

TinyKubeML: Orchestrating TinyML Models on Far-Edge Clusters

João Oliveira
joao.oliveira@fraunhofer.pt
Fraunhofer Portugal AICOS
Porto, Portugal

Fernando Rego
Filipe Sousa
fernando.rego@fraunhofer.pt
filipe.sousa@fraunhofer.pt
Fraunhofer Portugal AICOS
Porto, Portugal

Luís Almeida
lda@fe.up.pt
CISTER / FEUP - University of Porto
Porto, Portugal

Abstract

The Internet of Things (IoT) is rapidly materializing, but the growing volume of data generated by Far-Edge devices, often microcontroller-based, poses challenges for cloud-centric processing. TinyML addresses this challenge by enabling on-device ML inference, thereby reducing communication latency and cost. However, current solutions largely overlook deployment and management challenges, especially in heterogeneous, resource-constrained environments. This paper introduces TinyKubeML, a Kubernetes-based framework that enables resource-aware deployment of TinyML models on Far-Edge clusters. It abstracts device heterogeneity and automates model partitioning, artifact generation, and deployment using a custom Kubernetes Operator. TinyKubeML supports distributed inference and includes recovery mechanisms to ensure service continuity. Our evaluation shows that TinyKubeML can deploy distributed models efficiently with minimal impact on accuracy, while supporting automatic recovery in the case of device failures, demonstrating its potential to bridge the gap between scalable orchestration and TinyML deployment in IoT scenarios.

CCS Concepts

• **Networks** → **Cloud computing**; • **Computing methodologies** → **Distributed artificial intelligence**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

Keywords

Internet of Things Orchestration, Distributed TinyML, Far-Edge Device Management

1 Introduction

The Internet of Things (IoT) envisions a future in which a ubiquitous network of interconnected devices collects and infers valuable information across domains, from smart cities to industry. This vision is rapidly materializing, with projections estimating 32.1 billion connected devices by 2030, a 100% increase from 2023 (15.9 billion)[1]. The amount of data produced by these devices is also growing exponentially, creating pressure in Cloud-based infrastructures, which are often the solution used to process the generated data. Edge Computing emerged as a response [2], shifting computation closer to the data source to improve latency, bandwidth usage, and security [3]. Yet, data generation typically originates at the Far-Edge, where resource-constrained microcontroller-based devices sense and interact with their environments. Researchers have been exploring mechanisms to integrate Far-Edge devices into the Edge Computing paradigm to further exploit data locality and reduce communication costs. This effort led to the emergence of

TinyML, a technology that enables inference with simple Machine Learning (ML) models in resource-constrained devices [4].

Despite the recent advances in TinyML model training, optimization, and inference, managing an infrastructure composed of heterogeneous, resource-constrained Far-Edge devices remains a significant challenge. Fully realizing the IoT vision requires extending orchestration methodologies to span the entire continuum—from the Cloud to the Far-Edge. Some efforts targeted this integration using Kubernetes (k8s), the leading orchestration tool. Solutions such as Akri¹ and KubeEdge² incorporate Far-Edge devices as data sources but are not capable of deploying services to them. In contrast, FITA [5] enables the deployment of services on heterogeneous Far-Edge devices via the Kubernetes API, but lacks support for TinyML applications. TinyML-specific proposals often overlook deployment and management challenges or rely on proprietary solutions that hinder wide adoption. Moreover, most approaches assume monolithic deployments, overlooking the potential of distributing model components across multiple Far-Edge devices to effectively utilise available resources. There is a clear gap in orchestration solutions that integrate standard frameworks like Kubernetes with resource-aware TinyML deployments on the Far-Edge.

In this paper, we introduce TinyKubeML, a framework that combines TinyML with Kubernetes, enabling the deployment of ML models on Far-Edge clusters based on available resources. Our approach enables users to deploy neural-network-based TinyML models developed using traditional frameworks (e.g., TensorFlow, Keras), abstracting the heterogeneity of Far-Edge devices and the current network status. We extend k8s with a new Operator³ that manages a new Custom Resource that abstracts Far-Edge specific deployment details and implements TinyML orchestration logic where models can be automatically partitioned, depending on the Far-Edge cluster state, converted into artifacts compliant with the Far-Edge runtime, and deployed according to a configurable heuristic. Our approach also improves the availability of running TinyML-based applications, ensuring continuity despite device malfunctions, by leveraging k8s recovery features. To support the new Operator, we leverage the FITA platform and extend it with a TinyML interpreter in the Far-Edge device runtime, as well as support for deploying device routes through k8s, which is crucial for interconnecting Pods deployed in Far-Edge devices for distributed inference.

Therefore, this paper makes the following contributions:

¹<https://docs.akri.sh/>

²<https://kubedge.io/>

³Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components.

- An extension to FITA that adds support for the deployment of TinyML-based services;
- A Kubernetes Operator enabling resource-aware ML model partition and deployment in Far-Edge clusters.

In the remainder of the paper, we survey existing work on IoT orchestration and TinyML deployment (Section 2), detail TinyKubeML and its underlying components (Section 3), and describe our evaluation methodology, present results, and discuss implications (Section 4). Section 5 concludes the paper and outlines future work.

2 Related Work

Integrating TinyML deployments with IoT orchestration frameworks is fundamental to support dynamic applications that can adapt to heterogeneous and evolving Far-Edge clusters. Thus, we begin by surveying current IoT orchestration frameworks in Section 2.1, followed by the current efforts on TinyML deployment frameworks in Section 2.2.

2.1 Orchestration in the Far-Edge

IoT orchestration solutions abound in the literature, primarily relying on container-based virtualization technologies like Docker [6–8] to manage service deployment across heterogeneous devices. However, Far-Edge devices are often excluded from such solutions due to their constraints and lack of support for virtualization. In some cases, like the work presented by Donassolo et al. [7], Far-Edge devices are considered but treated merely as data producers, not supporting the deployment of services, given the reliance on Docker. GeneSIS [9], however, proposes an architecture capable of deploying services to Edge and Far-Edge devices by leveraging a domain-specific language that is compiled to deployable artifacts for the supported platforms. Still, this approach does not address the communication heterogeneity typical of Far-Edge environments and is incompatible with standard orchestration frameworks, limiting its applicability. Similarly, Baccelli et al. [10] propose a full-stack architecture using a JavaScript interpreter to enable deployments from the Cloud. Though enabling deployment on Far-Edge devices, this approach lacks integration with standard orchestration frameworks. Conversely, FITA [5] integrates Far-Edge devices in Kubernetes and supports service deployments using a container-like runtime. It also addresses communication heterogeneity by employing a middleware that abstracts Far-Edge protocols and data models into a standard-based protocol. Beyond academic work, Cloud Native Computing Foundation initiatives like Akri¹ and KubeEdge² provide some support for Far-Edge devices. Akri exposes Far-Edge resources to Kubernetes to enable function chaining, but does not support direct service deployment or address heterogeneity. KubeEdge introduces device twins for managing Far-Edge state via custom mappers and proprietary data models, but similarly excludes deployment capabilities on such devices.

Since FITA [5] is the only solution that supports service deployments to Far-Edge devices, integrates with standard orchestration frameworks (Kubernetes), and addresses communication protocol heterogeneity, we use it as the main building block of our proposal. Section 3.1 provides an overview of its components and how we leverage them in the context of TinyKubeML.

