Future Generation Computer Systems 155 (2024) 447–462

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Expanding the cloud-to-edge continuum to the IoT in serverless federated learning☆

Davide Loconte [a], Saverio Ieva [a], Agnese Pinto [a], Giuseppe Loseto [b], Floriano Scioscia [a,*], Michele Ruta [a]

[a] Polytechnic University of Bari, Via Orabona 4, Bari, I-70125, BA, Italy
[b] LUM "Giuseppe Degennaro" University, Strada Statale 100 km 18, Casamassima, I-70010, BA, Italy

ARTICLE INFO

Keywords:
Cloud-to-edge continuum
Cloud-to-things
Serverless computing
Internet of Things
Federated learning

ABSTRACT

Serverless computing enables greater flexibility and efficiency in the cloud-to-edge continuum. Artificial Intelligence and Machine Learning (AI/ML) applications benefit greatly from this paradigm, as they need to gather, preprocess, aggregate and analyze data at various scales. In such contexts, the increasing hardware/software resource availability of Internet of Things (IoT) devices provides the opportunity to exploit them not only as data sources in AI/ML infrastructures, but also as computational nodes for model training and inference; nevertheless, comprehensive frameworks are still mostly missing. This work introduces an innovative serverless computing architecture which expands the cloud-to-edge continuum toward IoT devices. The same functions can run on IoT, edge and cloud nodes with minimal to no code modification and they can be invoked through a uniform interface. A federated learning framework is defined based on the proposed architecture, exploiting an existing IoT-oriented ML algorithm in a novel way. Notably, IoT nodes are used for both federated training and local inference tasks. A full prototype implementation has been built with off-the-shelf technologies and devices. A case study on federated machine learning for activity recognition and experiments have been conducted to validate key elements of the proposal.

1. Introduction

The growth of real-time data streams generated by Internet of Things (IoT) devices poses availability, scalability and performance issues to cloud-based large-scale centralized collection and processing. The *edge computing* paradigm enables intelligent analysis at a smaller scale, in a more localized way, closer to data sources in the field, thereby decreasing latency and bandwidth usage, improving privacy and mitigating the effects of infrastructure failures and security vulnerabilities. The impact of edge computing is revealed by several indicators [1], including: (i) worldwide hardware market size, projected to grow from 9 to 146 billion U.S. dollars in the 2019–2028 decade; (ii) organizations' rising awareness of benefits and opportunities of improving latency, bandwidth, security, and dependability issues through edge computing; (iii) research interest, with a 35-fold increase in the number of published papers in the last 7 years. The integration of edge and cloud computing can leverage the best of both models, by defining a more flexible data gathering and management infrastructure based on applications' workloads, infrastructure status and devices' resource availability. Market research shows the majority of businesses are already implementing an integrated edge–cloud strategy [1]. In latest years, more advanced paradigms like *Osmotic Computing* [2] can establish a *cloud-to-edge continuum*, [3] where application microservice containers can be orchestrated and migrate dynamically across the two tiers [4].

Machine Learning (ML) and Artificial Intelligence (AI) applications are among the main use cases for cloud-to-edge architectures [5]. In early approaches, the more computationally demanding task of model training occurred in a centralized way in the cloud, while inference could be performed at the edge. More recently, model training and inference can be executed at both the cloud and edge layers – possibly

☆ This work has been supported by projects TEBAKA (TErritorial BAsic Knowledge Acquisition), funded by the Italian Ministry of University and Research, and SCIAME (Smart City Integrated Air Mobility Evolution), funded by the Italian Ministry of Enterprises and Made in Italy.

* Corresponding author.
E-mail addresses: davide.loconte@poliba.it (D. Loconte), saverio.ieva@poliba.it (S. Ieva), agnese.pinto@poliba.it (A. Pinto), loseto@lum.it (G. Loseto), floriano.scioscia@poliba.it (F. Scioscia), michele.ruta@poliba.it (M. Ruta).
URLs: https://sisinflab.poliba.it/loconte (D. Loconte), https://sisinflab.poliba.it/ieva (S. Ieva), https://sisinflab.poliba.it/pinto (A. Pinto), https://www.lum.it/docenti/giuseppe-loseto/ (G. Loseto), https://sisinflab.poliba.it/scioscia (F. Scioscia), https://sisinflab.poliba.it/ruta (M. Ruta).

https://doi.org/10.1016/j.future.2024.02.024
Received 10 September 2023; Received in revised form 29 January 2024; Accepted 23 February 2024
Available online 28 February 2024

with different trade-offs between latency, efficiency and accuracy – as data can be partitioned and aggregated at various levels and edge nodes are increasingly equipped with low-power specialized coprocessors for machine learning [4,6]. This kind of processing infrastructures enables *Federated Learning* (FL) [7] in the cloud-to-edge continuum, where multiple edge nodes collaborate to solve machine learning problems on different data subsets, under the coordination of a central cloud-based aggregator.

As the next step of the evolution of cloud-based and cloud-to-edge applications, in the *serverless computing* paradigm [8] application microservices encapsulate and expose stateless *functions* (*a.k.a. lambdas*, with a term borrowed from functional programming languages). Functions can be invoked without explicitly dealing with server or container provisioning, as it is managed by the infrastructure automatically. Serverless computing requires decomposing applications into a set of atomic stateless functions, but it pays off with increased availability, scalability, fault tolerance and cost-effectiveness, as it enables a more elastic adaptation to workload shifts while reducing risks of resource over- or under-provisioning.

The ever-increasing computational resources of IoT devices, supporting the availability of high-level programming language runtimes, provides an opportunity to expand the cloud-to-edge continuum toward the periphery of the network, right where new data are generated. By distributing computation even more pervasively, further latency and bandwidth savings can be achieved and overall scalability and resilience can be improved. This model fits AI/ML applications particularly well, since they are based on data stream aggregation from multiple sources, filtering, and analysis. If the same operations could be run at the IoT, edge and cloud layers as serverless functions, applications would be able to distribute workloads across all three layers, based on application- and context-specific availability, latency, data privacy and security criteria. While topics such as *TinyML* [9] and the *Semantic Web of Everything* [10] denote research results to bring advanced ML and AI technologies on board IoT and embedded devices, comprehensive frameworks for integrating them into a *cloud-to-thing continuum* [11] are still in their infancy.

Striving to fill that gap, this work proposes a novel general-purpose framework for expanding the cloud-to-edge continuum to IoT devices through serverless computing. The main contributions of the work can be summarized as follows:

- The infrastructure is transparent to the user and devices, that is, a node can collect and send data or queries to a local edge device or cloud endpoint and the response will be the same regardless of the device which processed the message. This seamless cloud-to-edge continuum is achieved by (i) ensuring that functions on both edge and cloud are equivalent in terms of inputs, outputs, topics, and messaging schema and (ii) enabling the full execution of the FL workflow across both layers. Moreover, the continuum expands to the IoT, as serverless functions for ML training and inference can be executed in the cloud, in edge nodes and in IoT field devices with minimal to no code difference.
- The framework is basically general-purpose, but a federated learning scenario has been adopted as a significant motivation, use case, and testbed for experiments. The federated learning architecture is also new, extending a semantic-enhanced IoT-oriented ML algorithm [12] by leveraging the amenability of its data summarization structure to aggregation operations before model generation. Although other ML and Deep Learning algorithms may exhibit higher accuracy than the adopted ML algorithm, it allows models to be generated out of larger and larger datasets from the IoT to edge to cloud nodes, by aggregating intermediate summaries. Therefore IoT nodes execute federated training tasks in addition to local inference (*a.k.a.* prediction) ones.
- The architectural design aims to be able to grant device and application security as well as data privacy, which is crucial in sensitive IoT contexts such as healthcare or vehicular networks. Security is enforced by the IoT and edge device management platform, by means of authentication mechanisms and encrypted communications. Privacy is supported by the devised federated learning architecture and protocol, which allow training ML models locally in IoT field devices or edge nodes, without sending them across the Internet to the cloud infrastructure.
- A platform prototype has been implemented using Commercial Of-The-Shelf (COTS) tools. The reference cloud provider is *Amazon Web Services* (AWS) since it provides all the technological building blocks required to implement components of the logical architecture, with *AWS Greengrass* (https://aws.amazon.com/it/greengrass/) that allows deploying serverless functions to the edge. Edge and IoT nodes have been implemented with Raspberry Pi single-board computers and STM-32 microcontrollers, respectively. An analysis has been conducted about portability to other providers and technologies.

In order to validate and clarify the proposed approach, a case study is conducted on the well-known *MotionSense* large-scale dataset [13] for activity recognition from data gathered in wearable devices. Illustrative examples clarify the framework and experimental analysis validates the core contributions of the research.

The proposed federated learning approach spanning cloud, edge and IoT layers has a wide range of possible applications. In telemedicine and active ageing scenarios, wearable devices can process data from individual subjects locally, sharing only anonymized aggregates with the edge and then the cloud; significant training can be performed in place with minimal latency, while waiting for larger and more accurate models from upper layers. In smart manufacturing scenarios, the proposed framework can manage Industrial IoT deployments for product quality control and equipment predictive maintenance across different production lines, shopfloors and plants. In the *Internet of Drones* [14], individual Unmanned Aerial Vehicles (UAVs) can perform AI/ML data processing on board, and share only significant high-level model information with peers and ground stations rather than raw data streams, thus improving latency as well as bandwidth and energy consumption in scenarios including search and rescue, [15] environmental and urban monitoring, precision farming and logistics.

The remainder of the paper is as follows. Section 2 provides background information on the adopted ML algorithm and relevant related work. Section 3 describes the framework, outlining the logical architecture, model training and inference tasks in federated learning. Portability considerations for integrating further cloud technology providers and ML tools are in Section 4. The prototypical implemented testbed is described in Section 5, including an illustrative example of federated learning for activity recognition through wearable devices. Early experiments are reported in Section 6, before conclusion.

2. Background

This section briefly recalls the reference ML algorithm which is exploited in a novel way to enable federated learning in the cloud-to-thing continuum. Relevant related work is also discussed.

2.1. Matchmaking features for (federated) mAchine learning data analysis

This work leverages *Mafalda* (*MAtchmaking Features for mAchine Learning Data Analysis*) [12], an IoT-oriented semantic-enhanced ML algorithm. Mafalda supports the typical ML pipeline for classification problems: data collection and preparation, feature extraction and selection, model training and refinement with hyperparameter optimization, model evaluation and deployment to generate predictions. Nevertheless, the semantic-enhanced approach changes the way each step is

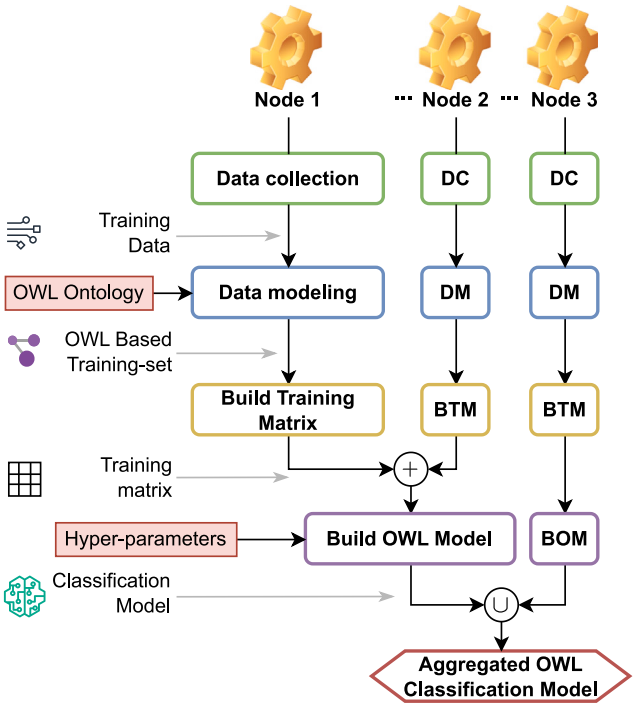


Fig. 1. Mafalda workflow for federated learning.

performed w.r.t. purely stochastic methods. The proposed workflow of Mafalda for FL is depicted in Fig. 1 and described in detail in what follows.

