b).

deployment of a new model may take hours to be completed since the model is stored in a non-optimized format for the target device. Moreover, the system may be sensitive to false positives since the pool of models may fail in the detection of anomalies.

The work presented in this paper paves the way to a novel approach to design and deliver intelligence in the far edge. Thus, it opens multiple challenges and opportunities like how to optimize the deployment of models? How to select the best model to deploy in the far edge? Is it possible to exploit federated approaches to improve the delivered model? And, how?

ACKNOWLEDGMENT

This work was partially co-funded by the European Union through the EIT Manufacturing's 2022 KAVA call: MOLIERE - MultipurpOse fLexible hIgh rEsolution stRain sEnsor (project number 22101); and by "Legge provinciale 6/99 sugli incentivi alle imprese" (L.P. 6/99) della Provincia Autonoma di Trento, through the "EdgeMetrix 5Gi4.0" project. The co-authors want to thank the Phox and Cinetix Group engineering team for their support, especially A. De Bernardi, E. Salandin, L. Carnaghi, and D. Colombo.

REFERENCES

[1] Z. Chang, S. Liu, X. Xiong, Z. Cai, and G. Tu, "A survey of recent advances in edge-computing-powered artificial intelligence of things," *IEEE Internet Things J.*, vol. 8, no. 18, pp. 13 849–13 875, 2021.

[2] D. Snaidauf, "Connected vehicles shift an industry - on the road to predictive maintenance and safety," Deloitte, Tech. Rep., 2018.

[3] M. Vecchio, P. Azzoni, A. Menychtas, I. Maglogiannis, and A. Felfernig, "A fully open-source approach to intelligent edge computing: AGILE's lesson," *Sens.*, vol. 21, no. 4, p. 1309, 2021.

[4] Z. K. Wazir, A. Ejaz, H. Saqib, Y. Ibrar, and A. Arif, "Edge computing: A survey," *Future Gener Comp Sy*, vol. 97, pp. 219–235, 2019.

[5] H. Doyu, R. Morabito, and J. Holler, "Bringing machine learning to the deepest IoT edge with TinyML as-a-service," *IEEE IoT Newsletter*, vol. March, 2020.

[6] I. Sittón-Candanedo, R. S. Alonso, J. M. Corchado, S. Rodríguez-González, and R. Casado-Vara, "A review of edge computing reference architectures and a new global edge proposal," *Future Gener Comp Sy*, vol. 99, pp. 278–294, 2019.

[7] M. Treveil, N. Omont, C. Stenac, K. Lefevre, D. Phan, J. Zentici, A. Lavoillotte, M. Miyazaki, and L. Heidmann, *Introducing MLOps*. O'Reilly Media, 2020.

[8] S. Alla and S. Adari, "What is mlops?" in *Beginning MLOps with MLFlow*. Berkeley, CA: Apress, 2021.

[9] M. M. John, H. H. Olsson, and J. Bosch, "Towards MLOps: A framework and maturity model," in *Proc. of the 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021, pp. 1–8.

[10] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Softw.*, vol. 34, no. 1, pp. 91–98, 2017.

[11] E. Raj, D. Buffoni, M. Westerlund, and K. Ahola, "Edge MLOps: An automation framework for aiot applications," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 191–200.

[12] C. Min, A. Mathur, U. G. Acer, A. Montanari, and F. Kawsar, "SensiX++: Bringing mlops and multi-tenant model serving to sensory edge devices," *arXiv preprint arXiv:2109.03947*, 2021.

[13] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review," *Sensors*, vol. 20, no. 9, 2020.

[14] L. Liu, D. Liu, Y. Zhang, and Y. Peng, "Effective sensor selection and data anomaly detection for condition monitoring of aircraft engines," *Sensors*, vol. 16, no. 5, p. 623, 2016.

[15] H. Qiu, J. Lee, J. Lin, and G. Yu, "Wavelet filter-based weak signature detection method and its application on rolling element bearing prognostics," *Journal of sound and vibration*, vol. 289, no. 4-5, pp. 1066–1090, 2006.

[16] J. Lee, H. Qiu, G. Yu, J. Lin *et al.*, "Rexnord technical services: Bearing data set," http://ti.arc.nasa.gov/project/prognostic-data-repository, Moffett Field, CA: IMS, Univ. Cincinnati. NASA Ames Prognostics Data Repository, NASA Ames, Tech. Rep., 2007.

[17] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[18] P. Virtanen *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in python," *Nat. Methods*, vol. 17, pp. 261–272, 2020.

[19] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proc. of the 8th IEEE International Conference on Data Mining (ICDM)*. IEEE, 2008, pp. 413–422.

[20] J. Cavalaglio Camargo Molano, M. Strozzi, R. Rubini, and M. Cocconcelli, "Analysis of nasa bearing dataset of the university of cincinnati by means of hjorth's parameters," in *International Conference on Structural Engineering Dynamics ICEDyn 2019*, 2019.