A Serverless Cloud-to-Thing Framework for TinyMLOps Workflows

Davide Loconte[1,†], Saverio Ieva[1,†], Agnese Pinto[1,†], Filippo Gramegna[1,†], Giuseppe Loseto[2,*,†], Floriano Scioscia[1,*,†] and Michele Ruta[1,†]

[1]*Polytechnic University of Bari, Via Orabona 4, I-70125 Bari, Italy*
[2]*LUM "Giuseppe Degennaro" University, Strada Statale 100 km 18, I-70010 Casamassima (BA), Italy*

Abstract

The deployment of Machine Learning (ML) across the Cloud-to-Thing continuum is a significant challenge, particularly when considering the heterogeneity of the available devices. This work introduces a general-purpose framework for Tiny Machine Learning Operations (TinyMLOps) that enables the orchestration of ML workflows in distributed, serverless environments spanning cloud, edge, and Internet of Things (IoT) nodes. The architecture follows an event-driven model in which each node—depending on its capabilities—implements a minimal mandatory set of components and an optional set of extended functionalities. Nodes advertise their capabilities and collaboratively fulfill MLOps tasks by handling requests they can satisfy. This decentralized approach allows for dynamic, context-aware distribution of operations across networks of heterogeneous resource-constrained devices. The framework is validated by means of a prototype implementation and early experiments involving STM32-based microcontrollers and Raspberry Pi edge devices.

Keywords

TinyMLOps, Edge Intelligence, Microcontroller Deployment, Model Compression, Cloud-to-Thing, Serverless Inference

1. Introduction

Machine Learning Operations (MLOps) frameworks offer solutions for the continuous integration, deployment, and monitoring of Machine Learning (ML) models. Basically, they combine Development Operations (DevOps) methodologies and data engineering techniques to automate and standardize ML workflows across multiple environments. Conventional MLOps frameworks rely on persistent connectivity and substantial computational resources, which are not always available across the Cloud-to-Thing continuum, which includes Internet of Things (IoT) devices with constrained processing, memory and network resources. Applying MLOps in this context also introduces specific challenges due to heterogeneity of compute nodes, ranging from centralized cloud servers to edge and microcontroller-class IoT devices [1]. In IoT scenarios, Tiny Machine Learning (TinyML) represents an emerging research area focusing on the deployment of compact ML models on resource-constrained devices, in order to enable real-time analysis and intelligence directly where new data are sensed or generated [2]. Potential applications of TinyML impact on several sectors, including predictive maintenance in Industrial IoT [3], anomaly detection in smart infrastructure [4] and human activity recognition via embedded and wearable devices [5]. TinyML can also integrate with the Semantic Web of Everything (SWoE) paradigm [6], which extends the Semantic Web vision to everyday physical objects for creating a new class of intelligent,

MLOps25 Workshop on Machine Learning Operations – the 1st ECAI Workshop on MLOps, October 25–30, 2025, Bologna, Italy.
*Corresponding authors.
†These authors contributed equally.
davide.loconte@poliba.it (D. Loconte); saverio.ieva@poliba.it (S. Ieva); agnese.pinto@poliba.it (A. Pinto); filippo.gramegna@poliba.it (F. Gramegna); loseto@lum.it (G. Loseto); floriano.scioscia@poliba.it (F. Scioscia); michele.ruta@poliba.it (M. Ruta)
https://sisinflab.poliba.it/loconte (D. Loconte); https://sisinflab.poliba.it/ieva (S. Ieva); https://sisinflab.poliba.it/pinto (A. Pinto); https://sisinflab.poliba.it/gramegna (F. Gramegna); https://www.lum.it/docenti/giuseppe-loseto/ (G. Loseto); https://sisinflab.poliba.it/scioscia (F. Scioscia); https://sisinflab.poliba.it/ruta (M. Ruta)
0000-0002-0182-4672 (D. Loconte); 0000-0001-8598-5504 (S. Ieva); 0000-0002-2502-7467 (A. Pinto); 0000-0003-4162-957X (F. Gramegna); 0000-0002-7995-8494 (G. Loseto); 0000-0002-7859-9602 (F. Scioscia); 0000-0003-1070-2279 (M. Ruta)

interoperable and context-aware autonomous systems able to perform local knowledge-based inference [7], facilitating dynamic distributed decision-making and process automation across heterogeneous environments. Nonetheless, the integration of TinyML into complete ML pipelines remains an open challenge. Continuous model improvement depends on feedback received from deployed models. On the contrary, edge devices cannot always store or transmit raw data due to bandwidth, power or privacy constraints. This limits automated retraining workflows that represent an important aspect of MLOps. Additionally, the development of compressed models often requires specialized tools, which may not integrate smoothly with Off-The-Shelf (OTS) ML development platforms. Finally, managing the complete lifecycle of TinyML models, consisting of training, versioning, deployment and validation, requires a lightweight yet robust MLOps framework adapted to the constraints of edge and IoT devices.

This paper attempts to address these limitations by proposing a distributed, serverless, event- and message-driven framework for TinyMLOps (TinyMLOps), aiming to support end-to-end ML lifecycle execution across the Cloud-to-Thing continuum. The framework defines a modular architecture that exposes a set of minimal and optional components and communicates through a Publish/Subscribe (pub/sub) pattern. The primary contributions can be summarized as follows: (i) a computing node architecture supporting serverless TinyMLOps across heterogeneous cloud, edge, and IoT devices; (ii) a prototype implementation leveraging Amazon Web Services (AWS) IoT Core, AWS Greengrass, and STM32-based nodes running the Zephyr Real Time Operating System (RTOS); (iii) an early experimental evaluation quantifying latency and processing time across networked nodes under realistic constraints.