Data collection (DC) and **Data modeling (DM).** After training set gathering and cleaning, feature selection is supported by an ontology which models the domain conceptualization along properly defined patterns, in order to allow semantic-based data annotation and interpretation. The ontology is expressed in a restriction of the *Web Ontology Language* (OWL) version 2 [16], corresponding to the *Attributive Language with unqualified Number restrictions* ($\mathcal{ALN}$) Description Logic. Specifically, for each feature f_q the ontology must include a corresponding class A_q and a partonomy of it, *i.e.*, a set of subclasses $A_{q,1}, A_{q,2}, \ldots, A_{q,p_q}$ with $\forall j = 1, \ldots, p_q : A_{q,j} \sqsubseteq A_i$, representing all meaningful configurations or value ranges the feature can take; p_q is different for each q as partonomies for different features are not constrained to have the same number of classes. As a minimal example, consider the ontology in Fig. 2, which models room temperature and humidity features.

Build Training Matrix (BTM). The subsequent training step builds an intermediate aggregate data structure named *Training Matrix* (TM), which is then used to generate the *model* as a set semantic annotations, one denoting each possible output class in the training set. Training is executed on a set S of n training samples, each with (at most) m features, and w distinct output classes. Processing the ith sample, its qth feature value is mapped to a *concept component* $C_{i,q}$ constructed over A_{q,j_q} in the reference ontology, which is the subclass in the partonomy for A_q containing the particular feature value. For the sake of conciseness, the definition of concept components and details for constructing them in [12] are not recalled here. Overall, the ith sample $\forall\, i = 1, \ldots, n$ is composed of: (a) up to m *concept components* $C_{i,1}, \ldots, C_{i,m}$ annotating its features; (b) an observed output O_i labeled with a class name in the ontology. Samples are processed sequentially in order to build the TM, which is a $(w+1)\times(k+1)$ matrix. All the different output classes are in the first column while the k distinct concept components occurring in the training set are in the first row. Each element of the TM represents the number of occurrences of the column header concept component in the samples having the row header output. Basically, the construction algorithm (detailed in [12]) takes the ith training sample and first checks its output class O_i: if no previous sample has been associated to that class, it appends a row to the TM setting its values to zeros. Analogously, for each concept component $C_{i,j}$, if no previous sample includes it, then the algorithm appends a column to the TM and sets its values to zeros. Finally, the algorithm increases by 1 the value of the cell corresponding to O_i and $C_{i,j}$. Continuing the above example, suppose two output classes exist, named *Spring* and *Winter*, and the training dataset contains the four samples shown in Table 1. Then the TM is computed as reported in Table 2.

Table 1
Example training set.

Temperature	Humidity	Output
17	40	Spring
21	60	Spring
3	20	Winter
8	50	Winter

Table 2
Example training matrix.

HighHum.	LowHum.	HighTemp.	LowTemp.	O
1	1	2	0	Spring
1	1	0	2	Winter

Build OWL Model (BOM). Once the TM is built, the model can be generated to be used for prediction tasks by means of semantic matchmaking, as explained in [12]. Basically the model consists in an $\mathcal{ALN}$ expression E_i for each output class O_i, given from the logical conjunction of concept components $C_{i,j}$ appearing in the TM row of O_i. In order to improve model accuracy, Mafalda defines a set of dynamic thresholds over each row and each column of the TM in order to exclude from the expression the concept components which occur too infrequently [12]. These thresholds are the subject of hyperparameter optimization for model refinement, in which typical techniques can be applied [17]. Anyway, the final model is just a set of high-level formal OWL 2 expressions, summarizing even large datasets in a compact and meaningful way. In the running example, supposing for the sake of simplicity a fixed threshold $\theta = 0.7$ is adopted for all rows and columns of the TM, the *HighHumidity* and *LowHumidity* concept components are discarded for both output classes, as they have a frequency of $\frac{1}{1+1} = 0.5 < \theta$. Therefore the final trained model consists in the following pair of OWL class expressions:

- $Spring \equiv HighTemperature$
- $Winter \equiv LowTemperature$

Aggregation for federated learning. As recalled above, the TM is an intermediate structure which summarizes – and anonymizes – input data exploiting a reference domain ontology. This work leverages one of its fundamental properties: if a training set is partitioned in two or more subsets and the corresponding TMs are generated, they can be aggregated simply by summing the values in cells corresponding to the same concept component and the same output class, *e.g.*, in our running example, the cells for *Spring* row and *LowHumidity* column can be summed across multiple TMs. The result will be identical to the generation of a single TM from the whole training set. Based on this property, the algorithm supports federated learning by aggregating TMs computed locally by different nodes and summing them, without exchanging training data, as pictured in Fig. 1. Additionally, the fact that the trained model consists in a set of OWL 2 individuals for the various output classes allows a second (optional) level of aggregation in federated learning, basically given by the conjunction of sets of individuals generated by independent nodes but referring to the same domain ontology. The latter method, also shown in Fig. 1, grants even

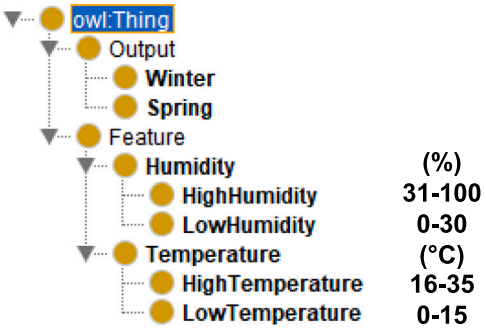


Fig. 2. Example ontology.

smaller data exchanges, by establishing a unified knowledge base from the collective knowledge of the various nodes.

Overall, the adopted Mafalda algorithm enables:

- *incremental learning*, in which training samples can be processed in batches: for each batch the same TM is updated, and at the end of each batch a new model is generated from the TM;
- *federated learning*, where individual nodes process different yet homogeneous datasets (*i.e.*, referring to the same ontology) and construct independent TMs. Each TM is then used in two ways: (i) define a model for carrying out predictions locally; (ii) send the TM to an upper-level aggregator, which will combine multiple TMs into a single one and generate a (presumably more accurate) model. This approach can be applied recursively with two or more levels of aggregation, and both the TM and the model generated at an upper level can be fed back to lower-level nodes, in a continuous loop aimed to improve performance as well as follow possible drifts and long-term dynamics of the monitored phenomenons.

While the work in [12] hinted at the possibility of federated learning for Mafalda but did not formalize it, this work exploits the original algorithm by constructing on top of it a serverless federated learning framework, which expands the cloud-to-edge continuum to include IoT field devices as computational nodes capable of running both training and inference tasks.

2.2. Related work

This subsection analyzes relevant works concerning serverless computing in the cloud-to-edge and cloud-to-thing continuum, as well as IoT-oriented federated machine learning.

2.2.1. Serverless computing in the cloud-to-edge continuum

Cloud–edge [18] computing has emerged as an innovative paradigm aiming to address the diverse and evolving demands of modern applications and services. The *cloud-to-edge continuum* [19] framework extends from localized edge devices to centralized cloud infrastructures, enabling seamless data and task flow across the cloud–edge interface. For instance, [20] describes a unified resource orchestration strategy for effectively managing cloud–edge resources, treating them as a single abstracted entity for executing distributed services. At the network edge, nodes process data in real-time to minimize latency and network bandwidth usage. Conversely, centralized cloud infrastructures offer extensive computational power and storage for demanding tasks. Bridging these extremes into a continuum enhances performance, by tailoring architectural designs to the specific requirements of different applications, distributing data and processing tasks across the network.

Recent research has focused on the integration of edge and cloud technologies in several areas like cyber–physical systems [4,21], healthcare [22] and intelligent transportation systems [23]. Nevertheless, cloud–edge computing architectures pose several challenges: (i) reducing *latency* is a primary goal, but achieving ultra-low latency between edge and cloud in real-world scenarios can be challenging due to varying network conditions and processing demands; (ii) as the number of edge devices and applications increases, managing *scalability* becomes a significant challenge, as edge infrastructures must be able to handle the dynamic growth in data processing requirements while maintaining performance and reliability; (iii) edge devices generate large volumes of information requiring an efficient *data management*, *i.e.*, deciding what data to process locally, what to transmit to the cloud, and how to store, retrieve, synchronize and orchestrate data effectively.

Combining compute continuum architectures with serverless frameworks represents one of the more promising research areas. Compared with traditional cloud computing approaches, *serverless computing* [8] aims to create dynamic environments where both infrastructure and platforms in which the services are running are hidden from customers. In this way, users of cloud services can invoke the desired functionality of their application only paying for the resources they actually use. The invocation of the functions is delegated to one of the available computation nodes (*e.g.*, cloud containers, decentralized edge environments or specific IoT devices) and the obtained results are sent back to the user. The fusion of serverless architecture with the cloud-to-edge continuum holds a great potential in the field of IoT-based federated learning scenarios [24]. By deploying serverless functions strategically across the continuum, edge devices can perform initial model training, leveraging their proximity to data sources. On the contrary, resource-intensive tasks like model aggregation and global updates can be executed in the cloud. This approach optimizes the federated learning workflow, while reducing the burden on individual edge devices and ensuring low-latency data processing thanks to the virtually unlimited computational resources available in the cloud.

2.2.2. IoT-oriented federated machine learning frameworks

New challenging application scenarios based on the cloud-to-edge continuum are emerging in the field of distributed intelligence [25]. The rapid dissemination [26] of IoT infrastructures and the requirements for low-latency, fault-tolerant and secure processing has motivated the development of edge analytics services.

A basic architecture for the orchestration of containerized microservices and cloud–edge intelligence was proposed in [4]. It is based on Osmotic Computing principles [2] allowing data mining with predictive Machine Learning models trained and executed on edge and on cloud, exploiting computational resources opportunistically to reach the best prediction accuracy. However, the proposed work did not fully utilize IoT devices for either training or inference. Additionally, the architecture did not employ federated learning approaches, as the data needed to be sent to centralized locations for processing.

In [27], a novel cloud–edge AI framework and architecture was presented to enhance ML efficiency, focusing on reducing transmission latency and bandwidth consumption. By means of container orchestration, the framework efficiently manages task allocation and data processing. It employs a specialized BranchyNet Deep Neural Network (DNN) model with early-exit branches for rapid inference on edge devices, optimizing response times and system load. Nevertheless, IoT devices within this framework are used only as data stream sources, being able to run neither training nor inference tasks.

FLoX [28] is a federated learning framework aiming to train and deploy neural network models over heterogeneous and distributed computing resources. It is built on the *funcX* [29] federated serverless computing platform in order to decouple FL model training/inference from infrastructure management. In this way, users can easily deploy models on different network devices. The approach is rather similar to this work, but it does not include training or inference on IoT devices.

The work in [30] also exploits funcX for defining *Rural AI*, a prototype system based on a federated Function-as-a-Service (FaaS) architecture aiming to demonstrate the capabilities serverless computing

Table 3
Related works comparison.

Reference	COTS tools	FaaS	FL	Cloud AI	Edge AI	IoT AI
[4]	✓	✗	✗	✓	✓	✗
[27]	✓	✗	✓	✓	✓	✗
[28]	✓	✓	✓	✓	✓	✗
[31]	✓	✓	✓	✓	✓	✗
[32]	✗	✓	✓	✓	✓	✗
This work	✓	✓	✓	✓	✓	✓

offers to traditional FL in rural precision agriculture scenarios, often characterized by limited and unreliable networking infrastructures.

Similar approaches are also in *FedLess* [31], a serverless framework for training and using DNN models on heterogeneous FaaS platforms, and in the Serverless Hierarchical Federated Learning (SHFL) framework [32], which adopts a two-layer FL architecture where nodes are grouped into clusters under cluster heads exchanging the local model parameters among a neighborhood of workers.

Even though the aforementioned frameworks provide several benefits, they all exhibit some limitations w.r.t. the proposed approach, as summarized in Table 3. The framework described in this paper is the only one which simultaneously grants: (i) serverless Function-as-a-Service architecture; (ii) support for federated learning; (iii) AI training and inference in cloud, edge and IoT nodes; (iv) implementation facilitated via COTS software tools.