2.2 TinyML Deployment

While several proposals dedicated to AI models targeting Far-Edge devices can be found in the literature, the deployment phase is often overlooked and has not seen as much interest from the scientific community. These proposals usually provide tools for training and developing AI models, which are then optimized for devices with constrained resources and integrated into their firmware. TensorFlow Lite Micro [11] is one of these proposals. It is a port of TensorFlow Lite, designed to run AI models on constrained devices. It provides tools to quantize and convert models developed using the TensorFlow framework, as well as an interpreter that must be integrated into the device firmware. While TensorFlow Lite Micro tools are extensive and provide wide support for different types of models, it does not provide or define any mechanisms to deploy and update models in Far-Edge devices, leaving the implementation of these mechanisms to the developer. The emlearn [12] and uTensor⁴ frameworks follow the same model conversion approach as TensorFlow Lite Micro, but they convert the models directly to native code (C and C++, respectively), which is then integrated into the device firmware. Nevertheless, these proposals also do not fully address the deployment process since the integration of models is performed at the code level, leaving the firmware update procedure up to the developer. TVM⁵ is another proposal that uses a more holistic approach by providing an end-to-end compiler framework to enable the deployment of models on any hardware. The MicroTVM extension is focused on constrained devices, such as those used at the Far-Edge, and also converts models to native C code. Significant focus is given to the optimization process, where host-driven model execution can be used to improve the model performance. However, like the previous proposals, models are expected to be integrated and compiled with the firmware code. A traditional full firmware update approach can be used to overcome this gap while using the previously presented proposals, however, it limits the flexibility in dynamically modifying device behavior, as it is a cumbersome operation. Therefore, TinyML model deployment remains as one of the major challenges that must be tackled for the widespread adoption of TinyML, as highlighted by Szidlo and Nagy [13]. A concrete step was taken by Doyu et al. [14] introducing TinyML-as-a-Service (TinyMLaaS), an architecture that provides a higher-level abstraction of TinyML that is *as hardware and software agnostic as possible*. While the concept defines the high-level architecture of a TinyMLaaS system, it does not define communication protocols, model formats, or device runtimes. Therefore, existing academic proposals that implement this concept [15, 16] often use proprietary software and/or fail to ensure interoperability. Other commercial-focused approaches, such as EdgeImpulse⁶ or NanoEdgeAIStudio⁷, provide platforms capable of training and converting models into native code to be embedded in the Far-Edge devices, leveraging automatic code generation tools. While an attractive approach, these tools use proprietary toolchains that hamper interoperability and also leave the deployment and update procedures to the developer.

Additionally, most of the available TinyML frameworks focus on monolithic models, overlooking the possibility of distributing the

⁴<https://github.com/uTensor/uTensor>

⁵<https://tvm.apache.org/>

⁶<https://www.edgeimpulse.com/>

⁷<https://www.st.com/en/development-tools/nanoedgeaistudio.html>

model over several Far-Edge devices, thus exploiting the resources available. DIANNE [17] supports the development of distributed neural networks across heterogeneous devices but requires manual assignment of the DIANNE blocks to Far-Edge devices. AR-MDI [18] proposes a decentralized approach for distributing model layers to sequential workers based on computational and transmission delays. However, it is focused on Edge devices. NAMP [19] is a framework that partitions neural networks into smaller deployable submodels that are distributed across Far-Edge devices following a well-defined heuristic, such as communication minimization. While this framework focuses on Far-Edge devices, it uses proprietary deployment procedures and communication protocols, hampering interoperability.

There is a clear gap for a solution that leverages standard IoT orchestration tools for distributed TinyML deployments, while ensuring interoperability and minimizing vendor lock-in. We propose TinyKubeML to address these shortcomings.

3 Proposed Solution

Considering the analysis of the state-of-art presented in Section 2, our solution addresses the following gaps:

- IoT orchestration tools disregard the deployment of TinyML-based applications on Far-Edge devices. Deploying TinyML models differs significantly from deploying traditional ML models due to the constraints of Far-Edge devices, which often require runtime support for conversion or compilation operations. Section 3.1 explains how we leverage FITA and extend it to enable the deployment of TinyML services. We detail our changes to integrate a TinyML framework (TensorFlow Lite Micro) and the new Operator added to FITA, which is responsible for managing and deploying communication routes to Far-Edge devices, enabling device-to-device communication.
- Current TinyML proposals mostly focus on AI-related processes and overlook the challenges of managing an infrastructure of TinyML-enabled Far-Edge devices. Models need to be deployed, updated, and monitored in heterogeneous device networks in a standardized way to avoid vendor lock-in. Current proposals also overlook the distributed deployment of TinyML models as a way to cope with device resource constraints. Section 3.2 presents the TinyML-focused Operator built on top of FITA, showing how it simplifies the deployment and management of TinyML models in clusters of Far-Edge devices.

Figure 1 presents the architecture of our solution and how each module interacts to deploy TinyML models to constrained Far-Edge devices. We highlight the contributions of this paper and their relevance to the tools we leverage to develop our solution. The remainder of the section details each module and its extensions, where applicable.

3.1 FITA Framework

FITA is a framework that enables service deployment on Far-Edge devices that do not support traditional virtualization techniques, using the standard Kubernetes interfaces. To achieve this, FITA uses several components.

embServe [20] is the Far-Edge device runtime, built on top of the Zephyr RTOS⁸. It targets microcontroller-based Class 2 [21] and higher devices, and abstracts the device application as a set of services that can be deployed and updated without requiring full firmware updates using standard protocols. Services are modules composed of pre-compiled native code that are loaded by the Service Manager using a dynamic linker. Services can interact with OS services, such as sensors or the network stack, using the embServe Service SDK, which links the available resources in the base firmware image with the Service code. The Data Router module allows the creation of routes between internal or external Services, enabling service linkage without the intervention of Edge devices. Interaction with the Edge layer is achieved using an LwM2M⁹ connector.

NextGenGW [22] is the component handling the interaction with Far-Edge devices. It is an IoT middleware that follows a standards-based approach regarding communication protocols and data models. It streamlines communication with heterogeneous Far-Edge devices by utilising the Message Queuing Telemetry Transport protocol (MQTT) [23] and leveraging IETF SDF [24] as the data representation model. Clients interact with the Far-Edge devices using a single protocol, while NextGenGW uses translators to convert from its protocol into the Far-Edge device protocol. Within the scope of this work, NextGenGW serves as an LwM2M server for Far-Edge devices, as it is the current protocol supported by embServe.

Far-Edge Kubelet is a module that creates the virtual representation of the Far-Edge node in the Kubernetes infrastructure. Due to the constraints of the Far-Edge devices, the standard Kubelet cannot run on the device itself. Therefore, FITA's Kubelet runs in the Edge and acts as a proxy for the Node requests. The Node is registered with the Far-Edge device real resources, such as CPU and memory, and with some details relevant to Far-Edge devices, such as the available sensors and actuators. For Far-Edge devices, a Container is mapped to an embServe service, and a Pod can contain several services. When a Pod is deployed to a Node representing a Far-Edge device, the custom Kubelet fetches the embServe compatible service, which is stored in an Open Container Initiative (OCI) registry as an OCI artifact, and deploys it using the NextGenGW protocol. This process, enabled by the Far-Edge Kubelet, abstracts all Far-Edge device intricacies, allowing users to deploy services to these devices in the same way as they would deploy services to other devices supported by Kubernetes. Additionally, the Far-Edge Kubelet also collects and provides runtime metrics related to the Far-Edge device and its services, such as used memory or CPU load. These metrics can be used by Kubernetes to perform resource-based orchestration. This is especially important for TinyKubeML to achieve an overview of the available resources in the Far-Edge network.

Finally, the **Far-Edge Node Watcher** is a simple module that listens to NextGenGW messages advertising Far-Edge node connections and disconnections. When a new node connects, the watcher deploys the respective Far-Edge Kubelet. When a node disconnects,