The reminder of this paper is structured as follows. Section 2 reviews the foundational concepts underlying this work. Section 3 describes the proposed conceptual architecture, which is used in Section 4 to implement a prototype and perform an initial functional performance evaluation. Finally, conclusion and future work directions are discussed in Section 5.

2. Background

To ensure the paper is self-contained, this section presents the key technologies discussed throughout, before a discussion of relevant related work.

2.1. Serverless computing in the Cloud-to-Thing Continuum

The Cloud-to-Thing continuum is a distributed computing paradigm that integrates IoT field devices, edge nodes, and centralized cloud data centers. Serverless computing has emerged as a compelling architectural paradigm encapsulating code in stateless *functions*, which can be deployed, invoked and scaled automatically by the infrastructure, without the requirement for the user to manage the underlying servers. Although initially limited to cloud environments, this paradigm has since expanded to comprehend both edge and IoT contexts. In the cloud, serverless platforms automatically scale and allocate resources in response to incoming request workloads. Similarly, these principles are now being applied at the network edge [8]. Deploying serverless functions across the cloud, fog, and edge continuum offers substantial benefits. This methodology enables data processing closer to the point of generation of data, aligning with emerging paradigms in distributed computing [9]. It leads to improvements in system reliability and end-to-end response times.

Although serverless computing provides additional flexibility in edge environments, there are significant technical challenges to solve. Compared to cloud servers, edge devices and IoT nodes typically have restricted Central Processing Unit (CPU), memory, and power resources. If not properly managed, these constraints can lead to increased response times or higher operational costs. Additionally, the "design once, provide anywhere" goal [10] for serverless functions is hindered by hardware and software heterogeneity, requiring attention to resource management and compatibility. Addressing these challenges often involves the use of intelligent function allocation strategies and lightweight virtualization techniques [11].

2.2. Artificial Intelligence at the Network's Edge with TinyML

Edge Intelligence is the process of implementing inference capabilities directly on resource-constrained nodes, thereby enabling local data analysis and actuation without requiring persistent cloud connectivity. This paradigm addresses critical requirements in IoT deployments, including low latency, limited bandwidth, and data locality. A key enabler is TinyML, which allows microcontroller-class devices to perform machine learning inference and, in some cases, training while complying with the resource constraints of such environments. By executing models directly on-device, TinyML minimizes communication overhead and latency, supporting real-time or offline use cases, such as industrial anomaly detection, environmental monitoring, and wearable computing [12].

However, deploying Artificial Intelligence (AI) to edge hardware introduces constraints which do not affect traditional ML pipelines. Models must be executed on bare-metal or thin RTOS runtime environments that lack standard libraries and frameworks. As a result, model rollout and maintenance are complicated by device heterogeneity and limited Application Programming Interfaces (APIs). Additionally, they must be extensively re-engineered to meet resource constraints using techniques such as quantization and pruning. To support embedded environments, Edge Intelligence requires rethinking conventional AI workflows, with the focus shifting to the availability of lightweight deployment mechanisms, support for heterogeneous endpoints, and runtime efficiency.

2.3. Related Work

When extending MLOps to the Cloud-to-Thing continuum, the reproducibility, version control, and continuous model delivery practices [13] have to be applied to distributed infrastructures featuring resource-constrained devices. This implies a smart and context-aware distribution of tasks among network nodes according to their energy, compute, and latency requirements. Typically, real-time inference is performed closer to the data source, while training and storage are carried out in the cloud [14].

A primary objective is to optimize network usage and resource consumption by decomposing ML pipelines into modular steps that can be dynamically allocated across the cloud, fog, and edge layers. These steps are deployed on platforms capable of programmatically instantiating runtime environments, configuring heterogeneous hardware and orchestrating deployment [15].

TinyML oriented frameworks further specialize these operations for highly constrained IoT devices. Emerging solutions target MicroController Units (MCUs) with limited memory and energy budgets, aiming to orchestrate ML pipelines that include inference, monitoring, and limited training. These systems address challenges such as intermittent connectivity, firmware-level deployment, and model adaptation in response to real-world operational variability [16].

Recent contributions have also detailed operational challenges and proposed enabling toolkits for TinyMLOps in real-world deployments. In particular, secure Over The Air (OTA) updates and performance evaluation of deployed models on embedded platforms have been addressed through dedicated platforms such as *RIOT-ML*, a toolkit built on the RIOT operating system (https://www.riot-os.org/) that enables secure model deployment and runtime assessment on constrained devices [17].

The issue of effective model update on energy-constrained devices has been investigated in intelligent vehicular networks, emphasizing mechanisms that tolerate high mobility, variable latency, and intermittent connectivity while preserving model integrity and consistency [18]. Moreover, event-driven architectures on ultra-low-power MCUs have been proposed for real-time classification tasks, highlighting the feasibility of continuous inference and partial reconfiguration even on constrained edge devices [19].

A broader analysis of the current limitations and requirements for scaling Edge Intelligence with TinyMLOps across heterogeneous hardware infrastructures is provided in [20], which outlines the foundational components needed for widespread adoption.