3. Proposed framework

The proposed approach relies on serverless computing to define a cloud-to-thing framework for data collection, ML model training, and inference. The proposal aims to achieve three key properties:

1. **Federated learning flexibility:** the federated learning workflows extend to IoT devices as nodes for ML training and inference, harnessing the Mafalda algorithm recalled in Section 2.1.
2. **Unified execution:** functions can seamlessly run in the cloud, in on-premise edge devices and in IoT devices with minimal to no difference in code.
3. **User and device transparency:** the infrastructure operates transparently for users and devices. This means that nodes have the capability to collect and dispatch data to either local edge devices or the cloud endpoint, with the response remaining consistent, regardless of the processing device.

Further details on the proposed framework are discussed in the following sections. While description details refer to federated learning scenarios for the purpose of clarity and accuracy, the core architecture presented in Section 3.1 is basically general-purpose, as it can support distributed serverless functions for any type of application involving IoT, edge and cloud layers.

3.1. Architecture

Fig. 3 depicts a high-level conceptual representation of the architecture, which encompasses the cloud infrastructure, edge devices and IoT field nodes. Specifically, it includes the following components:

- **Cloud Provider**: has the responsibility of efficient provisioning and elastic scaling of the underlying cloud infrastructure.
- **Edge Node**: scaled-down counterpart to traditional cloud data center, delivering computation, communication, and storage capabilities.
- **Field Device**: IoT node characterized by a lightweight Real-Time Operating System (RTOS) as well as limited computational and energy resources.

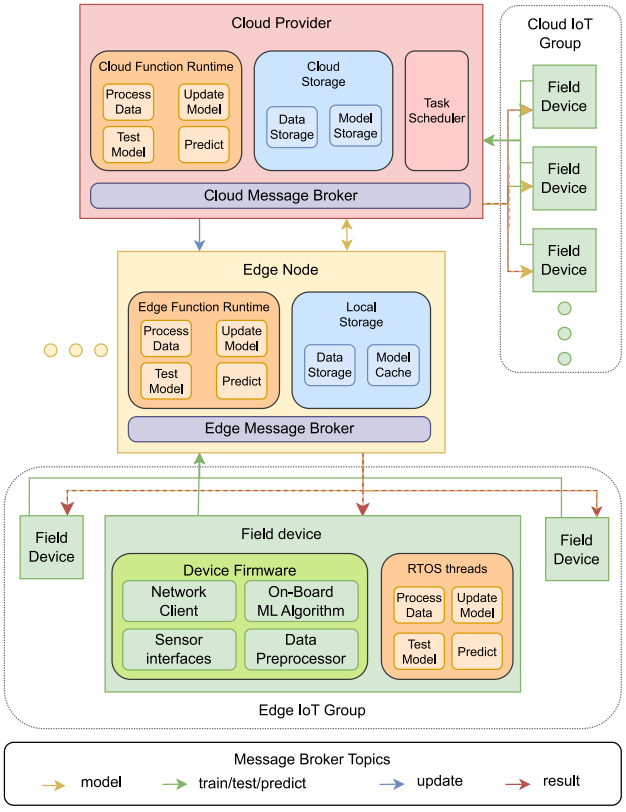


Fig. 3. Reference framework architecture.

- **Cloud/Edge IoT Group**: defines a set of field devices, connected either to the global cloud provider or to a local edge Node. Mutual authentication and authorization mechanisms are enforced, using certificates to manage roles and permissions for publishing and reading messages on available topics.
- **Function Runtime**: executes the stateless tasks in the Cloud Provider or in the Edge Node for the federated learning workflow in an event-driven and distributed manner. In Field Devices the same functions map to RTOS threads directly, for more efficient resource usage.
- **Storage**: stores the collected data and the machine learning models used for federated learning tasks. In federated learning scenarios, data is often partitioned into smaller, manageable batches to optimize communication efficiency. In particular, **Cloud Storage** holds the latest and most complete version of the federated learning model in persistent memory, whereas **Local Storage** acts as a cache: if an Edge Node must carry out a prediction task but lacks a local model, it retrieves the one from Cloud Storage.
- **Message Broker**: orchestrates event-driven message transmission and reception over the *MQTT* (Message Queuing Telemetry Transport) standard protocol (https://mqtt.org/), adopting the publish/subscribe paradigm. Nodes publish packets to specific topics, and the message broker routes received messages to subscribed services.
- **Task Scheduler**: it can invoke serverless functions in accordance with user-configured policies or timers. The main role in the proposed architecture is to invoke the *Update Model* function periodically, in order to decouple updates to the aggregated data for federated learning (in Mafalda's case, Training Matrices) from the task of training a new version of the ML model. Other triggers can be configured to activate the function, such as events related to the connection/disconnection of nodes in the architecture, requests from Field Devices or Edge Nodes, or when a certain amount of data is uploaded to the Data Store.

A noteworthy feature is the capability to perform machine learning tasks not only on edge nodes but also on IoT devices, without depending on the cloud infrastructure. This flexibility ensures that outcomes remain consistent, with variations primarily in response times. The following functions are executed in the Function Runtime:

- **Process Data**: the primary task is to read the data published by Field Devices. This function plays a pivotal role in the framework's data processing pipeline, ensuring that incoming data is efficiently archived in the Data Storage component for subsequent model updates and inference. The function is invoked for each incoming message from the `train` MQTT topic;
- **Update Model**: enhances predictive capabilities over time through incremental learning from newly available data batches. It retrieves locally stored data batches from the Data Storage, and performs an incremental model update, as described in Section 2.1. By adopting a *mini-batches* approach for incremental updates, the model can be trained in short, lightweight bursts of computation, complying with the execution time and memory constraints of serverless functions. The updated model is saved into the Model Storage. This function is started periodically by the Task Scheduler service.
- **Test Model**: the function responds to MQTT `test` messages containing labeled samples for prediction. It utilizes this data to calculate a confusion matrix and evaluation metrics, including precision, recall, F1-score, and overall accuracy for the most up-to-date model available on the node.
- **Predict**: responds to MQTT `predict` messages, which contain a series of unclassified samples, by delivering the classification results based on the most up-to-date model available on the node.

In order to validate the proposed serverless architecture, an off-the-shelf infrastructure based on AWS technologies has been employed for implementing all the components in a complete prototype, as shown in Fig. 4:

- **IoT Core**: as the central component of the architecture, the *AWS IoT Core* (https://aws.amazon.com/it/iot-core/) service is responsible for managing MQTT communications, device provisioning, and authentication within the network. It can interface with and manage both *on-premise* IoT device groups and edge nodes, which in turn can manage other IoT device groups.
- **Model Storage**: the *Amazon S3* (https://aws.amazon.com/s3/) object storage service is used to store and manage ML models. When an updated model is loaded, it replaces the previous version.
- **Data Storage**: the *DynamoDB* (https://aws.amazon.com/dynamodb/) scalable NoSQL key–value database handles data storage in the form of batches. A configurable batch ageing policy is established to limit the amount of relevant data stored for model training and update.
- **Lambda Functions**: a set of *AWS Lambda* (https://aws.amazon.com/lambda/) functions has been defined to map the aforementioned four tasks: process data, update model, test model, and predict. These lambdas are invoked and executed in the same way both in the cloud and on edge nodes. Moreover, the same source code implementing the Mafalda algorithm is exploited to run functions locally in Field Devices on data collected from available sensors. Further implementation details are provided in Section 5.1.
- **Scheduler**: the *AWS EventBridge* (https://aws.amazon.com/eventbridge/) service enables real-time data change notifications from AWS services, personal applications, and Software as a Service (SaaS) applications without coding. In the architecture, AWS EventBridge serves as the Task Scheduler component, orchestrating and triggering lambda functions automatically, based on a pre-configured timer.

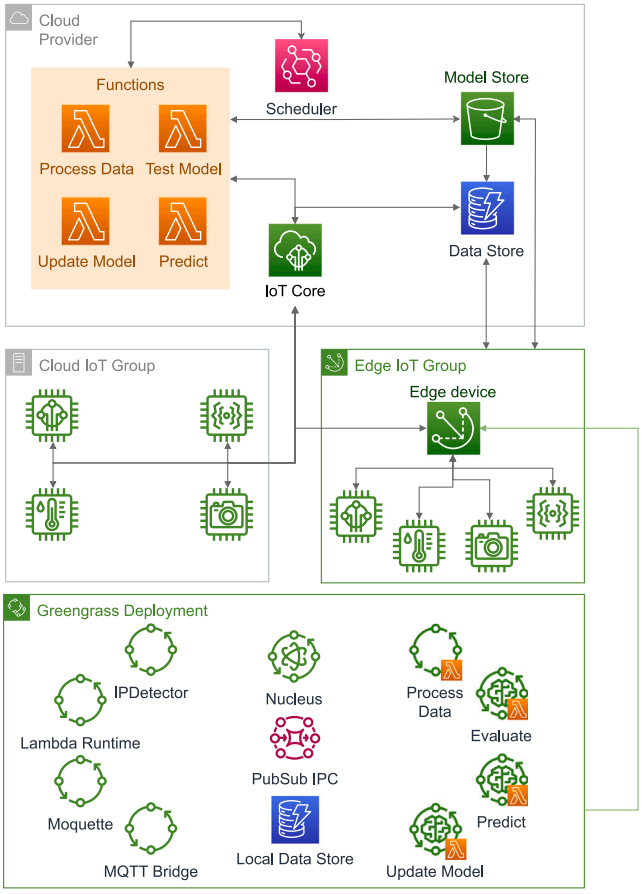


Fig. 4. Proposed AWS-based prototype architecture.

- **Edge device**: a sufficiently capable edge device, such as a single-board computer or a PC, acts as *AWS IoT Greengrass* (https://aws.amazon.com/greengrass/) *CoreDevice* to run Greengrass services and manage its IoT device group.
- **Greengrass Deployment**: A group of components executed on AWS CoreDevices, including:
  - **Nucleus**: manages the device's lifecycle and control communications with the cloud.
  - **PubSub IPC**: enables event-driven distributed Inter-Process Communication (IPC) among the internal components within the Greengrass node.
  - **Moquette**: a local MQTT broker allocated for the subnet.
  - **MQTT Bridge**: serves as an intermediary for MQTT messages among Moquette, PubSub IPC and AWS IoT Core, enhancing communication within the Greengrass ecosystem and connecting it to the broader AWS infrastructure.
  - **Process Data**, **Update Model**, **Test** and **Predict**: the corresponding Lambdas imported into the device that react to messages received on the local broker.
  - **Local Data Store**: managed through a local instance of DynamoDB, serves the dual purpose of storing the data batches received from the `train` topic and the updated models received from the cloud.
  - **IPDetector**: A component that manages the Cloud Discovery procedure.

Details on how these components interact in the training and testing/prediction phases of serverless federated learning are explained in the following two sections.

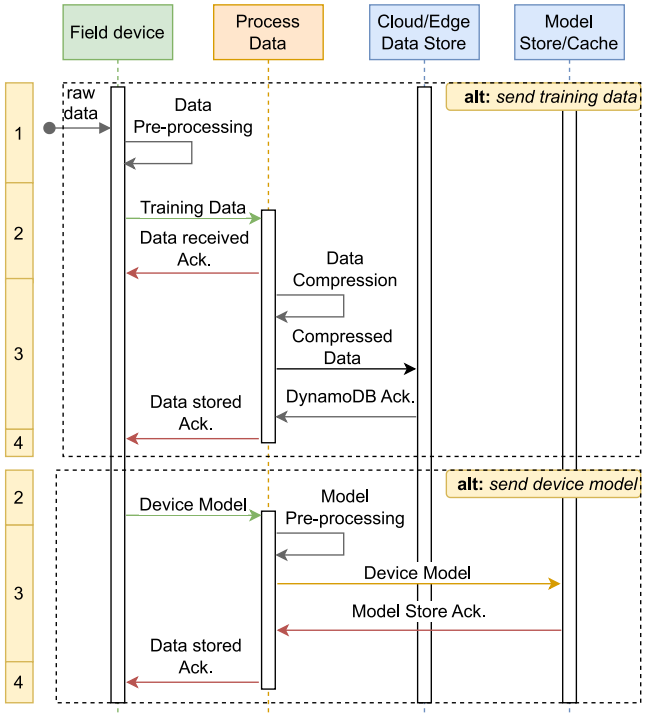


Fig. 5. Model training sequence diagram.