⁸<https://www.zephyrproject.org/>

⁹<https://www.openmobilealliance.org/specifications/lwm2m>

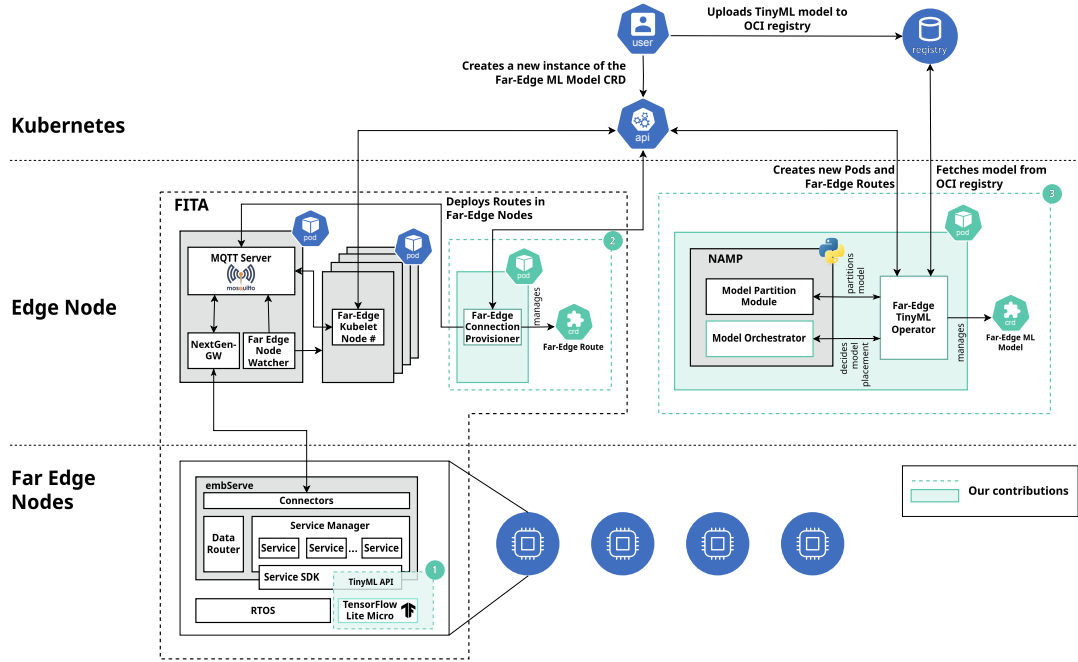


Figure 1: High-level system architecture showing the main blocks of the proposed solution: FITA with our extensions and the Far-Edge TinyML Operator, which leverages the NAMP framework.

it signals the rest of the system that the node has disappeared, triggering cleanup routines.

FITA enables the usage of Kubernetes as the deployment mechanism for TinyKubeML, allowing it to utilise standard Kubernetes interfaces for deploying TinyML models. Since TinyKubeML aims to be agnostic regarding the Far-Edge infrastructure, using k8s allows it to support solutions other than FITA, assuming that they allow deployments to Far-Edge devices. However, by adopting FITA, we are also introducing some dependencies. Services deployed to embServe-enabled devices need to be compiled using its toolchain and SDK thus TinyKubeML needs to be able to generate these services. FITA provides information about the Far-Edge runtime to k8s, which allows TinyKubeML to adapt according to the runtime and generate compatible deployable services. Within the scope of this work, we have only added support for embServe-based services, as FITA served as the basis of our work. Future extensions can integrate support for different runtimes by extending the deployment module of TinyKubeML.

While FITA already provides an extensive set of features that we leverage in this work, some are still missing, namely, i) embServe has no support for TinyML-based tools or libraries, and ii) FITA lacks a mechanism to deploy Service routes. In this work, we extend FITA to cover these two missing features as required for TinyKubeML.

3.1.1 TinyML Support. The current embServe SDK provides access to OS resources to services. This SDK provides APIs to access various communication buses (e.g., UART, I2C), sensors, and sockets, among other features. Services use the SDK to call functions that interact with these subsystems, which are dynamically linked when the service is loaded by embServe. Thus, we added a new

API to the embServe SDK that provides the functions required for TinyML inference (1 in Figure 1). Services include the model in a C buffer and call *tinymodel_load* passing the model buffer and the expected tensor arena. An opaque pointer representing the model is returned to the service, abstracting the underlying interpreter or library. Afterwards, the input of the model can be configured with *tinymodel_input* and the model inference invoked with *tinymodel_invoke*. The output can be obtained with *tinymodel_output*. Runtime information about the input and output tensors, including eventual quantization details, can be obtained with *tinymodel_tensor_info* and the model unloaded with *tinymodel_unload*. This API was then implemented in embServe. Among several possible TinyML frameworks that could be supported, we selected TensorFlow Lite Micro (TFLM) [11] for its widespread adoption and compact model format. Our embServe extension converts between the generic API and the TFLM API, manages the tensor arena for each service, and supports inference of multiple models independently. While our current proposed solution uses TFLM as the model runtime, other solutions, such as those introduced in Section 2.2, can be easily integrated into embServe by developing new implementations of the previously described API, if required by the runtime. For solutions that convert models to native code, no alterations are needed to embServe since the model is compiled into executable code. To distinguish between TinyML-enabled nodes and their model runtime, two annotations are added to the node representation in Kubernetes: (i) *embserve.fhp.pt/ml_support: true* signals that the node has support for TinyML-based services; (ii) *embserve.fhp.pt/ml_runtime: tflm* indicates that the supported model runtime of the node is TFLM.

This information allows TinyKubeML to generate services compatible with the node runtimes, even though only TFLM in embServe is currently supported.

3.1.2 Far-Edge Connection Provisioner. As explained earlier in this section, embServe supports linkage between services through its Data Router, enabling stream-like computations where the output of a service is used as input to another service. This is an important feature of service-oriented architectures, as it enables the deployment of complex, distributed applications using a set of services deployed on different devices and managed independently. TinyKubeML aims to support the distributed deployment of TinyML models, so service linkage is crucial to enable model inference without intervention of the Edge. Although embServe already supports service linkage, it is not abstracted in Kubernetes and requires users to interact with NextGenGW or embServe directly. An abstraction through Kubernetes is required to improve the user experience and support future extensions of FITA to devices using different runtimes. Thus, we add another module to FITA, the Far-Edge Connection Provisioner (FECF), a Kubernetes Operator that provides service linkage (2 in Figure 1).

Service linkage is defined in embServe as Routes, which are managed by the Data Router module. Routes define a pair of Uniform Resource Identifiers (URIs) that represent the origin of an event (Origin URI) and where it should be delivered (Destination URI). Figure 2 shows an example where the output of Service #1 from Far-Edge Node #1 is routed as the input of Service #1 from Far-Edge Node #2 using a CoAP [25] connector. Each device requires a Route instance: Far-Edge Node #1 contains the egress part of the Route which defines that the output (*out*) of local Service #1 (*local://1*) should be sent through the CoAP connector (*coap://*) to the device with address *10.10.1.1* in endpoint *e1*; Far-Edge Node #2 contains the ingress part of the Route which defines that events sent to endpoint *e1* of the CoAP connector should be delivered to the input *in* of local Service #1. These Routes are created and configured using a custom LwM2M object [20].

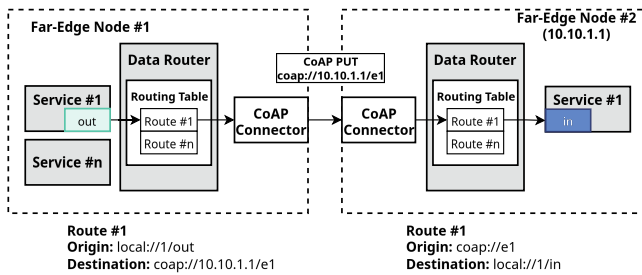


Figure 2: embServe Route example. Shows how a service output (*out*) is used as an input of another service (*in*).

We abstract Routes in Kubernetes by creating a Custom Resource Definition, *farederoutes.embserve.fhp.pt*, that defines new k8s objects with a spec containing several parameters:

- *egress* and *ingress*: these objects define the egress and the ingress Routes, respectively, and contain parameters that uniquely identify the output (egress) and the input (ingress) of the target services. The first parameter is *pod*, which identifies the Pod by name, followed by *container* that identifies the target service.

Listing 1: Far-Edge Route Custom Resource example.

```

1  apiVersion: embserve.fhp.pt/v1
2  kind: FarEdgeRoute
3  metadata:
4    name: my-route
5  spec:
6    egress:
7      pod: pod-far-edge-1
8      container: service-1
9      key: out
10   ingress:
11     pod: pod-far-edge-2
12     container: service-2
13     key: in
14   type: single

```

The *key* parameter defines which output/input should be used. These parameters are known if the Pods are already deployed, but k8s deployments can lead to scenarios where the Pod name is generated during deployment. Thus, we also define a *podSelector* parameter that can be used with Labels in alternative to the *pod* parameter.

- *type*: when a service can be deployed in multiple Nodes, the *podSelector* parameter can match multiple Pods. Thus, the *route_type* parameter defines how the Operator should handle these cases. Defining the *type* as *single* creates a single Route even if we match multiple Pods. Using *all* creates a Route for all matched Pods, one per Pod pair.