3. Serverless Cloud-to-IoT Framework for TinyMLOps

Starting from a summary of the main challenges of moving from MLOps to TinyMLOps, this section describes the proposed framework based on serverless computing. The goal is to provide a general-purpose platform, supporting distributed serverless functions for any type of application involving IoT, edge, and cloud layers.

3.1. From MLOps to TinyMLOps

To transition from standard MLOps to TinyMLOps architectures in production, the ML lifecycle must be adjusted to accommodate the following constraints of embedded deployments regarding the model footprint, runtime compatibility, deployment mechanisms, and monitoring infrastructure [16].
a. Hardware-aware model packaging. To ensure compatibility with IoT, ML models must ensure deterministic behavior, bounded latency, and fixed memory allocation. Techniques such as fixed-point quantization, pruning, and code generation are required to match microcontroller-level constraints [19].
b. Cross-platform build and deployment. Heterogeneous toolchains, memory layouts, and instruction sets complicate portability [19]. Due to the absence of a standardized runtime across microcontroller architectures, it is necessary to develop cross-compilation pipelines and platform-specific abstractions.
c. OTA and lifecycle management. Updating deployed models in the field requires OTA mechanisms compatible with limited bandwidth and memory. Some MCUs lack OTA support or cannot afford update operations due to energy and storage constraints, requiring pre-baked models during provisioning and fallback strategies for critical updates [19].
d. Minimalist runtime monitoring. Continuous telemetry is infeasible on resource-constrained nodes. Instead, lightweight mechanisms such as error flags, status counters, or sampled summaries must be used. These are typically aggregated through gateways or edge nodes [21].
e. Operational resilience across nodes. Devices at different layers of the Cloud-to-Thing spectrum exhibit variability in compute, memory, connectivity, and supported programming models (e.g., native code, containers, or serverless blueprints). TinyMLOps must account for this heterogeneity when planning deployment strategies, fallback behaviors, and runtime coordination [21].
Despite significant advances in TinyMLOps, existing platforms encounter difficulties in handling deployment and lifecycle management across multiple environments. Specifically, typical solutions frequently assume a homogeneous and reliable software infrastructure, which is not available in many IoT contexts. The subsequent section introduces a modular architecture aimed at addressing these gaps.

3.2. Architecture

The fundamental element of the architecture is the `Computing Node`, representing any computing-capable component. All nodes communicate over a Message Bus, which adheres to a pub/sub communication model. In this setup, a `Message Bus Server` manages multiple topics. Multiple `Message Bus Clients` can subscribe to or publish messages on any topic—publication is allowed even without a prior subscription. When the Message Bus Server receives a message on a given topic, it broadcasts it to all subscribed Message Bus Clients. This model aligns with widely adopted IoT protocols such as Constrained Application Protocol (CoAP) (via the observer pattern) [22] and Message Queuing Telemetry Transport (MQTT) [23], thereby facilitating practical implementation. This architecture defines the internal structure and responsibilities of the Computing Node. Within the Cloud-to-Thing continuum, it is unfeasible to enforce a uniform software stack across all nodes. For instance, MCUs are constrained in both memory and storage, making deployment of modules such as Local Storage impractical. Furthermore, due to their low duty cycle and energy constraints, these devices are generally unsuitable to act as Message Bus Servers. Based on these observations, the framework comprises two classes of computing nodes, depicted in Figure 1 and Figure 2 respectively: the `Minimal Computing Node`, which includes only the mandatory software components defined by the architecture, and the `Complete Computing Node`, which showcases both required and optional components.

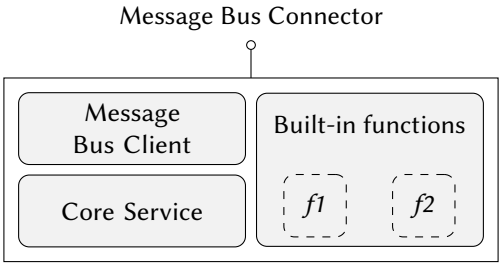

Figure 1: Minimal Computing Node

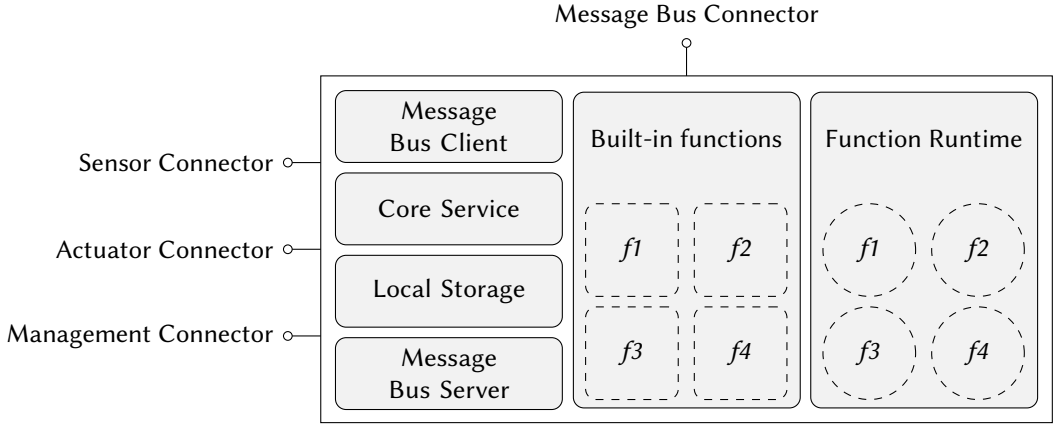

Figure 2: Complete Computing Node

Those two examples illustrate two extremes. Concrete implementations may include any configuration that integrates the mandatory modules shown in Figure 1, along with any combination of the optional modules and connectors depicted in Figure 2. This design choice allows the proposal to be flexible enough and to accommodate a large spectrum of hardware devices, spanning from less capable MCUs to cloud nodes. In particular, the mandatory functional blocks are the following:

- *Message Bus Client*: receives messages from the subscribed topics, publishes messages on the desired topic, and manages the connection with the nearest Message Bus Server;
- *Message Bus*: the only mandatory networking interface for the framework. It must implement the communication stack required to allow the Message Bus Client to connect and exchange data with the closest Message Bus Server;
- *Core Service*: manages the node and orchestrates the lifecycle of the other internal components. It processes incoming messages from the Message Bus Client, parses the response, and ascertains whether the current node has the ability to respond to the request or the message can be forwarded to another Message Bus Server that is capable of satisfying it. If both solutions fail, an error message must be returned. In addition, this component allows third parties to schedule more complex tasks, transfer data, move functions, and manage resources effectively by exposing the local node's capabilities to the external world;
- *Built-in functions*: zero or more hard-coded functions that are embedded in the device. They cannot be migrated, moved, or updated to other nodes, making them particularly relevant for devices that cannot support OTA updates or do not have sufficient resources to run a serverless runtime. Despite their static nature, these functions remain valuable within the overall architecture. This block can be skipped if the device does not do anything, like fog nodes that just relay between the edge and the cloud platform.

A more capable node can incorporate additional components. While they are not essential for basic compliance, they provide the architecture with the flexibility to execute complex tasks throughout the entire Cloud-to-Thing continuum by opportunistically utilizing all available resources.

- *Local Storage*: provides persistent or semi-persistent storage capabilities. It can be used to cache intermediate results, store logs, or maintain state across reboots or network partitions. Nodes equipped with this module can support more complex functions and participate in data-centric workflows.
- *Message Bus Server*: in addition to acting as a client, a node may optionally implement the message bus server component, enabling it to work as a communication hub for other nodes. This is particularly useful for edge clusters, where a device can act as a local coordinator or aggregation point for nearby constrained nodes.
- *Function Runtime*: enables the dynamic deployment, execution, and migration of serverless functions across nodes. Unlike Built-in functions functions, which are statically embedded, the function runtime supports computational offloading, adaptive deployment, and orchestration of distributed tasks.
- *Sensor Connector* and *Actuator Connector*: logical connectors allowing the node to interact with the physical environment. Sensor connectors receive input from attached sensing peripherals, while actuator connectors allow the node to command physical components.
- *Management Connector*: allows external systems and human interfaces to carry out advanced node administration. Through this connector, remote entities can inspect status, retrieve metrics, configure components, update deployed functions, or control lifecycle transitions.

3.3. Lifecycle Operations

Several workflows can be implemented using the architecture outlined in Section 3.2. Operations are executed and coordinated in a decentralized manner. In general, a node joins the network by establishing a connection with a neighboring Message Bus Server. A node can establish connections with multiple neighbors. A unique identifier is always assigned to the node. When the connection is established, the node publishes its capabilities. The Core Service is responsible for storing this information and routing the message appropriately.

The topics are organized following the pub/sub communication pattern, with requests being sent to the Message Bus Server in a general topic and responses being published on topics exclusively designated for the node-server pair. As shown in Figure 3, the message is forwarded to a node in the network that possesses the desired capability in a dedicated and private topic if the server-acting node is unable to fulfill the request. The server may attempt to forward the message to another Message Bus Server if none are found. The node is issued an error message if an error occurs or if no node capable of responding is found.

The following high-level operations form the basis of typical TinyMLOps workflows. Each operation is executed via message exchange over the Message Bus Server, and routed by the Core Service component based on the discovery of available capabilities. All data and model artifacts are associated with descriptors that specify their type and structure. Data descriptors may specify dimensionality, format, and usage type (e.g., training, test, validation). For models, descriptors define input shape, output type, and associated task metadata. The Core Service is responsible for registering and matching these descriptors against operation requirements. An operation is executed only if the associated descriptors satisfy the input constraints of the function to be applied.

Table 1 summarizes the operations available in the proposed framework that define the basic building blocks of TinyMLOps workflows. These functions are either built-in or serverless, and they are each initiated by the Core Service based on messages received in the Message Bus. The logic behind the