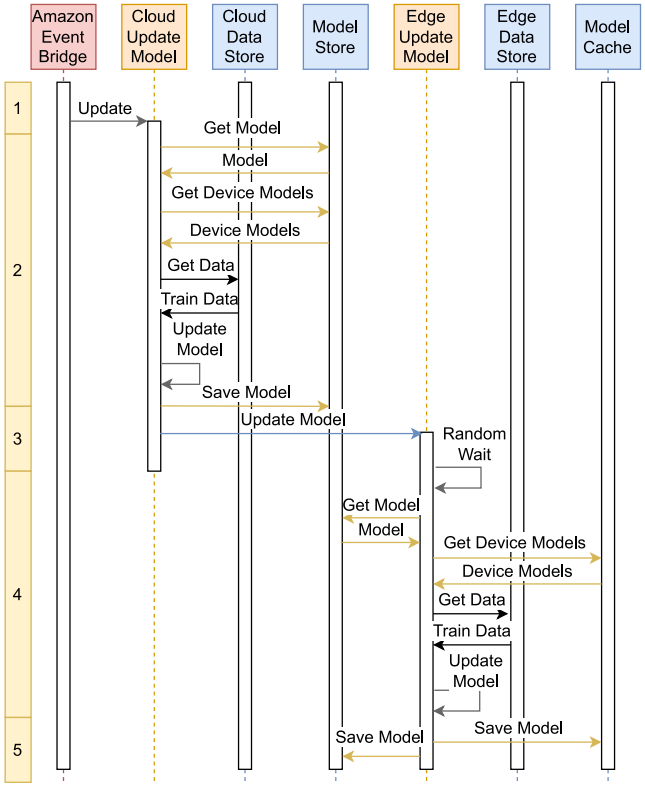


Fig. 6. Model update sequence diagram.

3.2. Training

This section describes the sequence of operations outlined by the training task, managed through the *Process Data* lambda function within the framework described in Section 3.1. Fig. 5 sketches the sequence diagram of operations and interactions among cloud/edge/field components: numbered steps are outlined in what follows.

1. The Field Device (FD) locally performs preprocessing on sensor data within a temporal window. During this phase, the intermediate aggregated data structure is generated, *i.e.*, the Training Matrix (TM) in case of Mafalda, or a local classification model is trained as described in Section 2.1.
2. The FD publishes a message serialized in binary format on the `train` MQTT topic. Depending on application-specific concerns about privacy and bandwidth availability, the federated learning framwework is configured so that the message contains either the training data batches, the Training Matrix or the local classification model, as shown by the two alternative sequences in Fig. 5.
3. The message triggers the execution of the *Process Data* function on the Edge Node or Cloud Provider managing the IoT group of the FD, which compresses and stores the received data on the relevant data/model store. The architecture does not constrain the way FDs are associated: for configuration simplicity, in the current AWS-based prototype the association is static either to an Edge Node in the local network or to the cloud, but proper dynamic criteria can be considered for future revisions.
4. Upon completion, the *Process Data* function publishes an acknowledgment message, notifying the FD of the task conclusion.

Independently from the data upload phase, there exists a model update phase, which allows updating the ML model to achieve a more accurate version based on the whole dataset available in the Data Store. Fig. 6 illustrates the sequence diagram of the interactions among the components.

1. The invocation of the *Update Model* serverless function is managed by the *AWS EventBridge* (https://aws.amazon.com/eventbridge/) component, a serverless event router which periodically schedules an event to trigger the function in the cloud.
2. The function retrieves the most recent TMs and models from the Model Store (managed through an S3 instance), along with the available data from the Data Store (managed through DynamoDB). TMs sent by IoT or edge devices are also retrieved from the Model Store at that time, in order to perform TM aggregation. The aggregated TM is used to generate the overall updated model, which is aggregated with Device Models and sent back to the Model Store. This aggregated model will be used for future prediction and training tasks.
3. Upon conclusion, cloud *Update Model* function sends a message on the `update` topic.
4. On the Edge Node, the same *Update Model* function is invoked upon the arrival of a message on the `update` topic, to update the reference model with the data available on Edge Data and Model Store. Initiating simultaneous updates from multiple devices can lead to write conflicts to the Model Store. To reduce the likelihood of conflicts, the function waits for a random interval before execution. Additionally, an Optimistic Concurrency Control strategy is adopted: the model is overwritten only if the object's version number has not changed from the initial read to the moment of writing the updated model. In case of a conflict, the training must be repeated on the new version.
5. The updated model is stored in the Local Model Storage of the Edge Node, which, as previously described in Section 3.1, plays the role of caching objects locally to reduce frequent access to the Cloud storage. Finally, the function sends the updated model to the cloud-based Model Store.

Serverless runtimes are typically configured to execute functions for short periods of time. In AWS Lambda execution time is capped

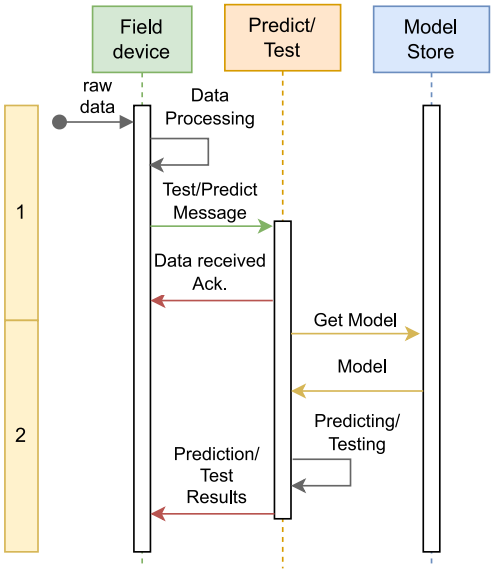


Fig. 7. Predict and Test functions on cloud.

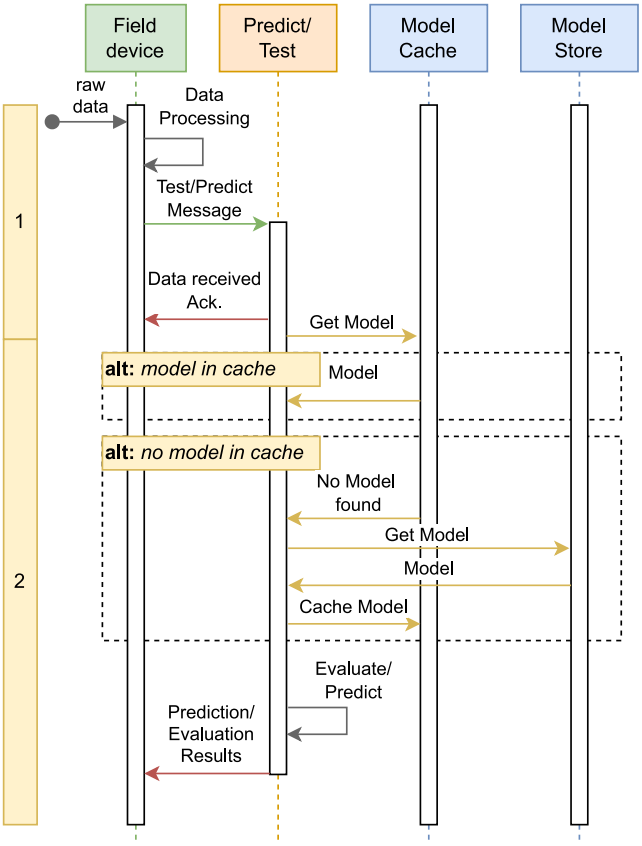


Fig. 8. Predict and Test functions on edge.

at 15 min. To address this limitation, it is possible to allocate larger amounts of resources to more intensive tasks such as the *Update Model* function.

3.3. Testing and prediction

This section provides a detailed overview of the workflow for carrying out ML inferences on distributed models, such as performing classification tasks to predict events based on sensor data connected to FDs. The sequence diagram in Fig. 7 shows the workflow in the case of cloud execution.

1. The Field Device, following data processing like in Section 3.2, publishes a message on the `predict` (respectively, `test`) topic.
2. The corresponding function is initiated in the cloud. The *Predict* (resp. *Test Model*) function retrieves the latest trained model instance from the Model Store, carries out classification (resp. evaluation), and dispatches the outcome as a message to the `result` topic.

When performing prediction or testing on the edge side, the sequence of operations becomes more complex, as illustrated in the sequence diagram in Fig. 8. Specifically, upon receiving a message on the `predict` (resp. `test`) topic, the Edge Node function initiates its workflow by attempting to retrieve the training model from the Local Data Storage, which operates as a cache, as elaborated in Section 3.1. Two alternative execution flows are possible:

- if the model is available locally, the function directly loads it to carry out the prediction (resp. test) task, mirroring the previous scenario;
- if the model is not in the cache, the function must then retrieve it from the cloud-based Model Store. This action incurs latency and data network traffic penalties. Once obtained, the model is cached to streamline future runs, before executing the requested prediction (resp. test) and returning the results.

4. Portability

AWS has been selected as the technology provider for the prototypical implementation of the proposed framework, due to its comprehensive feature set, which has allowed to map directly all components of the architecture to available building blocks at the IoT, edge and cloud layers. However, it is essential to note this is not a mandatory choice. The same framework is portable to other providers by substituting the adopted managed services with equivalent offerings. The considerations below summarize some relevant examples of available offerings, but many other players exist in the serverless computing and cloud-to-thing services markets and they are accelerating their pace of innovation, therefore new or improved off-the-shelf solutions can become available.

Microsoft Azure includes *Azure Functions* supporting multiple languages, *Azure Service Bus* for message management, *Blob Storage* and *Table Storage* for data storage, and *Azure IoT Edge* for deploying workloads on edge devices with enhanced device management capabilities. Unlike AWS Greengrass, Azure IoT Edge requires user's code to be run in containers: this can limit the choice and increase the cost of devices to be used as IoT Field Devices. Conversely, Azure IoT Edge automatically manages multiple-level hierarchies of devices, whereas AWS Greengrass requires the user to configure the device group hierarchy: this can be an important feature for large-scale federated learning applications. **IBM Cloud** services include *IBM Cloud Functions* based on *OpenWhisk*,[1] *IBM Event Streams* using *Kafka*,[2] *IBM Cloud Object Storage* and *Cloudant* –based on *CouchDB*[3]– for data storage, and *IBM Edge Application Manager* for managing edge device services, similar to Azure IoT Edge, with additional support for serverless computing through *Edge functions*. Edge computing solutions rely on the *IBM Cloud Satellite* service for managing hybrid cloud deployments, which is more powerful but more complex than AWS Greengrass or Azure IoT Edge.

[1] Apache OpenWhisk: https://openwhisk.apache.org/.
[2] Apache Kafka: https://kafka.apache.org/.
[3] Apache CouchDB: https://couchdb.apache.org/.

Other Platform-as-a-Service offerings, focused on the cloud-to-edge and the cloud-to-thing continuum, have been introduced recently. For instance, **Particle**[4] provides a cloud-based management platform for devices running their *Device OS*, which simplifies user applications integration into firmware and lifecycle management. Similarly, the **Arancino**[5] [33] platform comprises: (i) a family of open-hardware dual-board devices, which can perform as edge computing devices as well as IoT field devices for sensing and actuation; (ii) a cloud-based IoT management platform grounded on OpenStack. **EdgeImpulse**[6] supports cloud-to-edge ML training and can integrate *Arduino*[7] microcontrollers for inference tasks.

Ideally, the machine learning deployment framework should encompass several key characteristics: support for incremental learning, federated learning, and capabilities for both training and inference from cloud to IoT environments. Additionally, an important requirement is that the framework's installation size must be sufficiently compact to fit within the constraints of a serverless function runtime. Specifically, for AWS Lambda, this means the compressed package size must be kept under 50 MB.