Listing 1 shows how the Route from Figure 2 can be described using the proposed Custom Resource. When this resource is created, FECF uses the Kubernetes API server to collect the current list of Pods that match the user configurations. Each Far-Edge Pod contains an Annotation detailing its services and respective outputs/inputs, which is used by FECF to validate the configurations. After validation, FECF creates the new Route instances in the Far-Edge devices using the NextGenGW protocol. With this Operator and Custom Resource, we enable the configuration of device-to-device links using the k8s APIs.

3.2 Far-Edge TinyML Operator

A generic TinyML pipeline (top of Figure 3) consists of two main phases: model training and deployment [2]. The training phase includes selecting the algorithm, collecting and preparing data for training, training the model, and optimizing it to meet the resource constraints of target devices. Common optimization techniques include quantization and pruning, which reduce the model memory and computational footprint [3]. The deployment phase involves the mechanisms required to deliver and perform the inference of the model on the target device. As discussed in Section 2, while many frameworks address the training phase, deployment is often overlooked or treated as an afterthought.

While FITA with our extensions supports the deployment of TinyML models using the standard Kubernetes APIs on heterogeneous Far-Edge devices, it still requires users to manually develop, compile, and push services compatible with the embServe runtime. This might not be an issue when developing applications that depend on monolithic models, but it quickly becomes a burden for scenarios that leverage distributed computing over heterogeneous devices, such as those envisioned by TinyKubeML.

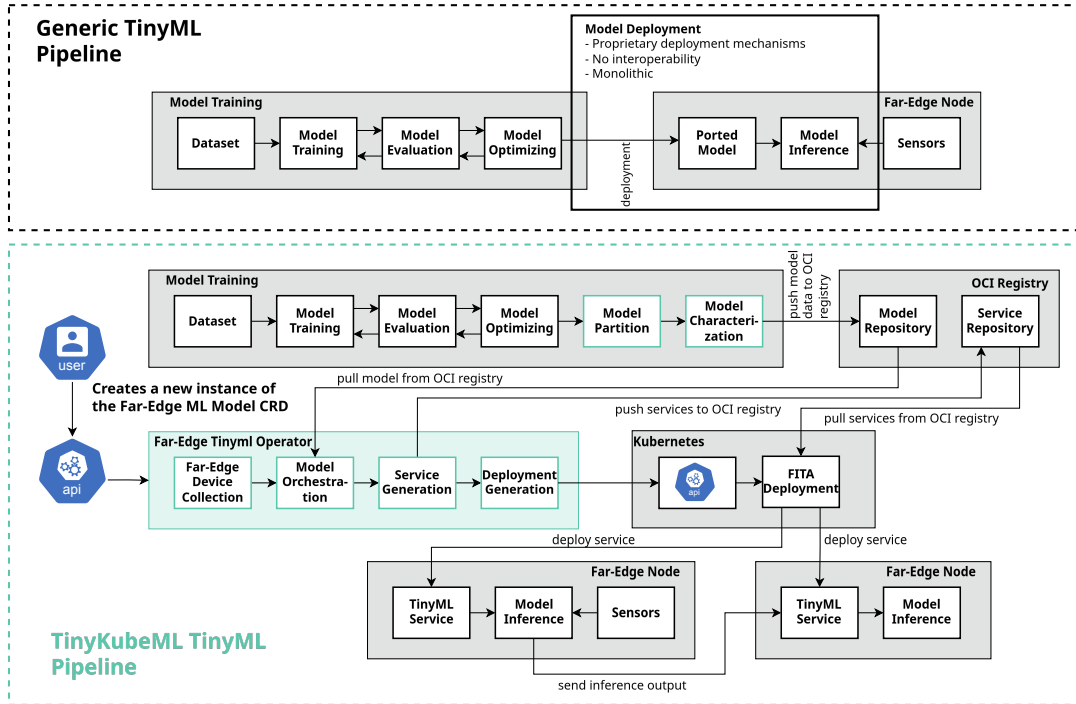


Figure 3: Generic TinyML pipeline (top, adapted from [2]) and our proposed pipeline, with the Far-Edge TinyML workflow highlighted (bottom).

Therefore, our proposed TinyML pipeline maintains the traditional training phase of the generic TinyML pipeline but extends the optimization and deployment phases by leveraging the NAMP framework [19], which we modified to fit TinyKubeML. The bottom of Figure 3 highlights the proposed Far-Edge TinyML Operator (FETMLO) workflow and its interactions with FITA.

3.2.1 Model Partition and Characterization. TinyKubeML adds two new operations to the model training phase, namely model partition and model characterization. These operations are performed using a set of Python scripts after the model is created, which collect the necessary data and prepare it for upload to an OCI registry.

Model partition uses the Model Partitioning Module of NAMP, which provides mechanisms that follow a per-layer partition approach to split a neural network into smaller submodels. The splitting procedure iterates over all possible combinations of layers, while also providing replacements for fully-connected layers composed of multiple smaller layers that contain a section of the larger layer. Currently, the partitioning module supports models in TensorFlow format, as this is the runtime we have selected to integrate into embServe. However, models in other formats, such as ONNX¹⁰, can also be used, as several tools provide mechanisms to convert or export them into a TensorFlow-compliant format.

The partition is followed by the model characterization, which collects the model resource requirements. Besides the size of the model to be deployed, the required tensor arena is another characteristic that must be considered during the deployment process, as it can consume a significant amount of memory. The tensor arena is a block of memory that is used by the runtime to allocate tensors.

Therefore, a simplified memory requirement for a model is the sum of the size of the deployable service with the tensor arena requirement. Unfortunately, TFLM does not provide a mechanism to calculate this value for an arbitrary model, requiring users to benchmark the models to determine an accurate tensor arena requirement, since the TFLM runtime implementation varies depending on the device CPU architecture and supporting libraries. To overcome this issue, we collect the required tensor arena by emulating the submodels using IoTNetEmu [26]. It is a tool for testing and evaluating IoT applications that utilizes QEMU¹¹ to emulate Far-Edge devices. We emulate a device running embServe with extensive memory capabilities and deploy the submodel services to it. After deployment, the submodel is loaded by embServe, and the required tensor arena is collected. The process is then repeated for all submodels. Currently, we only characterize the submodels on a single device type, a Cortex-M-based machine, since it is the type of device we used for validation (see Section 4.2.1). While this process provides exact tensor arena requirements, it is cumbersome because it requires emulating all submodels across all supported device architectures. Estimating the tensor arena is possible by analyzing each of the model layers, but this approach may introduce inaccuracies that impact the deployment process, as different runtime overheads are not accounted for. Nevertheless, since the characterization process is not performed during the FETMLO runtime, we opted for the emulation approach to get the most accurate model requirements. After this process, the original model, the generated submodels, and the resource metadata are packaged as an OCI artifact and pushed to the OCI-compliant registry.

¹⁰<https://onnx.ai/>

¹¹<https://www.qemu.org/>

Listing 2: Far-Edge Model Custom Resource example.

```

1  apiVersion: embserve.fhp.pt/v1
2  kind: FarEdgeMLModel
3  metadata:
4    name: anomaly-detection
5  spec:
6    image: fhp/anomaly-detection:latest
7    distributed: true
8    algorithm: kShortestPath
9    heuristic: minMemory
10   nodeSelector:
11     machine: "machine1"

```

3.2.2 Model Deployment. The deployment workflow starts when the user creates a new instance of the Custom Resource, *faredegml-models.embserve.fhp.pt*, which defines the model to be deployed and configures the model orchestration procedure. Listing 2 shows an example of a Custom Resource instance that deploys an anomaly detection model. The example shows the parameters that compose the custom resource specification:

- *image*: defines the artifact containing the model data, uploaded to a registry during the Model Characterization phase.
- *distributed*: enables model partition and distribution to multiple devices whenever the model cannot be deployed to a single device due to a lack of resources. Users can also disable model partitioning if their application requires it.
- *algorithm*: configures the algorithm used for the deployment. The example uses the currently supported *kShortestPath* algorithm, which finds the shortest path between the first layer and the last layer.
- *heuristic*: configures the heuristic used by the algorithm. The example configures the *minMemory* heuristic to minimize memory usage.
- *nodeSelector*: allows limiting the nodes considered for deployment using k8s Labels. The example limits the deployment to Far-Edge nodes installed in *machine1*.