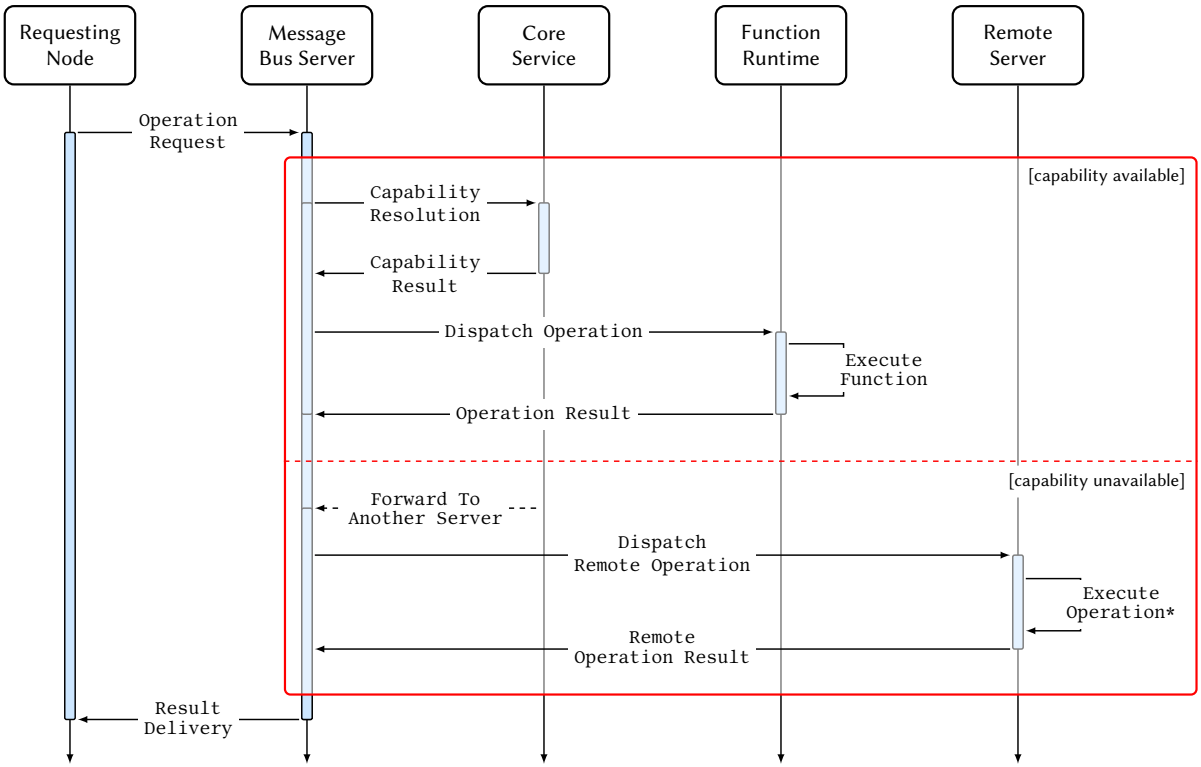

Figure 3: Sequence diagram of a generic TinyMLOps operation dispatch. Capability discovery, communication with remote core service, and remote function invocation are omitted for brevity, as they work like the local operation execution case.

removal and deletion of old, stale data is implemented of the Core Service. In the context of TinyMLOps, this set of operations can be combined to implement complex distributed architectural patterns.

4. Early Experiments

This section presents a preliminary experimental evaluation carried out to assess the feasibility of a prototype implementation of the proposed TinyMLOps framework. This prototype has undergone functional testing to verify whether the proposed architecture accurately responds to system events and user requests, as well as to identify the potential benefits and limitations of adopting a Cloud-to-Thing approach to TinyMLOps.

4.1. Prototype Implementation

The prototype implementation adopts the Cloud-to-Thing TinyMLOps architecture presented in Section 3.2. A guiding design principle during experimentation is the reuse of OTS components whenever possible. The prototype consists of the following nodes:

- *Cloud Node:* AWS (https://aws.amazon.com/) serves as the public cloud infrastructure used to implement the Cloud Node for this experiment.
- *Edge Node:* a Raspberry Pi 4 Model B, representative of a generic microprocessor-class device.
- *IoT Nodes:* twelve STM32 B-L4S5I-IOT01A Discovery kits, simulating a network of distributed IoT devices equipped with sensors for environmental monitoring.

Hardware specifications for the Edge and IoT nodes are listed in Table 2. The AWS platform is used to implement the Cloud Node, which supports the complete set of optional modules and complies

Table 1
TinyMLOps operations grouped by functional category

Category	Operation	Description
Data Gathering	`request_data(desc)`	Request data matching the description from neighbors.
	`push_data(payload, desc)`	Send raw or preprocessed data to a target node.
Model Training	`update(model, data, config)`	Train or fine-tune a model.
	`store(model, desc)`	Save a model (trained or empty) to external storage.
	`load(desc)`	Retrieve a model matching a descriptor.
Inference	`infer(input, model)`	Run inference on input data using a specified model.
	`batch_infer(data, model)`	Apply inference across a dataset.
Evaluation	`evaluate(testset, model)`	Evaluate a model's performance using validation data.
	`log_metrics(query)`	Report evaluation results for monitoring or analysis.
Management	`announce(node_desc)`	Announce the node's available resources and functions.
	`list_datasets(desc)`	List datasets accessible from the current node.
	`list_models(desc)`	List available models or descriptors on the current node.

Table 2
Hardware specifications and interfaces of Edge and IoT nodes

Feature	Edge Node	IoT Nodes
Device	Raspberry Pi 4 Model B	STM32 B-L4S5I-IOT01A
Processor	Quad-core ARM Cortex-A72 @ 1.5 GHz	ARM Cortex-M4 @ 120 MHz
RAM	8 GB LPDDR4-3200 SDRAM	640 KB SRAM
Flash/Storage	microSD (user-defined)	1 MB Flash
Connectivity	Gigabit Ethernet, Wi-Fi 802.11ac, Bluetooth 5.0	Wi-Fi 802.11 b/g/n, Bluetooth 4.1, Sub-GHz RF
External Interfaces	2× micro-HDMI, USB 3.0/2.0, GPIO, UART, SPI, I^2C	Arduino and STMod+ connectors, GPIO, UART, SPI, I^2C
Onboard Sensors	Not included (external via HATs or USB)	Temperature, humidity, pressure, 6-axis Inertial Measurement Unit, magnetometer, microphone, time-of-flight sensor