A significant gap has been identified in this regard, as there is a lack of a comprehensive framework that encompasses all these essential features, which becomes evident when considering the capability for both training and inference on IoT environments. For the deployment on cloud-to-edge, some state-of-the-art libraries such as **scikit-learn**[8] can be easily integrated into an AWS Lambda function, however larger frameworks such as **TensorFlow**[9] and **PyTorch**[10] exceed size limits and are not as easily integrated. **TensorFlow Lite** is exploitable for inference and fine-tuning on both cloud and edge devices, but its models can be run on IoT field devices only for inference.

5. Case study: federated learning for activity recognition

In order to clarify the proposal and highlight its features, a prototypical testbed has been fully developed for a federated learning case study concerning the domain of activity recognition. *Activity recognition* holds significant relevance, since its applications span from health and fitness monitoring to smart homes and public safety, encompassing the ability to discern human activities like standing, walking, running, and more. This case study aims to illustrate how the proposed federated learning framework, with its decentralized model training approach, can be applied to an activity recognition dataset.

5.1. Prototype deployment

The prototype closely adheres to the description provided in Section 3.1. It includes: 1 AWS Cloud Provider node, 1 Edge Node, 3 IoT Field Devices attached to the Edge Device and 3 IoT Field Devices attached directly to the Cloud Provider node. The deployment strategy has followed a top-down approach, started by configuring the cloud infrastructure on AWS, then by provisioning the Edge Device, and finally by programming the Field Devices.

In the cloud infrastructure setup, the primary focus has been on developing the four AWS Lambda functions described in Section 3.1. Each lambda has been programmed and packaged in a standalone *.zip* archive, and subsequently uploaded to the cloud using the AWS Console. The Lambda functions responsible for data processing, model updating, and prediction are configured to be triggered by their corresponding MQTT messages, meanwhile the Update Model function is scheduled for periodic invocation through Amazon EventBridge.

Field and Edge devices have been organized into IoT Groups at this stage. Within these groups, the necessary permissions and policies have been outlined, authorizing the nodes to access relevant resources. This includes authorizing the connectivity to Cloud MQTT Broker and allowing Edge Nodes to authenticate edge devices within their respective group.

To guarantee seamless compatibility between edge and cloud Lambda runtimes, certain requirements must be carefully considered during the Lambda development process. Specifically, the following key considerations must be kept in mind:

- Lambda functions should be either uploaded as standalone *.zip* files, with a maximum file size of 50 MB, or developed directly using the inline code editor in the AWS Console. This is crucial because AWS Greengrass service does not support alternative Lambda formats, such as layered packages and container functions;
- the chosen runtime for the Lambda function must be compatible and available with both AWS Greengrass CoreDevice and the AWS Lambda execution environment;
- if the Lambda package includes native code – such as libraries that have bindings to native libraries – it is crucial to align the CPU architecture of the cloud Lambda runner with that of the edge device. As an alternative, separate Lambda functions should be deployed for each distinct CPU architecture to be supported.

To meet these requirements, Python (version 3.9) has been selected as the programming language to implement the aforementioned functions. This language ensures compatibility across both edge and cloud environments. Additionally, the *aarch64* ARM 64-bit architecture has been chosen for Cloud Provider node instances to match the CPU architecture of the Edge Node. The Mafalda tool, originally implemented in Java [12], has been re-implemented in C with a Python wrapper: its compactness (357 kB overall, libraries included) is a beneficial feature, as it helps minimize the package size, thereby reducing load times.

For the Edge Node, a Raspberry Pi 4 Model B[11] hosts the GreenGrass CoreDevice service. This service enables remote control and monitoring of the Edge Device from the AWS Console, as well as the deployment of custom components. In a nutshell, CoreDevice provisioning can be split in two key parts:

1. **Setup of the Operating System**: installation of *Raspbian OS Bullseye* (version 11) is required, along with OpenJDK version 11.0.20, to meet the prerequisites of the AWS Greengrass CoreDevice installer.
2. **AWS Console CoreDevice Setup**: configuration of the CoreDevice software is performed via the AWS Console, which allows to download the installer package. Executing this installer on the Raspberry Pi completes the setup process.

Upon edge device registration, the configuration described earlier in Section 3.1 can be imported. This is achieved in two steps:

1. import the four Lambda functions into AWS Greengrass as *Components*;
2. create a new *Greengrass Deployment* with all the parts listed in Section 3.1, and specifically in Fig. 4, including the newly created four Lambda Components;

Minimal component configuration is mandatory to complete the CoreDevice setup, specifically:

[4] Particle: https://www.particle.io/.
[5] Arancino: https://arancino.cc/.
[6] EdgeImpulse: https://edgeimpulse.com/.
[7] Arduino: https://www.arduino.cc/.
[8] Scikit-learn: https://scikit-learn.org.
[9] TensorFlow: https://www.tensorflow.org/.
[10] PyTorch: https://pytorch.org/.

[11] ARM® Cortex®-A72 Quad-Core CPU @ 1.5 GHz, 8 GB of RAM, and 32 GB of Secure Digital (SD) storage memory.

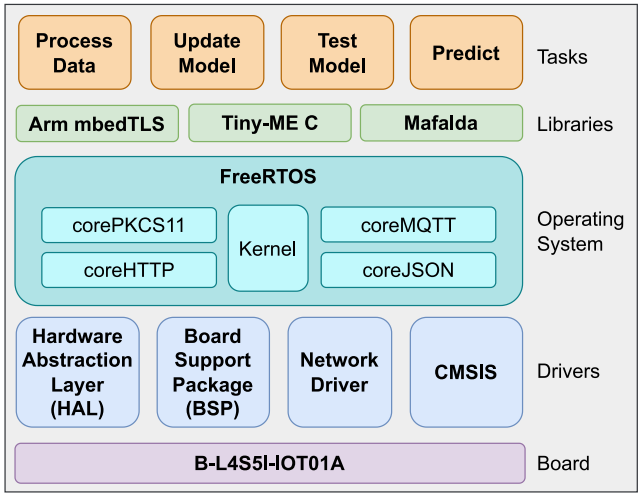


Fig. 9. IoT Field Device firmware.

- the Nucleus authentication component requires permissions to enable the IoT Group to both publish and subscribe to MQTT topics;
- MQTT Bridge needs to be configured to relay messages from client devices (*i.e.*, `train`, `test` and `predict` topics) to the PubSub broker and messages from Lambda functions (with `result` topic) to the local MQTT broker, allowing client devices to communicate with Greengrass component;
- setup local MQTT topic to trigger Lambdas on the edge device;
- specify the local MQTT broker endpoint to the *Greengrass Discovery API*.[12]

The last task has been needed to develop FDs firmware, whose architecture is shown in Fig. 9. It is designed following a layered software architecture. Starting from the bottom layer, the reference development board chosen for this experimentation has been the *STM32 Discovery kit IoT Node B-L4S5I-IOT01A*, having the following hardware configuration: STM32L4S5VIT6 Micro Controller Unit (MCU) with 120 MHz Arm Cortex-M4 core; 2 MB of flash memory; 640 kB of SRAM; wireless connectivity with Wi-Fi, NFC and Bluetooth Low Energy (BLE); a wide range of sensors, including a gyroscope, accelerometer, magnetometer, proximity, pressure, humidity and a microphone; embedded *ST-LINK* debugger and programmer.

Drivers are the lowest layer of software. They include:

- **Hardware abstraction Layer**, specific for the processor, which provides human-readable names and functions to access hardware components of the MCU;
- **Board Support Package**, specific for the board, which provides higher-level interfaces to access sensors and hardware features;
- **Network Driver** for the ISM43362-M3G-L44 Wi-Fi module of the board, providing a complete TCP/IP network stack (mandatory requirement to support AWS Greengrass);
- **Common Micro-controller Software Interface Standard** (CMSIS), which is a set of standard APIs and interfaces for ARM microprocessors to promote code reusability and interoperability across MCU vendors.

CMSIS is specifically required by **FreeRTOS**+ (https://www.freertos.org/FreeRTOS-Plus/) which is the main building block of the upper software layers. FreeRTOS is composed by the FreeRTOS kernel and some utility libraries. The FreeRTOS kernel provides concurrency as well as primitives and data structures for synchronization (Mutexes, Semaphores, Timers, Queues...). FreeRTOS+ libraries include: (i) *corePKCS11*, implementing a subset of the PKCS11 API to access cryptographic objects; (ii) *coreMQTT*, providing a MQTT client; (iii) *coreHTTP* and *coreJSON* to serialize/deserialize HTTP and JSON messages, respectively. *ARM mbedTLS* is an additional library needed to support mutual authentication mechanisms and encrypted communications, which are mandatory when communicating with AWS IoT Core and GreenGrass CoreDevice. The IDE of choice has been STM32CubeIDE, provided by the board vendor, with the *X-Cube-AWS* expansion pack that provides ports of the aforementioned FD building blocks.

Manual porting has been required of the other components, namely:

- **Mafalda** [12] and the **Tiny-ME C** [10] reasoning engine libraries have been ported to STM32 in order to support on-device ML training and inference;
- **Process data**, **Update Model**, **Test Model** and **Predict** tasks, described in Section 3; unlike the edge and cloud functions, which are invoked via MQTT messages, here the tasks are triggered when new sensor data are acquired.

The firmware, developed exclusively in C, is uploaded on the device using the ST-LINK Programmer. To correctly provision the device, and specifically to be authenticated into AWS IoT Core, it is necessary to follow this procedure:

- provision the secure element and retrieve the client certificate;
- sign and upload the firmware to the device;
- create a new *thing* on AWS IoT Core with the certificate from the secure element;
- attach policies to the certificate allowing the relative device to connect and subscribe to MQTT Brokers;
- add the device to the IoT Group;
- add the node to the Discovery API list on Greengrass;

Upon completion of these steps, all devices in the final prototype can communicate through an encrypted and authenticated connection.

5.2. Reference dataset

To illustrate the usefulness of the proposed federated learning framework, a small case study has been developed leveraging the *MotionSense* [13] dataset, which is publicly accessible under the Open Data Commons Open Database License (ODbL) v1.0 on *Kaggle*.[13]

MotionSense comprises accelerometer and gyroscope sensor data collected from iPhone 6s devices through the *Core Motion API*.[14] Specifically, the dataset consists of the following sensor measurements:

- **Attitude**: device orientation in terms of roll, pitch, and yaw;
- **Rotation Rate**: angular velocity of the device;
- **Gravity**: acceleration vector relative to gravity, expressed in the device own reference frame;
- **User Acceleration**: acceleration imparted to the device by the user.

Each type of sensor measurement is captured independently for each axis, for a total of 12 features.

These data were gathered from 24 different participants, each instructed to perform one of six activities: going downstairs, going upstairs, walking, jogging, sitting and standing. Each individual performed 15 different trials, during which the data were collected at

[12] https://docs.aws.amazon.com/greengrass/v2/developerguide/greengrass-discover-api.html.

[13] https://www.kaggle.com/datasets/malekzadeh/motionsense-dataset.

[14] https://developer.apple.com/documentation/coremotion/cmdevicemotion.

50 Hz sampling rate. Additionally, the dataset includes a label indicating which of the six activities was performed by the subject during the data collection process. MotionSense has been chosen in this work because it represents a realistic use case of data collection from multiple field devices, as the iPhone 6s could be replaced by a smaller wearable device. It can be assumed that each device monitors the activities of a subject.

Data have been preprocessed by aggregating 50 individual samples, corresponding to one second of data, into a single composite sample, incorporating both the mean and the standard deviation for each sensor across all its axes, thus resulting in 24 distinct features. It is important to note that this data aggregation was performed prior to the transfer of data to the designated field devices, thereby simplifying subsequent experimental procedures. In practical applications, the field devices are expected to conduct such preprocessing tasks in real-time. Consequently, it is essential that the computational requirements are compatible with the capabilities of the reference MCU. A preliminary test conducted with an STM32 board has validated this point: the board has been preloaded with raw data and instructed to execute the aggregation procedure. The computation time has been found to be, on average, 0.2 ms for each set of 50 samples, equating to one second of data capture. Furthermore, the available onboard memory has been sufficiently large to buffer the generated batch of samples, in accordance with the experimental settings. These results support the capability of the STM32 board to effectively handle the specified preprocessing tasks.