FETMLO detects the creation of the new resource and collects the current list of Nodes that match the user configurations, together with their current resource state, using the k8s Metrics API served by the Far-Edge Kubelet of each node. Then, the Operator pulls the model from the OCI registry, validates the artifact structure, and loads the model metadata, which contains the memory requirements and the required tensor arena for each submodel. Model orchestration is then started. During this process, the model is analyzed to determine the optimal model path based on the user configurations, and each model or submodel is assigned to a specific device. While NAMP already provides an orchestration module, its implementation is naive, as it validates every path, even those that do not fit the current network state. We opted instead to implement a new algorithm (*kShortestPath*) that models the orchestration process as a directed acyclic graph (DAG) as in Eq. 1 where $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ is the set of nodes, each representing a model layer, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of valid edges, each representing a possible submodel segment from v_i to v_j .

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \quad (1)$$

Each edge $e = (v_i, v_j)$ is assigned a weight following the selected heuristic. In the case of *minMemory* shown in Listing 2 example, the

edge weight $w(e)$ can be described by Eq. 2 where $\text{Mem}(v_i, v_j)$ is the memory requirement of the submodel spanning layers v_i to v_j , including the tensor arena, and $\text{Comm}(v_j)$ is the communication overhead of the output of layer v_j to other device.

$$w(e) = \text{Mem}(v_i, v_j) + \text{Comm}(v_j) \quad (2)$$

To minimize the possible paths, an edge e is only included in $\mathcal{E}$ if the submodel can be deployed in the current set of devices. This constraint can be modeled by Eq. 3 where $\mathcal{D}$ is the set of Far-Edge nodes and $\text{Mem}_{\text{avail}}(d_k)$ is the memory available on the Far-Edge node d_k .

$$\exists d_k \in \mathcal{D} : \text{Mem}(v_i, v_j) \leq \text{Mem}_{\text{avail}}(d_k) \quad (3)$$

The goal is then to find a path $P = \{(v_1, v_i), \dots, (v_j, v_n)\}$ from input layer v_1 to output layer v_n that minimizes the sum of edge weights along the path. This is expressed in Eq. 4 where $\mathcal{P}$ is the set of all valid paths from layer v_1 to layer v_n , and P^* is the path that minimizes the sum of edge weights. Thus, the model partition minimizes both the amount of memory used by the model and the amount of communications performed during an inference.

$$P^* = \arg \min_{P \in \mathcal{P}} \sum_{e \in P} w(e) \quad (4)$$

To compute the shortest path, we use the *shortest simple paths* algorithm from the networkx¹² Python library, based on Yen's algorithm [27].

Then we extract the sequence of edges $\{(v_1, v_i), \dots, (v_j, v_n)\}$ from path P^* , where edge (v_1, v_i) represents the submodel that spans layer v_1 to v_i . The next step is to assign each of the submodels in the path to a specific device in set $\mathcal{D}$. This is achieved using Algorithm 1, which relies on greedy backtracking. This approach attempts to map each submodel in P^* to a device while respecting memory constraints. The algorithm first sorts the list of submodels in descending order by their memory requirements and then sorts the devices by their available memory in descending order. This prioritizes allocating more memory-intensive submodels, reducing the chances of early allocation decisions blocking larger submodels later.

The recursive function *Assign* begins at the first submodel e_l ($l = 1$) and iteratively tries to place it on a device with enough memory. If a feasible assignment is found, the function proceeds recursively to the next submodel e_l ($l = 2$), etc. If, at any point, no valid placement is found, the function backtracks, removing the last assignment and trying the next device. The process continues until all n^* submodels are assigned, yielding a valid mapping A , or until no feasible assignment exists, in which case the algorithm ends with an empty set ($\emptyset$). In this case path P^* is removed from $\mathcal{P}$ and the minimization in Eq. 4 is re-triggered to find a new path P^* with the next minimum cost. This cycle repeats until a valid assignment is found or all candidate paths are exhausted. After a valid assignment is found, the service generation starts. FETMLO iterates over all assignments, extracting the inputs and outputs of each submodel, and creating a service source file compatible with the embServe runtime and corresponding TinyML runtime, which includes the submodel as a C array and the code to load and invoke the model. This service also includes the logic to handle the inputs

¹²<https://networkx.org/>

Algorithm 1: Greedy Backtracking device assignment.

Input: Partitioned model
 $P^* = \{(v_1, v_i), \dots, (v_j, v_n)\} = \{e_1, \dots, e_l, \dots, e_{n^*}\}$

Input: Devices $\mathcal{D} = \{d_1, \dots, d_k, \dots, d_n\}$

Output: Valid assignment $A : P^* \rightarrow \mathcal{D}$

Sort P by $Mem(e_l)$ descending;
Sort $\mathcal{D}$ by $Mem_{avail}(d_k)$ descending;
 $l=1; A=\emptyset;$

Function Assign(l, A):

```

  if  $l > n^*$  then
    return  $A$ 
  foreach  $d_k \in \mathcal{D}$  do
    if  $Mem(e_l) \leq Mem_{avail}(d_k)$  then
       $A[l] \leftarrow d_k;$ 
      if Assign( $l+1, A$ )  $\neq \emptyset$  then
        return  $A$ 
      Remove assignment  $A[l];$ 
  return  $\emptyset$ 

```

and outputs, and to automatically invoke the model when a valid input is received. Each service is then compiled and packaged using the embServe SDK tools. Afterwards, FETMLO encapsulates the files as OCI artifacts and pushes them to the OCI registry.

Finally, FETMLO uses the k8s API to create one Pod per sub-model assignment and the required Routes using the custom resource described in Section 3.1.2. An example deployment created by FETMLO can be observed in Listing 3. It shows a distributed TinyML model where the submodel spanning layers 0-16 is deployed to the node *fare-edge-1* (lines 1-12) and the submodel with the remaining layers is deployed in the node *fare-edge-2* (lines 14-25). FETMLO also adds a *femlm.operator.fhp.pt/model* annotation (lines 6 and 19) to identify the model a Pod belongs to, and a *femlm.operator.fhp.pt/input(output)* annotation (lines 7 and 20) that indicates if the Pod is the input or output of the complete model. A Route is also created, which links the output of the first Pod as the input of the second Pod (lines 27-42). The deployment is then performed by FITA, as explained in Section 3.1. After deployment, FETMLO continues to monitor the state of the deployed Pods and Routes to detect any potential malfunctions. If a Pod or Route belonging to a model deployment changes state, FETMLO automatically deletes the current deployment and triggers a new orchestration cycle since the device network conditions have changed. This ensures that the model deployment recovers from malfunctions and ensures the model availability in the cluster, in line with Kubernetes recovery features.

3.2.3 Assignment Metrics and Heuristics. Our current approach to partitioning and assigning TinyML models focuses on minimizing the memory usage after deployment, taking into account the available memory resources on a cluster of Far-Edge nodes. For this, we utilize the algorithm and heuristic presented previously, which decouples the partition and assignment operations. While this approach enables us to deploy TinyML models in scenarios where a monolithic model deployment would be impractical, other approaches can be explored by introducing additional metrics or by adopting different heuristics with alternative optimization targets. In terms of new metrics, it can be interesting to introduce

Listing 3: Example distributed TinyML deployment generated by the Far-Edge TinyML Operator.