with all mandatory components. AWS services such as *Lambda* (for the Function Runtime), *Simple Storage Service (S3)* (for Local Storage), and *IoT Core* (for the Message Bus Client and Message Bus Connector) map directly to the Complete Computing Node components described in Section 3.2 and shown in Figure 2. The Management Connector is represented by the *AWS Console* and AWS Software Development Kit (SDK), which provide global visibility and control over the system. The Core Service can be interpreted as the backbone code of the cloud provider to orchestrate and expose those services. Sensor Connector and Actuator Connector are not implemented at this level, as AWS cloud nodes do not directly interface with the physical environment.

The Edge Node is implemented using a Raspberry Pi 4 Model B running *AWS IoT Greengrass*. The node is provisioned directly from the AWS Console, along with *Moquette*, a Java-based MQTT server acting as both a Message Bus Server and a Message Bus Client for nearby IoT devices. The Local Storage component is realized as a custom *Greengrass* component wrapping a *Redis* (https://redis.io/) instance. The Greengrass Core Service itself represents the Core Service, which includes a Function Runtime capable of executing and migrating serverless Lambda functions from the Cloud Node. The node can

also be configured to communicate with physical sensors and actuators; however, these capabilities have not been implemented in the proposed experiments.

The IoT Node is a resource-constrained device based on an STM32 B-L4S5I-IOT01A development board, running custom firmware built with the *Zephyr* RTOS (https://zephyrproject.org/). Combined with some domain-specific logic, this firmware acts as the Core Service. Due to its limited resources, the node supports only the minimal configuration required for compliance. Specifically, it includes a Message Bus Client implemented using the built-in Zephyr API[1] and a collection of statically defined Built-in Functions. The onboard sensors are accessed using the integrated STM32 Hardware Abstraction Layer (HAL) and Zephyr drivers.

4.2. Materials, Methods and Results

This section reports on a set of integration and communication tests conducted with the prototype described in Section 4.1 to validate the feasibility of the TinyMLOps architecture. Furthermore, a simplified data collection task has been implemented to evaluate the messaging pipeline performance in terms of communication latency and node-level processing time.

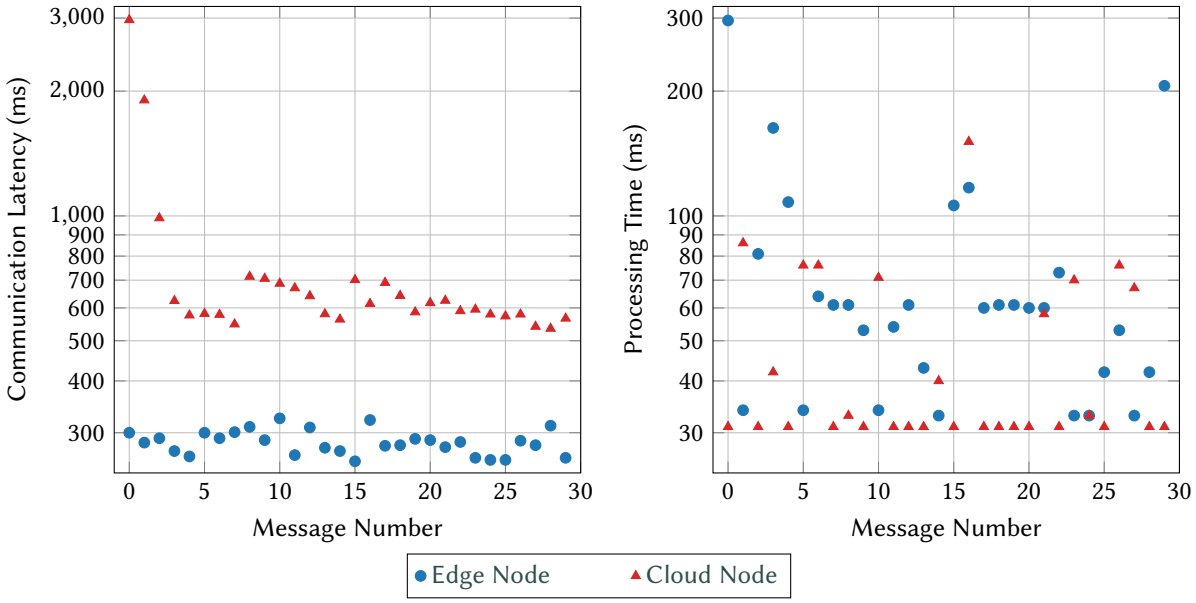

Figure 4: Latency and processing time for edge and cloud nodes over message sequence

The experimental setup begins with the configuration of both AWS IoT Core and AWS IoT Greengrass. This involves the registration of each device as a *Thing* in IoT Core, the provisioning of X.509 certificates for mutual Transport Layer Security (TLS) authentication, and the attachment of proper IoT policies to establish permissions. To authorize peer-to-peer communication and access to other AWS services, such as S3, each device is also assigned appropriate Identity and Access Management (IAM) roles. The second step involves configuring the edge device by installing the *Greengrass Core Device* software. Once the device appears on the AWS Console, a Greengrass deployment is initiated, including all essential components: Message Bus Server, Message Bus Client, Function Runtime, Local Storage, and the relevant Lambda functions. The third setup step focuses on the IoT devices. While the edge device benefits from OTS modules provided by AWS Greengrass, the IoT devices requires a custom software stack developed from scratch. To abstract board-specific hardware and facilitate code portability, Zephyr OS is exploited as the firmware's core RTOS. Identical firmware is flashed onto all twelve devices, differing only in authentication credentials and target destination: six devices are configured to send messages to the cloud node directly (C_i, $i = 1, \dots, 6$), and six to the edge node (E_i, $i = 1, \dots, 6$).