Afterwards the preprocessed dataset has been divided by subjects. Specifically, data on subjects 1 to 6 are reserved to perform the initial Mafalda model selection. This model resulting from the initial data is referred as the **bootstrap model**. While not strictly mandatory in the context of this framework, hyperparameter optimization at this stage may enhance model accuracy down the line. Also, it is reasonable to assume that limited data are available to choose an initial model. In any case, subsequent updates to the model are possible as more data are collected.

The remaining data have been split among all Field Devices as follows:

- *c1*: subjects 7 to 9;
- *c2*: subjects 10 to 12;
- *c3*: subjects 13 to 15;
- *e1*: subjects 16 to 18;
- *e2*: subjects 19 to 20;
- *e3*: subjects 21 to 24.

where FDs *c1*, *c2* and *c3* are associated to a Cloud IoT Group and FDs *e1*, *e2* and *e3* to an Edge IoT Group.

5.3. Illustrative examples

The versatility of the proposed framework extends its applicability to a wide array of use-cases, including privacy-sensitive domains like industrial workplace safety. In such environments, the worker can be equipped with wearable IoT sensors that monitor movements and environmental conditions in real time. Each wearable could be configured to either locally update its machine learning model for immediate inference or to send the data to an on-site edge node, such as a Raspberry Pi. This edge node could perform more complex inferences and, if necessary, share only the aggregated and anonymized model updates with a centralized cloud service. In turn, the cloud could merge these updates with other models received by additional edge nodes from multiple sites and, if necessary, perform further data analytics.

One of the key features of this framework is its flexibility, since it offers training and inference across different layers of the network. This multi-layer approach enables the system to be fine-tuned according to varying requirements and limitations. For instance, if the network conditions are challenging or unreliable, the edge and IoT devices can still carry on with essential monitoring and prediction tasks. Moreover, this architecture supports scenarios where conventional cloud-based solutions might fall short, such as in compliance with privacy regulations that prohibit the fine-grained tracking of employees.

Another compelling application for this framework is in the area of elderly care, particularly for in-home or ambient-assisted living environments. In those settings, a network of IoT sensors could be strategically placed around the living space or even worn by the elderly individuals. The sensors could monitor a variety of metrics such as movement, heart rate, and even ambient conditions like room temperature or air quality. Like in industrial settings, these IoT devices could either update their machine learning models locally or transmit data to a nearby edge node for more complex analysis. An edge node could be a dedicated home server or a smart home hub capable of ML computations. The aggregated anonymized model data could then be sent to a centralized cloud service for larger-scale analytics, such as predictive health assessments or emergency event recognition.

The true advantage of this federated architecture becomes apparent when considering the delicate balance between the need for high-quality care and the privacy concerns often associated with monitoring vulnerable populations. By enabling machine learning to occur at the device or edge level, sensitive data can be processed locally, thereby reducing the amount of personal information that needs to be sent to the cloud, and for this reason, this approach aligns well with privacy regulations and ethical guidelines.

Extending considerations beyond activity recognition, the proposed federated cloud-to-thing approach offers advantages in scenarios characterized by challenging network conditions. For example, considering a remote farming setting, where network connectivity is inconsistent. IoT sensors can be deployed throughout the farmland to monitor soil moisture, temperature, and other vital parameters for crop health. Analogously in smart grid systems, where network connectivity can often be unreliable, especially in remote areas, IoT devices embedded in transformers or substations can locally process data for anomaly or fault detection. When network conditions allow, these devices can send essential data to a local edge node for further analysis. This setup ensures monitoring is performed continuously, even when connectivity to the cloud is unstable.

6. Experiments

Following the case study described in Section 5, an experimental campaign on real devices has been carried out to prove the feasibility of the proposed approach and to assess its performance.

6.1. Materials and methods

The experimental testbed has been set up as described in Section 5.1. A Raspberry Pi 4 Model B+ has acted as the edge CoreDevice, while IoT Field Devices have been implemented with three B-L4S5I-IOT01A boards, chosen due to their compatibility with the AWS stack. In particular, AWS documentation provides instructions to install their Greengrass software specifically for Raspberry Pi and B-L4S5I-IOT01A boards have certified compatibility with AWS, meaning that STM32 provides a set of libraries to facilitate the integration with AWS IoT Core. Using other boards or MCUs is still possible, but requires to port FreeRTOS libraries to the new device manually, or to find existing working ports. For instance, the Arduino and Arancino families of development boards have a compatible implementation of FreeRTOS.[15] Greengrass is generally simpler to install since it requires only a Linux operating system on the target device and a compatible Java Runtime Environment.

[15] https://www.arduino.cc/reference/en/libraries/freertos/.

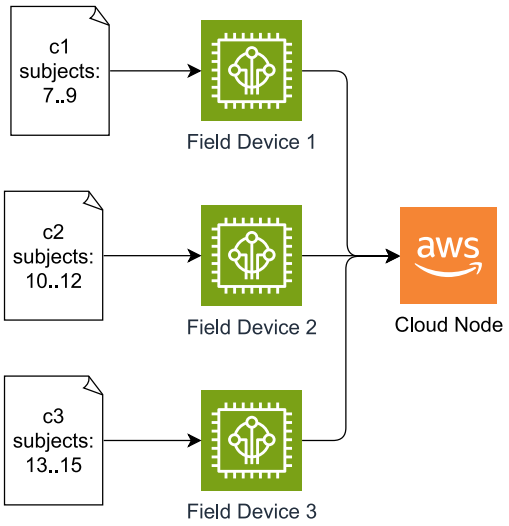


Fig. 10. Experimental setup — cloud only.

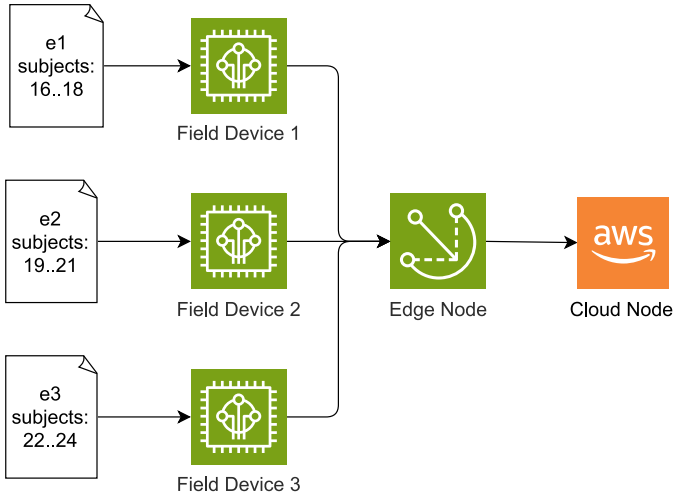


Fig. 11. Experimental setup — cloud and edge.

The first step involves data preparation, as described in Section 5.2. Briefly, data have been partitioned by subject and, for each partition, 20% of data have been held out to evaluate model accuracy. Initial data of subjects 1 to 6 have been used to train the Mafalda **bootstrap model** and select an appropriate threshold value, which has been found to be 0.11, maximizing the accuracy of the model w.r.t. the test samples of subjects 1 to 6. The bootstrap model has been therefore loaded on the Model Store.

In the first experiments, the Edge Node has been turned off and the three physically available STM32 boards have been connected directly to the cloud AWS IoT Core. Devices have been loaded with *c1*, *c2* and *c3* data subsets, respectively, and programmed to send data in batches of 256 samples each second. The training data have been sent on the `train` topic, while test data have been sent on the `test` topic; they all have been stored into the Cloud Data Store, as explained in Section 3.2 and Fig. 5. This setup is sketched in Fig. 10. Data has been also used to update and test the on-board ML model.

It is useful to note that Mafalda's accuracy does not really depend on batch size, as training samples are processed one by one to build the TM incrementally, as explained in Section 2.1. For other algorithms, however, it might be necessary to determine the optimal value while training the bootstrap model. In this phase, FDs have logged the time elapsed between the publish operation and the receipt of each of the two acknowledgements sent by the Cloud Lambda function, as shown in Fig. 5, in order to assess network latencies. Lambda functions add to acknowledgment messages of prediction and test the time elapsed during model inference, in order to profile the computational overhead due to Mafalda invocation.

Subsequently the configuration has been modified as shown in Fig. 11, by adding an Edge Node to the network. The same experiment has been repeated using *e1*, *e2* and *e3* data subsets, now measuring communication latencies between the Greengrass Core Device and the FDs. All six data subsets are therefore uploaded to the Cloud Data Store.

For the last test, AWS EventBridge fires an event that triggers the *Update Model*, updating the model contained in Model Store with data from the cloud Data Store. Upon completion, notifications are sent to each Edge Node and Field Device, instructing them to perform the update with their respective data. The model that has been trained with data from all devices is identified hereafter as the **final model**. This final model has been evaluated against the bootstrap model to assess any improvements in accuracy.

Overall, the test has been conducted in two phases. In the first phase, the three available B-L4S5I-IOT01A devices have been attached to the AWS cloud in a Cloud IoT Group. At the second stage, they have been connected to the local network and configured in the Edge IoT Group managed via the Greengrass Edge Node. Despite the limitation of not having all six IoT devices simultaneously, the experimentation and the comparisons are still valid, as: (i) the test methods in the two phases are coherent; (ii) tests comparing the two configurations do not require all IoT devices to be online at the same time; (iii) the final model update occurs by processing all the six parts of the dataset mapped to the six IoT Field Devices anyway, as explained above.

Table 4
Communication latencies for cloud-connected Field Devices (ms).

		Dataset subset			Overall
		c1	*c2*	*c3*	
First Ack.	*Avg.*	992	609.72	600	**741.29**
	Max	2969	714	687	**2969**
	Std. Dev.	725.32	51.25	52.37	**472.25**
Second Ack.	*Avg.*	47.9	47.18	47.18	**46.35**
	Max	151	76	79	**151**
	Std. Dev.	36.18	18.47	20.06	**26.36**
Total	*Avg.*	1039.91	656.91	647.18	**787.64**
	Max	3000	747	758	**3000**
	Std. Dev.	726.29	48.51	61.05	**437.34**

6.2. Results

Building upon the prototype described in Section 5.1 and test methodology outlined in Section 6.1, this section reports results, focusing on three critical performance metrics: communication latencies, processing times, and model accuracy. The aim is to assess the overall framework performance, strengths and weaknesses and to evaluate its applicability in real-world scenarios.

Latency metrics: Latency data contains the round-trip time between the initiation of a request and the receipt of the acknowledgment messages, providing insights into the network overhead of data transfer and processing within the system. Measurement has been conducted at the Field Device level, as they are the most directly impacted by the communication latency. Average, maximum and standard deviation results are reported in Table 4 for Field Devices connected to the cloud, and in Table 5 and for edge-connected FDs. The data are visualized in Figs. 12 and 13 for cloud and edge-connected Field Devices respectively, meanwhile Fig. 14 provides an aggregated, side-by-side comparison of overall mean network latency for the two cases.

Specifically, with reference to the interactions shown in Fig. 5, the rows in Tables 4 and 5 represent the following latency metrics:

Table 5
Communication latencies for edge-connected field devices (ms).

		Dataset subset			Overall
		e1	*e2*	*e3*	
First Ack.	*Avg.*	284.9	285.9	284.4	**285.06**
	Max	322	310	325	**325**
	Std. Dev.	17.59	15.87	22.16	**18.7**
Second Ack.	*Avg.*	101.63	53.2	60.1	**72.61**
	Max	296	73	206	**296**
	Std. Dev.	71.89	13.81	49.47	**56.18**
Total	*Avg.*	386.54	308.27	313	**357**
	Max	596	371	467	**596**
	Std. Dev.	76	25.71	46.78	**58.58**

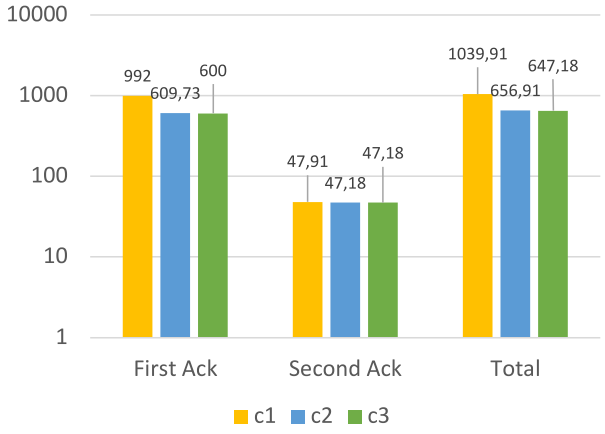


Fig. 12. Network Latency — FD to cloud (ms).