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: anomaly-detection-0-to-16
5    annotations:
6      femlm.operator.fhp.pt/model: anomaly-detection
7      femlm.operator.fhp.pt/input: true
8  spec:
9    containers:
10     - name: anomaly-detection-0-to-16
11       image: fhp/anomaly-detection-0-to-16:latest
12     nodeName: fare-edge-1
13 ---
14 apiVersion: v1
15 kind: Pod
16 metadata:
17   name: anomaly-detection-16-to-29
18   annotations:
19     femlm.operator.fhp.pt/model: anomaly-detection
20     femlm.operator.fhp.pt/output: true
21 spec:
22   containers:
23     - name: anomaly-detection-16-to-29
24       image: fhp/anomaly-detection-16-to-29:latest
25     nodeName: fare-edge-2
26 ---
27 apiVersion: embserve.fhp.pt/v1
28 kind: FarEdgeRoute
29 metadata:
30   name: route-57e88598
31   annotations:
32     femlm.operator.fhp.pt/model: anomaly-detection
33 spec:
34   type: single
35   egress:
36     pod: anomaly-detection-0-to-16
37     container: anomaly-detection-0-to-16
38     key: model_output_0
39   ingress:
40     pod: anomaly-detection-16-to-29
41     container: anomaly-detection-16-to-29
42     key: model_input_0

```

network information, such as bandwidth or link quality, between devices. This information can better describe the network conditions of the cluster, where heterogeneous devices can have different communication resources. Another interesting metric is related to the available energy resources. Adding the power supply type or the amount of battery left as metrics, along with an inference energy consumption estimator, can enable energy-aware orchestration. Likewise, introducing an estimated inference time based on the device's computing characteristics can also improve the orchestration process, as it enables the calculation of an estimated end-to-end inference latency in conjunction with the network information. Additionally, AI accelerators, such as neural processing units, are becoming a reality in the microcontroller space, with recently released devices (e.g., STM32N6) featuring these units. These accelerators can significantly reduce the inference time of TinyML models, therefore, the assignment algorithm needs to be aware of devices with acceleration capabilities to leverage them during the orchestration process.

As for new heuristics, the optimal deployment will depend on the use case. Our current approach minimizes memory usage with the goal of deploying a distributed model using the minimum number

of devices. This ensures that the number of communications between devices is minimized, which has several advantages in terms of the total inference time and bandwidth usage. However, our approach also implements a greedy assignment that can limit further deployments by exhausting the capabilities of the devices in use. We envision use cases where the resource usage must be balanced between a set of devices to enable further deployments. Therefore, a heuristic that attempts to balance memory, CPU, and energy usage can be beneficial for use cases that can tolerate increased total inference time. Time-sensitive use cases would benefit from a heuristic that achieves the lowest inference time, considering both network and device characteristics. Additionally, energy-focused heuristics are also interesting for IoT scenarios, as Far-Edge devices can be battery-powered. Heuristics that minimize or balance power usage, or that deploy the most computationally intensive parts of the model to mains-powered devices, could enable new scenarios and can be implemented using the metrics discussed above. Finally, being able to mix several of these optimization targets can also be relevant, where a heuristic tries to achieve the lowest inference time possible while balancing energy consumption, for example. In conclusion, TinyKubeML enables the exploration of all these scenarios due to its modular architecture, which allows future work to integrate these new metrics and heuristics.

4 Evaluation

This section evaluates the feasibility of TinyKubeML using a set of tests designed to characterize its impact on model accuracy, deployment performance, and resource requirements.

4.1 Experimental Scenario

For our evaluation, we considered a predictive maintenance scenario, one of the current key applications of the Industrial IoT [2]. Our scenario focuses on anomalous sound detection, where Far-Edge devices equipped with microphones are used to identify whether sound emitted from a target machine is normal or anomalous using a TinyML model, which can indicate or predict mechanical failures.

4.1.1 Anomaly Detection Model. Our tests used the Deep AutoEncoder model from MLPerf Tiny [28]. This model was designed for microcontroller-based devices and is provided to the community as a benchmark for evaluating TinyML performance. The auto-encoder model encodes the input data into a feature representation, which is then used to reconstruct the input data. Since the model is only trained on normal data, error characterization between the original and reconstructed input data is used to classify whether the data is anomalous, with anomalous data presenting a higher error. We trained our model using the MLPerf Tiny tools and the ToyCar data from the ToyADMOS [29] dataset, which comprises 7000 samples of normal sound data for training and 1400/1600 normal/abnormal sound samples for testing. The training process uses the baseline configurations of MLPerf Tiny. To reduce the memory requirements, we leverage the NAMP tools to quantize the output model and all generated submodels using full integer post-training quantization (int8). The details of the original and quantized model versions are shown in Table 1, where AUC refers to the area under the ROC curve and pAUC is the adjusted AUC if the maximum false positive rate is set as 0.1. As expected, the quantization reduces

the size of the final model to 25% of its original size, with a minor reduction in accuracy. We use this version of the model for our tests. Table 2 shows a summary of the most relevant model layers.

Model	Layers	Parameters	Size (KB)	AUC	pAUC
ToyCar	29	269992	1061	0.87	0.751
ToyCar (Quantized)	29	269992	278	0.865	0.684

Table 1: Anomaly Detection Model details.

Layer	Type	Size (B)	Tensor Arena (B)	Inputs	Outputs
29	Dense	85704	1280	128	640
2	Dense	83656	1280	640	128
5, 8, 11, 20, 23, 25	Dense	18120	768	128	128
14	Dense	18120	768	128	8
17	Dense	2280	768	8	128
others	Activation Normalization	912 - 1760	768 - 1024	8 - 128	8 - 128

Table 2: Anomaly Detection Model layer details.

Scenario	Devices	Available Memory
1	1	350KB
2	2	2 x 256KB
3	4	4 x 128KB
4	5	2 x 128KB 3 x 64KB

Table 3: Evaluation scenarios.

4.1.2 Device Scenarios. We defined four different scenarios that vary in the number of Far-Edge devices and their available memory. The details of each scenario are shown in Table 3. Scenario 1 is used as a baseline, as the amount of memory in the single Far-Edge device is sufficient for the full model. The others represent more constrained scenarios, requiring model partitioning to deploy the model, and are therefore useful for evaluating TinyKubeML.

4.2 Experimental Setup

4.2.1 Hardware. Our experimental setup uses a server and a test host computer. The server runs the Kubernetes cluster and hosts both FITA and the container registry. It is a generic server equipped with an Intel Core i7-4600 processor, 32GB of RAM, and a 256GB SSD. The test host computer is a generic laptop with an Intel Core i7-8665U CPU, 32GB of RAM, and a 256GB NVMe SSD, which is used to coordinate the tests. Both machines are running Ubuntu 22.04 LTS and are connected to the same local network via Ethernet running at 1 Gbps. We use several nRF7002 development kits¹³ as Far-Edge devices. These kits use an nRF5340 SoC, which includes an ARM-Cortex M33 CPU with 1MB of flash and 512KB of RAM, and a nRF7002 Wi-Fi companion chip. The devices connect to the server’s local network over 802.11ax/HE using the 5GHz band. The round-trip latency measured from the server to a Far-Edge device is approximately 5 ms.

¹³<https://www.nordicsemi.com/Products/Development-hardware/nRF7002-DK>

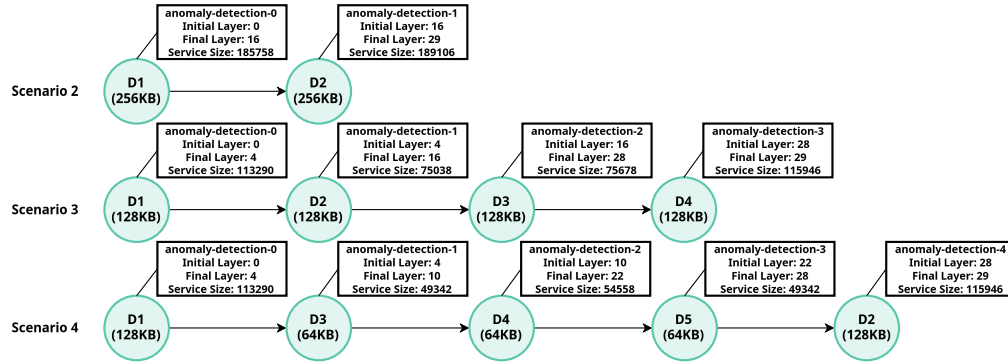


Figure 4: TinyKubeML deployment solutions for several scenarios.