[1] https://docs.zephyrproject.org/latest/connectivity/networking/api/mqtt.html

Table 3
Average and standard deviation (in milliseconds) of communication latency and processing time per node

	Metric	E1	E2	E3	E4	E5	E6	Edge
Communication Latency	Avg.	293.80	287.80	293.00	279.60	274.00	279.60	284.63
	Std.Dev.	25.06	21.39	12.08	18.42	14.47	21.89	19.19
Processing Time	Avg.	106.00	65.80	67.20	72.20	57.60	75.20	74.00
	Std.Dev.	110.21	30.86	9.39	52.17	30.75	73.46	57.55
	Metric	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**Cloud**
Communication Latency	Avg.	1111.00	877.60	691.40	631.00	602.00	558.80	745.30
	Std.Dev.	1039.83	573.04	174.18	52.34	58.73	19.65	487.74
Processing Time	Avg.	48.00	80.40	33.60	38.80	33.20	47.20	46.87
	Std.Dev.	23.35	44.66	4.77	17.44	3.90	22.41	27.10

The edge and cloud nodes are configured to run a Lambda function that saves incoming messages in local storage during the experiment. Each IoT device sends one message per second for five seconds, using data from onboard sensors and publishing it via the message client. Upon receipt, the edge (resp. cloud) node sends an acknowledgment on a dedicated topic through its Message Bus Client. Communication latency is measured by the IoT device as the time elapsed between sending the message and receiving the first acknowledgment. A second acknowledgment is sent by the edge (resp. cloud) node after data is successfully stored, allowing the IoT device to calculate the processing time as the interval between the two acknowledgments.

Figure 4 and Table 3 present the experimental results. In the scatter plot, messages are ordered by the timestamp of the data message received from the IoT devices, as recorded by the respective Message Bus Server. The cloud node has experienced a substantially higher latency during the initial few messages it receives, which is likely due to the cold start behavior of Lambda functions. As expected, the edge node demonstrates significantly lower communication latency compared to the cloud. In contrast, the cloud node's superior computational resources are reflected in its shorter processing times. In some cases, the edge node has achieved processing times comparable to the cloud, but with greater variance. Considering communication latency, the edge node tends to have a shorter overall turnaround time. These findings suggest the overall operational efficiency can be optimized through intelligent and resource-aware task allocation. However, without careful management, task distribution may lead to suboptimal outcomes and even degrade performance compared to a purely cloud-based solution.

5. Conclusions

This article introduced a serverless, modular framework for TinyMLOps that enables the execution of ML workflows in Cloud-to-Thing architectures. The system employs a lightweight event-driven architecture and defines a set of minimal and optional software components, along with a corresponding set of operations to support a variety of devices, including microcontroller-class IoT nodes, edge nodes and cloud infrastructures. The feasibility of the proposed architecture has been demonstrated by means of a prototype implementation that employed MCU nodes running Zephyr RTOS-based firmware and AWS services. A preliminary experiment has validated the prototype's functional compliance and highlighted both the benefits and limitations of the edge computing approach compared to a centralized architecture. In particular, response time and resource utilization can be optimized by intelligently offloading certain real-time critical operations from the cloud to edge devices, despite the latter's limited performance capabilities.

Future work will concern several areas. Firstly, an extension of the orchestration capabilities will be defined through lightweight and resource-aware approaches to the distribution of computational

tasks [24]. The main goal is to allow the system to automate complex tasks based on user requests and contextual requirements. Integrating support for federated learning and formalizing service-level guarantees under resource constraints are other two key areas of development. Thorough case studies will also be conducted to assess the impact of task distribution across the cloud, edge, and IoT nodes. This study will include a scalability analysis and the characterization of the framework's fault tolerance and responsiveness.

Acknowledgments

The work has been partially supported by the *Space It Up* project, funded by the Italian Space Agency (ASI) and the Ministry of University and Research (MUR), under contract n. 2024-5-E.0 - CUP n. I53D24000060005", and by the *NXT Digital Platform* project (grant M1ERDW2), co-funded by NTT DATA Italia S.p.A. and European Regional Development Fund for Apulia Region 2014/2020 Operating Program.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] D. Loconte, S. Ieva, F. Gramegna, I. Bilenchi, C. Fasciano, A. Pinto, G. Loseto, F. Scioscia, M. Ruta, E. Di Sciascio, Serverless Microservice Architecture for Cloud-Edge Intelligence in Sensor Networks, IEEE Sensors Journal 25 (2024) 7875–7885.

[2] V. Rajapakse, I. Karunanayake, N. Ahmed, Intelligence at the extreme edge: A survey on reformable TinyML, ACM Computing Surveys 55 (2023) 1–30.

[3] E. Njor, M. A. Hasanpour, J. Madsen, X. Fafoutis, A Holistic Review of the TinyML Stack for Predictive Maintenance, IEEE Access (2024).