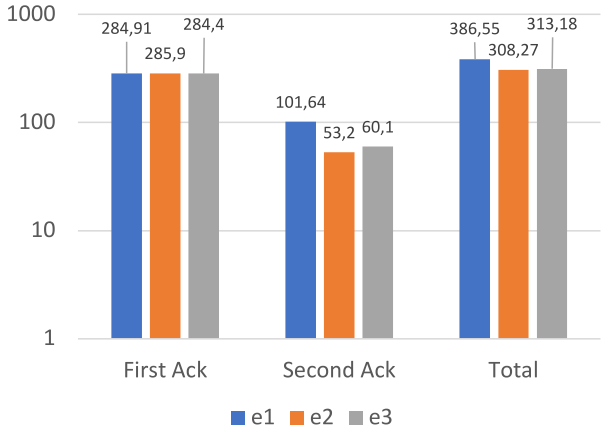


Fig. 13. Network Latency — FD to edge (ms).

- **First Ack.**: the time interval between the moment raw training data is published and when an acknowledgment for data receipt is received.
- **Second Ack.**: the elapsed time between receiving the first acknowledgment (data received) and the second acknowledgment (data stored).
- **Total**: the cumulative time from the point of raw training data publication to when an acknowledgment for data storage is received.

The columns in these tables are categorized by device, each identified by their respective data subsets. Additionally, an *Overall* column reports aggregate statistics across all three devices.

As expected, the latency tests show edge computing beats cloud computing in response times. The main reason appears to be the network overhead associated with the cloud infrastructure, since the edge is located in proximity to the IoT devices and in the same network, in contrast with the cloud which is more distant, leading to inherent delays in data transmission.

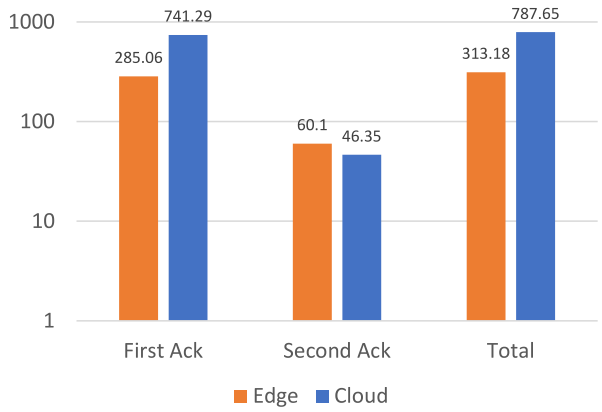


Fig. 14. Network Latency — Comparison between FD to cloud and FD to edge (ms).

It is worth noting the initial spike in latency observed in the cloud node, particularly during the first invocation of the Lambda function. This latency peak, evident under the 'c1 Max Response Time' metric, can be attributed to the cold-boot problem, a known limitation in serverless architectures. However, this limitation does not critically undermine the proposed framework efficiency. The Edge Node has not exhibited the cold-boot latency problem, highlighting a distinct advantage of the Edge Lambda Runtime against the Cloud Runtime.

Processing time: computational turnaround times have been assessed to understand the overhead associated with processing tasks executed on the different nodes. Measurements have been taken directly on the node performing the computation, bypassing factors such as Lambda startup delays and network latencies. The focus is strictly on assessing whether Mafalda can be effectively and efficiently run across all layers of the architecture, ensuring that the framework is not only lightweight but also practical for real-world deployment.

In serverless computing, the configuration of resources allocated to execute functions is a critical aspect. In AWS Lambda, the user can control the CPU tier assigned to a function indirectly, by setting the amount of required RAM.[16] This also increases the cost per second billed to the account[17]: Table 6 recalls pricing for executing Lambda functions in the *eu-central-1* region (as of 20 December, 2023) across five RAM tiers using ARM CPUs. AWS charges clients proportionally to the duration of the function. Table 7 reports on execution time measurements and costs for three operations on the cloud for each memory tier: downloading and decompressing a 256-batch, updating the model with a single batch, and predicting a batch of data. The 2048 MB configuration is always the fastest, as expected. However, in terms of cost-effectiveness, the 512 MB tier emerges as the overall best choice. For training tasks specifically, the 1024 MB tier is the most cost-effective. These findings highlight the importance of function testing to identify the optimal cloud node configuration for specific tasks.

Table 8 and Fig. 15 compare results on the average, standard deviation, and maximum execution time among Field Devices, Edge Devices, and the 512 MB tier Cloud Nodes.

In the experimental setup, the Edge Node has shown faster execution times compared to the cloud in terms of computational speed, meanwhile the time required to download and prepare the batch for processing is higher on edge. Both the edge and cloud infrastructures

[16] https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[17] https://aws.amazon.com/lambda/pricing/.

Table 6
AWS lambda pricing in eu-central-1 region ($10^{-7}$$/ms).

Memory config. (MB)	ARM CPU pricing
128	0.021
512	0.083
1024	0.167
1536	0.25
2048	0.333

Table 7
Cloud node: execution time (ms) and cost ($10^{-7}$$).

Memory config (MB)		Retrieve batch	Train on batch	Predict batch	Total
128	*Time*	285.72	448.14	193.23	927.09
	Cost	6.00	9.41	4.06	19.47
512	*Time*	71.16	108.62	48.31	228.09
	Cost	**5.91**	9.02	**4.01**	**18.93**
1024	*Time*	60.84	49.74	31.83	142.41
	Cost	10.16	**8.31**	5.32	23.78
1536	*Time*	57.00	34.41	17.89	109.30
	Cost	14.25	8.60	4.47	27.33
2048	*Time*	51.31	31.26	11.18	93.75
	Cost	17.09	10.41	3.72	31.22

Table 8
Mafalda functions training and prediction performance (ms).

		Retrieve batch	Train on batch	Predict batch
Field device	*Avg.*	*N/A*	10 550	9286.2
	Max	*N/A*	10 625	9527
	Std. Dev.	*N/A*	66.67	160.51
Edge device	*Avg.*	166.61	68.97	26.47
	Max	185.14	73.22	28.07
	Std. Dev.	14.09	2.29	1.44
Cloud node	*Avg.*	71.16	108.62	48.31
	Max	116.16	135.53	64.60
	Std. Dev.	23.22	16.17	21.15

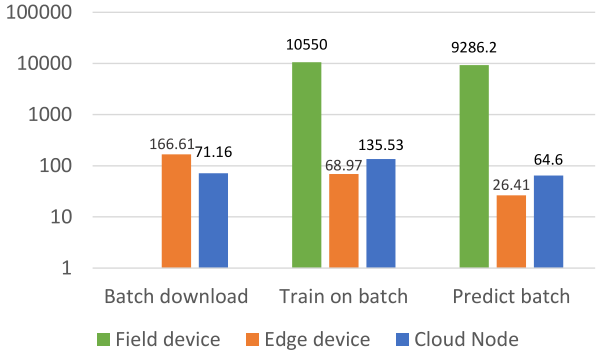


Fig. 15. Mafalda train and predict performance (ms).

offer vertical scalability, since the Greengrass software stack can be installed on more capable hardware at the edge, while the cloud AWS Lambda execution tier can be increased. A direct cost comparison is a complex task, since it should consider edge hardware and energy cost models as well as IoT Core messaging and storage expenses. AWS charges each device a fixed cost of $0.18 per month, regardless of its capabilities. Therefore, when the number of requests increases significantly, edge devices may become more cost-effective.

An expected advantage of the cloud infrastructure is its superior horizontal scalability compared to the edge devices, although the limited scope of this study, focusing on a small number of devices, could not provide the conditions under which this advantage could be clearly exposed and measured.

Table 9
Bootstrap model — Confusion matrix.

dws	jog	sit	std	ups	wlk	<– *Classified as*
134	3	1	4	11	12	**dws**
23	138	0	1	4	18	**jog**
0	0	431	21	0	0	**sit**
0	0	5	395	0	0	**std**
40	1	0	7	122	13	**ups**
91	18	7	3	25	260	**wlk**

Table 10
Bootstrap model — Evaluation summary.

Precision	Recall	F1Score	Accuracy	Class
0.465	0.812	0.592	–	*dws*
0.863	0.750	0.802	–	*jog*
0.971	0.954	0.962	–	*sit*
0.916	0.988	0.951	–	*std*
0.753	0.667	0.707	–	*ups*
0.858	0.644	0.736	–	*wlk*
0.804	0.802	0.792	0.828	***Average***

Table 11
Final model — Confusion matrix.

dws	jog	sit	std	ups	wlk	<– *Classified as*
128	2	1	4	5	25	**dws**
21	145	1	0	3	14	**jog**
0	0	452	0	0	0	**sit**
0	0	0	400	0	0	**std**
17	0	0	4	132	30	**ups**
82	5	11	2	39	265	**wlk**

In addition, it is possible to note how IoT devices have been able to perform machine learning tasks, thanks to Mafalda's ability to operate in resource-constrained environments. Although the processing time on IoT devices is considerably higher than both the edge and the cloud, even when accounting for network latencies, considering that every batch amounts to 256 s of data acquisitions w.r.t. the reference dataset, IoT model training and prediction times can be deemed as acceptable in a realistic scenario. The capability of Mafalda to execute on these devices is a distinct advantage of the proposed framework. This is especially relevant in harsh or challenging environments where network communications may be limited or unreliable, thereby enhancing the framework versatility.

Model accuracy: finally, the effectiveness of the ML algorithm and the impact of the federated learning environment have been evaluated through prediction accuracy measurements. This addresses the algorithm's ability to incrementally learn across the architecture by evaluating its accuracy metrics derived from the confusion matrix. The evaluation consists in three steps: initially, a bootstrap model is selected using a subset of the original dataset, as described in Section 5.2; this model is trained and evaluated to record accuracy metrics; subsequently, the same metrics are captured for the final model that has undergone updates with partial models and data from the Field Devices. By comparing these two sets of measurements, the analysis aims to assess whether the framework can achieve incremental improvement. Table 9 reports the confusion matrix for the bootstrap model and Table 10 shows the associated performance metrics, including precision, recall, and F1 score for each class, as well as the overall accuracy. Tables 11 and 12 provide the same data for the final model.

The prediction accuracy has exhibited only a slight improvement of ∼2% from the initial bootstrap model to the final checkpoint. This modest gain can be deemed as more indicative of limitations of Mafalda itself rather than a shortcoming of the federated learning approach. One significant limitation is Mafalda's need for incremental training across all layers of the architecture without increasing its expressiveness, which may not optimally leverage the computational capabilities at each layer. It is worth noting that the framework enables a

Table 12
Final model — Evaluation summary.

Precision	Recall	F1Score	Accuracy	Class
0.516	0.776	0.620	–	*dws*
0.954	0.788	0.863	–	*jog*
0.972	1.000	0.986	–	*sit*
0.976	1.000	0.988	–	*std*
0.737	0.721	0.729	–	*ups*
0.793	0.656	0.718	–	*wlk*
0.825	0.824	0.817	0.851	***Average***

model update feedback loop for progressive performance improvement in realistic applications based on continuous data streams. Furthermore, it is able to execute other ML algorithms, including neural networks that are trained incrementally in mini-batches. However, a careful trade-off with the computational demands of more complex algorithms is required in order to not preclude their deployment to resource-constrained IoT devices.

Based on the application requirements, a two-tiered approach could also be envisioned: deploying a more expressive model on the cloud and edge layers while retaining a less computationally intensive model like Mafalda on Field Devices. This strategy would aim to balance the trade-offs between computational resources and prediction accuracy, thereby providing a good compromise between the two solutions.