4.2.2 Software. We deployed a single-node Kubernetes cluster using MicroK8s (v1.32.3), with the Control Plane, Kubelet, FITA, and application Pods all running on the same server. Although a more realistic setup would have the Control Plane in a separate device and include multiple nodes, our configuration provides a representative approximation of real-world performance. We enabled the MicroK8s registry add-on to host our local OCI registry. To simulate different Far-Edge nodes resource constraints, we artificially limit their available memory by configuring the embServe firmware image at compile time based on the Scenario under test.

4.3 Experimental Results

4.3.1 Model Deployment. To evaluate the model deployment procedure of TinyKubeML, we started by creating an instance of the Far-Edge Model Custom Resource described in Listing 2. Figure 4 shows the solutions provided by the Far-Edge TinyML Operator, with Scenario 1 omitted since the model is deployed in the only available device. The solution for Scenario 2 is expected since it divides the model into two, minimizing communication costs by splitting the model where the number of outputs is lower (layer 16). Scenario 3 follows the same logic but now also divides each half into two to cope with the increased resource constraints. As for Scenario 4, the largest layers (1 and 29) are assigned to the devices with the highest memory available (D1 and D2), while the remaining layers are evenly distributed among the other devices. For each deployment, we measured the deployment cycle time of the Far-Edge TinyML Operator, from the creation of the CRD to the submission of the deployment to Kubernetes. Figure 5 shows the results for 100 deployments on each Scenario. Results show that, for Scenario 1, the operator takes approximately 11 seconds to fully create the deployment, with most of the time (81%) spent loading the model. During this process, we utilize TensorFlow to load and validate each submodel, which significantly contributes to the overall time. Fetching the model from the OCI registry and creating the Kubernetes deployment are other significant procedures, which account for 17% of the cycle time due to the required network requests. The remaining time is utilized by the Far-Edge device collection, as no orchestration is required in this scenario. To better contextualize the results, we can compare a TinyKubeML deployment with a NAMP deployment without Kubernetes integration. Compared to the base NAMP, TinyKubeML added the Model OCI Pull, Far-Edge Collection, and Model Deployment operations, which account for

18% of the total deployment time in Scenario 1, showing a low overhead in this scenario. When we move to Scenario 2, the cycle time increases. We can observe an increased Far-Edge collection time, a predictable result since the number of requests to k8s increases with the number of devices, similar to what occurs with the model deployment. The model loading time also increases marginally since there is an additional model to prepare for deployment. The Kubernetes integration overhead accounts for 23.6% of the total deployment time in this scenario. For Scenarios 3 and 4, the trends from Scenario 2 continue, with model orchestration representing roughly 2% of the total cycle time in Scenario 4. The integration overhead peaks at 37% of the total deployment time in Scenario 4. These results show that TinyKubeML can autonomously deploy a fully distributed TinyML model in a constrained scenario under 15 seconds. We also measured the operator resource usage during each deployment cycle. Figure 6 shows the CPU time results while Figure 7 shows the peak memory usage. Results for the CPU time show that Scenario 1 is mostly CPU-bound, as the CPU time accounts for 89% of the total cycle time. This trend is attenuated when evaluating the other scenarios, reaching a value of 76% for Scenario 4 due to the increased number of models to load and deploy, which require waiting for I/O operations. These results indicate that a more capable CPU can reduce the deployment time. Finally, the memory usage results show that Scenarios with more submodels to deploy require more memory, with an increase of around 50 MB per submodel when compared with Scenario 1 (single model). These results show that, although resource usage increases for more complex scenarios, this increase is mostly negligible in our tests, with a 37% increase in memory usage and an 18% increase in CPU time when comparing Scenario 1 with Scenario 4. Still, the peak memory consumption was 920 MB, indicating that our solution can scale for more complex scenarios while still fitting on current low-spec devices that support Kubernetes.

4.3.2 Inference Characterization. The partition of the quantized model in distributed scenarios is not lossless since it requires the creation of submodels that can lead to performance degradation due to approximations. Therefore, we evaluated the distributed inference by feeding the test data as input to the first device and by collecting the inference output to assess the AUC and pAUC for the distributed scenarios. Our results show an AUC of 0.864 and a pAUC of 0.683 across all distributed scenarios (2, 3, and 4),

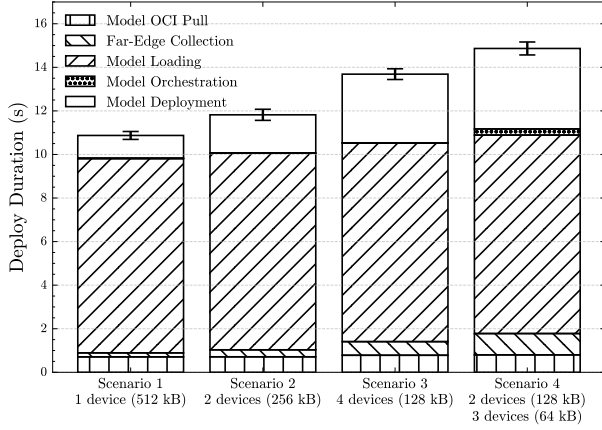


Figure 5: FETMLO deployment time under several scenarios.

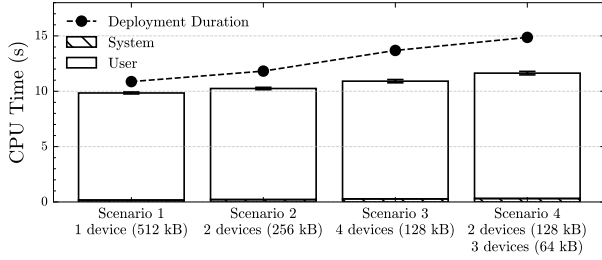


Figure 6: FETMLO CPU time during a deployment cycle.

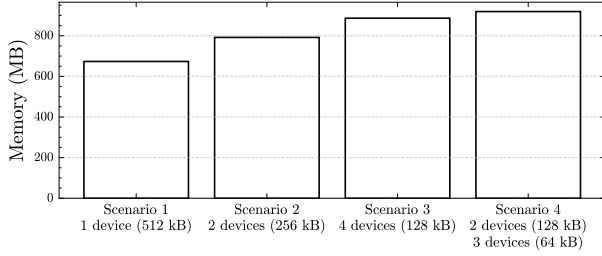


Figure 7: FETMLO peak memory usage during a deployment cycle.

which represents a negligible accuracy loss compared to the non-distributed results (0.865, 0.684). We also collected the inference time during the test, presented in Figure 8. Results show that the total inference time is dominated by the communication and runtime overhead, representing 63% of the total time in Scenario 1. Looking into Scenario 2, we observe a minimal increase in communication time, as the additional communication hop packet is relatively small (26 bytes). The inference time increases from 12 ms to 18.6 ms, which we attribute to the overhead of the inference runtime, as it needs to perform two separate inferences and removes tensor arena optimizations, since the layers that require more tensor operations (2 and 29) are deployed to different devices. The inference time does not increase when we look into Scenarios 3 and 4, which seems to corroborate our hypothesis. As the number of devices increases, the communication time also increases, representing 77% of the total time in Scenario 4, which highlights the importance of using the communication packet size in the partitioning heuristic. Finally,

we also observed that most of the variability in our results comes from the communication time, accounting for 85% of the standard deviation observed in Scenario 4.

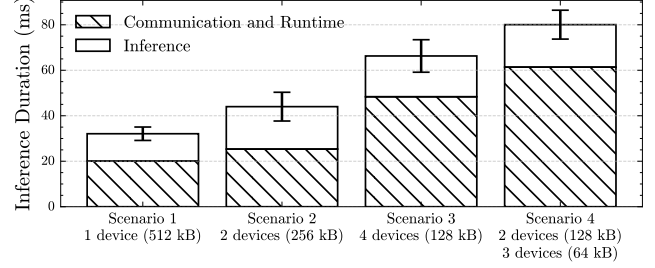


Figure 8: Inference time under several scenarios.

4.3.3 Deployment Recovery. One key feature of the Far-Edge TinyML Operator is its ability to automatically redeploy a model in the event of a malfunction. The operator continuously monitors the deployment state and triggers a full redeployment if it detects that any Pod or Route associated with the model has been removed. Upon detecting such a failure, the operator cleans up any remaining Pods and Routes to reset the deployment. A new deployment cycle then begins, and the model is redeployed. To evaluate this recovery mechanism, we measured recovery times across different scenarios by force-deleting one of the model Pods. Results from 100 recovery cycles are shown in Figure 9.

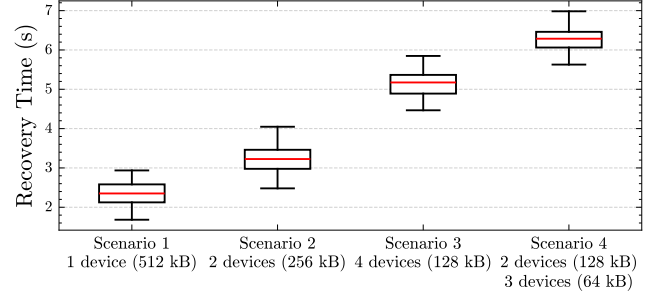


Figure 9: Recovery time for a deployment under several scenarios.

The recovery time follows the same trends as the model deployment results, where scenarios with more devices require longer re-deployment times. However, redeployments are significantly faster than deployments since all model data is cached by the Operator, allowing it to skip the Model OCI Pull and Model Loading phases shown in Figure 5. Our results show that TinyKubeML can redeploy a model after detecting a malfunction in under 3 seconds in a single-device scenario (Scenario 1). In a more complex scenario (Scenario 4), the time increases to 7 seconds due to the increased number of devices. These results demonstrate TinyKubeML’s ability to handle deployment failures and maintain application continuity with minor service interruptions after malfunctions.