[4] M. Ficco, A. Guerriero, E. Milite, F. Palmieri, R. Pietrantuono, S. Russo, Federated learning for IoT devices: Enhancing TinyML with on-board training, Information Fusion 104 (2024) 102189.

[5] D. Loconte, S. Ieva, A. Pinto, G. Loseto, F. Scioscia, M. Ruta, Expanding the cloud-to-edge continuum to the IoT in serverless federated learning, Future Generation Computer Systems 155 (2024) 447–462.

[6] M. Ruta, F. Scioscia, I. Bilenchi, F. Gramegna, G. Loseto, S. Ieva, A. Pinto, A multiplatform reasoning engine for the Semantic Web of Everything, Journal of Web Semantics 73 (2022) 100709:1–14.

[7] M. Ruta, F. Scioscia, G. Loseto, A. Pinto, E. Di Sciascio, Machine Learning in the Internet of Things: a Semantic-enhanced Approach, Semantic Web Journal 10 (2019) 183–204.

[8] M. Li, J. Zhang, J. Lin, Z. Chen, X. Zheng, FireFace: Leveraging Internal Function Features for Configuration of Functions on Serverless Edge Platforms, Sensors 23 (2023) 7829.

[9] C. Calavaro, V. Cardellini, F. Lo Presti, G. Russo Russo, Beyond Cloud: Serverless Functions in the Compute Continuum, SN Computer Science 6 (2025). doi:10.1007/s42979-025-03699-7.

[10] M. A. Ahmadon, S. Yamaguchi, The Design Once, Provide Anywhere Concept for the Internet of Things Service Implementation, Evolution of Information, Communication and Computing System (2023) 43–57.

[11] B. Oliveira, N. Ferry, H. Song, R. Dautov, A. Barišić, A. R. Da Rocha, Function-as-a-service for the cloud-to-thing continuum: a systematic mapping study, in: IoTBDS 2023-8th International Conference on Internet of Things, Big Data and Security, 2023, pp. 82–93.

[12] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, A. S. Hafid, A Comprehensive Survey on TinyML, IEEE Access 11 (2023) 96892–96922. doi:10.1109/ACCESS.2023.3294111.

[13] S. Alla, S. K. Adari, What is MLOps?, in: Beginning MLOps with MLFlow: Deploy Models in AWS SageMaker, Google Cloud, and Microsoft Azure, Springer, 2020, pp. 79–124.

[14] E. Raj, D. Buffoni, M. Westerlund, K. Ahola, Edge MLOps: An Automation Framework for AIoT Applications, in: 2021 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2021, pp. 191–200.

[15] R. Miñón, J. Diaz-de Arcaya, A. I. Torre-Bastida, P. Hartlieb, Pangea: an MLOps tool for automatically generating infrastructure and deploying analytic pipelines in edge, fog and cloud layers, Sensors 22 (2022) 4425.

[16] M. Antonini, M. Pincheira, M. Vecchio, F. Antonelli, Tiny-MLOps: a framework for orchestrating ML applications at the far edge of IoT systems, in: 2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS), 2022, pp. 1–8. doi:10.1109/EAIS51927.2022.9787703.

[17] Z. Huang, K. Zandberg, K. Schleiser, E. Baccelli, RIOT-ML: toolkit for over-the-air secure updates and performance evaluation of TinyML models, Annals of Telecommunications 80 (2025) 283–297. doi:10.1007/s12243-024-01041-5.

[18] T. K. S. Flores, I. Silva, M. B. Azevedo, T. d. A. d. Medeiros, M. d. A. Medeiros, D. G. Costa, P. Ferrari, E. Sisinni, Advancing Tiny Machine Learning Operations: Robust Model Updates in the Internet of Intelligent Vehicles, IEEE Micro 45 (2025) 76–86. doi:10.1109/MM.2024.3354323.

[19] M. T. Lê, J. Arbel, TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification, in: Proceedings of the 3rd Workshop on Machine Learning and Systems, EuroMLSys '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 148–153. doi:10.1145/3578356.3592586.

[20] S. Leroux, P. Simoens, M. Lootus, K. Thakore, A. Sharma, TinyMLOps: Operational Challenges for Widespread Edge AI Adoption, in: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2022, pp. 1003–1010. doi:10.1109/IPDPSW55747.2022.00160.

[21] C. Banbury, V. Janapa Reddi, A. Elium, S. Hymel, D. Tischler, D. Situnayake, C. Ward, L. Moreau, J. Plunkett, M. Kelcey, M. Baaijens, A. Grande, D. Maslov, A. Beavis, J. Jongboom, J. Quaye, Edge Impulse: An MLOps Platform for Tiny Machine Learning, in: D. Song, M. Carbin, T. Chen (Eds.), Proceedings of Machine Learning and Systems, volume 5, Curan, 2023, pp. 254–268.

[22] C. Bormann, A. P. Castellani, Z. Shelby, CoAP: An application protocol for billions of tiny Internet nodes, IEEE Internet Computing 16 (2012) 62–67.

[23] OASIS Standard, MQTT Version 3.1.1, https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014. OASIS Standard, 29 October 2014.

[24] G. Loseto, F. Scioscia, M. Ruta, F. Gramegna, S. Ieva, C. Fasciano, I. Bilenchi, D. Loconte, Osmotic Cloud-Edge Intelligence for IoT-based Cyber-Physical Systems, Sensors 22 (2022) 2166.