7. Conclusion and future work

This paper has introduced a novel serverless computing architecture extending the cloud-to-edge continuum to Internet of Things field devices. A federated learning framework, leveraging an existing lightweight IoT-oriented machine learning algorithm, has been defined on top of it. A noteworthy feature of the proposal is that IoT devices are involved in both ML model training and prediction tasks, which are implemented as serverless functions at the edge and in the cloud using the same codebase as the IoT device. The complete framework has been implemented on a prototypical testbed exploiting AWS cloud technologies, a Raspberry Pi 4 edge device and three STM32 microcontrollers as IoT devices. A case study with a well-known activity recognition dataset has provided an illustrative motivating example, while experimental tests on communication latencies, training and prediction turnaround time, and accuracy improvement across the federated architecture suggest the feasibility and usefulness of the proposed approach.

Future work will involve several aspects. A formal analysis of the cost model of the proposed architecture for cloud-to-thing infrastructures will be useful to characterize the proposal, highlight its limitations and improve it. Experimental tests at a larger scale, which more data and a higher number of devices, will be carried out to analyze stress conditions for edge devices in the proposed architecture and to assess cloud scalability. Furthermore, porting the prototype to other cloud–edge stacks and IoT device families will corroborate the provided considerations about real-world feasibility of the approach. The framework itself is susceptible to improvements, by introducing more dynamic and context-aware resource allocation policies. Finally, the adopted Mafalda ML algorithm itself is ongoing investigations to increase its effectiveness for federated learning, by including support for data streams, fine-grained incremental learning and further hyperparameter optimizations concerning thresholds on the training matrix and entropy-based conditions on the model generation. These improvements shall increase model accuracy and generalization capabilities.

CRediT authorship contribution statement

Davide Loconte: Methodology, Software, Investigation, Data curation, Visualization, Writing – original draft. **Saverio Ieva:** Methodology, Software, Investigation, Resources, Writing – original draft. **Agnese Pinto:** Investigation, Visualization, Validation, Writing – original draft, Writing – review & editing, Project administration. **Giuseppe Loseto:** Conceptualization, Methodology, Software, Investigation, Validation, Writing – original draft, Writing – review & editing. **Floriano Scioscia:** Conceptualization, Methodology, Investigation, Validation, Writing – original draft, Writing – review & editing. **Michele Ruta:** Conceptualization, Validation, Writing – original draft, Writing – review and editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: All authors reports financial support was provided by Government of Italy Ministry of Enterprises and Made in Italy. All authors reports financial support was provided by Government of Italy Ministry of University and Research.

Data availability

Used data are already publicly available.

References

[1] Shanhong Liu, Edge computing, 2021, https://www.statista.com/study/82641/edge-computing/.
[2] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, Rajiv Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Comput. 3 (6) (2016) 76–83.
[3] Tsozen Yeh, Shengchieh Yu, Realizing dynamic resource orchestration on cloud systems in the cloud-to-edge continuum, J. Parallel Distrib. Comput. 160 (2022) 100–109.
[4] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, Osmotic cloud-edge intelligence for IoT-based cyber-physical systems, Sensors 22 (6) (2022).
[5] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, Albert Y. Zomaya, Edge intelligence: The confluence of edge computing and artificial intelligence, IEEE Internet Things J. 7 (8) (2020) 7457–7469.
[6] Jiangchao Yao, Shengyu Zhang, Yang Yao, Feng Wang, Jianxin Ma, Jianwei Zhang, Yunfei Chu, Luo Ji, Kunyang Jia, Tao Shen, et al., Edge-cloud polarization and collaboration: A comprehensive survey for AI, IEEE Trans. Knowl. Data Eng. 35 (7) (2022) 6866–6886.
[7] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, Yuan Gao, A survey on federated learning, Knowl.-Based Syst. 216 (2021) 106775.
[8] Hossein Shafiei, Ahmad Khonsari, Payam Mousavi, Serverless computing: A survey of opportunities, challenges, and applications, ACM Comput. Surv. 54 (11s) (2022).
[9] Lachit Dutta, Swapna Bharali, TinyML meets IoT: a comprehensive survey, Internet Things 16 (2021) 100461.
[10] Michele Ruta, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, Agnese Pinto, A multiplatform reasoning engine for the semantic web of everything, J. Web Semant. 73 (2022) 100709.
[11] Theo Lynn, John G. Mooney, Brian Lee, Patricia Takako Endo, The Cloud-to-Thing Continuum, Palgrave Macmillan, Cham, Switzerland, 2020.
[12] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Agnese Pinto, Eugenio Di Sciascio, Machine learning in the internet of things: A semantic-enhanced approach, Semant. Web 10 (1) (2019) 183–204.
[13] Mohammad Malekzadeh, Richard G. Clegg, Andrea Cavallaro, Hamed Haddadi, Mobile sensor data anonymization, in: Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI '19, ACM, New York, NY, USA, 2019, pp. 49–58.
[14] Laith Abualigah, Ali Diabat, Putra Sumari, Amir H. Gandomi, Applications, deployments, and integration of Internet of Drones (IoD): a review, IEEE Sens. J. 21 (22) (2021) 25532–25546.
[15] Leonardo Militano, Adriana Arteaga, Giovanni Toffetti, Nathalie Mitton, The cloud-to-edge-to-IoT continuum as an enabler for search and rescue operations, Future Internet 15 (2) (2023) 55.
[16] Bijan Parsia, Sebastian Rudolph, Markus Krötzsch, Peter Patel-Schneider, Pascal Hitzler, OWL 2 Web Ontology Language Primer, second ed., W3C Recommendation, W3C, 2012, http://www.w3.org/TR/owl2-primer.
[17] Li Yang, Abdallah Shami, On hyperparameter optimization of machine learning algorithms: Theory and practice, Neurocomputing 415 (2020) 295–316.
[18] Keyan Cao, Yefan Liu, Gongjie Meng, Qimeng Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728.

[19] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, Omer Rana, The internet of things, fog and cloud continuum: integration and challenges, Internet Things 3-4 (2018) 134–155.
[20] Francesco Tusa, Stuart Clayman, End-to-end slices to orchestrate resources and services in the cloud-to-edge continuum, Future Gener. Comput. Syst. 141 (2023) 473–488.
[21] Kun Cao, Shiyan Hu, Yang Shi, Armando Walter Colombo, Stamatis Karnouskos, Xin Li, A survey on edge and edge-cloud computing assisted cyber-physical systems, IEEE Trans. Ind. Inform. 17 (11) (2021) 7806–7819.
[22] Lanfang Sun, Xin Jiang, Huixia Ren, Yi Guo, Edge-cloud computing and artificial intelligence in internet of medical things: Architecture, technology and application, IEEE Access 8 (2020) 101079–101092.
[23] Peter Arthurs, Lee Gillam, Paul Krause, Ning Wang, Kaushik Halder, Alexandros Mouzakitis, A taxonomy and survey of edge cloud computing for intelligent transportation systems and connected vehicles, IEEE Trans. Intell. Transp. Syst. 23 (7) (2022) 6206–6221.
[24] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, Virginia Smith, Federated learning: Challenges, methods, and future directions, IEEE Signal Process. Mag. 37 (3) (2020) 50–60.
[25] Victor Casamayor Pujol, Praveen Kumar Donta, Andrea Morichetta, Ilir Murturi, Schahram Dustdar, Edge intelligence-research opportunities for distributed computing continuum systems, IEEE Internet Comput. 27 (4) (2023) 53–74.
[26] Daniel Rosendo, Alexandru Costan, Patrick Valduriez, Gabriel Antoniu, Distributed intelligence on the edge-to-cloud continuum: A systematic literature review, J. Parallel Distrib. Comput. 166 (2022) 71–94.
[27] Daniel R. Torres, Cristian Martín, Bartolomé Rubio, Manuel Díaz, An open source framework based on kafka-ML for distributed DNN inference over the cloud-to-things continuum, J. Syst. Archit. 118 (2021) 102214.
[28] Nikita Kotsehub, Matt Baughman, Ryan Chard, Nathaniel Hudson, Panos Patros, Omer Rana, Ian Foster, Kyle Chard, Flox: Federated learning with faas at the edge, in: 2022 IEEE 18th International Conference on E-Science (E-Science), 2022, pp. 11–20.
[29] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, Kyle Chard, Funcx: A federated function serving fabric for science, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020, pp. 65–76.
[30] P. Patros, M. Ooi, V. Huang, M. Mayo, C. Anderson, S. Burroughs, M. Baughman, O. Almurshed, O. Rana, R. Chard, K. Chard, I. Foster, Rural AI: Serverless-powered federated learning for remote applications, IEEE Internet Comput. 27 (02) (2023) 28–34.
[31] Andreas Grafberger, Mohak Chadha, Anshul Jindal, Jianfeng Gu, Michael Gerndt, FedLess: Secure and scalable federated learning using serverless computing, in: 2021 IEEE International Conference on Big Data (Big Data), 2021, pp. 164–173.
[32] Jer Shyuan Ng, Wei Yang Bryan Lim, Zehui Xiong, Xianbin Cao, Jiangming Jin, Dusit Niyato, Cyril Leung, Chunyan Miao, Reputation-aware hedonic coalition formation for efficient serverless hierarchical federated learning, IEEE Trans. Parallel Distrib. Syst. 33 (11) (2022) 2675–2686.
[33] Maurizio Giacobbe, Francesco Alessi, Angelo Zaia, Antonio Puliafito, et al., Arancino.cc: an open hardware platform for urban regeneration, Int. J. Simul. Process Model. 15 (4) (2020) 343–357.

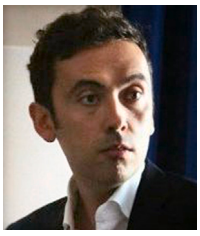
Davide Loconte received the M.S. degree in Information Technology Engineering from the Polytechnic University of Bari, Bari, Italy. He is currently a Ph.D. student at the Information Systems Laboratory in the same institution. His current research interests include Edge Computing, Machine Learning, and Semantic Web Technologies.

Saverio Ieva received the M.S. degree in Information Technology Engineering from the Polytechnic University of Bari, Bari, Italy. He is currently a Ph.D. student at the Information Systems Laboratory in the same institution. His current research interests include knowledge representation systems, artificial intelligence for cyber–physical systems and distributed ledger technologies. He has co-authored several papers in international workshops and conferences.

Agnese Pinto received the master's degree in Management Engineering and the Ph.D. in Electrical and Information Engineering from Polytechnic University of Bari in 2004 and 2015, respectively. She is currently an assistant professor with the same institution. Research activity of Agnese Pinto started in 2004 soon after graduation. Her research interests include Description Logics, semantic matchmaking, ubiquitous knowledge management and storage, machine learning for data mining in pervasive environments. She has co-authored papers about her research interests in international journals and conferences.

Giuseppe Loseto received the master's degree in Information Technology Engineering from Polytechnic University of Bari in 2009, and the Ph.D. in Information Technology Engineering in 2013 from the same institution. He is currently an assistant professor at the LUM University "Giuseppe Degennaro". His research interests include pervasive computing and the Internet of Things, knowledge representation systems and applications for ubiquitous smart environments. On these topics, he has co-authored over 50 papers in international journals and conferences.

Floriano Scioscia received the master's degree in Information Technology Engineering with honors from Polytechnic University of Bari in 2006, and the Ph.D. in Information Engineering in 2010 from the same institution. He is currently an associate professor at the Information Systems Laboratory (SisInfLab) in the same university. His research interests include knowledge representation systems and applications for pervasive computing, cyber–physical systems and the Internet of Things. He co-authored over 80 papers in international journals, edited books and conferences.



Michele Ruta received the master's degree in Electronics Engineering from Polytechnic University of Bari in 2002 and the Ph.D. in Computer Science in 2007. He is currently full professor at the same institution. His research interests include pervasive computing and ubiquitous web, knowledge representation systems and applications for wireless ad-hoc contexts. On these topics, he has co-authored more than 140 papers in international journals, edited books and conferences. He is involved in international and national research projects and is Program Committee member of several international conferences and workshops. He is also editorial board member of journals in areas related to his research interests.