5 Conclusion

In this paper, we presented TinyKubeML, a Kubernetes-based framework that enables resource-aware deployment of TinyML models

on Far-Edge clusters. Our solution enables users to easily deploy TinyML models while abstracting the heterogeneity of Far-Edge devices by leveraging a custom Operator capable of automatically partitioning models, converting them into Far-Edge deployable artifacts, and deploying them according to configurable heuristics. To the best of our knowledge, TinyKubeML is the first solution to seamlessly integrate TinyML deployments into a standard orchestration framework, thereby bridging the gap between the Cloud and the Far-Edge. Our evaluation demonstrates that TinyKubeML can deploy a distributed Anomaly Detection model across five devices in under 15 seconds, with no loss in accuracy, while maintaining service continuity through automated recovery mechanisms.

Future work will focus on validating TinyKubeML with additional models typical of TinyML scenarios (image classification, keyword detection) and with different cluster compositions (protocols, architectures). Additionally, we will explore new orchestration strategies to support broader application scenarios by integrating new metrics, heuristics, and optimization targets in the Far-Edge TinyML Operator. Furthermore, we want to validate our solution in real-world environments with diverse and heterogeneous Far-Edge devices.

Acknowledgments

This work has received funding from the Alliance for the Energy Transition (56) co-financed by the Recovery and Resilience Plan (PRR) through the European Union, from the Horizon Europe research and innovation programme of the European Union, under grant 101092912, project MLSysOps, and from the Research Centre in Real-Time and Embedded Computing Systems – CISTER, funded by national funds through the FCT/MCTES (PIDDAC): Base Funding - UIDB/04234/2020.

References

- [1] Lionel Vailshery. Number of internet of things (iot) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033, 2024. URL <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>.
- [2] Visal Rajapakse, Ishan Karunanayake, and Nadeem Ahmed. Intelligence at the extreme edge: A survey on reformable tinyml. *ACM Computing Surveys*, 55, 7 2023. doi: 10.1145/3583683.
- [3] Dr Lachit Dutta and Swapna Bharali. Tinyml meets iot: A comprehensive survey. *Internet of Things (Netherlands)*, 16:100461, 2021. doi: 10.1016/j.iot.2021.100461.
- [4] Ramon Sanchez-Iborra and Antonio F. Skarmeta. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20: 4–18, 2020. ISSN 15580830. doi: 10.1109/MCAS.2020.3005467.
- [5] Carlos Resende, João Oliveira, Filipe Sousa, Waldir Moreira, and Luis Almeida. Improving far-edge device management in iot applications using kubernetes. *IEEE Open Journal of the Industrial Electronics Society*, pages 1–24, 2025. doi: 10.1109/OJIES.2025.3581076.
- [6] Muhammad Alam, Joao Rufino, Joaquim Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018. doi: 10.1109/MCOM.2018.1701233.
- [7] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. Fog based framework for iot service provisioning. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2019. doi: 10.1109/CCNC.2019.8651835.
- [8] Sami Yangui, Pradeep Ravindran, Ons Bibani, Roch H Glitho, Nejib Ben Hadj-Alouane, Monique J Morrow, and Paul A Polakos. A platform as-a-service for hybrid cloud/fog environments. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–7. IEEE, 2016. doi: 10.1109/LANMAN.2016.7548853.
- [9] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Genesis: Continuous orchestration and deployment of smart iot systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875. IEEE, 2019. doi: 10.1109/COMPSAC.2019.00127.
- [10] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. Reprogramming low-end IoT devices from the cloud. In *3rd Cloudification of the Internet of Things Conference, CIoT 2018*, pages 1–6. IEEE, jul 2019. doi: 10.1109/CIOT.2018.8627129.
- [11] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems. 2021. doi: 10.48550/arXiv.2010.08678.
- [12] emlearn: Machine learning inference engine for microcontrollers and embedded devices, 2010. URL <https://doi.org/10.5281/zenodo.2589394>.
- [13] Tomasz Szydlo and Marcin Nagy. Management of tinyml-enabled internet of things devices. *IEEE Micro*, 45(1):87–94, 2025. doi: 10.1109/MM.2024.3382488.
- [14] Hiroshi Doyo, Roberto Morabito, and Martina Brachmann. A tinyml ecosystem for machine learning in iot: Overview and research challenges. In *2021 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–5, 2021. doi: 10.1109/VLSI-DAT52063.2021.9427352.
- [15] Syed Ali Raza Zaidi, Ali M. Hayajneh, Maryam Hafeez, and Q. Z. Ahmed. Unlocking edge intelligence through tiny machine learning (tinyml). *IEEE Access*, 10:100867–100877, 2022. doi: 10.1109/ACCESS.2022.3207200.
- [16] Mayank Gulati, Koen Zandberg, Zhaolan Huang, Gerhard Wunder, Cedric Adjih, and Emmanuel Baccelli. Tdmil: Tiny distributed machine learning for microcontroller-based interconnected devices. *IEEE Access*, 12:167810–167826, 2024. doi: 10.1109/ACCESS.2024.3492921.
- [17] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. Dianne: Distributed artificial neural networks for the internet of things. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT, M4IoT 2015*, page 19–24, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2836127.2836130.
- [18] Pengzhen Li, Erdem Koyuncu, and Hulya Seferoglu. Adaptive and resilient model-distributed inference in edge computing systems. *IEEE Open Journal of the Communications Society*, 4:1263–1273, 2023. doi: 10.1109/OJCOMS.2023.3280174.
- [19] Fernando Rego, João Oliveira, Filipe Sousa, and Luis Almeida. Namp: A network-aware model partitioning framework for constrained devices. In *Proceedings of the 4th International Workshop on Real-Time and IntelliGent Edge Computing, RAGE '25*, New York, NY, USA, 2025. Association for Computing Machinery. doi: 10.1145/3722567.3727844.
- [20] João Oliveira, Filipe Sousa, and Luis Almeida. embservice: Embedded services for constrained devices. In *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, pages 1–8, 2023. doi: 10.1109/WFCS57264.2023.10144123.
- [21] Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for Constrained-Node Networks. RFC 7228, May 2014. URL <https://www.rfc-editor.org/info/rfc7228>.
- [22] Carlos Resende, Waldir Moreira, and Luis Almeida. Nextgengw - a software framework based on mqtt and semantic definition format. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 141–148, 2023. doi: 10.1109/SMARTCOMP58114.2023.00035.
- [23] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. MQTT Version 5.0. OASIS Standard mqtt-v5.0-os, Organization for the Advancement of Structured Information Standards, 2019. URL <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf>.
- [24] Michael Koster, Carsten Bormann, and Ari Keränen. Semantic Definition Format (SDF) for Data and Interactions of Things. Internet-Draft draft-ietf-asdf-sdf-18, Internet Engineering Task Force, February 2024. URL <https://datatracker.ietf.org/doc/draft-ietf-asdf-sdf/18/>. Work in Progress.
- [25] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014. URL <https://www.rfc-editor.org/info/rfc7252>.
- [26] João Oliveira, Filipe Sousa, and Luis Almeida. Iotnetemu - a framework to emulate and test iot applications. In *Proceedings of the 19th ACM International Symposium on QoS and Security for Wireless and Mobile Networks, Q2SWinet '23*, page 33–37, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3616391.3622774.
- [27] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, pages 712–716, 1971. doi: 10.1287/mnsc.17.11.712.
- [28] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021. doi: 10.48550/arXiv.2106.07597.
- [29] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In *Proceedings of IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 308–312, November 2019. doi: 10.48550/arXiv.1908.03299.